Version Control and Git Workflows

[DT-0540] Metodi di sviluppo agile

Daniele Di Pompeo

daniele.dipompeo@univaq.it

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

A.A. 2024-2025

Introduction to Version Control

Definition:

- Version control is a system that records changes to files over time, allowing you to recall specific versions later.

Purpose:

- Helps manage changes in code, documents, and large projects.
- Enables multiple people to work on a project without conflicting changes.

Examples:

- Git, SVN, Mercurial

Architecture and Versioning Model

- Git:
  - Distributed Version Control System (DVCS) - each developer has a complete local copy of the repository.
  - Uses snapshots for versioning, storing the entire state of files at each commit.
- SVN (Subversion):
  - Centralized Version Control System (CVCS) - a central repository tracks changes, and developers work with server-synced copies.
  - Tracks version history by differences (deltas) rather than snapshots.
- Mercurial:
  - Also a Distributed Version Control System (DVCS), similar to Git, with each developer having a full copy.
  - Uses snapshots like Git, but focuses on simplicity and usability.

Branching and Merging

- Git:
  - Supports lightweight and flexible branching, allowing frequent branching and merging.
  - Branches are easy to manage, with merge conflict resolution tools built-in.
- SVN (Subversion):
  - Branching is possible but heavier and more complex due to centralized architecture.
  - Merging can be cumbersome and often requires manual conflict resolution.
- Mercurial:
  - Supports branching similar to Git but uses “named branches” to organize branches.
  - Merging is straightforward but can be less flexible than Git’s branching model.

Speed, Usability, and Community

- Git:
  - Very fast, especially for local operations, since each developer has a full repository.
  - Steep learning curve due to complex command structure but widely adopted with strong community support.
- SVN (Subversion):
  - Slower for large projects due to reliance on a central server.
  - Easier to learn and use for teams transitioning from centralized systems.
  - Popular in legacy and enterprise applications but declining in popularity.
- Mercurial:
  - Fast and efficient for local operations, similar to Git.
  - Known for a simpler command structure and ease of use, making it more beginner-friendly.
  - Smaller community compared to Git but strong support for Windows environments.

Benefits of Version Control

- **Collaboration:** Multiple contributors can work on the same codebase simultaneously.
- **History Tracking:** All changes are tracked with timestamps, author information, and descriptions.
- **Backup and Recovery:** Restores previous versions if issues arise in newer versions.
- **Branching and Merging:** Allows developers to work on features independently and merge when ready.

What is Git?

Definition:

- Git is a distributed version control system created by Linus Torvalds in 2005.

Features:

- Tracks changes and enables multiple contributors to collaborate on projects.
- Allows for offline work, as each contributor has a full copy of the repository.

Why Git?

- It's fast, secure, and has become the standard for version control in software development.

Git commands

Setting Up Git

- `git --version` - Check the installed Git version.
- `git config --global user.name "Your Name"` - Set the global username.
- `git config --global user.email "your.email@example.com"` - Set the global email.
- `git config --global core.editor "editor"` - Set the default text editor for Git.
- `git config --list` - Display all configured settings.

Purpose: Configure Git to use your name, email, and editor settings for all repositories.

Initializing and Cloning Repositories

- `git init` - Initialize a new Git repository in the current directory.
- `git clone <URL>` - Clone an existing remote repository.
- `git clone <URL> <folder>` - Clone a repository into a specific folder.

Purpose: Create new repositories or obtain a local copy of a remote repository to start working on it.

Basic Git Commands

- `git status` - Display the current status of the working directory.
- `git add <file>` - Stage changes for commit.
- `git add .` - Stage all modified files for commit.
- `git commit -m "message"` - Commit staged changes with a message.
- `git commit -am "message"` - Add and commit all modified files in one step.

Purpose: Track changes, stage updates, and create commits with descriptive messages.

Viewing History

- `git log` - View the commit history.
- `git log --oneline` - View a compact log of commit messages.
- `git log --graph` - Display a graphical representation of branch history.
- `git show <commit>` - Show details of a specific commit.
- `git diff` - Show differences between working directory and staged files.
- `git diff <commit1> <commit2>` - Compare two commits.

Purpose: Review project history and examine changes between commits.

Branching

- `git branch` - List all branches in the repository.
- `git branch <branch_name>` - Create a new branch.
- `git checkout <branch_name>` - Switch to an existing branch.
- `git checkout -b <branch_name>` - Create and switch to a new branch.
- `git branch -d <branch_name>` - Delete a branch locally.

Purpose: Use branches to develop features in isolation before merging them into the main codebase.

Merging and Rebasing

Merging:

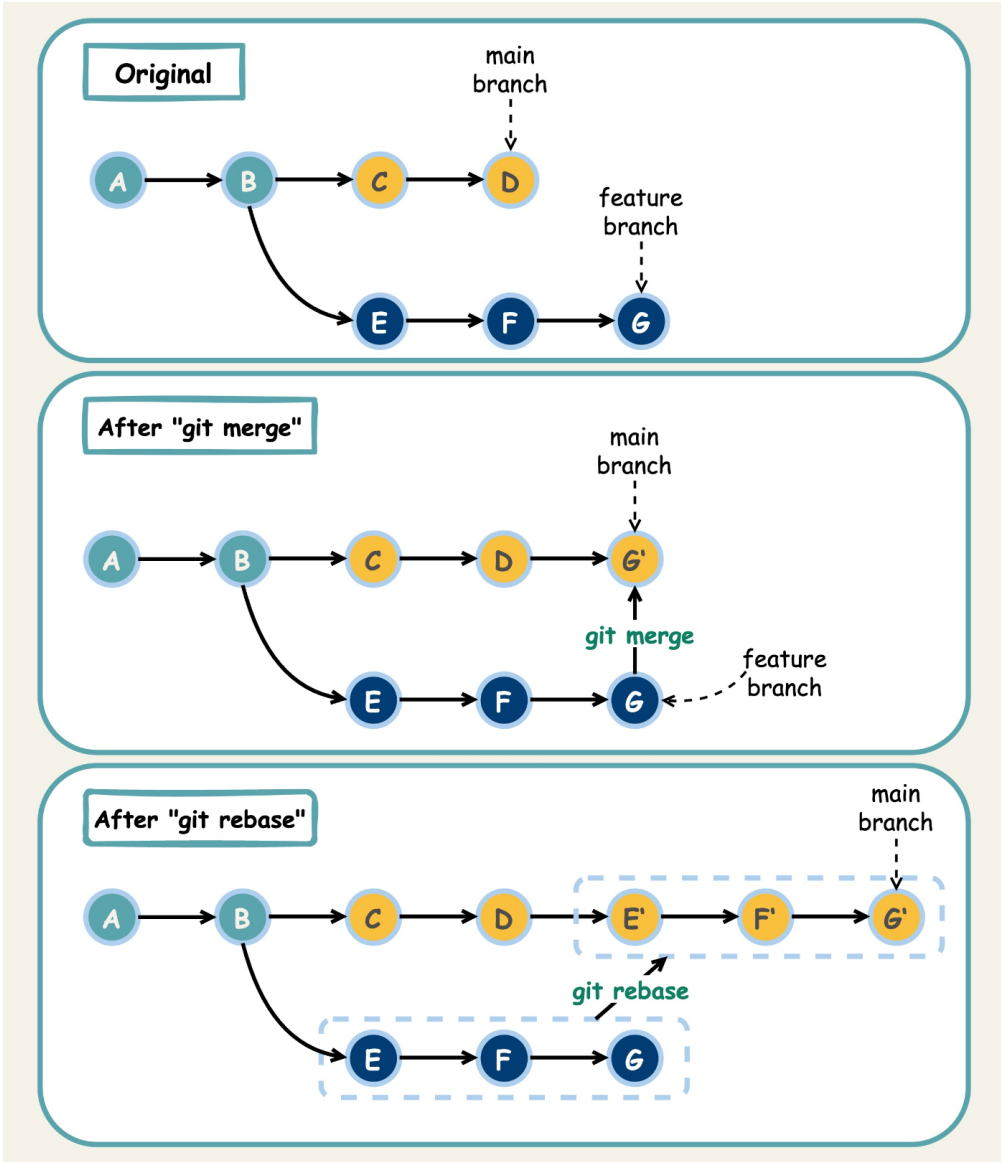
- `git merge <branch_name>` - Merge a branch into the current branch.
- `git merge --no-ff <branch_name>` - Create a merge commit even if the merge is a fast-forward.

Rebasing:

- `git rebase <branch_name>` - Apply changes from one branch onto another.
- `git rebase -i <commit>` - Start an interactive rebase to edit, squash, or reorder commits.

Purpose: Incorporate changes from one branch into another, either by merging them or rebasing for a cleaner history.

Merging and Rebasing



Working with Remote Repositories

- `git remote add origin <URL>` - Link a local repository to a remote repository.
- `git push origin <branch_name>` - Push local changes to a remote branch.
- `git fetch` - Retrieve updates from the remote repository without merging.
- `git pull` - Fetch and merge changes from the remote repository.
- `git remote -v` - List remote connections for the repository.

Purpose: Collaborate with others by synchronizing changes between local and remote repositories.

Stashing Changes

- `git stash` - Save uncommitted changes for later.
- `git stash list` - View all stashes.
- `git stash apply` - Reapply the most recent stash.
- `git stash pop` - Reapply and remove the most recent stash.
- `git stash drop` - Delete a specific stash.

Purpose: Temporarily store changes you're not ready to commit, allowing you to switch branches or update code.

Undoing Changes

- `git checkout -- <file>` - Discard changes in the working directory.
- `git reset <file>` - Unstage a file that was added.
- `git reset --soft <commit>` - Move HEAD to an earlier commit, keeping changes staged.
- `git reset --hard <commit>` - Move HEAD to an earlier commit and discard all changes.
- `git revert <commit>` - Create a new commit that undoes a previous commit.

Purpose: Safely or forcefully revert to previous states in your repository if needed.

Advanced Git Commands

- `git cherry-pick <commit>` - Apply a specific commit from another branch.
- `git reflog` - View the history of HEAD changes, useful for recovering lost commits.
- `git bisect` - Use binary search to find the commit that introduced a bug.
- `git blame <file>` - Show commit information for each line in a file.
- `git tag <tag_name>` - Add a tag to mark a specific commit.
- `git archive` - Create an archive of files from a particular commit.

Purpose: Advanced commands for debugging, recovering lost work, tagging releases, and more.

Summary of Git Commands

- Git provides a powerful set of tools for version control, collaboration, and history management.
- Basic commands cover setup, staging, committing, branching, and merging.
- Advanced commands help with debugging, history tracking, and release management.

Remember: Practice these commands in a test repository to get comfortable with Git!

Git-Flow

What is Git Flow?

Definition:

- Git Flow is a branching strategy that helps manage large projects by organizing different types of branches for different stages of development.

Purpose:

- Provides structure and consistency for development, testing, and deployment workflows.
- Separates feature development, releases, hotfixes, and long-term maintenance.

Core Branches in Git Flow

Main Branches:

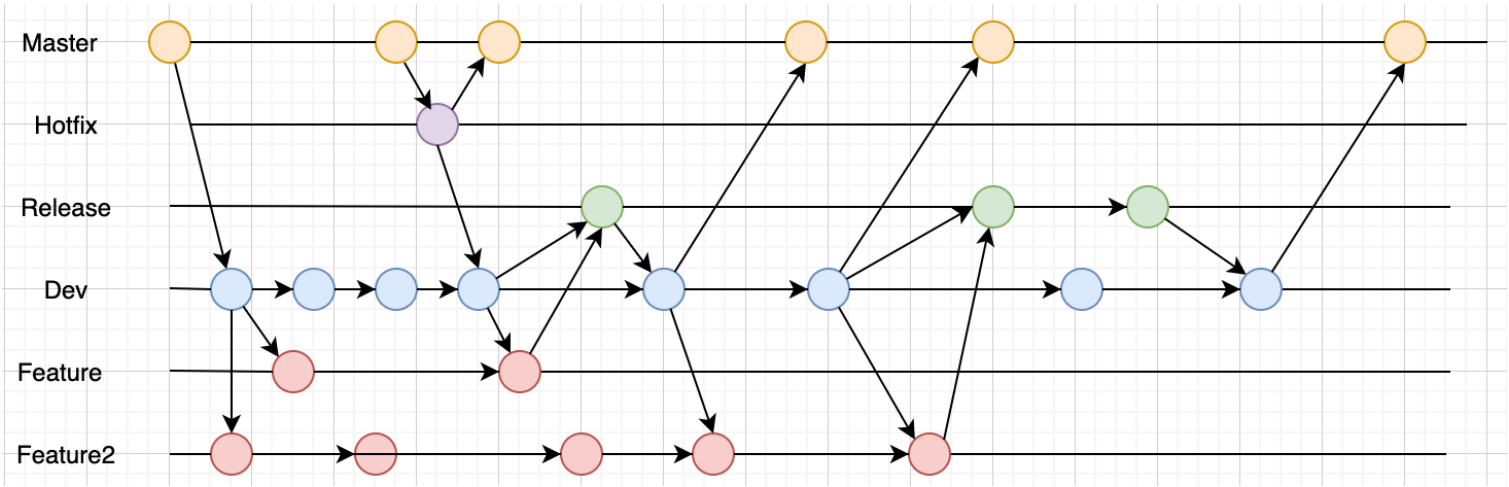
- **Master**: The production-ready code is always on this branch. Each commit on ‘master‘ is a new release.
- **Develop**: The main development branch where completed features are merged before release.

Supporting Branches:

- **Feature Branches**: Separate branches for developing new features, created off ‘develop‘.
- **Release Branches**: Used to finalize and test releases before they go to production.
- **Hotfix Branches**: Created from ‘master‘ to quickly address production issues.

Git-Flow

https://i.sstatic.net



Git Flow Workflow Overview

Steps:

1. Feature Development: Developers create feature branches for new features.
2. Merge to Develop: Features are merged into ‘develop‘ once complete.
3. Create Release: When ‘develop‘ is stable, create a release branch for testing.
4. Merge to Master: After testing, merge the release branch into ‘master‘ and ‘develop‘.
5. Hotfixes: Critical fixes are done on a hotfix branch created from ‘master‘.

Release Control:

- Each type of branch serves a specific role, helping to control the release process and avoid conflicts.

Feature Branches

Purpose:

- Used to develop individual features in isolation from other features.
- Created off the ‘develop‘ branch and named descriptively (e.g., ‘feature/login-page‘).

Example Commands:

- `git checkout develop` - Switch to ‘develop‘.
- `git checkout -b feature/login-page` - Create a new feature branch.
- `git push origin feature/login-page` - Push to the remote repository.

When Complete:

- Merge the feature branch back into ‘develop‘ with a pull request or ‘git merge‘.

Release Branches

Purpose:

- Created from ‘develop‘ when it is stable and ready for release.
- Used for testing and preparing the final release version.

Example Commands:

- `git checkout develop` - Switch to ‘develop‘.
- `git checkout -b release/1.0.0` - Create a release branch.
- `git push origin release/1.0.0` - Push for testing and review.

Final Steps:

- After testing, merge into both ‘master‘ and ‘develop‘, then tag the release.

Hotfix Branches

Purpose:

- Created from ‘master‘ to quickly fix critical issues in production.
- Hotfixes allow immediate production patches without disrupting the ‘develop‘ branch.

Example Commands:

- `git checkout master` - Switch to ‘master‘.
- `git checkout -b hotfix/urgent-bug-fix` - Create a hotfix branch.
- `git push origin hotfix/urgent-bug-fix` - Push for review and testing.

When Complete:

- Merge into both ‘master‘ and ‘develop‘, then tag with a version update if necessary.

Git Flow Commands (Using git-flow)

Installing git-flow:

- Many
  teams use the ‘git-flow‘ extension to automate the process. Install with:
  `brew install git-flow` (macOS) or `apt-get install git-flow` (Linux).

Common Commands:

- `git flow init` - Initializes the git-flow structure in the repository.
- `git flow feature start <name>` - Starts a new feature branch.
- `git flow release start <version>` - Creates a release branch.
- `git flow hotfix start <name>` - Creates a hotfix branch.

Benefits of Git Flow

- **Organized Workflow:** Each branch has a defined purpose, reducing merge conflicts and improving clarity.
- **Stable Releases:** Release branches allow comprehensive testing before production.
- **Parallel Development:** Developers can work on features independently without impacting the main codebase.
- **Quick Hotfixes:** Critical production issues can be patched immediately without delaying new features.

When to Use Git Flow

Best Suited For:

- Long-term Projects: Useful for projects with regular feature releases and maintenance.
- Team Environments: Ensures clear structure when multiple developers are contributing.
- Complex Release Management: Helpful when managing multiple environments (e.g., staging, production).

Consider Alternatives If:

- Continuous Delivery is needed, as Git Flow can be too structured for rapid deployment.
- Small Teams or Projects: May be unnecessarily complex for quick or lightweight projects.

Summary of Git Flow

- Git Flow is a branching strategy that organizes work into distinct branches: **feature**, **develop**, **release**, **master**, and **hotfix**.
- Provides a structured approach for teams to develop features, prepare releases, and quickly address production issues.
- Ideal for teams working on complex projects that require clear branching and release management.

Introduction to GitHub

Definition:

- GitHub is a web-based platform for version control and collaboration using Git.

Features:

- Allows developers to host and review code, manage projects, and build software.
- Provides issue tracking, pull requests, and code reviews.

Why GitHub?

- GitHub is widely used in the open-source community and provides a collaborative environment for developers.

Using Git within Scrum

Git Branching Strategies in Scrum

Feature Branching:

- Each user story or feature is developed in its own branch.
- Branch naming convention: `feature/<story-name>`.

Sprint Branching:

- Create a Sprint branch to group completed features for each Sprint.
- Useful for reviewing all changes associated with a specific Sprint.

Release Branching:

- Create a release branch at the end of the Sprint to prepare for production.
- Allows testing and bug fixing before merging into the main branch.

Git in Sprint Planning

Creating Feature Branches:

- At the start of a Sprint, the team creates feature branches for each user story in the Sprint Backlog.
- This allows each feature to be developed independently.

Estimating Workload:

- Git history (e.g., past commits, issue resolutions) helps in estimating the effort required for similar user stories.

Example Workflow:

- `git checkout -b feature/<story-name>` - Create a feature branch for each user story.

Git in Daily Scrum

Tracking Progress:

- Team members update the progress of their feature branches during Daily Scrum.
- Git status and commit history provide a snapshot of each feature's development.

Collaboration and Issue Resolution:

- Developers push code daily and review each other's pull requests.
- Git can reveal merge conflicts early, allowing them to be resolved quickly.

Example:

- `git push origin feature/<story-name>` - Push feature branch updates.
- `git pull origin develop` - Pull recent changes from the develop branch.

Git for Code Review and Quality Control

Pull Requests:

- Before merging feature branches into develop, developers create pull requests (PRs).
- PRs allow team members to review code and provide feedback.

Code Quality Checks:

- Automated tests can run on PRs to ensure code quality and functionality.
- Code review guidelines ensure consistent code quality across the team.

Example Workflow:

- `git push origin feature/<story-name>` - Push changes.
- Open a pull request and assign reviewers.

Git in Sprint Review

Demonstrating Completed Work:

- The team showcases completed features from the Sprint using merged branches.
- The release branch can be used for testing or staging environments.

Collecting Feedback:

- Based on stakeholder feedback, new issues or enhancements are created for future Sprints.
- Git commits and history help track features and issues discussed.

Example Commands:

- `git checkout develop` - Switch to the branch with completed features.
- `git merge feature/<story-name>` - Merge completed features before review.

Git in Sprint Retrospective

Reviewing Commit History:

- Analyze commit history to review coding practices, frequency of commits, and collaboration patterns.

Identifying Improvement Areas:

- Use Git stats to identify areas to improve, such as reducing conflicts or improving code review times.

Example:

- `git log --oneline --since="2 weeks ago"` - Review recent commit history for the Sprint.

Example Workflow in Scrum with Git

Step-by-Step Workflow:

1. During Sprint Planning, create feature branches for each user story.
2. During development, commit changes frequently and push updates.
3. Submit a pull request when a feature is complete, and request code review.
4. Merge approved features into develop before the Sprint Review.
5. After stakeholder feedback, finalize a release branch if required.

Command Examples:

- `git checkout -b feature/<story-name>`
- `git commit -m "Implemented user login feature"`
- `git push origin feature/<story-name>`

Best Practices for Using Git in Scrum

- **Frequent Commits:** Commit changes frequently to keep track of progress and make issues easier to resolve.
- **Use Descriptive Commit Messages:** Ensure commit messages explain what was done and why.
- **Branch Naming Conventions:** Use clear and consistent branch names (e.g., `feature/login-page`).
- **Review and Merge Regularly:** Use pull requests for code review and merge branches only after review.
- **Sync Often with Develop:** Regularly sync feature branches with the develop branch to avoid conflicts.

Summary of Git in Scrum

- Git supports Scrum practices by providing structure for branching, collaboration, and quality control.
- Feature branches enable independent work on user stories during Sprints.
- Pull requests and code reviews ensure code quality before merging into main branches.
- Using Git throughout Scrum ceremonies enhances visibility, collaboration, and accountability within the team.

Github and Gitlab

Working with GitHub

- **Repositories:** Store project code and history.
- **Issues:** Track bugs, feature requests, and tasks.
- **Pull Requests:** Propose changes to a repository.
- **Code Review:** Enables team members to review changes before merging.

Example Workflow:

- Create a branch, make changes, push to GitHub, open a pull request, review and merge.

Introduction to GitLab

Definition:

- GitLab is an open-source DevOps platform that provides Git-based version control along with CI/CD capabilities.

Features:

- Offers integrated continuous integration (CI) and continuous deployment (CD).
- Provides tools for issue tracking, code reviews, and project management.

Why GitLab?

- GitLab is popular for its end-to-end DevOps features, supporting the entire software development lifecycle.

Working with GitLab

- **Repositories:** Host code and track changes.
- **Issues and Boards:** Track tasks and use Kanban-style boards.
- **Merge Requests:** Propose and review changes to code.
- **CI/CD Pipelines:** Automate building, testing, and deployment.

Example Workflow:

- Create a branch, push changes, create a merge request, review, and deploy using GitLab CI/CD.

Comparing GitHub and GitLab

GitHub:

- Strong focus on open-source projects and community.
- Pull requests and extensive integrations.

GitLab:

- End-to-end DevOps platform with built-in CI/CD.
- Merge requests and comprehensive project management.

Conclusion:

- Both platforms are powerful tools for version control and collaboration, with GitLab offering more DevOps features.

Version Control and Git Workflows

[DT-0540] Metodi di sviluppo agile

Daniele Di Pompeo

daniele.dipompeo@univaq.it

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

A.A. 2024-2025