01 git - VCS DVCS

Source Control Management (SCM): Git and GitHub

Introduction to Source Control Management (SCM)

Source Control Management (SCM) is a system that manages changes to source code over time. It allows developers to track modifications, collaborate efficiently, and maintain a history of their work. SCM tools are essential for software development teams to ensure code integrity, collaboration, and versioning. SCM allows developers to:

- **Track Changes:** Record every modification made to the codebase, including who made the change, when, and why.
- **Collaborate Effectively:** Enable multiple developers to work on the same codebase simultaneously without stepping on each other's toes.
- **Revert to Previous Versions:** Easily roll back to earlier versions of the code if errors are introduced or if a feature needs to be undone.
- **Manage Releases:** Create and manage different versions of the software for releases and deployments.
- **Branch and Merge:** Develop new features or fix bugs in isolated branches and then merge them back into the main codebase.
- **Audit and Compliance:** Maintain a detailed history of changes for auditing and compliance purposes.
- `Version Control System (VCS):A general term encompassing all systems that manage versions of files.`

Evolution of Version Control Systems

1. **Local Version Control**
   - Early systems stored patch sets (file differences) on local machines
   - Limited collaboration capabilities
   - Examples: RCS (Revision Control System), Manual backups or simple scripts.
2. **Centralized Version Control Systems (CVCS)**
   - Single central repository on a server
   - Developers check out and commit to this central repository
   - Requires network access for most operations
   - Advantages:
   - Centralized repository ensures everyone works on the same codebase.

- Disadvantages:
  - Single point of failure (if the server goes down, no one can collaborate).
  - Limited offline capabilities.
  - Examples: SVN (Subversion), CVS (Concurrent Versions System)

3. **Distributed Version Control Systems (DVCS)**
   - Each developer has a complete copy of the repository including history
   - Can work offline and synchronize later
   - No single point of failure
   - Examples: Git, Mercurial, Bazaar

Git: The Distributed Version Control System

Git was created by Linus Torvalds in 2005 for Linux kernel development. It was designed to be fast, efficient, and support distributed, non-linear workflows.

Key Features of Git

- **Distributed architecture** - full repository copies on each developer's machine
- **Speed** - most operations are local, making them extremely fast

- **Data integrity** - SHA-1 checksums ensure content is uncorrupted
- **Non-linear development** - powerful branching and merging capabilities
- **Staging area** - intermediate area between working directory and repository

Git's Three States

1. **Modified** - Changes made in working directory but not committed
2. **Staged** - Modified files marked to go into the next commit
3. **Committed** - Data safely stored in the local repository

Git Storage Model

- Git stores data as a series of snapshots, not differences
- Each commit represents a complete snapshot of all tracked files at that point
- Files that haven't changed are stored as references to previous identical files

GitHub and Remote Repositories

GitHub is a web-based hosting service for Git repositories that adds collaboration features:

- **Remote repository hosting**
- **Pull requests** for code review
- **Issue tracking**
- **Wikis** for documentation
- **Actions** for CI/CD workflows
- **Project management tools**

Other similar platforms include GitLab, Bitbucket, and Azure DevOps.

Essential Git Commands

Setting Up Git

```
# Configure user information
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"

# Set default editor
git config --global core.editor "vim"

# List all configurations
git config --list
```

When to use: When setting up Git on a new machine or changing your identity information.

Creating Repositories

```
# Initialize a new Git repository
git init

# Clone an existing repository
git clone https://github.com/username/repository.git

# Clone to a specific folder
git clone https://github.com/username/repository.git my-folder
```

When to use:

- `git init` : When starting a new project from scratch
- `git clone` : When you want to work on an existing project

Basic Workflow Commands

```
# Check repository status
git status

# Add files to staging area
git add filename.txt         # Add specific file
git add directory/           # Add all files in directory
git add .                    # Add all files in current directory

# Commit staged changes
git commit -m "Commit message"
git commit -a -m "Commit message"  # Add and commit in one step

# View commit history
git log
git log --oneline            # Compact view
git log --graph              # Visual representation of branches
git log -p                   # Show patches (changes) with each commit
```

When to use:

- `git status` : Frequently, to see which files are modified/staged/untracked
- `git add` : After making changes you want to include in the next commit
- `git commit` : After staging changes you want to save to history
- `git log` : When you need to review commit history

Working with Remote Repositories

```
# List remote repositories
git remote -v

# Add remote repository
git remote add origin https://github.com/username/repository.git

# Fetch changes from remote
git fetch origin

# Pull changes from remote (fetch + merge)
git pull origin main

# Push local changes to remote
git push origin main

# Set tracking relationship
git branch --set-upstream-to=origin/main main
```

When to use:

- `git remote add` : When connecting local repo to a remote for the first time
- `git fetch` : To download changes without merging them
- `git pull` : When you want to update your local branch with remote changes
- `git push` : After making commits you want to share with others

Branching and Merging

```
# List branches
git branch                   # Local branches
git branch -r                # Remote branches
git branch -a                # All branches

# Create new branch
```

```
git branch feature-x

# Switch to a branch
git checkout feature-x
git checkout -b feature-y    # Create and switch in one command

# Git 2.23+ alternative to checkout for switching
git switch feature-x
git switch -c feature-y      # Create and switch

# Merge branch into current branch
git merge feature-x

# Delete branch
git branch -d feature-x      # Safe delete (only if merged)
git branch -D feature-x      # Force delete
```

When to use:

- Create branches when developing new features, fixing bugs, or experimenting
- `git checkout` or `git switch`: When changing between branches
- `git merge`: When integrating changes from one branch into another
- Branch deletion: After successful integration or abandoning work

Examining Changes

```
# View changes between working directory and staging
git diff

# View changes between staging and last commit
git diff --staged

# View changes between branches
git diff main..feature-x

# View changes in a specific file
git diff -- path/to/file.txt
```

When to use: Before committing, to review exactly what will be included.

Undoing Changes

```
# Discard changes in working directory
git restore file.txt
git checkout -- file.txt     # Pre-Git 2.23 syntax

# Unstage files
git restore --staged file.txt
git reset HEAD file.txt      # Pre-Git 2.23 syntax

# Amend last commit
git commit --amend -m "New commit message"

# Revert a commit (creates new commit that undoes changes)
git revert commit-hash

# Reset to a previous state (CAUTION: rewrites history)
git reset --soft commit-hash   # Keep changes staged
git reset commit-hash          # Keep changes unstaged
git reset --hard commit-hash   # Discard changes
```

When to use:

- `git restore` / `git checkout` : When you want to discard local changes
- `git restore --staged` / `git reset HEAD` : When you staged files by mistake
- `git commit --amend` : For small fixes to the most recent commit
- `git revert` : To undo a commit while preserving history
- `git reset` : When you need to move back to a previous state (careful with shared branches)

Stashing Changes

```
# Save changes temporarily
git stash

# Save with description
git stash save "Work in progress for feature X"

# List stashes
git stash list

# Apply most recent stash without removing it
git stash apply

# Apply specific stash
git stash apply stash@{2}

# Apply and remove most recent stash
git stash pop

# Remove a stash
git stash drop stash@{1}

# Clear all stashes
git stash clear
```

When to use: When you need to switch contexts but aren't ready to commit current changes.

Advanced Commands

```
# Interactive rebase for rewriting history
git rebase -i HEAD~3

# Cherry-pick specific commits
git cherry-pick commit-hash

# Squash last 3 commits interactively
git rebase -i HEAD~3

# Find which commit introduced a bug
git bisect start
git bisect bad                # Current version has the bug
git bisect good commit-hash   # Known good version
# Git will checkout commits, and you mark them good/bad until found

# Show who modified each line
git blame file.txt
```

When to use:

- `git rebase` : When you need to reorganize commit history
- `git cherry-pick` : To apply specific commits from one branch to another

- `git bisect` : To find which commit introduced a bug
- `git blame` : To determine who changed particular lines and when

Git Tags

```
# List tags
git tag

# Create lightweight tag
git tag v1.0.0

# Create annotated tag
git tag -a v1.0.0 -m "Version 1.0.0"

# Push tags to remote
git push origin v1.0.0
git push origin --tags       # Push all tags

# Delete tag
git tag -d v1.0.0
```

When to use: To mark specific points in history, typically for releases.

Git Submodules

```
# Add submodule
git submodule add https://github.com/username/repository.git path/to/submodule

# Initialize and update submodules (after cloning a repo with submodules)
git submodule init
git submodule update

# Clone repository with all submodules
git clone --recurse-submodules https://github.com/username/repository.git
```

When to use: When your project depends on another project that you want to keep as a separate repository.

Common Git Workflows

Feature Branch Workflow

1. Create a branch for a feature: `git checkout -b feature-x`
2. Make changes and commit: `git add .` and `git commit`
3. Push to remote: `git push -u origin feature-x`
4. Create pull request on GitHub
5. Review, address feedback, merge to main branch

Gitflow Workflow

A more structured approach with dedicated branches:

- `main` - production code
- `develop` - development code
- `feature/*` - for new features
- `release/*` - preparing for a release
- `hotfix/*` - for urgent production fixes

GitHub Flow

A simpler workflow:

1. Branch from `main`
2. Add commits
3. Open a pull request
4. Discuss and review
5. Deploy and test
6. Merge to `main`

Best Practices for Git

1. **Write clear commit messages**
   - Start with a short, descriptive summary (50 chars or less)
   - Followed by a blank line and a more detailed explanation
2. **Commit early, commit often**
   - Make small, focused commits rather than large ones
   - Each commit should represent a single logical change
3. **Use branches effectively**
   - Create branches for features, bugs, experiments
   - Keep `main` branch stable and deployable
4. **Pull before pushing**
   - Always synchronize with remote before pushing to avoid conflicts
5. **Perform code reviews via pull requests**
   - Use GitHub's pull request features for team feedback
   - Address comments and improve code quality
6. **Use .gitignore**
   - Exclude build artifacts, dependencies, and local configuration
   - Only track source files and important configuration
7. **Protect sensitive data**
   - Never commit credentials, API keys, or secrets
   - Use environment variables or secure storage

Troubleshooting Common Git Issues

Resolving Merge Conflicts

```
# After a failed merge, fix conflicts in files
# Files will have conflict markers: <<<<<<< HEAD, =======, >>>>>>>
# Edit files to resolve, then:
git add resolved-file.txt
git commit
```

Recovering Lost Commits

```
# Find "dangling" commits
git reflog

# Recover by creating a branch at that commit
git branch recovery-branch commit-hash
```

Fixing Detached HEAD State

```
# Create a branch at current position
git branch temp-branch
```

```
git checkout temp-branch
```

Cleaning Untracked Files

```
# Preview what will be removed
git clean -n

# Remove untracked files
git clean -f

# Remove untracked files and directories
git clean -fd
```

Advanced Git Concepts

Git Hooks

Scripts that run automatically on specific Git events:

- `pre-commit` : Run before a commit is created
- `post-commit` : Run after a commit is created
- `pre-push` : Run before pushing to a remote
- `post-merge` : Run after merging

Git Attributes

Configure how Git handles specific files:

- Line ending normalization
- Diff strategies for binary files
- Merge strategies

Git LFS (Large File Storage)

Extension for handling large files:

```
# Install Git LFS
git lfs install

# Track large file types
git lfs track "*.psd"

# Use Git normally
git add file.psd
git commit -m "Add design file"
```

Conclusion

Git is a powerful tool that supports many different workflows and development styles. Mastering Git commands and understanding their appropriate use cases allows developers to work more efficiently, collaborate effectively, and maintain a clean project history.

Learning Git is a continuous process - start with the basic commands, gradually incorporate more advanced techniques...