# GIT/GITHUB Training

Duration : 18 hours

Contents:

**Day 1: Git Commands, Concepts & TFS vs Git (3 Hours)**

**1️. Git Basics - Review & Comparison with TFS**

* **TFS Git vs. Git vs. GitHub**
* Quick review of **basic Git commands** and workflows
* **Key differences between Git and TFS** (Centralized vs. Distributed)
* Mapping **TFS commands** to **Git equivalents**
* Hands-on: **Setting up a Git repository from TFS**

**2️. Deep Dive into Branching & Merging**

* Understanding Advanced branching strategies (Git Flow, Trunk-Based Development)
* git merge vs. git rebase (pros, cons & when to use each)
* Handling conflicts effectively in Git

**3️. Rewriting History**

* git commit --amend
* git rebase -i (Interactive Rebase)
* git reflog and recovering lost commits

**4️. Overview of Git Stash & Cleaning Untracked Files**

* What is **stashing**?
* Using git stash, git stash pop, git stash list
* git stash options (--include-untracked, --patch)
* git stash branch: Creating a branch from stash
* git clean for removing untracked files

**Day 2: Advance Git Commands, Concepts (3 Hours)**

**5️. Overview of Cherry Picking, Rebasing & Tags**

* git cherry-pick: Selective commit integration
* Resolving conflicts during **cherry-picking**
* git rebase for a cleaner commit history
* **Overview of Git Tags**: Lightweight vs. Annotated
* Creating, listing, and deleting tags (git tag, git push --tags) (Extra time required)
* Using tags in release management (Extra time required)

**6️. Overview of Git Aliases**

* Creating **custom Git shortcuts**
* Managing & sharing aliases (git config --global alias)

**7. Markdown (.md) Summary (Extra time required)**

* Writing project documentation with **Markdown (.md)**
* Best practices for .md files (Readme, Contributing, License, etc.)

**8. Hands-on Practice**

* Exercises on **stashing, cherry-picking, tags and aliases**

**9. Overview of Git Submodules & Patches**

* Introduction to **submodules**
* Adding & updating submodules (git submodule add, git submodule update)
* Pitfalls & best practices (Extra time required)
* Creating & applying patches (git format-patch, git apply, git am)

**Day 3: Advanced Git, Workflows (3 Hours)**

**10. Overview of Git Hooks**

* **Introduction to Git hooks**: Client-side vs. Server-side
* Setting up & customizing hooks (pre-commit, pre-push, post-merge)
* Examples: Linting before commit, checking commit messages, preventing secrets from being committed
* Using Husky for managing hooks in a team (Extra time required)

**11. Advanced Collaboration Techniques**

* Working with **forks** & upstream repositories
* Managing **multiple remotes (Extra time required)**
* Best **pull request strategies** for large teams (Extra time required)

**12. Performance Optimization (Extra time required)**

* **Shallow clones** and **sparse-checkouts**
* Large repository management strategies

**Day 4: GitHub Deep Dive (3 Hours)**

**13. GitHub: Deep Dive & Workflow with .yml file**

* Understanding **GitHub repositories & settings**
* Using **GitHub Projects** for issue tracking
* Understanding **.yml file structure** for GitHub Actions
* Hands-on: Creating an **automated workflow with .yml file**

**1️3️. Hands-on Practice: GitHub Actions & Hooks**

* Setting up **GitHub Actions** for CI/CD
* Implementing **Git Hooks** in projects

**1️4. Debugging & Troubleshooting**

* **Using git bisect** for bug identification
* **Analyzing history** with git blame & git log
* **Recovering from merge & rebase conflicts**

**Day 5: Git with Jira and API Integration (3 Hours)**

**1️5. Git with Jira & Automation (Extra time required)**

* **Issue tracking integration**
* Linking **commits & pull requests** to Jira issues
* **Webhook-based workflow automation**

**1️6. GitHub API & Integrations (Extra time required)**

* **Using the GitHub API for automation**
* **Integrating GitHub with third-party tools**

**Day 6: Integrations, Security & Troubleshooting (3 Hours)**

**1️7. Security & Compliance**

* **GPG signing** for commits (Extra time required)
* **Managing API Tokens (Extra time required)**
* Handling **sensitive data** (.gitignore, GitHub secrets)
* **Audit & compliance practices (Extra time required)**

**1️8. Performance Optimization & Customization (Extra time required)**

* **Overview of Shallow clones & sparse-checkouts** for large repositories
* **Customizing Git with aliases & hooks**
* **Optimizing Git workflows for team productivity**

**1️9️. Best Practices & Final Q&A**

* **Commit message conventions**
* **Code review best practices**
* **Scaling Git for large teams & projects**
* **Real-world case studies (Extra time required)**
* Open Q&A session

| **Software** | **Purpose** | **Download Link** |
| --- | --- | --- |
| **Git** | Version control system | <https://git-scm.com> |
| **GitHub Account** | Code repository and collaboration | <https://github.com> |
| **GitHub Desktop (Optional)** | GUI for GitHub | <https://desktop.github.com> |
| **VS Code** | Code editor with Git integration | <https://code.visualstudio.com> |
| **Husky (Optional)** | Git hook management | npm install husky --save-dev |
| **Node.js (Optional)** | Required for Husky and GitHub Actions | <https://nodejs.org> |
| **Jira (Optional)** | Issue tracking and project management | https://www.atlassian.com/software/jira |
| **Python (Optional)** | GitHub API automation | <https://www.python.org> |
| **Postman (Optional)** | API testing for GitHub integration | <https://www.postman.com> |
| **Markdown Viewer** | Preview .md files | Built into GitHub/VS Code |
| **GPG (Optional)** | Commit signing and encryption | <https://gnupg.org> |

**. Git Basics - Review & Comparison with TFS**

* **TFS Git vs. Git vs. GitHub**
* Quick review of **basic Git commands** and workflows
* **Key differences between Git and TFS** (Centralized vs. Distributed)
* Mapping **TFS commands** to **Git equivalents**
* Hands-on: **Setting up a Git repository from TFS**

**1. TFS Git vs. Git vs. GitHub**

* **TFS Git**: A Git-based repository hosted in **Azure DevOps (formerly TFS)** with additional Microsoft integrations like work item tracking.
* **Git**: A **distributed version control system** that allows decentralized collaboration and local repositories.
* **GitHub**: A **cloud-based hosting service** for Git repositories, adding collaboration features like issues, pull requests, and CI/CD.

**2. Quick Review of Basic Git Commands and Workflows**

* **Initializing a repository**

git init

* **Cloning an existing repository**

git clone <repo\_url>

* **Staging and committing changes**

git add .

git commit -m "Commit message"

* **Branching and merging**

git branch new-feature

git checkout new-feature # or 'git switch new-feature'

git merge new-feature

* **Pushing and pulling changes**

git push origin main

git pull origin main

**3. Key Differences Between Git and TFS (Centralized vs. Distributed)**

| **Feature** | **TFS (Centralized)** | **Git (Distributed)** |
| --- | --- | --- |
| **Architecture** | Centralized (Server-based) | Distributed (Each user has a full repo copy) |
| **Connectivity** | Requires server connection | Works offline, syncs later |
| **Branching** | Heavyweight & expensive | Lightweight & fast |
| **Merging** | More complex & limited | Easier, supports rebasing |
| **Collaboration** | Requires TFS Server | Uses remote repositories (GitHub, GitLab, etc.) |
| **Speed** | Slower due to centralization | Faster due to local operations |

**4. Mapping TFS Commands to Git Equivalents**

| **Action** | **TFS Command** | **Git Equivalent** |
| --- | --- | --- |
| Create Repository | TFS Version Control (VSTS) | git init |
| Check Out | tf get latest | git pull |
| Check In / Commit | tf checkin | git commit -m "message" |
| Branching | tf branch | git branch <branch-name> |
| Merge Changes | tf merge | git merge <branch> |
| Undo Changes | tf undo | git checkout -- <file> |
| View History | tf history | git log |

**5. Hands-on: Setting Up a Git Repository from TFS**

**Step 1: Clone the TFS Repository**

git clone <TFS\_repo\_URL>

**Step 2: Add a Remote for GitHub or another Git Server**

git remote add origin <Git\_repo\_URL>

**Step 3: Fetch the Latest Changes**

git fetch --all

**Step 4: Convert and Push to Git**

git checkout -b main

git push -u origin main

**Deep Dive into Branching & Merging**

* Understanding Advanced branching strategies (Git Flow, Trunk-Based Development)
* git merge vs. git rebase (pros, cons & when to use each)
* Handling conflicts effectively in Git

**1. Understanding Advanced Branching Strategies**

There are multiple strategies used in different workflows. The two most popular are **Git Flow** and **Trunk-Based Development**.

**🔹 Git Flow (Feature Branch Workflow)**

* **Ideal for**: Large teams, structured release cycles.
* **Branches Used**:
  + main → Stable, production-ready branch.
  + develop → Ongoing development.
  + feature/\* → Individual feature branches.
  + release/\* → For preparing releases.
  + hotfix/\* → Quick fixes for production.
* **Pros**:
  + Clear branch separation.
  + Safer for large teams.
* **Cons**:
  + Can be complex with too many branches.
  + Frequent merges may create conflicts.

**Git Flow Commands Example**

# Initialize Git Flow

git flow init

# Create a feature branch

git flow feature start new-feature

# Complete and merge feature

git flow feature finish new-feature

**🔹 Trunk-Based Development**

* **Ideal for**: CI/CD, high-speed teams, DevOps.
* **Branches Used**:
  + main (or trunk) → All changes merged directly.
  + Short-lived feature branches.
* **Pros**:
  + Faster delivery.
  + Fewer merge conflicts.
* **Cons**:
  + Risk of breaking main.
  + Requires strong CI/CD automation.

**Trunk-Based Workflow Example**

# Create a short-lived feature branch

git checkout -b feature-xyz

# Merge quickly after testing

git checkout main

git merge feature-xyz --no-ff

**2. git merge vs. git rebase**

Both are used to integrate changes, but they work differently.

| **Feature** | **git merge** | **git rebase** |
| --- | --- | --- |
| **Workflow** | Creates a new merge commit | Rewrites history |
| **Commit History** | Preserves full branch history | Creates a linear history |
| **Use Case** | Collaborative projects, multiple contributors | Cleaning up local branches before pushing |
| **Pros** | Safe, retains original commits | Cleaner history, avoids unnecessary merges |
| **Cons** | Can result in merge commits cluttering history | Risky if used incorrectly (rewrites history) |

**🔹 git merge Example**

git checkout main

git merge feature-branch

* Creates a merge commit.
* Maintains the original commit history.

**🔹 git rebase Example**

git checkout feature-branch

git rebase main

* Moves the feature branch commits on top of main.
* Creates a **linear** commit history.

**When to Use What?**

| **Scenario** | **Use git merge** | **Use git rebase** |
| --- | --- | --- |
| Merging feature branch into main | ✅ | ❌ |
| Keeping a local feature branch up-to-date | ❌ | ✅ |
| Preserving commit history for collaboration | ✅ | ❌ |

**3. Handling Conflicts Effectively in Git**

**🔹 Identifying a Merge Conflict**

If there’s a conflict:

Auto-merging file.txt

CONFLICT (content): Merge conflict in file.txt

Use:

git status

to see the conflicted files.

**🔹 Resolving a Conflict**

1. Open the conflicted file:

<<<<<<< HEAD

Your current branch changes.

=======

Incoming changes from merged branch.

>>>>>>> feature-branch

1. Manually edit the file to keep the correct version.
2. Add the resolved file:

git add file.txt

1. Complete the merge:

git commit -m "Resolved conflict in file.txt"

**🔹 Aborting a Merge**

If you want to cancel a merge:

git merge --abort

**🔹 Aborting a Rebase**

If you encounter issues during a rebase:

git rebase --abort

**Final Tips**

* **Use git log --graph --oneline --all** to visualize branch merges.
* **Use git stash** before rebasing to save uncommitted changes.
* **Avoid rebasing shared branches**—it rewrites history.

**Rewriting History**

* git commit --amend
* git rebase -i (Interactive Rebase)
* git reflog and recovering lost commits

**Rewriting History in Git**

Rewriting history in Git is a powerful technique to modify commits after they have been made. However, rewriting history in shared branches can be **dangerous** since it alters commit hashes. Below are key methods for rewriting history.

**1. git commit --amend (Modifying the Last Commit)**

This command allows you to **edit the last commit**, either to:

* Modify the commit message.
* Add more changes to the last commit.
* Change the author of a commit.

**🔹 Example: Changing the Last Commit Message**

git commit --amend -m "Updated commit message"

**Before:**

Commit #123abc - "Initial commit"

**After:**

Commit #123abc - "Updated commit message"

**🔹 Example: Adding Changes to the Last Commit**

git add updated-file.txt

git commit --amend --no-edit

✅ This keeps the existing commit message but includes the new changes.

**🔹 Example: Changing the Author of the Last Commit**

git commit --amend --author="New Name <newemail@example.com>"

**2. git rebase -i (Interactive Rebase)**

Interactive rebase allows you to **rewrite multiple commits**, such as:

* Squashing commits together.
* Reordering commits.
* Editing commit messages.

**🔹 Running Interactive Rebase**

git rebase -i HEAD~3

This opens an editor with the last 3 commits:

pick 123abc First commit

pick 456def Second commit

pick 789ghi Third commit

**Interactive Rebase Options**

| **Command** | **Description** |
| --- | --- |
| pick | Keep the commit as is. |
| reword | Modify the commit message. |
| edit | Modify commit content. |
| squash (s) | Merge this commit into the previous one. |
| fixup (f) | Like squash, but discards this commit's message. |
| drop | Delete the commit. |

**🔹 Example: Squashing Commits**

Change:

pick 123abc First commit

squash 456def Second commit

squash 789ghi Third commit

* After saving, Git will prompt you to enter a new commit message.
* The three commits are merged into **one commit**.

**3. git reflog (Recovering Lost Commits)**

If you accidentally delete or rewrite a commit, **Git reflog** allows you to **recover it**.

**🔹 Viewing Reflog**

git reflog

Example output:

abc123 HEAD@{0}: commit: Added a new feature

def456 HEAD@{1}: rebase (finish): returning to main

ghi789 HEAD@{2}: reset: moving to previous commit

* Each line shows a previous state of the repository.

**🔹 Recovering a Lost Commit**

If you accidentally delete a commit:

git reset --hard abc123 # Reset to commit abc123

**🔹 Undoing a Hard Reset**

If you ran:

git reset --hard HEAD~2

to remove the last 2 commits, **you can recover them**:

git reflog

git reset --hard <commit\_hash>

**Final Tips**

* **Be cautious when rewriting history in shared branches!**
* **Use git commit --amend for quick fixes to the last commit.**
* **Use git rebase -i for modifying multiple commits.**
* **Use git reflog if you need to recover lost commits.**

**Overview of Git Stash & Cleaning Untracked Files**

* What is **stashing**?
* Using git stash, git stash pop, git stash list
* git stash options (--include-untracked, --patch)
* git stash branch: Creating a branch from stash
* git clean for removing untracked files

**Overview of Git Stash & Cleaning Untracked Files**

Sometimes, you might be working on a feature but need to switch branches or pull updates. Git **stash** allows you to save your uncommitted changes temporarily and return to them later without committing.

**1. What is Git Stashing?**

* Stashing saves uncommitted changes in a temporary stack.
* Useful when you need a clean working directory but don’t want to commit yet.
* Allows you to **switch branches** or **pull changes** without losing your work.

**2. Using git stash Commands**

**🔹 Stashing Changes**

git stash

* Moves uncommitted changes (both tracked and staged files) into a **stash stack**.
* Leaves you with a clean working directory.

**🔹 Viewing Stashes**

git stash list

Example output:

stash@{0}: WIP on main: 123abc Modified README

stash@{1}: WIP on feature-branch: 456def Updated script.js

* Each stash is identified by **index** (stash@{0} is the latest).

**🔹 Applying a Stash**

git stash pop

* Restores the most recent stash and **removes it from the list**.
* If you want to keep the stash in the list, use:

git stash apply stash@{0}

**🔹 Deleting a Stash**

git stash drop stash@{0}

* Removes a specific stash.
* To **clear all stashes**:

git stash clear

**3. git stash Options**

**🔹 Stashing Untracked Files**

By default, git stash ignores untracked files. To include them:

git stash --include-untracked

or

git stash -u

**🔹 Stashing with a Message**

git stash push -m "WIP: Refactoring API calls"

* Helps keep track of why you stashed changes.

**🔹 Stashing Part of a File (--patch)**

If you want to stash only some changes:

git stash push --patch

* Allows you to **interactively select** which changes to stash.

**4. Creating a Branch from Stash (git stash branch)**

You can create a new branch with the stashed changes.

git stash branch new-feature stash@{0}

* This **creates a branch** and **applies the stashed changes** to it.

**5. Cleaning Untracked Files (git clean)**

Sometimes, you want to remove **untracked** files (files not in version control).

**🔹 Checking What Will Be Deleted (--dry-run)**

git clean -n

* Shows which files would be deleted, **without deleting them**.

**🔹 Deleting Untracked Files**

git clean -f

* Deletes **untracked** files (but not directories).

**🔹 Deleting Untracked Files & Directories**

git clean -fd

* -f → Force deletion.
* -d → Removes **untracked directories** too.

**🔹 Deleting Only Ignored Files**

git clean -X

* Removes **only ignored files** (e.g., files in .gitignore).

**Final Thoughts**

✅ Use git stash to **save and restore changes temporarily**.  
✅ Use git stash list to **view saved stashes**.  
✅ Use git stash branch to **resume work from a stash in a new branch**.  
✅ Use git clean -n before deleting to **avoid accidental data loss**.

**Overview of Cherry Picking, Rebasing & Tags**

* git cherry-pick: Selective commit integration
* Resolving conflicts during **cherry-picking**
* git rebase for a cleaner commit history
* **Overview of Git Tags**: Lightweight vs. Annotated
* Creating, listing, and deleting tags (git tag, git push --tags)
* Using tags in release management

**Overview of Cherry Picking, Rebasing & Tags**

Git provides several techniques to manage commits efficiently. **Cherry-picking**, **rebasing**, and **tags** help maintain a cleaner commit history and facilitate structured releases.

**1. git cherry-pick: Selective Commit Integration**

**🔹 What is Cherry-Picking?**

* Allows you to apply a specific commit from one branch to another.
* Useful when you want to **pick a single feature or fix** without merging the entire branch.

**🔹 Cherry-Picking a Commit**

git cherry-pick <commit-hash>

Example:

git cherry-pick 1a2b3c4d

This applies commit 1a2b3c4d to your current branch.

**🔹 Cherry-Picking Multiple Commits**

git cherry-pick <commit1> <commit2>

Example:

git cherry-pick 1a2b3c4d 5d6e7f8g

* Picks two commits and applies them in order.

**🔹 Resolving Conflicts During Cherry-Picking**

If conflicts occur:

1. **Resolve conflicts manually**
2. **Mark the conflict as resolved**

git add <conflicted-file>

1. **Continue cherry-picking**

git cherry-pick --continue

1. **Abort cherry-picking (if needed)**

git cherry-pick --abort

**2. git rebase: Cleaner Commit History**

**🔹 What is Rebasing?**

* Moves commits to a new base, ensuring a **linear history**.
* Removes unnecessary merge commits from feature branches.

**🔹 Basic Rebasing**

git rebase main

Moves your branch commits **on top of** the main branch.

**🔹 Interactive Rebasing (-i)**

Allows you to **edit, reorder, squash, or delete commits**.

git rebase -i HEAD~3

Options during interactive rebase:

| **Command** | **Description** |
| --- | --- |
| pick | Keep the commit as is. |
| reword | Modify the commit message. |
| edit | Edit the commit content. |
| squash | Merge this commit into the previous one. |
| drop | Delete the commit. |

**🔹 Resolving Conflicts During Rebase**

If conflicts occur:

1. **Resolve conflicts manually**
2. **Mark the conflict as resolved**

git add <conflicted-file>

1. **Continue rebasing**

git rebase --continue

1. **Abort rebasing (if needed)**

git rebase --abort

**3. Git Tags: Lightweight vs. Annotated**

**🔹 What Are Git Tags?**

* Tags are **pointers to a specific commit**, commonly used for releases.
* Two types of tags:
  + **Lightweight Tags** → Simple commit reference.
  + **Annotated Tags** → Stores additional metadata (author, date, message).

**🔹 Creating Tags**

**Creating a Lightweight Tag**

git tag v1.0.0

**Creating an Annotated Tag**

git tag -a v1.0.0 -m "Release version 1.0.0"

**🔹 Listing Tags**

git tag

or

git tag --list "v1.\*"

Lists all tags matching v1.\*.

**🔹 Deleting Tags**

**Delete a Local Tag**

git tag -d v1.0.0

**Delete a Remote Tag**

git push origin --delete v1.0.0

**🔹 Pushing Tags to Remote**

git push origin v1.0.0

or

git push --tags

Pushes **all local tags** to the remote repository.

**4. Using Tags in Release Management**

**🔹 Checking Out a Tagged Version**

git checkout v1.0.0

* Switches to a specific release.

**🔹 Creating a Branch from a Tag**

git checkout -b hotfix-v1.0.1 v1.0.0

* Creates a hotfix-v1.0.1 branch from v1.0.0.

**Final Thoughts**

✅ Use **cherry-picking** to apply selective commits.  
✅ Use **rebasing** for a clean, linear commit history.  
✅ Use **tags** to mark important commits for release management.

**Overview of Git Aliases**

* Creating **custom Git shortcuts**
* Managing & sharing aliases (git config --global alias)

**Overview of Git Aliases**

Git aliases allow you to create **custom shortcuts** for frequently used Git commands. This improves efficiency and simplifies workflows.

**1. Creating Custom Git Shortcuts (git alias)**

Aliases are defined using the git config command. You can add them at:

* **Local level** (specific to a repository)
* **Global level** (available across all repositories)

**🔹 Creating a Simple Alias**

Example: Instead of typing git status, create a shortcut:

git config --global alias.s status

Now, you can just type:

git s

**🔹 Aliases for Common Commands**

| **Alias** | **Full Command** | **Usage** |
| --- | --- | --- |
| git st | git status | Show working tree status |
| git co | git checkout | Switch branches |
| git br | git branch | List branches |
| git cm | git commit -m | Commit with a message |
| git lg | git log --oneline --graph --all --decorate | Pretty log output |
| git amend | git commit --amend | Amend last commit |
| git rb | git rebase | Rebase commits |
| git pl | git pull | Pull latest changes |

**2. Managing & Sharing Git Aliases**

**🔹 Viewing All Configured Aliases**

git config --global --list | grep alias

**🔹 Editing Aliases Manually**

You can edit the .gitconfig file directly:

nano ~/.gitconfig

Example .gitconfig snippet:

[alias]

s = status

co = checkout

cm = commit -m

lg = log --oneline --graph --all --decorate

**🔹 Sharing Aliases with Others**

* Copy your .gitconfig file and share it.
* Alternatively, export aliases using:

git config --global --list | grep alias > git-aliases.txt

**🔹 Importing Aliases from a File**

To load shared aliases:

cat git-aliases.txt | while read line; do git config --global $line; done

**Final Thoughts**

✅ **Aliases** save time and reduce repetitive typing.  
✅ **Use meaningful shortcuts** to improve workflow.  
✅ **Easily share & import** aliases across different machines.

**Markdown (.md) Summary**

* Writing project documentation with **Markdown (.md)**
* Best practices for .md files (Readme, Contributing, License, etc.)

**Markdown (.md) Summary for Project Documentation**

Markdown (.md) is a lightweight markup language widely used for **writing project documentation** in repositories like GitHub, GitLab, and Bitbucket.

**1. Writing Project Documentation with Markdown**

Markdown is commonly used for:  
✅ **README.md** → Project introduction and setup instructions  
✅ **CONTRIBUTING.md** → Guidelines for contributing to the project  
✅ **LICENSE.md** → Defines the licensing terms  
✅ **CHANGELOG.md** → Lists version updates and changes  
✅ **CODE\_OF\_CONDUCT.md** → Defines project behavior rules

**🔹 Basic Markdown Syntax**

| **Feature** | **Syntax** | **Example** |
| --- | --- | --- |
| **Headings** | # H1, ## H2, ### H3 | # Project Title |
| **Bold** | \*\*bold text\*\* | **bold text** |
| **Italic** | \*italic text\* | *italic text* |
| **Lists** | - Item 1 or 1. Item 1 | - Item 1  1. Item 1 |
| **Code Blocks** | ```code``` | print("Hello, Markdown!") |
| **Links** | [Text](URL) | [Visit GitHub](https://github.com) |
| **Images** | ![Alt Text](image\_url) |  |
| **Tables** | ` | Col1 |
| **Task Lists** | - [ ] Task 1 | - [ ] Task 1 |

**2. Best Practices for .md Files**

**🔹 README.md: Essential Information**

A well-structured README.md should include:  
✅ **Project Title**  
✅ **Description** (What the project does)  
✅ **Installation Guide**  
✅ **Usage Instructions**  
✅ **Contribution Guidelines**  
✅ **License Information**

Example README.md:

# Project Name

## Description

A brief description of what the project does.

## Installation

```sh

git clone https://github.com/user/repo.git

cd repo

npm install

**Usage**

npm start

**Contributing**

See <CONTRIBUTING.md>.

**License**

This project is licensed under the MIT License - see <LICENSE.md>.

---

### \*\*🔹 CONTRIBUTING.md: Contribution Guidelines\*\*

- Define contribution rules (e.g., pull requests, code style, branching).

- Example:

```md

# Contributing Guidelines

1. Fork the repository.

2. Clone your forked repo.

3. Create a feature branch (`git checkout -b feature-name`).

4. Commit and push your changes.

5. Submit a pull request.

**🔹 LICENSE.md: Licensing Information**

* Use an **open-source license** (MIT, Apache, GPL).
* Example MIT License:

MIT License

Copyright (c) 2024 [Your Name]

Permission is hereby granted, free of charge, to any person obtaining a copy...

**3. Tools for Editing Markdown**

🛠 **VS Code** → Built-in Markdown preview  
🛠 **Typora** → WYSIWYG Markdown editor  
🛠 **GitHub/GitLab** → Online Markdown rendering

**Final Thoughts**

✅ Use **Markdown** for clear, structured documentation.  
✅ Keep your README.md simple and **informative**.  
✅ Maintain **consistent formatting** across .md files.

**Hands-on Practice**

* Exercises on **stashing, cherry-picking, tags and aliases**

**Hands-on Practice: Git Stashing, Cherry-Picking, Tags, and Aliases**

These exercises will help you practice essential Git commands related to **stashing, cherry-picking, tags, and aliases**.

**1. Stashing Changes in Git**

**Exercise 1: Basic Stash and Apply**

1️. Create a new branch and make some changes.

git checkout -b stash-practice

echo "This is a temporary change" >> file.txt

git status

2️. Stash the changes.

git stash

3️. List the available stashes.

git stash list

4️. Apply the last stash and remove it.

git stash pop

5️. Apply a specific stash without removing it.

git stash apply stash@{0}

6️. Drop the stash.

git stash drop stash@{0}

7️. Clear all stashes.

git stash clear

**2. Cherry-Picking Specific Commits**

**Exercise 2: Selectively Apply Commits**

1️. Switch to the main branch and create a feature branch.

git checkout main

git checkout -b cherry-pick-demo

2️. Make multiple commits.

echo "Change 1" >> file1.txt && git add file1.txt && git commit -m "Commit 1"

echo "Change 2" >> file2.txt && git add file2.txt && git commit -m "Commit 2"

echo "Change 3" >> file3.txt && git add file3.txt && git commit -m "Commit 3"

3️. Switch to main and cherry-pick **only the second commit**.

git checkout main

git cherry-pick <commit-hash-of-commit-2>

4️. If there are conflicts, resolve them and continue:

git cherry-pick --continue

5️. Undo a cherry-picked commit:

git reset --hard HEAD~1

**3. Working with Git Tags**

**Exercise 3: Creating and Managing Tags**

1️. Switch to the main branch and create a versioned tag.

git checkout main

git tag -a v1.0 -m "Version 1.0 Release"

2️. Verify the tag.

git tag

3️. Push the tag to the remote repository.

git push origin v1.0

4️. Delete a tag locally and remotely.

git tag -d v1.0

git push origin --delete v1.0

5️. List all tags.

git tag --list

**4. Creating and Using Git Aliases**

**Exercise 4: Custom Git Shortcuts**

1️. Create aliases for frequently used commands.

git config --global alias.st status

git config --global alias.co checkout

git config --global alias.cm "commit -m"

git config --global alias.br branch

git config --global alias.last "log -1 HEAD"

2️. Use the aliases.

git st # Equivalent to git status

git co main # Equivalent to git checkout main

git cm "Quick commit" # Equivalent to git commit -m "Quick commit"

git br # Equivalent to git branch

git last # Equivalent to git log -1 HEAD

3️. View all aliases.

git config --global --list | grep alias

4️. Remove an alias.

git config --global --unset alias.st

**Final Task: Combine Everything**

1️. Create a feature branch, make changes, and stash them.  
2️. Switch to another branch and cherry-pick a commit.  
3️. Tag a commit and push it.  
4️. Use Git aliases to speed up commands.

**Overview of Git Submodules & Patches**

* Introduction to **submodules**
* Adding & updating submodules (git submodule add, git submodule update)
* Pitfalls & best practices
* Creating & applying patches (git format-patch, git apply, git am)

**Overview of Git Submodules & Patches**

**1. Git Submodules**

A **Git submodule** allows you to include one Git repository inside another as a dependency. This is useful when working with shared libraries or external dependencies in a project.

**🔹 Adding & Updating Submodules**

**Step 1: Adding a Submodule**

1. Navigate to your Git repository.  
2. Add a submodule using:

git submodule add https://github.com/example/repo.git submodule-folder

3. Initialize and update the submodule:

git submodule update --init --recursive

4️. Commit the change:

git commit -m "Added submodule"

5️. Push changes:

git push origin main

**Step 2: Cloning a Repository with Submodules**

If you clone a repo that contains submodules, you must initialize them separately:

git clone --recurse-submodules <repo-url>

**Step 3: Updating Submodules**

To update a submodule to the latest commit:

cd submodule-folder

git pull origin main

cd ..

git add submodule-folder

git commit -m "Updated submodule"

git push origin main

To update all submodules at once:

git submodule update --remote --merge

**🔹 Pitfalls & Best Practices**

✅ Always **initialize** submodules after cloning (git submodule update --init --recursive).  
✅ Use **submodules only for truly external dependencies** (otherwise, consider using a monorepo).  
✅ Ensure all contributors **update submodules** before working (git submodule update).  
✅ Avoid making direct commits inside a submodule; instead, update its reference in the parent repo.

**2. Creating & Applying Patches in Git**

**🔹 Creating a Patch (git format-patch)**

A **patch** is a file that contains differences between commits.

**Step 1: Generate a Patch for the Last Commit**

git format-patch -1

This creates a .patch file, e.g., 0001-commit-message.patch.

**Step 2: Generate a Patch for Multiple Commits**

git format-patch HEAD~3 # Creates patches for the last 3 commits

**Step 3: Save Patch to a File**

git format-patch -1 -o patches/

**🔹 Applying a Patch (git apply & git am)**

**Step 1: Apply a Patch Locally**

git apply 0001-commit-message.patch

**Step 2: Apply a Patch and Commit Automatically**

git am < 0001-commit-message.patch

**Step 3: Revert an Applied Patch**

git reset --hard HEAD~1

**Final Thoughts**

✅ **Submodules** help manage external repositories inside a project.  
✅ **Patches** allow sharing commit changes without pushing to a remote repository.  
✅ Always test patches before applying them.

**Overview of Git Hooks**

* **Introduction to Git hooks**: Client-side vs. Server-side
* Setting up & customizing hooks (pre-commit, pre-push, post-merge)
* Examples: Linting before commit, checking commit messages, preventing secrets from being committed
* Using Husky for managing hooks in a team

**Overview of Git Hooks**

**🔹 What Are Git Hooks?**

**Git hooks** are scripts that **automate actions** at different stages of the Git workflow. They are stored in the .git/hooks/ directory and can be used for various purposes like enforcing coding standards, validating commit messages, and running tests before pushing.

Git hooks are classified into:

* **Client-side hooks** – Run locally on the developer's machine (e.g., pre-commit, pre-push).
* **Server-side hooks** – Run on the remote repository server (e.g., pre-receive, post-receive).

**🔹 Setting Up & Customizing Hooks**

**Step 1: Locate the Git Hooks Directory**

Hooks are stored inside your local repository:

ls .git/hooks/

By default, Git provides sample hooks like pre-commit.sample.

**Step 2: Create a Custom Hook**

1. Navigate to the hooks directory:

cd .git/hooks

2. Create a pre-commit hook:

touch pre-commit

chmod +x pre-commit

3. Add a simple script to prevent commits with "WIP" in the message:

#!/bin/bash

if git diff --cached | grep "WIP"; then

echo "❌ Commit contains 'WIP'. Please remove it."

exit 1

fi

4. Save the file and test by committing a change.

**🔹 Common Hooks & Use Cases**

**1. Pre-Commit Hook (Runs Before Commit)**

Example: **Run ESLint before committing JavaScript files**

#!/bin/bash

if ! eslint .; then

echo "❌ ESLint failed. Fix issues before committing."

exit 1

fi

**2. Pre-Push Hook (Runs Before Pushing)**

Example: **Prevent pushing to main branch directly**

#!/bin/bash

branch=$(git rev-parse --abbrev-ref HEAD)

if [[ "$branch" == "main" ]]; then

echo "❌ You cannot push directly to the main branch."

exit 1

fi

**3. Post-Merge Hook (Runs After Merging)**

Example: **Run npm install after merging**

#!/bin/bash

if [ -f package.json ]; then

echo "📦 Running npm install after merge..."

npm install

fi

**🔹 Using Husky for Managing Hooks in a Team**

[Husky](https://github.com/typicode/husky) is a tool that **manages Git hooks in JavaScript projects**.

**Step 1: Install Husky**

npm install husky --save-dev

**Step 2: Enable Git Hooks**

npx husky install

**Step 3: Add a Pre-Commit Hook with Husky**

npx husky add .husky/pre-commit "npm run lint"

git add .husky/pre-commit

**Step 4: Configure Husky in package.json**

"scripts": {

"prepare": "husky install"

}

**Step 5: Run Husky Automatically**

Now, npm run prepare will enable Husky for all developers on the team.

**Final Thoughts**

✅ **Git hooks** enforce code quality and security.  
✅ **Client-side hooks** prevent mistakes before committing.  
✅ **Server-side hooks** enforce policies before merging.  
✅ **Husky** simplifies Git hooks for JavaScript projects.

**Advanced Collaboration Techniques**

* Working with **forks** & upstream repositories
* Managing **multiple remotes**
* Best **pull request strategies** for large teams

**Advanced Collaboration Techniques in Git**

When working with large teams or open-source projects, Git provides several powerful features to collaborate effectively. This includes **forks, upstream repositories**, and **managing multiple remotes**. Here's a breakdown of advanced collaboration techniques:

**🔹 Working with Forks & Upstream Repositories**

**What is a Fork?**

A **fork** is a copy of a repository that allows you to freely experiment with changes without affecting the original project. It is commonly used in open-source projects to contribute by creating pull requests.

**Forking a Repository**

1. Navigate to the repository on GitHub (or another Git host) and click on the **Fork** button.  
2. After forking, clone the repository to your local machine:

git clone https://github.com/your-username/repository-name.git

cd repository-name

**Working with Upstream Repositories**

The **upstream repository** is the original repository you forked from. To stay updated with changes made by others in the original repository, you need to add it as a remote and sync your fork.

**Step 1: Add Upstream Remote**

1. First, check the current remotes:

git remote -v

You’ll see the URL of your fork (origin). Now add the **upstream** remote:

git remote add upstream https://github.com/original-owner/repository-name.git

**Step 2: Sync Your Fork with Upstream**

To keep your fork up-to-date with the original repository, fetch the changes from **upstream**:

git fetch upstream

Then, merge or rebase the upstream changes into your fork’s main branch:

git checkout main

git merge upstream/main

Alternatively, you can **rebase** instead of merging:

git rebase upstream/main

**Step 3: Push Changes to Your Fork**

Once you've updated your local fork with the latest upstream changes, push them to your GitHub fork:

git push origin main

**🔹 Managing Multiple Remotes**

**What is a Remote in Git?**

A **remote** is a reference to a remote repository hosted on a server. It is how Git allows you to collaborate by pulling and pushing code to remote repositories like GitHub, GitLab, or Bitbucket.

**Working with Multiple Remotes**

In large projects, you may need to work with multiple remotes (e.g., your fork, the upstream repository, and other collaborators' repositories).

**Step 1: Check Existing Remotes**

git remote -v

This will list your remotes (e.g., origin, upstream).

**Step 2: Add Another Remote**

To add another remote (e.g., a collaborator's repo):

git remote add collaborator https://github.com/collaborator/repository-name.git

**Step 3: Fetch from Multiple Remotes**

Fetch from all remotes:

git fetch --all

**Step 4: Push to a Specific Remote**

If you want to push changes to a specific remote, use:

git push collaborator branch-name

**🔹 Best Pull Request Strategies for Large Teams**

**1. Use Feature Branches**

Each team member should work on their own **feature branch** (instead of working on the main or master branch) to prevent conflicts and facilitate easy code reviews.

**Step 1: Create a Feature Branch**

git checkout -b feature/feature-name

**2. Open a Pull Request (PR) Early**

In large teams, it’s a good practice to open a pull request as soon as you’ve made meaningful progress. This allows early feedback and reduces the time taken to get the feature merged.

**3. Rebase Before Opening a PR**

Before opening a PR, rebase your feature branch to ensure your changes are based on the latest code in the main branch:

git fetch upstream

git rebase upstream/main

**4. Create Small, Focused Pull Requests**

Avoid large pull requests. Instead, break them down into smaller, focused PRs that solve specific problems. This makes it easier for reviewers to understand and provide feedback.

**5. Use PR Templates**

To ensure that pull requests are consistent, use **PR templates**. This can be added to the .github/PULL\_REQUEST\_TEMPLATE.md file in the repository.

Example PR template:

## Description

- What was changed and why.

## Related Issue

- Link to the related issue (e.g., #123).

## How to Test

- Steps to test the changes.

## Checklist

- [ ] I have tested my changes locally.

- [ ] I have added necessary documentation.

- [ ] I have reviewed the code for any issues.

**6. Require Reviews and Approvals**

Set up branch protection rules (on GitHub/GitLab) to require at least one approval from a team member before merging a pull request into the main branch.

**7. Use Continuous Integration (CI) Tools**

Integrate **CI tools** (like GitHub Actions, Travis CI, or CircleCI) to automatically run tests and linting checks before merging PRs. This ensures that code quality is maintained.

**8. Squash and Merge**

In large teams, squashing commits before merging helps keep the commit history clean. GitHub allows you to squash and merge commits directly via the GitHub interface.

**Final Thoughts**

✅ **Forks** are great for contributing to open-source projects.  
✅ **Upstream repositories** help keep your fork up-to-date.  
✅ Managing **multiple remotes** enables smoother collaboration.  
✅ **Feature branches** and **small PRs** are essential for effective teamwork.  
✅ **PR templates** and **CI tools** ensure consistency and code quality.

Performance Optimization

• Shallow clones and sparse-checkouts

• Large repository management strategies

**Performance Optimization in Git**

Working with large repositories or numerous branches can lead to performance issues in Git, such as slow clones or checkouts. Git provides various tools and strategies to optimize performance and manage large repositories efficiently. This section covers **shallow clones**, **sparse-checkouts**, and strategies for managing large repositories.

**🔹 Shallow Clones and Sparse-Checkouts**

**1. Shallow Clones**

A **shallow clone** is a Git clone that doesn't retrieve the full history of a repository. Instead, it retrieves only a limited number of commits from the repository's history. This is useful when you want to reduce the size of the clone or only need the latest changes.

**Why Use Shallow Clones?**

* **Faster Clones**: Shallow clones are faster to perform because they don’t include the entire repository history.
* **Smaller Size**: The size of the clone is smaller, which saves on disk space.
* **Ideal for CI/CD**: Shallow clones are beneficial for automated systems like CI/CD pipelines where you don’t need the full history.

**How to Create a Shallow Clone**

You can create a shallow clone with the --depth option to limit the history retrieved. For example, if you only want the latest 5 commits:

git clone --depth 5 https://github.com/user/repository.git

This will only fetch the latest 5 commits, reducing the clone size.

**Shallow Clone with a Specific Branch**

You can also specify a particular branch to shallow clone:

git clone --depth 5 --branch branch-name https://github.com/user/repository.git

**Shallow Clone and Fetch More History**

If you later need to fetch more commits from the history, you can use:

git fetch --unshallow

This command converts the shallow clone into a full clone, downloading all the history.

**2. Sparse-Checkout**

**Sparse-checkout** allows you to check out only specific directories or files from a repository, which can be beneficial for working with large repositories where you don’t need all files.

**Why Use Sparse-Checkout?**

* **Improves Performance**: Instead of checking out the entire repository, sparse-checkout allows you to focus only on the parts of the repository you need.
* **Reduces Disk Usage**: You avoid unnecessary files and directories from being checked out, saving disk space.

**How to Set Up Sparse-Checkout**

1. **Enable Sparse-Checkout**  
First, enable the sparse-checkout option:

git config core.sparseCheckout true

2. **Create a Sparse-Checkout File**  
Next, define which directories or files you want to check out by editing the .git/info/sparse-checkout file. For example:

# Include only the 'src' directory

/src/

3. **Checkout the Sparse Repository**  
Once you've defined your sparse-checkout settings, you can proceed with the checkout:

git checkout branch-name

Git will now only checkout the files and directories specified in .git/info/sparse-checkout.

**How to Use Sparse-Checkout in a Shallow Clone**

You can combine shallow clones and sparse-checkout for optimized performance when you only need certain parts of a repository:

git clone --depth 1 --filter=blob:none --no-checkout https://github.com/user/repository.git

cd repository

git sparse-checkout init --cone

git sparse-checkout set src/ # Only checkout the 'src' folder

**🔹 Large Repository Management Strategies**

**1. Large Binary Files with Git LFS (Large File Storage)**

Git is not optimized for managing large binary files, such as images, videos, or compiled assets. Using **Git LFS (Large File Storage)**, you can store large files outside of the main repository and manage them in a more efficient manner.

**How to Set Up Git LFS**

1. Install Git LFS:

git lfs install

2. Track Large Files:

git lfs track "\*.mp4" # Track .mp4 files with Git LFS

3. Add and Commit Files as Usual:

git add .gitattributes

git add large-video.mp4

git commit -m "Add large video file"

Git LFS will store the actual binary files on a separate server, reducing the repository size.

**2. Git Submodules for Large Projects**

For very large projects, especially when dealing with third-party dependencies or separate components, using **Git submodules** can be an effective way to manage large repositories. A submodule is a Git repository inside another Git repository.

**How to Use Git Submodules**

1. **Adding a Submodule**  
To add a submodule, use the git submodule add command:

git submodule add https://github.com/other-repo/repository.git path/to/submodule

2. **Cloning a Repository with Submodules**  
When cloning a repository with submodules, use the --recurse-submodules option:

git clone --recurse-submodules https://github.com/your-repo/repository.git

3. **Updating Submodules**  
To fetch and checkout the correct commit for the submodule:

git submodule update --remote

**3. Git’s filter-branch and filter-repo for Repository Cleanup**

In very large repositories, sometimes you need to reduce the size by removing large files from history (such as log files, assets, or old binaries). This can be achieved by using the git filter-branch or the more efficient git filter-repo tool.

**Using git filter-repo**

git filter-repo is a faster and more flexible tool than git filter-branch for cleaning up repository history.

1. **Install git filter-repo**:

pip install git-filter-repo

2. **Remove a File from History**:

git filter-repo --path path/to/large-file --invert-paths

This removes the specified file from the entire history of the repository.

**4. Git Repository Splitting (Monorepos)**

When dealing with monorepos (a large repository containing multiple projects), it’s useful to split the repository into smaller, more manageable parts. You can use tools like git-subsplit or even split your repository using filter-repo to create individual repositories for each project.

**How to Split a Git Repository**

1. **Split the Repository by Directory**:

git filter-repo --subdirectory-filter path/to/directory

This splits the repository, keeping only the history of the specified directory.

**Final Thoughts**

✅ **Shallow clones** and **sparse-checkouts** are great tools to optimize clone times and reduce disk usage.  
✅ Use **Git LFS** for managing large binary files.  
✅ Manage large projects effectively with **submodules** or **repository splitting**.  
✅ For history cleanup, **git filter-repo** is an efficient tool.

**GitHub: Deep Dive & Workflow with .yml file**

* Understanding **GitHub repositories & settings**
* Using **GitHub Projects** for issue tracking
* Understanding **.yml file structure** for GitHub Actions
* Hands-on: Creating an **automated workflow with .yml file**

**GitHub: Deep Dive & Workflow with .yml File**

GitHub is a powerful platform for version control and collaboration, offering various features for managing repositories, tracking issues, and automating workflows with GitHub Actions. In this section, we will explore key GitHub features like repositories, GitHub Projects for issue tracking, and creating automated workflows using .yml files.

**🔹 Understanding GitHub Repositories & Settings**

A **GitHub repository** is where your project code, files, and version history live. GitHub repositories are essential for version control and collaboration, allowing teams to contribute to projects and track changes over time.

**Key Features of GitHub Repositories**

1. **Repository Types**:
   * **Public**: Visible to everyone; can be forked and cloned by anyone.
   * **Private**: Only accessible by collaborators and organization members.
2. **Branching**: GitHub repositories support branching, allowing multiple contributors to work on separate features or fixes.
3. **Pull Requests**: Pull requests allow you to propose changes to a repository, review them, and merge them into the main branch.
4. **Settings**: GitHub repositories have several configurable settings:
   * **Branches**: Define branch protection rules.
   * **Actions**: Enable GitHub Actions to automate workflows.
   * **Collaborators**: Manage who can access and contribute to the repository.
   * **Webhooks**: Set up automated events triggered by specific actions (e.g., pushing to a branch).

**🔹 Using GitHub Projects for Issue Tracking**

**GitHub Projects** helps you manage tasks and track the progress of your work directly within your repository. It's an excellent tool for organizing issues, pull requests, and tasks in a Kanban-like board.

**Key Features of GitHub Projects**

1. **Project Boards**: GitHub Projects provides Kanban boards to manage tasks through different columns, like "To Do," "In Progress," and "Done."
2. **Issues and Pull Requests**: You can link GitHub Issues or Pull Requests to a specific project, making it easier to track their status.
3. **Automation**: You can automate workflows within the project by setting up triggers that automatically move tasks across columns when certain conditions are met (e.g., when a pull request is merged).
4. **Customizable Views**: You can set up custom fields for each project card, such as labels, assignees, and due dates.

**Creating a GitHub Project**

1. Go to your repository and click on the **Projects** tab.
2. Click **New Project** and choose a template (Kanban, Automated, or Basic).
3. Add cards, such as **Issues** or **Pull Requests**, and move them through your columns.

**🔹 Understanding .yml File Structure for GitHub Actions**

GitHub Actions allows you to automate workflows and CI/CD pipelines directly in your repository. These workflows are defined using .yml files, which specify the steps to be performed on certain events (e.g., pushing code, creating pull requests).

**Structure of a .yml File in GitHub Actions**

A typical .yml file for GitHub Actions follows this structure:

name: Workflow Name

on:

event: [push, pull\_request] # Triggers for the workflow (e.g., push, pull request, etc.)

branches:

- main # The branch that triggers the event

jobs:

job\_name:

runs-on: ubuntu-latest # Specify the runner environment

steps:

- name: Checkout code

uses: actions/checkout@v2 # Action to checkout the repository

- name: Setup Node.js

uses: actions/setup-node@v2 # Action to set up Node.js

with:

node-version: '14'

- name: Install dependencies

run: npm install

- name: Run tests

run: npm test

**Key Sections in .yml Files**

1. **name**: Defines the name of the workflow.
2. **on**: Specifies the event that triggers the workflow. Common events include:
   * push: Triggers on commits to a branch.
   * pull\_request: Triggers when a pull request is opened, synchronized, or closed.
   * schedule: Triggers on a schedule (e.g., cron jobs).
3. **jobs**: Defines the jobs that will run within the workflow.
   * job\_name: A unique name for the job.
   * runs-on: Specifies the environment for the job (e.g., ubuntu-latest, windows-latest, macos-latest).
   * **steps**: Defines the sequence of actions to be performed in the job, such as checking out code, setting up environments, installing dependencies, and running tests.

**Common GitHub Actions Keywords**

* **uses**: Specifies a pre-defined action to run.
* **run**: Executes a custom command or script.
* **with**: Passes input parameters to an action.

**🔹 Hands-on: Creating an Automated Workflow with .yml File**

Let’s create a simple automated workflow in GitHub Actions for a Node.js project. This workflow will be triggered on every push to the main branch, and it will install dependencies and run tests.

**Steps to Create an Automated Workflow**

1. **Create the GitHub Actions Workflow File**:
   * In your GitHub repository, create a directory named .github/workflows.
   * Inside that directory, create a file named nodejs-ci.yml.
2. **Define the Workflow**: Open nodejs-ci.yml and add the following contents:

name: Node.js CI

on:

push:

branches:

- main

jobs:

build:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v2

- name: Set up Node.js

uses: actions/setup-node@v2

with:

node-version: '14'

- name: Install dependencies

run: npm install

- name: Run tests

run: npm test

This .yml file specifies:

* + The workflow is triggered on a push to the main branch.
  + The job runs on an ubuntu-latest virtual machine.
  + The steps include checking out the code, setting up Node.js version 14, installing dependencies, and running tests.

1. **Push the Changes to GitHub**:
   * Commit and push your .yml file to the repository:

git add .github/workflows/nodejs-ci.yml

git commit -m "Add GitHub Actions workflow for Node.js CI"

git push origin main

1. **Check the Workflow**:
   * Once you push the changes, go to the **Actions** tab in your GitHub repository.
   * You will see the workflow running under the **"Node.js CI"** workflow.
   * If everything is configured correctly, it will check out the code, set up Node.js, install dependencies, and run the tests.

**🔹 Summary**

* **GitHub Repositories**: Manage your code, issues, and pull requests in a centralized location.
* **GitHub Projects**: Organize tasks and issues with project boards in a Kanban-like view.
* **GitHub Actions**: Automate workflows with .yml files, which define triggers, jobs, and steps.
* **.yml File**: Used to specify GitHub Actions workflows. These workflows define the automation of tasks like continuous integration (CI), deployment, and testing.

**Hands-on Practice: GitHub Actions & Hooks**

* Setting up **GitHub Actions** for CI/CD
* Implementing **Git Hooks** in projects

**Hands-on Practice: GitHub Actions & Hooks**

**In this session, you will learn to set up GitHub Actions for Continuous Integration (CI) and Continuous Deployment (CD) as well as implement Git Hooks in your project to automate tasks and enforce quality control.**

**1. Setting Up GitHub Actions for CI/CD**

**Objective: Set up an automated CI/CD pipeline using GitHub Actions for a simple Node.js project that will automatically run tests and deploy code on the main branch.**

**Step 1: Create a GitHub Repository**

**If you haven’t already, create a GitHub repository or use an existing one.**

1. **Go to** [**GitHub**](https://github.com/) **and create a new repository (or open an existing one).**
2. **Clone the repository to your local machine using:**

**git clone https://github.com/your-username/your-repo.git**

**cd your-repo**

**Step 2: Create a Workflow File for CI/CD**

1. **In the root of your repository, create the .github/workflows directory:**

**mkdir -p .github/workflows**

1. **Inside .github/workflows, create a new YAML file for the CI workflow:**

**touch .github/workflows/nodejs-ci.yml**

1. **Open the nodejs-ci.yml file and add the following contents:**

**name: Node.js CI/CD Pipeline**

**on:**

**push:**

**branches:**

**- main**

**pull\_request:**

**branches:**

**- main**

**jobs:**

**build:**

**runs-on: ubuntu-latest**

**steps:**

**- name: Checkout code**

**uses: actions/checkout@v2**

**- name: Set up Node.js**

**uses: actions/setup-node@v2**

**with:**

**node-version: '14'**

**- name: Install dependencies**

**run: npm install**

**- name: Run tests**

**run: npm test**

**- name: Deploy to Production**

**if: github.ref == 'refs/heads/main'**

**run: |**

**echo "Deploying to production server"**

**# Add your deployment commands here (e.g., AWS, Heroku)**

**Explanation:**

* + **on.push: Triggers the workflow when there’s a push to the main branch.**
  + **on.pull\_request: Also triggers the workflow when a pull request is made to the main branch.**
  + **jobs.build: Defines a job called build that runs on an ubuntu-latest runner.**
  + **steps: Each step performs a task like checking out code, setting up Node.js, installing dependencies, running tests, and deploying the code.**

**Step 3: Commit & Push the Workflow File**

1. **Add the .yml file to the staging area:**

**git add .github/workflows/nodejs-ci.yml**

1. **Commit the changes:**

**git commit -m "Add GitHub Actions CI/CD pipeline"**

1. **Push the changes to GitHub:**

**git push origin main**

**Step 4: Monitor the Workflow**

**Once pushed, go to the Actions tab of your GitHub repository. You should see the newly triggered CI/CD pipeline. GitHub will run the jobs defined in the .yml file automatically:**

* **It will check out the code.**
* **Set up Node.js.**
* **Install dependencies (npm install).**
* **Run tests (npm test).**
* **Deploy if the push is to the main branch.**

**2. Implementing Git Hooks in Projects**

**Git hooks are scripts that Git executes before or after events such as commits, pushes, and merges. They help automate common tasks and enforce policies.**

**Objective: Set up Git hooks using Husky to ensure code quality before commits and pushes.**

**Step 1: Install Husky**

**Husky is a popular tool for managing Git hooks. It allows you to set up hooks easily and run commands like linting or testing before code is committed.**

1. **Install Husky as a development dependency:**

**npm install husky --save-dev**

1. **Enable Husky hooks:**

**npx husky install**

1. **Add a post-install script to enable Husky after each install:**

**// In package.json, add the following in the "scripts" section:**

**"scripts": {**

**"prepare": "husky install"**

**}**

1. **Run the command to create a default .husky directory:**

**npx husky add .husky/pre-commit "npm test"**

**This command adds a pre-commit hook that runs the tests before every commit.**

**Step 2: Add Custom Git Hooks**

1. **Create additional Git hooks, such as a pre-push hook to check for linting errors before code is pushed to the repository:**

**npx husky add .husky/pre-push "npm run lint"**

1. **Your .husky directory should now contain two scripts:**
   * **pre-commit: Runs npm test before committing.**
   * **pre-push: Runs npm run lint before pushing code.**

**Step 3: Verify the Git Hooks**

1. **Pre-commit Hook:**
   * **Make a change to your code (e.g., add a bug or modify a test).**
   * **Try to commit:**

**git commit -am "Test pre-commit hook"**

* + **The pre-commit hook will run npm test. If tests fail, the commit will be rejected, and you will need to fix the tests first.**

1. **Pre-push Hook:**
   * **Before pushing changes to GitHub, the pre-push hook will run npm run lint. If linting fails, you won’t be able to push the changes until you fix the linting issues.**

**Step 4: Push Your Changes to GitHub**

**Once the Git hooks are configured, try making a change, committing, and pushing to GitHub. The hooks should be triggered automatically before the commit and push.**

**Summary**

* **GitHub Actions allows you to automate CI/CD workflows using .yml files, which can automate tasks such as testing, linting, and deploying code.**
* **Git Hooks allow you to automate tasks before or after certain Git actions, like committing or pushing code, ensuring code quality and consistency.**
* **Husky makes managing Git hooks easy, helping you integrate tasks like linting and testing directly into your workflow.**

**Debugging & Troubleshooting**

* **Using git bisect** for bug identification
* **Analyzing history** with git blame & git log
* **Recovering from merge & rebase conflicts**

**Debugging & Troubleshooting in Git**

Git provides several tools to help you track down and fix issues in your code, manage conflicts, and understand the history of changes. In this section, we will explore how to use **git bisect** for bug identification, **git blame** and **git log** to analyze history, and how to recover from merge and rebase conflicts.

**1. Using git bisect for Bug Identification**

**Git bisect** is a powerful tool that helps you identify which commit introduced a bug. It performs a binary search through your commit history, testing each commit until it finds the culprit.

**Objective: Use git bisect to find the commit that introduced a bug.**

**Step 1: Start Bisecting**

1. **Mark a known good commit**: First, you need to specify a commit where the code worked correctly. This will be your "good" commit.
2. **Mark a known bad commit**: Next, mark the commit where the bug was detected. This is your "bad" commit.

Run the following command to start the bisect process:

git bisect start

git bisect bad # Mark the current commit as bad

git bisect good <commit-hash> # Mark a known good commit

Git will now automatically checkout a commit that is roughly halfway between the "good" and "bad" commits.

**Step 2: Test and Mark Commits**

Git will check out a commit for you to test. At this point, run your tests or manually check if the bug is present.

* If the bug is present, mark the commit as **bad**:

git bisect bad

* If the bug is **not** present, mark the commit as **good**:

git bisect good

Git will continue to narrow down the search space, testing commits between the good and bad commits. You’ll need to continue running tests until Git identifies the exact commit that introduced the bug.

**Step 3: Finish Bisecting**

Once Git identifies the problematic commit, it will display the commit hash and message. To finish the bisect process, run:

git bisect reset

This will return you to the branch you were originally working on.

**2. Analyzing History with git blame & git log**

**Objective: Use git blame and git log to investigate and analyze the history of your code and understand who made specific changes.**

**Using git blame**

git blame helps you see who last modified each line of a file. This is helpful for understanding when a particular line of code was introduced or changed.

**Example Usage:**

1. To blame a specific file, run:

git blame <file-path>

This will show you a line-by-line annotation of the file, including:

* + Commit hash
  + Author of the commit
  + Date of the commit
  + The actual line of code

1. You can also blame specific lines of a file:

git blame -L <start-line>,<end-line> <file-path>

This will show the blame information for a specific range of lines in the file.

**Using git log**

git log is used to show the commit history of a repository. It provides information about the commits, including the commit hash, author, date, and commit message.

**Example Usage:**

1. To view the commit history of the repository:

git log

This will show the commits starting from the most recent commit.

1. To view a specific file's commit history:

git log <file-path>

1. To see a condensed log with one line per commit:

git log --oneline

1. You can filter the log by author:

git log --author="Author Name"

1. To view a graphical log with branches and merges:

git log --graph --oneline --all

This can be very useful when you need to track down where specific changes were introduced in a file or branch.

**3. Recovering from Merge & Rebase Conflicts**

**Objective: Learn how to handle and resolve conflicts during merges and rebases.**

**Handling Merge Conflicts**

A merge conflict occurs when two branches have changes in the same part of a file. Git cannot automatically merge them and requires manual intervention.

**Steps to Resolve Merge Conflicts:**

1. **Start a merge**:

git merge <branch-name>

If there are conflicts, Git will notify you, and the files with conflicts will be marked.

1. **Locate the conflict**: Git marks the conflicts in the file like this:

<<<<<<< HEAD

code from the current branch

=======

code from the other branch

>>>>>>> <branch-name>

1. **Manually resolve the conflict**:
   * Edit the file to resolve the conflict, choosing or combining the changes as necessary.
   * Remove the conflict markers (<<<<<<<, =======, >>>>>>>).
2. **Stage the resolved file**:

git add <file-path>

1. **Complete the merge**: After resolving the conflicts, commit the merge:

git commit

**Handling Rebase Conflicts**

During a rebase, Git attempts to apply each commit from one branch onto another. If a conflict occurs, Git will pause the rebase process and require you to resolve the conflict.

**Steps to Resolve Rebase Conflicts:**

1. **Start the rebase**:

git rebase <branch-name>

1. **Resolve conflicts**: If a conflict occurs, Git will stop and indicate which files have conflicts. Resolve them in the same way as merge conflicts.
2. **Continue the rebase**: After resolving conflicts, continue the rebase:

git rebase --continue

If there are further conflicts, Git will stop again. Repeat the resolution process until the rebase is complete.

1. **Abort the rebase**: If you decide you want to cancel the rebase process, you can abort:

git rebase --abort

This will return you to the state before the rebase started.

**Summary**

* **Git bisect** is a powerful tool for finding the commit that introduced a bug by performing a binary search.
* **Git blame** shows who modified each line of a file and when, helping to trace the history of specific changes.
* **Git log** provides a detailed history of commits, which can be used to understand changes in the repository over time.
* **Merge and rebase conflicts** are common when working with multiple branches. Understanding how to resolve these conflicts manually and continue the process is crucial for maintaining a clean project history.

**Git with Jira & Automation**

* **Issue tracking integration**
* Linking **commits & pull requests** to Jira issues
* **Webhook-based workflow automation**

**Git with Jira & Automation**

Integrating Git with Jira and automating workflows is a powerful combination that enhances the collaboration, issue tracking, and productivity of development teams. This integration allows developers to link their commits and pull requests directly to Jira issues, ensuring a seamless workflow between code and project management tools.

Let’s break down how you can integrate **Git with Jira** and set up **automated workflows** using webhooks.

**1. Issue Tracking Integration**

**Objective: Learn how to integrate Git with Jira for issue tracking.**

Jira is widely used for tracking tasks, bugs, user stories, and more. By integrating Jira with Git, you can streamline the tracking process and connect code changes directly to specific Jira issues.

**Steps for Integrating Git with Jira:**

**1. Set Up a Jira Integration**

Most popular Git platforms (like GitHub, GitLab, Bitbucket) provide built-in integration with Jira. Here’s how to integrate:

* **GitHub & Jira Integration**: Use the **GitHub for Jira** app to automatically link commits and pull requests to Jira issues. This is done by installing the app from the Atlassian Marketplace and configuring it in your GitHub repository settings.
* **Bitbucket & Jira Integration**: Bitbucket is an Atlassian product, so integration with Jira is seamless. You simply need to connect your Bitbucket account with Jira via the application links in the Jira settings.
* **GitLab & Jira Integration**: GitLab has its own Jira integration available. You can connect GitLab with Jira using the **Jira DevOps Integration** settings available in the project settings of Jira and GitLab.

Once integrated, you can link Git commits and pull requests to Jira issues.

**2. Link Commits and Pull Requests to Jira Issues**

Linking Git commits and pull requests to Jira issues ensures that every change in the codebase is associated with a specific task or bug.

**Linking Commits to Jira Issues:**

To link a commit to a Jira issue, you simply need to include the Jira issue key (e.g., PROJECT-123) in the commit message. For example:

git commit -m "Fix bug in authentication logic. Closes PROJECT-123"

* **PROJECT-123**: The issue key from Jira.
* This will automatically associate the commit with the Jira issue PROJECT-123 once the commit is pushed to the repository.

**Linking Pull Requests to Jira Issues:**

For pull requests, include the Jira issue key in the title or description of the pull request. For example:

Title: "Fix authentication bug - Resolves PROJECT-123"

By including the issue key, Jira will automatically link the pull request with the corresponding issue. When the pull request is merged, the status of the Jira issue can be automatically updated to reflect the completion of the task.

**2. Webhook-Based Workflow Automation**

**Objective: Automate workflows between Git and Jira using webhooks.**

Webhooks allow you to automate actions in response to certain events, such as when a commit is pushed or a pull request is created. You can use webhooks to trigger actions in Jira, such as automatically transitioning issues, updating issue statuses, or notifying team members about code changes.

**Steps to Set Up Webhooks:**

**1. Creating a Webhook in Jira**

You can create webhooks in Jira to automate workflows when specific events happen in your Git repository (e.g., commit, pull request). Follow these steps:

1. **Navigate to Jira Settings**: Go to **Jira Administration > System > Webhooks**.
2. **Create a New Webhook**: Click the "Create a Webhook" button.
3. **Configure the Webhook**:
   * Provide a name for the webhook.
   * Add the URL for the webhook handler (this will be the URL of your Git hosting service’s webhook listener or a custom handler you create).
   * Choose the events that should trigger the webhook (e.g., commit, pull request, merge).

**2. Configuring Git Webhooks**

Now that you’ve set up a webhook in Jira, you need to set up a corresponding webhook in your Git repository.

**For GitHub:**

1. Go to your repository in GitHub.
2. Click on **Settings** > **Webhooks** > **Add Webhook**.
3. Add the payload URL (provided by Jira) and configure the events (e.g., push events, pull request events).
4. Save the webhook.

**For Bitbucket:**

1. Go to your repository in Bitbucket.
2. Click on **Repository Settings** > **Webhooks** > **Add Webhook**.
3. Add the payload URL (provided by Jira).
4. Choose the events to trigger the webhook.
5. Save the webhook.

**For GitLab:**

1. Navigate to your GitLab project.
2. Go to **Settings** > **Webhooks**.
3. Enter the webhook URL (provided by Jira).
4. Select events to trigger the webhook (e.g., push, pull requests).
5. Save the webhook.

**3. Automating Jira Actions Based on Git Events**

Once the webhook is configured, you can define actions to trigger in Jira based on the events from Git.

For example, you could automate the following actions:

* **Automatically transition Jira issues when a pull request is merged**: When a pull request is merged, the corresponding Jira issue could be moved to the "Done" status automatically.
* **Update Jira issues when commits are made**: You could set up a workflow where a commit with a specific message automatically transitions an issue to the next stage (e.g., "In Progress" when a commit is made).
* **Send notifications to team members**: Notify team members via Slack, email, or in-app Jira notifications when specific events happen in the repository.

**Example Workflow:**

1. **Developer Pushes Code**: A developer pushes code to the Git repository with a commit message like "Fix bug in login, Closes PROJECT-123".
2. **Webhook Triggers Jira Action**: The webhook set up in the Git repository triggers a call to Jira’s webhook listener, which automatically updates the issue status to "In Progress".
3. **Pull Request Created**: The developer creates a pull request. The webhook sends the request details to Jira, and the corresponding Jira issue is linked to the pull request.
4. **Pull Request Merged**: When the pull request is merged, the webhook sends the event to Jira, and the issue status is updated to "Done".

**Best Practices for Git-Jira Integration**

1. **Use Consistent Naming Conventions**: Ensure that all commit messages and pull request titles include the correct Jira issue key. This makes it easier to track progress.
2. **Automate Transitions in Jira**: Use webhooks to automate the transition of Jira issues based on Git events (e.g., when a pull request is merged, automatically transition the issue to the "Done" state).
3. **Ensure Proper Permissions**: Ensure that the integration and webhook configurations have the necessary permissions to make changes to Jira issues.
4. **Use Jira Smart Commits**: With smart commits in Jira, you can include additional commands in your commit message (e.g., #time, #comment, #transition) to update Jira issues from Git directly.

**Conclusion**

By integrating Git with Jira, you can streamline your development process by automatically linking commits and pull requests to Jira issues and setting up automated workflows using webhooks. This integration enhances collaboration and helps ensure that the project management and version control systems stay in sync.

**GitHub API & Integrations**

* **Using the GitHub API for automation**
* **Integrating GitHub with third-party tools**

**GitHub API & Integrations**

GitHub provides a powerful **API** that allows developers to automate tasks, integrate with other services, and enhance their workflows. Whether you're automating tasks like issue creation or managing repositories, or integrating GitHub with third-party tools like CI/CD platforms, project management software, or custom applications, the GitHub API is essential for building a streamlined development process.

**1. Using the GitHub API for Automation**

**Objective: Learn how to use the GitHub API for automating tasks like managing issues, repositories, and pull requests.**

GitHub's REST API allows you to interact with various aspects of GitHub repositories, including creating issues, managing pull requests, and interacting with commit history.

**Authentication:**

Before using the GitHub API, you'll need to authenticate. GitHub supports several authentication methods:

* **Personal Access Token (PAT)**: You can create a PAT through GitHub’s website under **Settings > Developer Settings > Personal Access Tokens**. This is the recommended method for authentication.
* **OAuth**: OAuth is used when integrating third-party applications with GitHub.
* **GitHub Apps**: A GitHub App is an alternative to OAuth, providing more granular permissions and better security for integrations.

For most tasks, a **Personal Access Token** is sufficient.

**API Endpoints:**

GitHub provides a variety of endpoints to interact with repositories, issues, pull requests, and more. Below are some common tasks and their associated API endpoints:

**1. Managing Issues:**

* **Create an Issue**: You can create a new issue via the API by sending a POST request to the following endpoint:

POST https://api.github.com/repos/{owner}/{repo}/issues

Example:

{

"title": "Fix bug in authentication",

"body": "The authentication logic fails under certain conditions",

"labels": ["bug"]

}

* **List Issues**: You can fetch issues from a repository using a GET request:

GET https://api.github.com/repos/{owner}/{repo}/issues

**2. Managing Pull Requests:**

* **Create a Pull Request**:

POST https://api.github.com/repos/{owner}/{repo}/pulls

Example:

{

"title": "Fix bug in login",

"head": "bugfix-branch",

"base": "main",

"body": "This pull request fixes the authentication issue"

}

* **List Pull Requests**:

GET https://api.github.com/repos/{owner}/{repo}/pulls

**3. Manage Repository:**

* **Create a Repository**:

POST https://api.github.com/user/repos

Example:

{

"name": "new-repo",

"description": "This is a new repository",

"private": false

}

* **Get Repository Information**:

GET https://api.github.com/repos/{owner}/{repo}

**4. Managing Commits:**

* **List Commits**:

GET https://api.github.com/repos/{owner}/{repo}/commits

* **Get Commit Details**:

GET https://api.github.com/repos/{owner}/{repo}/commits/{sha}

**Example: Automating Issue Creation via API**

Here’s how to automate issue creation with GitHub API using Python:

import requests

# Authentication

headers = {

'Authorization': 'token YOUR\_PERSONAL\_ACCESS\_TOKEN'

}

# API endpoint to create an issue

url = 'https://api.github.com/repos/{owner}/{repo}/issues'

# Issue data

data = {

'title': 'Bug in login system',

'body': 'Authentication fails under certain conditions',

'labels': ['bug']

}

# Make POST request to create an issue

response = requests.post(url, headers=headers, json=data)

if response.status\_code == 201:

print("Issue created successfully!")

else:

print("Failed to create issue:", response.status\_code)

**2. Integrating GitHub with Third-Party Tools**

**Objective: Learn how to integrate GitHub with third-party tools like CI/CD platforms, project management software, and other integrations to streamline development workflows.**

Integrating GitHub with third-party tools can automate repetitive tasks, enhance collaboration, and improve productivity.

**Common Integrations:**

**1. GitHub with Jenkins (CI/CD):**

Jenkins is a popular CI/CD tool that can be integrated with GitHub for automated builds and deployments.

* **Install GitHub Plugin for Jenkins**: Jenkins provides a GitHub plugin that allows you to easily link your Jenkins jobs with GitHub repositories.
* **GitHub Webhooks**: Set up a webhook in GitHub that triggers Jenkins jobs when certain events occur (e.g., new commits, pull requests).

**2. GitHub with Slack:**

Slack notifications can be integrated with GitHub to notify teams about code changes, pull requests, and more.

* **GitHub for Slack Integration**: Install the GitHub app in Slack. This integration allows GitHub to send notifications to specific channels for events like new pull requests, issues, and commits.

**3. GitHub with Jira:**

GitHub and Jira can be integrated to automatically link commits, pull requests, and branches to Jira issues.

* **Jira GitHub Integration**: You can link your GitHub repository with Jira using the GitHub for Jira app, enabling you to automatically update issue statuses based on commit and pull request activity.

**4. GitHub with Trello:**

For teams using Trello to manage tasks, GitHub can be integrated to link commits and pull requests to Trello cards.

* **Zapier Integration**: Use Zapier to automate the integration between GitHub and Trello. For example, a new GitHub pull request can automatically create or update a Trello card.

**5. GitHub with CircleCI:**

CircleCI is another popular CI/CD tool that can be integrated with GitHub for automating testing, building, and deployment processes.

* **CircleCI GitHub Integration**: Link your CircleCI account with GitHub to automatically trigger builds based on GitHub events like pushes and pull requests.

**6. GitHub with GitLab:**

Sometimes, teams use GitLab for CI/CD and GitHub for code hosting. You can link the two platforms together for continuous integration.

* **GitLab CI Integration**: Use GitLab CI to monitor your GitHub repositories and trigger pipeline executions based on GitHub events.

**Example of GitHub API Integration with a Third-Party Tool (Jira):**

You can use the GitHub API in combination with Jira’s API to automatically create issues or update statuses in Jira when new commits or pull requests are made in GitHub.

Here’s an example workflow:

1. **GitHub Commit**: Developer commits changes and links the commit to a Jira issue (e.g., JIRA-123).
2. **Webhook**: A webhook triggers an event in Jira, updating the issue status to “In Progress”.
3. **Automated Workflow**: Once the pull request is merged, another webhook is triggered to update the Jira issue status to “Done”.

**Conclusion**

GitHub's API and its integrations with third-party tools can greatly improve your team's productivity, streamline your workflows, and automate repetitive tasks. Whether you’re automating issue creation, integrating with CI/CD platforms, or linking your repositories with project management tools like Jira, the GitHub API offers a versatile and scalable solution for building powerful automation workflows.

**Security & Compliance**

* **GPG signing** for commits
* **Managing API Tokens**
* Handling **sensitive data** (.gitignore, GitHub secrets)
* **Audit & compliance practices**

**Security & Compliance in GitHub**

Security and compliance are crucial components in any development workflow, especially when working with open-source repositories or enterprise-level projects. In this section, we'll discuss how to enhance the security and compliance of your Git repository, including signing commits with GPG, managing API tokens securely, handling sensitive data, and establishing audit & compliance practices.

**1. GPG Signing for Commits**

**Objective**: Sign your commits and tags with GPG to ensure their authenticity and integrity.

**What is GPG Signing?**

GPG (GNU Privacy Guard) signing allows you to cryptographically sign your commits and tags in Git, ensuring that the commits originate from a trusted source. This helps prevent malicious changes or unauthorized users from tampering with your repository’s history.

**How to Set Up GPG Signing for Commits:**

1. **Install GPG**: If you don't have GPG installed, you can install it by following the instructions for your operating system:
   * GPG Installation Guide
2. **Generate a GPG Key**: Generate a GPG key to sign your commits:

gpg --full-generate-key

Follow the instructions to create a key. Choose a key type (RSA and RSA), key size (2048 or 4096 bits), and expiration date.

1. **Get Your GPG Key ID**: After generating your key, you can list your keys and find your key ID:

gpg --list-secret-keys --keyid-format LONG

1. **Add Your GPG Key to Git**: Configure Git to use your GPG key by running:

git config --global user.signingkey YOUR\_KEY\_ID

1. **Enable Commit Signing**: To ensure that all commits are signed by default, configure Git:

git config --global commit.gpgSign true

1. **Sign Your Commits**: When committing changes, Git will sign the commit automatically:

git commit -m "Your commit message"

1. **Verify Commit Signing**: You can verify that a commit is signed by running:

git log --show-signature

**Integrating GPG with GitHub:**

To link your GPG key with GitHub:

1. **Export the Public Key**:

gpg --armor --export YOUR\_KEY\_ID

1. **Add the Public Key to GitHub**:
   * Go to your GitHub settings → **SSH and GPG keys** → **New GPG key**.
   * Paste your public key and save.

**Benefits:**

* **Authenticity**: Anyone can verify that the commit was made by the authorized user.
* **Integrity**: Ensures the commit has not been tampered with since it was made.

**2. Managing API Tokens**

**Objective**: Securely manage and store API tokens to prevent unauthorized access to your repositories and services.

API tokens are used for authentication to services like GitHub and other external integrations. These tokens should be kept secure and never exposed in public repositories.

**Best Practices for Managing API Tokens:**

1. **Avoid Hardcoding Tokens in Code**: Never hardcode API tokens directly in your source code. Instead, use environment variables or configuration management tools to store them securely.

Example (using .env file for local storage):

API\_TOKEN=your\_api\_token\_here

1. **Use GitHub Secrets**: For GitHub Actions and workflows, use **GitHub Secrets** to securely store sensitive data such as API tokens.
   * Go to **Settings > Secrets** in your GitHub repository.
   * Add your tokens under **New Repository Secret**.

Example in a GitHub Actions workflow:

jobs:

build:

runs-on: ubuntu-latest

steps:

- name: Checkout repository

uses: actions/checkout@v2

- name: Use API Token

run: echo "Using API token ${{ secrets.API\_TOKEN }}"

1. **Token Expiry & Rotation**: Set expiration dates on API tokens whenever possible, and rotate them regularly to reduce the risk of compromise.
2. **Scope the Token Permissions**: Restrict the scope of the API token to only the required permissions. For example, if a token is only used for reading data, limit its access to read-only permissions.

**3. Handling Sensitive Data**

**Objective**: Properly handle sensitive data in your repositories to avoid accidental exposure.

Sensitive data, such as passwords, API keys, and credentials, should never be committed to a Git repository, especially when it's hosted publicly.

**Best Practices for Handling Sensitive Data:**

1. **Use .gitignore to Prevent Sensitive Files from Being Tracked**: The .gitignore file tells Git which files or directories to ignore, ensuring that sensitive files like API keys or credentials are not committed.

Example .gitignore entry:

# Ignore .env files

.env

1. **GitHub Secrets**: As mentioned earlier, for CI/CD workflows and automation scripts, store sensitive data such as tokens or passwords in **GitHub Secrets** rather than committing them to the repository.
2. **Tools to Detect Sensitive Data**: Use tools like **GitGuardian** or **TruffleHog** to scan your Git history and repositories for sensitive data.
3. **Remove Sensitive Data from Git History**: If sensitive data was accidentally committed, you can remove it from the Git history using the git filter-branch or BFG Repo-Cleaner tool.

Example (using BFG Repo-Cleaner):

bfg --delete-files '.env' my-repo.git

**4. Audit & Compliance Practices**

**Objective**: Implement auditing and compliance practices to track and verify actions in your Git repository.

Audit and compliance practices are essential in ensuring the integrity of your project and tracking potential issues or unauthorized activities.

**Audit Practices:**

1. **Commit Signing**: Ensure that all commits are signed using GPG (as discussed earlier) to track who made each change and ensure the authenticity of the changes.
2. **Branch Protection Rules**: GitHub allows you to configure **branch protection rules** to enforce certain checks before changes can be merged into important branches (e.g., main or master).
   * Require status checks to pass (e.g., passing tests).
   * Require pull request reviews before merging.
   * Enforce commit signing.
3. **Monitoring Repository Access**: Regularly review repository access logs, especially if you are using GitHub Enterprise, to track who has access to the repository and what changes are being made.
4. **Security Audits**: Conduct periodic security audits of your GitHub repositories to ensure that they comply with security best practices and that sensitive data is not being exposed.

GitHub provides a **Security Advisory Database** to help identify known vulnerabilities in dependencies.

1. **Compliance Audits**: For teams that must adhere to specific compliance standards (e.g., GDPR, SOC 2, HIPAA), make sure to audit your GitHub workflows and practices regularly to ensure compliance. This may involve reviewing:
   * Access controls
   * Data handling procedures
   * Logging and monitoring systems
2. **Automating Compliance Checks**: Implement automated compliance checks using GitHub Actions, such as enforcing certain coding standards, checking for vulnerable dependencies, or verifying that all pull requests meet security criteria.

**Conclusion**

In today’s world, **security and compliance** are essential components of any development process. By signing your commits with GPG, properly managing your API tokens, handling sensitive data with care, and establishing audit and compliance practices, you can ensure that your projects are secure and trustworthy.

**Performance Optimization & Customization**

* **Overview of Shallow clones & sparse-checkouts** for large repositories
* **Customizing Git with aliases & hooks**
* **Optimizing Git workflows for team productivity**

**Performance Optimization & Customization in Git**

Optimizing Git performance and customizing Git to suit your needs are essential for improving productivity, especially when dealing with large repositories and complex workflows. This section covers strategies for optimizing Git for performance, such as using shallow clones and sparse-checkouts, as well as customizing Git with aliases and hooks to enhance your team's productivity.

**1. Overview of Shallow Clones & Sparse-Checkouts for Large Repositories**

**What are Shallow Clones?**

A **shallow clone** is a Git clone with a limited history, meaning you only clone the most recent commits rather than the entire repository history. This can drastically improve performance, especially when working with very large repositories.

**How to Perform a Shallow Clone:**

You can create a shallow clone by using the --depth option with the git clone command:

git clone --depth 1 <repository-url>

This command clones only the latest commit from the repository. The depth value can be increased if you need a deeper history.

* **Pros**:
  + Faster clone time.
  + Reduced disk usage as fewer commits are fetched.
  + Ideal for CI/CD pipelines where only the latest state of the repository is required.
* **Cons**:
  + Limited access to the full commit history (e.g., you cannot see older commits, and operations like git log will be limited).
  + Cannot push changes from a shallow clone unless you perform a full fetch.

**What are Sparse-Checkouts?**

**Sparse-checkout** allows you to selectively check out a portion of the repository, making it ideal for large repositories with many files where only a subset of files is needed.

**How to Set Up Sparse-Checkout:**

1. **Enable Sparse-Checkout**: First, you need to enable the sparse-checkout mode:

git config core.sparseCheckout true

1. **Set the Files or Directories to Checkout**: Define which files or directories you want to check out in the .git/info/sparse-checkout file. For example, if you only want to check out a specific folder, add that folder path to the file:

/path/to/your/folder/

1. **Checkout the Repository**: After configuring sparse-checkout, you can perform the checkout:

git checkout <branch-name>

* **Pros**:
  + Saves bandwidth and disk space by only checking out the files you need.
  + Increases performance when working with large repositories.
* **Cons**:
  + Complex setup, especially for users unfamiliar with sparse-checkouts.
  + Some operations may not work as expected (e.g., searching across unchecked files).

**2. Customizing Git with Aliases & Hooks**

**Git Aliases**

Git aliases allow you to create custom shortcuts for frequently used Git commands. This can save time and reduce the need for repetitive commands.

**How to Set Up Git Aliases:**

1. **Creating Aliases**: You can define an alias using the git config command. For example, you can create an alias for git status as git st:

git config --global alias.st status

1. **Listing Aliases**: To view your defined aliases:

git config --global --get-regexp alias

1. **Example Aliases**:
   * Shorten common commands:

git config --global alias.co checkout

git config --global alias.br branch

git config --global alias.ci commit

1. **Complex Aliases**: Aliases can also be complex, such as combining commands:

git config --global alias.last 'log -n 1 HEAD'

This creates an alias git last that shows the latest commit.

**Git Hooks**

Git hooks are custom scripts that are triggered by Git events (e.g., commit, push). They help automate tasks such as linting, testing, or verifying commit messages before allowing a commit or push.

**Common Git Hooks:**

* **Pre-commit Hook**: Runs before a commit is made, often used to lint code or run tests.
* **Pre-push Hook**: Runs before changes are pushed to a remote, useful for running checks or preventing invalid pushes.
* **Post-merge Hook**: Runs after a merge, useful for handling additional tasks like running tests or updating dependencies.

**Setting Up Git Hooks:**

1. **Creating a Hook**: Git hooks are stored in the .git/hooks directory of your repository. For example, to create a pre-commit hook, you would:

cd .git/hooks

touch pre-commit

chmod +x pre-commit

1. **Adding Logic to the Hook**: You can add custom logic to the hook. For example, a pre-commit hook to run eslint on your code could look like this:

#!/bin/sh

eslint . --ext .js,.jsx

1. **Using Husky for Git Hooks**: For more advanced management and sharing of hooks across a team, you can use **Husky**. Husky simplifies setting up Git hooks and ensures they are consistent across all developers.

Install Husky:

npm install husky --save-dev

Set up a pre-commit hook with Husky:

npx husky add .husky/pre-commit "npm test"

**3. Optimizing Git Workflows for Team Productivity**

**Optimizing Workflows:**

1. **Branching Strategies**:
   * **Git Flow**: A structured branching model that uses specific branches for feature development, releases, and hotfixes. This is ideal for teams with a defined release cycle.
   * **Trunk-Based Development**: A simpler branching strategy where developers work on short-lived branches and merge back into the main or trunk branch frequently. This encourages rapid integration and is suitable for continuous delivery teams.
2. **Code Reviews**: Implementing code review practices with tools like **GitHub Pull Requests** or **GitLab Merge Requests** helps ensure code quality and collaboration. It's essential to establish clear guidelines for how reviews should be conducted and to automate processes such as linting and testing using GitHub Actions or CI/CD pipelines.
3. **Automated CI/CD Pipelines**: Integrating automated **Continuous Integration (CI)** and **Continuous Deployment (CD)** workflows into your Git repositories allows for faster, more reliable delivery of code. Tools like **GitHub Actions**, **CircleCI**, and **Travis CI** help streamline the testing, build, and deployment processes.
4. **Branch Protection Rules**: GitHub and GitLab provide branch protection rules that enforce certain policies, such as requiring successful builds before merging, ensuring that all pull requests pass code reviews, or requiring signed commits.
5. **Git Submodules**: If you're working with a repository that contains other repositories, using **Git submodules** can help keep these dependencies in sync. However, submodules should be used carefully, as they can add complexity to your workflow.

**Conclusion**

Optimizing performance and customizing your Git workflows can significantly enhance team productivity, especially when working with large repositories. By leveraging shallow clones, sparse-checkouts, Git aliases, and hooks, you can improve your development environment. Additionally, establishing effective branching strategies and CI/CD pipelines ensures that your team works efficiently and delivers high-quality software.

**Best Practices & Final Q&A**

* **Commit message conventions**
* **Code review best practices**
* **Scaling Git for large teams & projects**
* **Real-world case studies**
* Open Q&A session

**Best Practices & Final Q&A in Git**

This session wraps up by reviewing the best practices for using Git in a professional and scalable manner. It also opens the floor for final questions and clarifications. We'll cover commit message conventions, code review best practices, scaling Git for large teams, and real-world case studies.

**1. Commit Message Conventions**

**Why Commit Message Conventions Matter**

Clear and consistent commit messages are crucial for understanding the purpose of each change, especially when working with large teams or long-term projects. They help improve collaboration, track issues, and maintain a readable project history.

**Recommended Commit Message Format:**

A well-structured commit message typically follows this format:

1. **Title (Short, Descriptive)**: A brief summary of what was changed.
2. **Body (Optional)**: A detailed explanation of the change and why it was made, especially for complex or large changes.
3. **Footer (Optional)**: References to issues, tickets, or pull requests (e.g., Fixes #123).

**Example:**

feat(user-auth): Add JWT authentication for login

- Implemented token-based authentication using JSON Web Tokens (JWT)

- Added middleware to verify tokens

- Updated login route to issue tokens after successful authentication

Fixes #452

**General Guidelines:**

* **Imperative Mood**: Use the present tense and imperative mood. For example, "Add" instead of "Added" or "Adding."
* **Keep it Concise**: The title should be a single line, 50-72 characters long. The body can be more detailed if necessary.
* **Use Labels**: Prefix commits with tags like feat, fix, chore, docs, style, etc., to clarify the type of change.

**Popular Commit Message Conventions:**

* **Conventional Commits**: A standardized format for commit messages that makes it easier to automate versioning and changelog generation.
* **Gitmoji**: A fun approach to commit messages, using emojis to represent types of changes (e.g., ✨ for features, 🐛 for bugs).

**2. Code Review Best Practices**

**Why Code Reviews Matter**

Code reviews are essential for maintaining code quality, promoting knowledge sharing, and identifying bugs or inefficiencies early in the development process.

**Key Code Review Practices:**

1. **Keep Reviews Small**: Review small, manageable chunks of code at a time. Large reviews can overwhelm the reviewer and increase the chances of missing issues.
2. **Be Respectful**: Provide constructive feedback and focus on the code, not the person. Be specific about what can be improved.
3. **Automate What You Can**: Use CI tools (e.g., GitHub Actions, CircleCI) to automate tasks like linting, testing, and formatting. This reduces the burden on reviewers.
4. **Focus on Functionality and Design**: Ensure the code works as expected and adheres to project guidelines. Review the design for maintainability and scalability.
5. **Review for Security**: Always check for potential security vulnerabilities (e.g., unescaped user inputs, hardcoded secrets).
6. **Follow Up**: After providing feedback, check if the suggested changes have been applied and that new issues haven’t been introduced.

**Good Code Review Process:**

* Review pull requests (PRs) frequently and ensure they are small.
* Make use of inline comments for specific lines of code that require attention.
* Test the changes locally to confirm the effectiveness of the changes.

**3. Scaling Git for Large Teams & Projects**

**Challenges in Large-Scale Git Repositories:**

* **Performance Issues**: Large repositories can be slow to clone, fetch, or merge. Techniques like shallow cloning, sparse-checkouts, and Git LFS (Large File Storage) can help mitigate these problems.
* **Complex Branching**: With a large team, managing branches can become complex. A clear branching strategy (like Git Flow or Trunk-Based Development) can help manage this complexity.
* **Conflict Management**: With many contributors, conflicts can arise frequently. Tools like **git rerere** (reuse recorded resolution) can help automate conflict resolution, while frequent rebasing helps avoid long-lived conflicts.

**Best Practices for Large Teams:**

* **Git Flow or Trunk-Based Development**: Choose a branching model that suits your team's needs. Git Flow is ideal for structured releases, while trunk-based development encourages frequent integration.
* **Clear Commit Guidelines**: Adopt consistent commit message conventions and enforce them via Git hooks or CI checks.
* **Automation**: Use CI/CD pipelines for automated testing, linting, and deployment to avoid bottlenecks in the review and merge process.
* **Branch Protection Rules**: On platforms like GitHub or GitLab, use branch protection rules to prevent direct commits to main/master, require pull request reviews, and enforce successful build checks before merging.

**Git Submodules & Monorepos:**

* For larger teams, you might want to use **Git submodules** or even a **monorepo** approach, where multiple projects or components live in the same repository. However, both approaches come with their own set of challenges regarding management, versioning, and collaboration.

**4. Real-World Case Studies**

**Case Study 1: Open Source Project Scaling**

An open-source project started with a simple Git repository. As contributors increased, it became difficult to manage the volume of contributions. The team implemented Git Flow and established a strict code review process, improving collaboration. They also moved to a monorepo approach for related projects, allowing for easier dependency management and faster integrations.

**Case Study 2: Large Enterprise Repo**

In a large enterprise environment, the repository grew so large that performance became an issue. The team introduced **shallow clones** for developers working on specific features, and **Git LFS** for storing large binaries. They also streamlined their branching strategy to use feature branches for new features and hotfixes for urgent issues. A dedicated CI/CD pipeline for automated testing and deployment was also set up to improve the workflow.

**Case Study 3: Cross-Repository Integration**

A team working with multiple repositories in a microservices architecture used Git submodules to maintain dependencies between services. However, they encountered issues with syncing submodules, especially when developers forgot to update submodule references. To address this, they wrote custom scripts to automate submodule updates and integrated them into their CI pipeline.

**5. Open Q&A Session**

This is the time to ask any lingering questions or dive deeper into specific topics discussed in this session, such as:

* **Best practices for dealing with merge conflicts**
* **Advanced Git strategies for managing large codebases**
* **Using Git with tools like Jira or Slack**
* **Specific questions on workflows, hooks, or performance tuning**