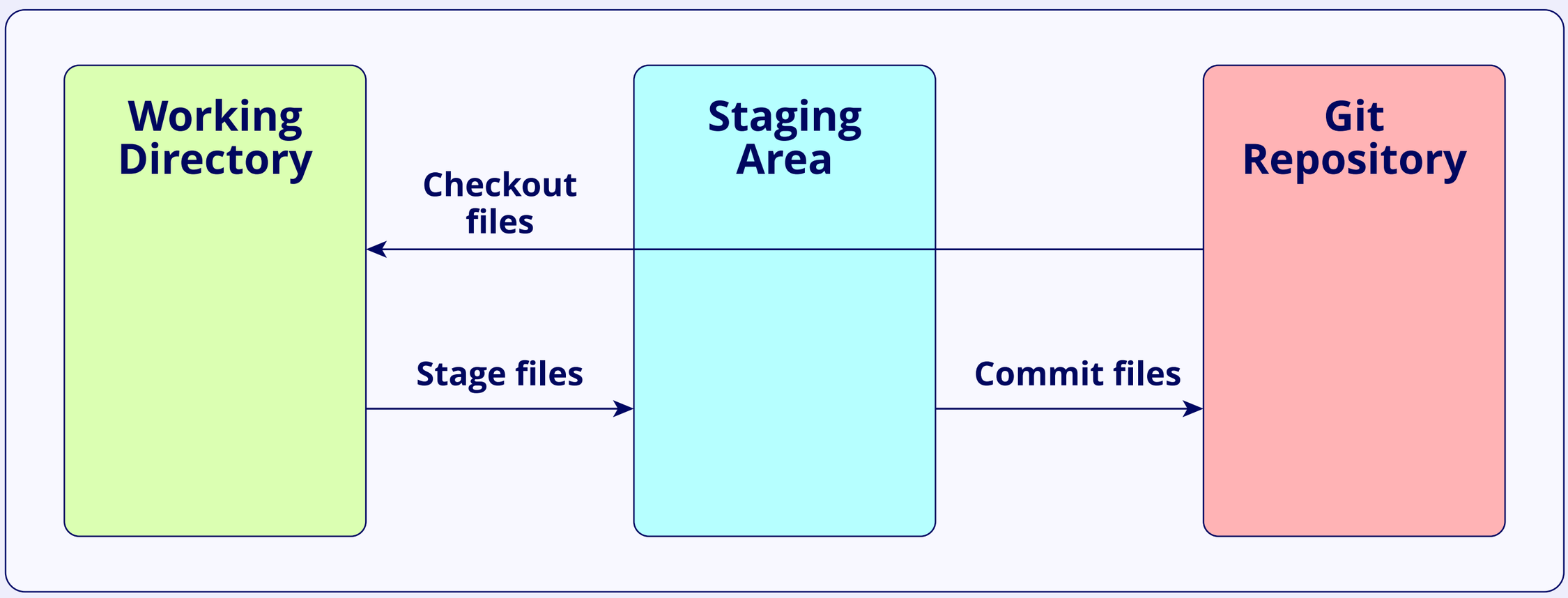
1. Introduction

Git is a tool that allows multiple people to work on the same project simultaneously and helps them track changes in the code.

- **Basic Teminology:**
  - **Repository (Repo):** A folder that contains all the project files and the version history.
  - **Branch:** A separate version of the project. The original version is usually called `main` or `master`.
  - **Commit:** A snapshot of changes made in the repository with a message describing what was changed at the given time.
  - **Staging area:** The staging area is where files (ready to be committed) go before they are saved (committed) to the Git repository. It helps to review the changes made and create meaningful commits.
  - **Check out:** This means switching to a different branch or commit in a project and making it available in the working directory.



  - **Commit hash:** It is a unique ID (a long string of letters and numbers) that Git generates for each commit made to the project.
  - **Reference:** Also known as a ref, it is a label that Git uses to keep track of important points in your project's history.
  - **Tag:** It is a reference that cannot be changed later to point elsewhere. It's commonly used to denote releases, versions, or significant milestones in a project's development.
  - **HEAD:** It is a reference that points to the latest change (commit) in the branch being worked on.
  - **Reference logs:** They track all updates to branches and pointers (like HEAD), including lost commits, merges, resets, and other changes.
  - **Logs:** They only track the history of commits in a branch.
  - **Remote-tracking branches:** They are branch references in the local Git repository that track the currently known state of branches in a remote repository. They keep the local repository in sync with the remote repository.
  - **Untracked files:** These files are not part of the repository and have not been added to the staging area or committed to the repository. They are often new files created by the build process.

- **Key Features:**
  - **Collaboration:** Git makes it easy for teams to work on the same project by allowing different workflows, such as centralized, feature-branch, and forking workflows.
  - **Distributed system:** Each person working on the project has a copy of the project's history. This means team members can work offline, and their data is safe.
  - **Branching and merging:** Branches can be created to test new ideas without altering the main branch. If necessary, these changes can be merged into the main branch.
  - **Commit history:** This records all the changes made to a project over time. It helps track changes, fix mistakes, and return to earlier versions.

2. Common Git Configurations

- **Edit the Configuration File:**

Command	Description
`git config --global --edit`	Opens the global Git configuration file in the default editor for manual editing. With the **`--global`** flag, the settings apply to all repositories for the current user.
`git config --local --edit`	Opens the local Git configuration file for the current repository. With the **`--local`** flag, the settings apply only to that repository and override any global settings.

- **User Information:**

Command	Description
`git config --global user.name “Name”`	Sets the name to be used in commits.
`git config --global user.email “email@example.com”`	Sets the email address to be used in commits.

- **Aliases:**

Command	Description
`git config --global alias.st status`	Sets up **`git st`** as a shorthand (called alias) for the command **`git status.`**
`git config --get-regexp alias`	Lists all created aliases.

- **Git Output Colorization:**

Command	Description
`git config --global color.ui auto`	Configures Git to automatically add a color to the output in the terminal for all commands, making it easier to read.
`git config --global color.branch.current "yellow"`	Sets the color of the current branch name to yellow.

- **Check All the Configurations:**

Command	Description
`git config --list`	Lists all the current configuration settings in Git, including both global and local configurations.

3. Initiating and Cloning Repositories

- **Initialize a New Git Repository:**

Command	Description
`git init`	Sets up a new Git repository in the project folder where the command is executed.

- **Clone an Existing Repository:**

Command	Description
`git clone <repository-URL>`	Copy an existing repository from a remote server to a local machine. The remote repository from which a project is cloned is, by default, named **`origin`**.

4. Core Git Operations

Command	Description
`git status`	Shows the current state of the working directory and staging area, including untracked, modified, and staged files.
`git add <filename>`	Adds a specific file to the staging area.
`git add`	Adds all changes in the current directory and its subdirectories to the staging area.
`git commit -m "commit message"`	Saves the staged changes to the repository with a descriptive message.
`git diff`	Displays the differences between the working directory and the staging area. It can be used with additional arguments to compare commits.

5. Working with Remotes

- **Manage Remote Repositories (`git remote`):**

Command	Description
`git remote`	Lists the names of the remote repositories configured for the local project.
`git remote add <name> <url>`	Adds a new remote repository with the specified name and URL.
`git remote remove <name>`	Removes the specified remote repository.

`git remote rename <old-name> <new-name>`	Renames a remote repository from **`<old-name>`** to **`<new-name>`**.
`git remote set-url <name> <new-url>`	Changes the URL for the specified remote repository.

- **Fetch Updates (`git fetch`):**

Command	Description
`git fetch`	Retrieves updates from the default remote repository without merging them into the local branches.
`git fetch <remote-name>`	Fetch from the specified remote repository.
`git fetch <remote-name> <branch-name>`	Fetch from a specific branch of the specified remote repository.
`git fetch --all`	Fetch from all remote repositories linked to the local repository.

- **Pull Changes (`git pull`):**

Command	Description
`git pull`	Retrieves updates from the default remote repository, usually called **`origin`**, and merges them into the current branch.
`git pull <remote-name>`	Retrieves updates from a specific remote repository and merges them into the current branch.
`git pull <remote-name> <branch-name>`	Retrieves updates from a specific remote repository branch and merges them into the current branch.

- **Push Changes (`git push`):**

Command	Description
`git push`	Sends the commits from the current branch to the corresponding branch on the remote repository.
`git push <remote-name>`	Pushes local changes to a specific remote repository.
`git push <remote-name> <branch-name>`	Pushes local changes to a specific branch of the remote repository.
`git push --all`	Pushes all local branches to the remote repository.
`git push --force`	Forcefully pushes changes to the remote repository, overwriting any remote changes that conflict with any of the local commits.

6. Managing Branches and Merges

- **Create and Delete Branches (`git branch`):**

Command	Description
`git branch <branch-name>`	Creates a new branch locally but stays in the current branch. (The local branch can be pushed to create it remotely.)
`git branch -d <branch-name>`	Deletes the specified branch. The **`-d`** option prevents deletion if the branch has commits that are not merged into another branch.
`git branch -D <branch-name>`	Using **`-D`** forcefully deletes the specified branch, even if it has unmerged changes.
`git push <remote-name> --delete <branch-name>`	Deletes the specified branch from the remote repository.

- **List and Rename Branches:**

Command	Description
`git branch`	Lists all branches in the local repository with the current branch marked with an asterisk (*).
`git branch -r`	Lists all remote-tracking branches.
`git branch -a`	Lists both local and remote branches.
`git branch -m <new-branch-name>`	Renames the current branch.
`git branch -m <old-branch-name> <new-branch-name>`	Renames the specified branch.

- **Switching Branches (`git switch`):**

Command	Description
`git switch <branch-name>`	Moves from the current branch to the specified one.
`git switch -b <branch-name>`	Creates a new branch named **`<branch-name>`** and switches to it.

- **Merge Branches (`git merge`):**

Command	Description
`git merge <branch-name>`	Merge changes from the specified branch into the current branch.

- **Using Rebase (`git rebase`):**

Command	Description
`git rebase <base-branch>`	Takes the commits from the current branch and re-applies them on top of the **`<base-branch>`**, creating a cleaner history of the commits.
`git rebase -i HEAD~<n>`	Opens an interactive editor to specify how to combine (squash), reorder, or drop the last few commits from the specified commit. Change the word “pick” listed against each commit to “squash” or “drop.” Finally, edit the commit message as needed.

- **Using cherry-pick (`git cherry-pick`):**

Command	Description
`git cherry-pick <commit-hash>`	Applies the changes introduced by **`<commit-hash>`** to the current branch.
`git cherry-pick <commit-hash1>.. <commit-hash2>`	Applies all commits from **`<commit-hash1>`** to **`<commit-hash2>`** to the current branch, excluding **`<commit-hash1>.`**

- **Replace Last Commits (`git commit --amend`)**

Command	Description
`git commit --amend`	Replaces the last commit with a new one. It can be an update in the commit message or some new changes.

7. Undoing Changes

- **Revert Commits (`git revert`):**

Command	Description
`git revert <commit-hash>`	Creates a new commit that reverts the changes made by the specified commit. The history of commits is not changed.
`git revert <commit-hash> -m "message"`	Specifies a commit message when reverting.
`git revert <commit-hash> --no-commit`	Reverts the specified commit but doesn’t automatically create a new commit.
`git revert <commit-hash1> <commit-hash2> ...`	Creates a new commit that reverts multiple specified commits in the order they are listed.
`git revert <oldest-commit-hash>.. <newest-commit-hash>`	Reverts a range of commits from **`<oldest-commit-hash>`** to **`<newest-commit-hash>`**, excluding **`<oldest-commit-hash>`** and creating a separate commit for each revert.

Git Cheat Sheet

educative

- **Reset Changes (`git reset`):**

Command	Description
`git reset --soft <commit-hash>`	The state of the current branch is reverted to the specified commit. However, the contents of the staging area and the working directory remain unchanged.
`git reset <commit-hash>`	Resets the current branch to a specific commit and discards the changes in the staging area to match but does not change the working directory.
`git reset --hard <commit-hash>`	Resets the current branch to a specific commit and discards all changes in the working directory and staging area.

- **Using (`git rm`):**

Command	Description
`git rm <file-name>`	Deletes the specified file from the working directory and stages the deletion so that it will be committed.
`git rm --cached <file-name>`	Removes the file from the staging area but keeps the file in the working directory
`git rm <file1> <file2> ...`	Removes multiple files in one command.
`git rm -r <directory-name>`	Deletes all files and subdirectories within the specified directory.

- **Clean Untracked Files (`git clean`)**

Configuration: To make **`git clean`** work without any flags, set **`requireForce`** to **`false`** in the **`.gitconfig`** file. Otherwise, use the **`-f`** or **`-i`** flag with the respective command.

Command	Description
`git clean -n`	Lists all the untracked files and directories that would be deleted if git clean commands are used.
`git clean -f`	Deletes untracked files from the working directory.
`git clean -fd`	Deletes both untracked files and directories.
`git clean -fx`	Deletes all untracked files, including those listed in the file **`.gitignore`**.
`git clean -fX`	Deletes only the files listed in the file **`.gitignore`** and are untracked.
`git clean -i`	This enters an interactive mode for selecting which files or directories to delete.

8. Saving Changes Temporarily

- **Stash Changes (`git stash`):**

Command	Description
`git stash`	Saves the changes in the working directory and staging area in a temporary store (a stash) and reverts the working directory to the latest commit.
`git stash list`	Displays a list of all stashes created, with names like **`stash@{0}`**, **`stash@{1}`**, etc., for those created in that order.
`git stash apply`	Restores the changes saved in the latest stash **`(stash@{0})`** to the working directory.
`git stash apply stash@{n}`	Restores the changes saved in the specific stash entry identified by index **`n`** to the working directory.
`git stash pop`	Restores the most recent stash and removes it from the list of stashes.
`git stash pop stash@{n}`	Restores the stash with a specific index, **`n`**, and removes it from the list of stashes
`git stash drop`	Deletes the most recent stash entry.
`git stash drop stash@{n}`	Deletes a specific saved stash.
`git stash clear`	Deletes all the saved stashes.

9. Marking Releases and Milestones

- **Using (`git tag`):**

Command	Description
`git tag`	Displays a list of all the existing tags in the repository.
`git tag <tag-name>`	Creates a new tag for the latest commit.
`git tag -a <tag-name> -m "message"`	Creates a tag with some additional message. Annotated tags include metadata like the tagger's name, email, and date.
`git tag <tag-name> <commit-hash>`	Creates a new tag named **`<tag-name>`** that points to the commit with the hash **`<commit-hash>`**.
`git tag -a <tag-name> <commit-id> -m "Tag message"`	Tags a specific commit with some additional message.

`git tag -d <tag-name>`	Deletes the specified tag from the local Git repository.
`git push <remote-name> --delete <tag-name>`	Deletes the tag **`<tag-name>`** from the remote repository **`<remote-name>`**.
`git push <remote-name> <tag-name>`	Pushes a specific tag **`<tag-name>`** to the remote repository **`<remote-name>`**.
`git push --tags`	Pushes all tags to the remote repository that is associated with the local repository.

10. Exploring Repository History

- **Working with the log (`git log`):**

Command	Description
`git log`	Displays the commit history, including a list of commits in the current branch with details like the commit hash, author, date, and commit message.
`git log --oneline`	Displays a simplified view of the commit history of the current branch with only the commit hash and message, one line per commit.
`git log -n <number>`	Displays the specified number of most recent commits of the current branch.
`git log <file-path>`	Displays the commit history for the specified file in the current branch.
`git log --since= "YYYY-MM-DD" --until="YYYY-MM-DD"`	Displays only the commits in the specified date range for the current branch.
`git log --author= "Author Name"`	Filters and displays the commits by a specific author.
`git log -p`	This includes the differences (called a patch) introduced by each commit in the commit history.
`git log --stat`	Provides a summary of changes in each commit, including file changes and line counts.
`git log --pretty= format:"%h %s`	Customizes the output format of the commit history.
`git log --graph`	Displays the commit history as a graph to visualize branch and merge history.

Git Cheat Sheet

educative

- **View Reference Logs (`git reflog`):**

Command	Description
`git reflog`	Displays the reference logs (reflogs) for the current branch. Each log includes an index, the commit hash, and a message describing the change.
`git reflog <reference>`	Shows the reflog for a specific reference (e.g., branch or HEAD).
`git reflog -n <number>`	Displays the specified number of most recent reflog entries.
`git reflog <branch-name>`	Displays the reflog entries for the specified branch.
`git reflog show ref@{n}`	Displays a specific entry from the reflog. Replace **`ref`** with the reference (e.g., HEAD) and **`n`** with the entry number.

- **Using (`git blame`):**

Command	Description
`git blame <file>`	Shows who last modified each line of the specified file.
`git blame <file> <commit>`	Shows the blame information for a file as of a specific commit.
`git blame -L <start>,<end> <file>`	Shows the blame information for a specific range of lines in a file.
`git blame -e <file>`	Displays the email addresses of the authors in the blame information.
`git blame -p <file>`	Shows more comprehensive blame information like full commit messages.
`git blame -w <file>`	Shows the blame information, ignoring the changes solely applied to whitespaces.

- **Using (`git bisect`):**

Command	Description
`git bisect start`	Initiates a binary search to pinpoint the commit that introduced a change or a bug. Next, the search range must be specified by marking two commits as good and bad.
`git bisect bad`	Marks the current commit as bad, implying that it contains the bug.
`git bisect bad <commit-hash>`	Marks the specified commit as bad.
`git bisect good <commit-hash>`	Marks the specified commit as good, implying the absence of the bug.

`git bisect status`	Displays the current status of the bisect session.
`git bisect reset`	Ends the bisect session and returns to the original branch.

11. Git Best Practices

- **Commit Messages and Best Practices:**
  - Write the commit message with clear, imperative subject lines and optional body details.
  - For the subject line:
    - Keep it short—ideally within 50 characters.
    - Write in the imperative mood (e.g., “Add feature” instead of “Added feature”).
    - Capitalize the first letter of the subject.
    - Avoid ending with a period
  - For the body:
    - Separate the subject line from the body with a blank line.
    - Use the body for more detailed explanations when necessary.
    - Wrap text at 72 characters per line for better readability.
    - Briefly explain the reasoning behind the changes, not just what was changed.
    - Add reference for any relevant issue numbers or tickets.
- **Workflow Recommendations:**
    - Use branches for feature development, following a strategy like Git Flow or GitHub Flow.
    - Commit small, focused changes regularly.
    - Frequently pull changes from the remote repository to stay updated.
    - Use pull requests for code review and collaboration; ensure they are approved before merging with the remote branch.
    - Combine related commits before merging for a clean history.
    - Tag important milestones or versions.
- **Git Flow**
    - **Organized branching and Git flow process:**
      - **Main branch:** It always holds the stable, production-ready code version deployed or released.
      - **Develop branch:** This is an integration branch where new features and fixes are combined before release. It’s a staging area for the next release.
      - **Feature branches:** These allow developers to work on new features or fixes independently. Start each new feature or task by creating a feature branch from **`develop`**. When the feature is complete, merge it back into **`develop`.**
      - **Release branches:** These are used to prepare for a new release. A release branch is created when the **`develop`** branch is stable and ready for release. Once the release is finalized (adjustments and bug fixes are made), the release branch is merged into both **`main`** and **`develop`** for deployment.
      - **Hotfix branches:** These are used for urgent fixes that must go directly to production. They are created from **`main`** and merged back into both **`main`** and **`develop`**.
    - **Handling Merge Conflicts:**
      - Once we have identified the file(s) with conflicts, use a text editor or merge tool to resolve conflicts in the files. Look for conflict markers (**`<<<<<<<, =======, >>>>>>>`**) in the affected files.
      - Add the file to the staging area: **`git add <file>`**
      - Commit the file to the remote: **`git commit -m "Resolve merge conflicts"`**