ZOHAIR BANOORI

GIT CHEAT SHEET

28 JULY 2025 / 11:44 PM

INSTALLATION & GUIs

GitHub for Windows

https://windows.github.com

GitHub for Mac:

https://mac.github.com

Git for All Platforms:

http://git-scm.com

SETUP

Configuring user information used across all local repositories

- git config --global user.name "[firstname lastname]"

  # Set a name that is identifiable for credit when reviewing version history

- git config --global user.email "[valid-email]"

  # Set an email address that will be associated with each history marker

- git config --global color.ui auto

  # Set automatic command line coloring for Git for easy reviewing

---

CREATE & INITIALIZE

Creating and cloning Git repositories

- `git init`

  `# Initialize an existing directory as a Git repository`

- `git clone [url]`

  `# Retrieve an entire repository from a hosted location via URL`

STAGE & SNAPSHOT

Working with snapshots and the Git staging area

- `git status`

  `# Show modified files in working directory, staged for your next commit`

- `git add [file]`

  `# Add a file as it looks now to your next commit (stage)`

- `git reset [file]`

  `# Unstage a file while retaining the changes in working directory`

- `git diff`

  `# Diff of what is changed but not staged`

- `git diff --staged`

  `# Diff of what is staged but not yet committed`

- `git commit -m "[descriptive message]"`

  `# Commit your staged content as a new commit snapshot`

UNDOING CHANGES

Mistake recovery and safe rollback options

- git checkout <branch>

  # Switch to another branch (e.g., git checkout main)

- git reset

  # Unstage staged files (after git add)

- git reset --hard [commit]

  # Reset to specific commit (dangerous — loses local changes)

- git stash

  # Temporarily shelve your changes to clean your working directory

- git commit --amend

  # Modify the most recent commit (do not use on published commits)

- git revert [commit]

  # Revert changes by creating a new commit that undoes a specific commit

Refer to Git Basics - Undoing Things for further details.

BRANCH & MERGE

Branch operations and history tracking

- git branch

  # List your branches. A * will appear next to the currently active branch

- git branch [branch-name]

  # Create a new branch at the current commit

- git branch -d [name]

  # Delete a branch from your repository

- git branch -D [name]

  # Force delete a branch from your repository

- git checkout [branch-name]

  # Switch to another branch and check it out into your working directory

- git checkout -b [new-branch]

  # Create a new branch and switch to it

- git merge [branch]

  # Merge the specified branch’s history into the current one

- git merge --abort

  # Abort a merge and return to the pre-merge state (use after merge conflicts)

- git log

  # Show all commits in the current branch’s history

- `git log --graph`

  `# Print an ASCII graph of the commit and merge history`

- `git log --oneline`

  `# Print each commit on a single line`

SHARE & UPDATE

Synchronizing your repository with remotes

- `git remote`

  `# Manage the set of tracked remote repositories`

- `git remote -v`

  `# Show remote URLs for fetch/push`

- `git remote show <name>`

  `# Display detailed information about a specific remote`

- `git remote update`

  `# Fetch updates from all remotes or a group of remotes`

- `git remote add [alias] [url]`

  `# Add a git URL as an alias`

- `git branch -r`

  `# List all remote-tracking branches`

- `git fetch [alias]`

  `# Fetch all the branches from that Git remote`

- git merge [alias]/[branch]

  # Merge a remote branch into your current branch to bring it up to date

- git push [alias] [branch]

  # Transmit local branch commits to the remote repository branch

- git pull

  # Fetch and merge any commits from the tracking remote branch

---

TEMPORARY COMMITS

Preserve work-in-progress using stash

- git stash

  # Save modified and staged changes

- git stash list

  # List stack-order of stashed file changes

- git stash pop

  # Write working from top of stash stack

- git stash drop

  # Discard the changes from top of stash stack

TRACKING PATH CHANGES

Tracking file renames and deletions

- `git rm [file]`

  `# Delete the file from project and stage the removal for commit`

- `git mv [existing-path] [new-path]`

  `# Change an existing file path and stage the move`

- `git log --stat -M`

  `# Show all commit logs with indication of any paths that moved`

INSPECT & COMPARE

Comparing branches, diffs, and commit history

- `git log`

  `# Show the commit history for the currently active branch`

- `git log branchB..branchA`

  `# Show the commits on branchA that are not on branchB`

- `git log --follow [file]`

  `# Show the commits that changed file, even across renames`

- `git diff branchB...branchA`

  `# Show the diff of what is in branchA that is not in branchB`

- `git show [SHA]`

  `# Show any object in Git in human-readable format`

REWRITE HISTORY

Rewriting and cleaning up commit history

- `git rebase [branch]`

  `# Apply any commits of current branch ahead of specified one`

- `git reset --hard [commit]`

  `# Clear staging area, rewrite working tree from specified commit`

Use with caution. **Do not** rewrite history on shared/public branches.

IGNORING PATTERNS

Preventing unintentional staging or committing of files

- `git config --global core.excludesfile [file]`

  `# System-wide ignore pattern for all local repositories`

`.gitignore` example:

```
logs/

*.notes

pattern*/
```

Save a file with desired patterns as `.gitignore` with either direct string matches or wildcard globs.

SHA-1 & OBJECTS

Git internal structure — commit identification and integrity

Git uses SHA-1 hashes for commit identification.

- SHA-1 is a cryptographic hash function
- It generates a unique digital fingerprint for each file/commit
- Ensures file integrity and serves as a reference (e.g., in `git revert [SHA]`)

Hashes are visible in `git log` or on GitHub pages and are used across many Git commands.

MERGE CONFLICT RESOLUTION GUIDE

In Git, merge conflicts, or conflicts that occur when merged branches have competing commits, are not uncommon when working with a team of developers or when working with open-source software. This study guide provides you with tips for resolving merge conflicts.

Tips for resolving merge conflicts

- After running `git merge`, a message will appear informing that a conflict occurred on the file.
- Read the error messages that imply you cannot push your local changes to GitHub, especially the remote changes with `git pull`.
- Use the command line or GitHub Desktop to push the change to your branch on GitHub after you make a local clone of the repository for all other types of merge conflicts.
- Before merging any commits to the `master` branch, push it into a remote repository so that collaborators can view the code, test it, and inform you that it's ready for merging.

- Use the `git rebase` command to replay the new commits on top of the new base and then merge the feature branch back into the `master`.

Key takeaways

It is important to effectively resolve merge conflicts because local changes cannot be made to Git until the merge conflicts have been locally resolved. Once all conflicts have been resolved, changes can be pushed to Git and merged in a pull request.

GIT FORKS AND PULL REQUESTS GUIDE

GitHub is an open-source platform for collaboration and knowledge sharing, allowing users to explore code created by others. This study guide will provide you with pointers on effectively using the platform to make pull requests in the Git environment.

Pull requests

Pull requests allow you to inform fellow contributors about changes that have been made to a branch in Git. When pulling requests, you can discuss and evaluate proposed changes before implementing changes to the primary branch.

You can eventually merge changes back into the main repository (or repo) by creating a pull request. However, it is important to note that before any changes are made to the original code, GitHub creates a fork (or a copy of the project), which allows changes to be committed to the fork copy even if changes cannot be pushed to the other repo. Anyone can suggest changes through the inline comment in pull requests, but the owner only has rights to review and approve changes before merging them. To create a pull request:

- Make changes to the file.
- Change the proposal and complete a description of the change.
- Click the Proposed File Change button to create a commit in the forked repo to send the change to the owner.

- Enter comments about the change. If more context is needed about the change, use the text box.
- Click Pull Request.

When creating multiple commits, a number next to the pull request serves as the identifier for accessing the pull requests in the future. This is important because it allows project maintainers to follow up with questions or comments.

For more information on creating pull requests, click the following link: Creating a pull request.

Pull request merges

You can merge pull requests by retaining the commits. Below is a list of pull request merge options that you can use when merging pull requests.

- **Merge commits**: All commits from the feature branch are added to the base branch in a merge commit using the `--no-ff` option.
- **Squash and merge commits**: Multiple commits of a pull request are squashed, or combined into a single commit, using the fast-forward option. It is recommended that when merging two branches, pull requests are squashed and merged to prevent the likelihood of conflicts due to redundancy.
- **Merge message for a squash merge**: GitHub generates a default commit message, which you can edit. This message may include the pull request title, pull request description, or information about the commits.
- **Rebase and merge commits**: All commits from the topic branch are added onto the base branch individually without a merge commit.
- **Indirect merges**: GitHub can merge a pull request automatically if the head branch is directly or indirectly merged into the base branch externally.

Key takeaways

Pull requests are a crucial tool you can use for efficiently capturing, implementing, and receiving approvals for changes. These

capabilities are made possible through collaboration. Practicing pull requests can help you hone your skills and contribute to a project.

CODE REVIEWS GUIDE

Code reviews are critical for producing high-quality, maintainable code, especially in large-scale or collaborative projects. Adopting consistent coding standards—like those in Google's style guides—ensures uniformity across teams and makes code easier to understand. This section introduces key code review strategies and highlights the role of pull request reviews in modern development workflows.

Google style guides

Every major open-source project includes a style guide, which is a set of norms for writing code for that project. When all the code in a huge codebase is written in the same manner, it is considerably simpler to understand.

You can find the project and style guide for Google code here.

Code review

Code review, also referred to as peer code review, is the deliberate and methodical gathering of other programmers to examine each other's code for errors. Code review can speed up and simplify the software development process, unlike other techniques. Peer reviews also save time and money, especially by catching the kinds of defects that could sneak through testing, production, and into the laptops of end users.

Common code review strategies

- **Pair programming**: Engineers work side-by-side on the same code. Useful for mentoring, but less objective and more resource-intensive.
- **The email thread**: Code is sent via email for review. Flexible but can lead to disorganized feedback.

- **Over the shoulder**: A developer walks a peer through their code directly. Informal and effective for quick feedback.
- **Tool assisted**: Using browser- or IDE-integrated tools to asynchronously review and track code feedback, enabling efficient, non-local review processes.

Pull request reviews

A pull request (PR) is a way to review new code before merging it into a main branch on GitHub. Contributors can comment on, approve, or request changes to proposed updates. Repository admins can require approval before merging.

Anyone with read access can review and suggest inline changes. Learn more about reviewing PRs here.

Five tips for pull request reviews

- **Be selective with reviewers**: Add only a reasonable number of reviewers to avoid inefficiency.
- **Timely reviews**: Ideally complete within two hours to reduce context switching and delays.
- **Constructive feedback**: Feedback should be specific, helpful, and respectful.
- **Detailed pull request description**: Include a comprehensive summary of changes, usage, design, and additional reviewer notes.
- **Interactive rebasings**: Keep commit history clean by editing commits before merge.

Key takeaways

- **Consistent coding standards** improve readability and maintenance.
- **Code reviews** catch defects early and improve quality through collaboration.
- **Review strategies** should match project context—pair programming, async tools, email, or over-the-shoulder.
- **Pull request reviews** enable structured collaboration and code validation before merging.

GIT TERMS & DEFINITIONS

- **Git**: A free open source version control system available for installation on Unix-based platforms, Windows and macOS.
- **GitHub**: A web-based Git repository hosting service that enables sharing, access, and cloning of repositories.
- **Version control systems (VCS)**: A tool to safely test code before releasing it, allow multiple people to collaborate on the same coding projects together, and stores the history of that code and configuration.
- **Source Control Management (SCM)**: A tool similar to VCS to store source code.
- **Repository**: An organization system of files that contain separate software projects.
- **Git directory**: A database for a Git project that stores the changes and the change history.
- **Distributed**: Each developer has a complete copy of the repository on their local machine.
- **Commit**: A command to make edits to multiple files and treat that collection of edits as a single change.
- **Commit files**: A stage where the changes made to files are safely stored in a snapshot in the Git directory.
- **Commit ID**: An identifier next to the word `commit` in the log.
- **Commit message**: A summary and description with contextual information on the parts of the code or configuration of the commit change.
- **Git staging area**: A file maintained by Git that contains all the information about what files and changes are going to go into the next commit.
- **Stage files**: A stage where the changes to files are ready to be committed.
- **Modified files**: A stage where changes have been made to a file, but they have not been stored or committed.
- **Tracked**: A file's changes are recorded.
- **Untracked**: A file's changes are not recorded.
- **Branch**: A pointer to a particular commit, representing an independent line of development in a project.
- **Head**: This points to the top of the branch that is being used.
- **Master**: The default branch that Git creates when a new repository is initialized; commonly used to place the approved pieces of a project.
- **Merge**: An operation that combines the origin/master branch into a local master branch.
- **Fast-forward merge**: A merge when all the commits in the checked out branch are also in the branch that's being merged.
- **Three-way merge**: A merge that uses the snapshots at the two branch tips along with their most recent common ancestor (the commit before the divergence).
- **Merge conflict**: This occurs when the changes are made on the same part of the same file, and Git won't know how to merge those changes.

- **Rollback**: The act of reverting changes made to software to a previous state.
- **Rebasing**: The act of changing the base commit used for a branch.
- **Patch**: A command that can detect that there were changes made to the file and will do its best to apply the changes.
- **Diff**: A command to find the differences between two files.
- **Git log**: A log that displays commit messages.
- **Remote repositories**: Repositories that enable developers to work independently on local copies while contributing to a shared project.
- **Remote branches**: Read-only branches that reflect data from a remote repository.
- **Private key**: A secret cryptographic key used to decrypt data encrypted with the corresponding public key.
- **Public key**: A cryptographic structure used for secure communication and validating digital signatures.
- **Secure Shell (SSH)**: A secure protocol for connecting to servers remotely.
- **SSH protocol**: A standard based on public-key encryption used for remote server access.
- **SSH key**: A credential used for SSH authentication.
- **SSH client**: Software that initiates a secure connection to an SSH server.
- **SSH server**: A system that accepts incoming SSH connections, authenticates them, and establishes secure sessions.
- **Application Programming Interface (API) key**: An authentication token used to call an API and identify the person, programmer, or program accessing a system.
- **Computer protocols**: Guidelines published as open standards that allow protocols to be implemented across various products.
- **DNS zone file**: A configuration file that specifies the mappings between IP addresses and host names in your network.

EDUCATION

GitHub is **free** for students and teachers. Discounts available for other educational uses.

- **Email:** education@github.com
- **Website:** https://education.github.com