Git and Version Control

Article Ideas

- Licenses on GitHub Repos

Getting Started

- Basics

Why Do We Need a Version Control System?

Q: Why do we need a version control system?
A: To track changes in project files over time (time-stamped history) and collaborate with others (merging work and resolving conflicts).

Types of Version Control Systems

Q: What are the two types of version control systems?
A:

- **Centralized VCS** (Single point of failure)
  - Examples: Subversion (SVN), Team Foundation Server (TFS)
  - Requires the central server to be online for access.
- **Distributed VCS** (Everyone has a copy)
  - Examples: Git, Mercurial
  - Snapshots are locally stored and can be synchronized over a network.

Using Git

Q: How can Git be used?
A:

- **CLI (Command Line Interface)** → More powerful and flexible.
- **GUI-based tools:**
  - GitKraken (Cross-platform & visually appealing)
  - Sourcetree (Popular for beginners)
- **IDE-based tools:**
  - VS Code Extensions (e.g., GitLens)

Note: GUIs have limitations—buttons, scroll bars, and graphs can only do so much.
Learning **CLI first** ensures conceptual clarity and allows seamless transition to GUI tools used by your team.

- Installing and Configs

Installing Git

- Download from git-scm.com/downloads
- Verify installation:
- `css`
- `CopyEdit`
- `git --version`

Configuring Git

Git requires initial setup for user identity, editor preferences, and line endings.

1. User Identity

```
arduinoCopyEditgit config --global user.name "your_name" git config --global user.email "your_email@
```

2. Default Editor (Use VS Code)

```
cssCopyEditgit config --global core.editor "code --wait"
```

- The --wait flag ensures the terminal waits until VS Code is closed.

3. Editing Git Configuration

```
luaCopyEditgit config --global -e
```

(This opens Git's global config in VS Code.)

Managing Line Endings (EOL)

Different OS handle end-of-line (EOL) characters differently:

- **Windows:** Uses \r\n (Carriage Return + Line Feed)
- **Mac/Linux:** Uses \n (Line Feed only)

To handle this automatically:

```
csharpCopyEditgit config --global core.autocrlf true # Windows users git config --global core.autocr
```

Further Documentation

For detailed Git commands and configurations, refer to:
Git Documentation → git-scm.com/docs

Creating Snapshots

Summary of Common Commands

Command	**Description**
git init	Initialize a Git repository
git add <file>	Stage changes
git commit -m "message"	Commit changes
git status	Show working and staging area status
git diff	Show unstaged changes
git diff --staged	Show staged changes
git log --oneline	Show commit history in a compact format
git rm <file>	Remove a file from the repository
git restore --staged <file>	Unstage a file
git clean -fd	Remove untracked files

Initializing

Initializing a Git Repository

- git init initializes a new Git repository.
- Creates a .git/ directory to store Git's internal data.
- Use ls -a to verify the .git/ directory.

Understanding .git/

Understanding .git/ Directory

- Contains folders like branches/, HEAD, config/, objects/, etc.
- Do not modify these files directly.

- Basic Git Workflow

  Basic Git Workflow

  Staging and Committing Changes

  - git add file1 file2 stages specific files.
  - git add . stages all changes.
  - git commit -m "Meaningful commit message" commits changes.
  - Running git commit without -m opens the default text editor.
  - Each commit has a unique SHA-256 hash, timestamp, and author details.

  Best Practices

  - Commit often, but keep changes self-contained and logical.
  - Avoid very small or very large commits.
  - Use simple past or present tense for commit messages.

  Checking Repository Status

  - git status -s shows a short status of modified files.
  - Left column = staging area, right column = working directory.

- Managing Staging Area

  Managing Staging Area

  - git ls-files lists files in the staging area.
  - git rm file removes a file from both the project and staging area.
  - git rm --cached file removes a file from staging but keeps it in the working directory.

- Viewing changes and commits

  Viewing Changes

  - git diff shows working directory vs. staging area changes.
  - git diff --staged shows staged changes vs. last commit.
  - Visual tools like VS Code, WinMerge, and KDiff3 can be configured for diffs.

  Viewing Commit History

  - git log shows commit history.
  - Useful options: --oneline, --graph, --reverse.

  Viewing a Specific Commit

  - git show <commit_hash> views details of a commit.
  - git show HEAD views the last commit.
  - git show HEAD~1 views the commit before the last one.

- Undoing Changes

  Undoing Changes

  Unstaging Files

  - git restore --staged file_name unstages a file.

  Discarding Local Changes

  - git restore file_name restores a file from the staging area.
  - git restore --source=HEAD~1 file restores to the previous commit version.

  Cleaning Untracked Files

  - git clean -fd removes all untracked files.

Browsing History

Command	**Description**
git log --oneline	View commit history concisely
git log --grep="term"	Search commits by message
git log -S"code"	Find commits that added/removed specific code
git show HEAD~2	View details of an earlier commit
git diff HEAD~2 HEAD	Compare two commits
git checkout <commit>	Checkout a commit (detached HEAD)
git bisect start	Start debugging commits using binary search
git shortlog -nse	Show contributors ranked by commit count
git blame <file>	Show who last modified each line
git tag v1.0	Tag the latest commit as v1.0

- Searching for Commits: git log

  Searching for Commits

  By Author, Date, Email, or Message

  - git log --oneline --stat --graph --reverse → Displays structured commit logs.
  - git log --oneline -3 → Shows the last 3 commits.
  - git log --author="Name" → Filters commits by author.
  - git log --before="YYYY-MM-DD" --after="YYYY-MM-DD" → Filters commits by date range.
  - git log --grep="search term" → Searches commit messages (case-sensitive).
  - git log -S"function_name()" → Finds commits that added or removed specific code.

  Searching in a Range

  - git log --oneline <commit1>..<commit2> → View commits between two commits.
  - git log --oneline <filename> → View commit history of a specific file.
  - git log --patch → Shows exact changes in commits.

- Creating customized commands: Aliasing

  Creating Git Aliases

  Define shortcuts for common commands:

  ```
  shCopyEditgit config --global alias.lg "log --oneline --graph" git config --global alias.unstage "re
  ```

  Usage:

  - git lg → Runs the custom log command.
  - git unstage → Restores all staged files.

- Viewing and Comparing Commits

  Viewing Commits

  - git show <commit> → Displays commit details.
  - git show HEAD~3 → Views the 3rd commit before HEAD.
  - git show HEAD~3:<file_path> → Displays a file's version from a past commit.
  - git show --name-only → Lists modified files in the commit.
  - git show --name-status → Shows modified files with their status.

  Comparing Commits

  - git difftool HEAD~4 HEAD~1 → Compares two commits.
  - git difftool HEAD~4 HEAD~1:<file_path> → Compares specific files.
  - --name-only → Shows only modified file names.
  - --name-status → Shows file names with modification types.

- Head and detached Head

  Understanding HEAD & Detached HEAD

  - git checkout <commit> → Enters **detached HEAD state** (temporary checkout).
  - Detached HEAD commits are not part of a branch and may be garbage collected.
  - git log will only show commits leading up to HEAD, not the full history.

- Binary Search for Bugs in Commits

  Finding Bugs with Git Bisect (Binary Search for Bugs)

  ```
  shCopyEditgit bisect start git bisect bad # Marks current commit as faulty git bisect good <commit_i
  ```

  - Git checks out a middle commit. If it's good, run git bisect good. If bad, run git bisect bad.
  - This process continues until Git identifies the faulty commit.
  - git bisect reset → Exits bisect mode and returns to the latest branch.

- Finding Active Contributors

  Finding the Most Active Contributors

  - git shortlog -nse --before="YYYY-MM-DD" --after="YYYY-MM-DD" → Shows contributors ranked by commits.

- Viewing FIle HIstory

  Viewing File History

  - git log --oneline --stat <file> → Shows commit history for a file.
  - git log --patch <file> → Shows exact changes made to the file.

- Restoring a deleted file

  Restoring a Deleted File

  If a file was deleted in a past commit:

  ```
  shCopyEditgit checkout HEAD~1 -- <file_name> git commit -m "Restored <file_name>"
  ```

- Blaming: Who changed a file

  Finding Who Changed a Line in a File

  - git blame -L 20,50 <file> → Shows who modified lines **20 to 50**.

- Tagging Commits

  Tagging Commits

  - git tag v1.0 <commit> → Tags a specific commit.
  - git tag → Lists all tags.
  - git show <tag> → Views details of a tag.
  - git checkout <tag> → Checks out a tag.
  - git tag -d <tag> → Deletes a tag.

Branching and Merging

CommandDescriptiongit switch -c <branch> | Create and switch to a branch
git branch -d <branch> | Delete a branch
git log master..<branch> | Show commits in a branch but not in master
git diff master..<branch> | Show file changes between branches
git stash push -m "msg" | Temporarily save changes
git merge <branch> | Merge a branch into the current branch
git merge --squash <branch> | Squash merge a branch
git rebase master | Rebase a branch onto master
git cherry-pick <commit> | Apply a specific commit to the current branch

- Branching

  Branching Basics

  - **Branching** allows you to develop features separately from the main code.
  - **Master branch (master)** → Stable version of the code.
  - **Feature branch (feature-branch)** → Isolated workspace for new development.
  - When the feature is done, merge it back into master.

  Creating and Managing Branches

  - git switch -c <branch_name> → Create and switch to a new branch.
  - git branch → List all branches.
  - git branch -m old_name new_name → Rename a branch.
  - git branch -d <branch_name> → Delete a branch.

  Comparing Branches

  - git log master..<branch_name> → Show commits in the branch but not in master.
  - git diff master..<branch_name> → Show file changes.

- Stashing

  Stashing Changes

  - Stashing temporarily saves your changes before switching branches.
  - git stash push -m "Message" → Save changes with a label.
  - git stash apply <inndex>
  - git stash list → List all stashes.
  - git stash show <index> → View a stash.
  - git stash drop <index> → Remove a stash.
  - git stash clear → Remove all stashes.

- Merging

  Merging Branches

  Merge Types:

  1. **Fast-forward Merge** → Moves master forward if no new commits exist in master.
  2. **Three-way Merge** → Merges branches that have diverged, creating a merge commit.

  Merging:

  ```
  shCopyEditgit switch master git merge <branch_name>
  ```

  Handling Merge Conflicts:

  - Manually edit the conflicting files.
  - Use graphical merge tools like P4Merge or Kdiff3:

  ```
  shCopyEditgit config --global merge.tool p4merge git mergetool
  ```

- Undoing a Merge

  Undoing a Merge:

  - git merge --abort → Cancel an ongoing merge.
  - git reset --hard HEAD~1 → Undo the merge before it is committed.
  - git revert -m 1 HEAD → Undo a pushed merge commit.

- Squash Merging

  Squash Merging

  - **Squash merging** combines multiple commits into a single commit.

  ```
  shCopyEditgit switch master git merge --squash <branch_name> git commit -m "Merged feature branch" g
  ```

  - This is useful for keeping a **clean, linear history**.

- Rebasing

  Rebasing (Rewriting History)

  - Rebasing moves a branch's starting point to the latest commit in master.
  - It **keeps history linear** and avoids unnecessary merge commits.

  Rebasing a branch onto master:

  ```
  shCopyEditgit switch <branch_name> git rebase master
  ```

  - If conflicts occur, resolve them and run:

  ```
  shCopyEditgit rebase --continue
  ```

  - If needed, cancel rebase:

  ```
  shCopyEditgit rebase --abort
  ```

  ⚠️ **Avoid rebasing after pushing to a shared repository.**

- Cherry-picking

  Cherry-picking

  - Apply a specific commit from another branch onto your current branch.

  ```
  shCopyEditgit cherry-pick <commit_hash>
  ```

- Restoring a file from another branch

  Restoring a File from Another Branch

  - Restore a specific file from another branch:

  ```
  shCopyEditgit restore --source=<branch_name> -- <filename>
  ```

Collaboration

CommandDescriptiongit clone <url> | Clone a repositorygit fetch origin master | Fetch master changes from remotegit pull | Fetch + merge changes from remotegit push origin master | Push master branch to remotegit push | Shortcut for git push origin mastergit push origin <tag> | Push a tag to remotegit push -d origin <branch> | Remove a branch from remotegit remote add upstream <url> | Add upstream remote repositorygit remote rm upstream | Remove upstream remote

- Understanding Remotes

  Understanding Remotes

  - origin → The central remote repository (GitHub, GitLab, etc.).
  - origin/master → The master branch of the remote repository.
  - git remote -v → View configured remotes.
  - git branch -vv → Check how far your local branch is behind the remote.

- Forking and Cloning

  Forking & Cloning

  1. **Fork the repository** (since you don't have push access to the original).
  2. **Clone the forked repository**
  3. `sh`
  4. `CopyEdit`
  5. `git clone <forked_repo_url>`
  6. **Set upstream to the original repository**
  7. `sh`
  8. `CopyEdit`
  9. `git remote add upstream <original_repo_url>`
  10. **Sync with the upstream repository**
  11. `sh`
  12. `CopyEdit`
  13. `git fetch upstream git merge upstream/master`

- Pushing Changes and Pull Requests PRs

  Pushing Changes & Pull Requests (PRs)

  1. **Create a new branch**
  2. `sh`
  3. `CopyEdit`
  4. `git switch -c feature-branch`
  5. **Make changes & commit**
  6. `sh`
  7. `CopyEdit`
  8. `git add . git commit -m "Description of changes"`
  9. **Push to your fork**
  10. `sh`
  11. `CopyEdit`
  12. `git push origin feature-branch`
  13. **Open a PR**
      - Navigate to your fork on GitHub/GitLab.
      - Click **New Pull Request**.
      - Compare feature-branch with upstream/master.
      - Add a description and request a reviewer.
      - Submit the PR.

- Reviewing and Merging a PR

  Reviewing & Merging a PR

  1. **Reviewer checks PR** → They may request changes.
  2. **Make changes & push again** → This updates the PR.
  3. **Approval & merge options**:
      - **Standard merge** → Preserves commit history.
      - **Squash merge** → Combines multiple commits into one.
      - **Rebase merge** → Rewrites history for a linear commit log.
  4. **Post-merge cleanup**
  5. `sh`
  6. `CopyEdit`
  7. `git remote prune origin # Clean up deleted remote branches git branch -d feature-branch # Delet`

- Pulling and Syncing Changes

  Pulling & Syncing Changes

  - git pull → Fetch + merge remote changes.
  - git pull --rebase → Replays local changes on top of remote changes.
  - Store credentials:
  - `sh`
  - `CopyEdit`
  - `git config --global credential.helper cache`

- Sharing Tags

  Sharing Tags

  - Tags are **not** pushed by default:
  - `sh`
  - `CopyEdit`
  - `git push origin <tag_name>`
  - Delete a remote tag:
  - `sh`
  - `CopyEdit`
  - `git push origin --delete <tag_name>`

- Managing Remote Branches

  Managing Remote Branches

  - git branch -r → Show remote tracking branches.
  - git push --set-upstream origin <branch_name> → Push a branch to remote.
  - git push -d origin <branch_name> → Delete a branch from remote.
  - git branch -d <branch_name> → Delete a local branch.

  Handling Remote Tracking Branches

  - If the remote branch exists but not locally:
  - `sh`
  - `CopyEdit`
  - `git switch -c <local_branch_name> origin/<remote_branch>`
  - Remove stale remote tracking branches:
  - `sh`
  - `CopyEdit`
  - `git remote prune origin`

- Issue Tracking and Milestones

  Issue Tracking & Milestones

  - **Issues** → Used for tracking tasks, bug fixes, and feature requests.
  - **Labels** → Categorize issues (e.g., bug, enhancement).
  - **Milestones** → Group issues under a deadline.
  - **Link issues to PRs** → Helps track progress.