Version Control (Git)

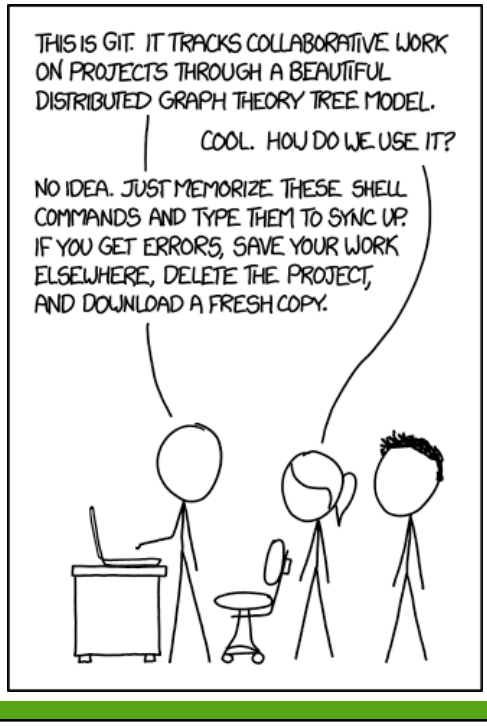
Kris.Hermans@pxl.be

DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Dep. PXL-IT – Elfde-Liniestraat 26 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

Contents

- Part 1: command line
  - Intro
  - Working Locally
  - Working Remote
  - Undoing Changes
  - Branch / Merge / Rebase
- Part 2: internals
  - How Git (really) works → To Be Continued…



https://xkcd.com/1597/

Introduction

- Created by Linus Torvalds, who also created Linux
- Prompted by Linux-BitKeeper separation
- Started in 2005
- Written in Perl and C
- Runs on Linux, OS X and Windows
- Design goals
  - Speed
  - Simplicity
  - Strong branch/merge support
  - Distributed
  - Scales well for large projects

Why distributed?

- Reliable branching/merging
  - Feature branches
  - Always work under version control
  - Applying fixes to different branches
- Full local history
  - Compute repository statistics
  - Analyze regressions
- New ideas (DevOps!)
  - CI/CD pipelines

New ideas: e.g. to be able to have a code repository and when you push a branch, your cloud production deploys a new version => automatically.

One speaks of “Continuous Integration” and “Continuous Delivery”.

Installation

- Windows
  - https://git-for-windows.github.io/
  - choco install git
    - Chocolatey: https://chocolatey.org/
- MacOS X
  - brew install git
    - Homebrew: https://brew.sh/index_nl.html
- Linux
  - apt-get git

There are several ways to install git.

Linux
This is de “mother OS” and git is easily installed using apt-get command

Windows/Mac
On the other OS, there are guides which go through some tools for imitating Bash etc. This is no “first class” experience. The Github team committed themselves to provide a good installer for the command line AND a graphical client. The graphical clients makes some tasks easy (e.g. .gitignore files), but it is strongly recommended to learn the commandline first.

Configuration

- System-level configuration
  - `git config --system`
  - Stored in /etc/gitconfig or c:\Program Files (x86)\Git\etc\gitconfig
- User-level configuration
  - `git config --global`
  - Stored in ~/.gitconfig or c:\Users\.gitconfig
- Repository-level configuration
  - `git config`
  - Stored in .git/config in each repo

PXL IT

You can edit these files with an editor or use the git config command.

https://git-scm.com/docs/git-config

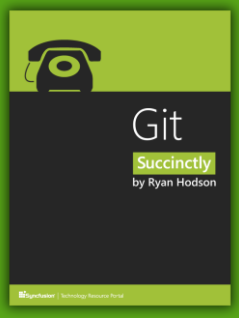
At least you should configure user.email and user.name

Nice to know:

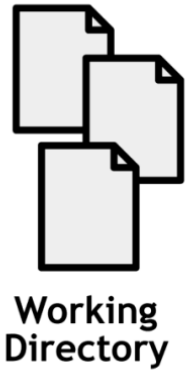
You can use VS Code to views diffs:

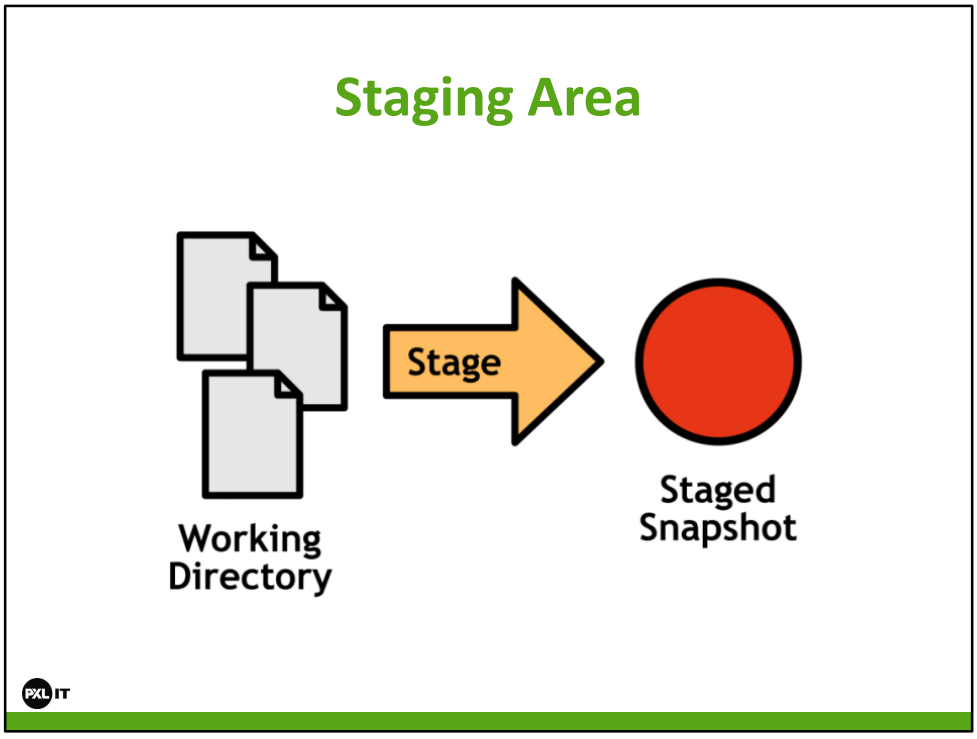
Source: https://medium.com/faun/using-vscode-as-git-mergetool-and-difftool-2e241123abe7

WORKING LOCALLY



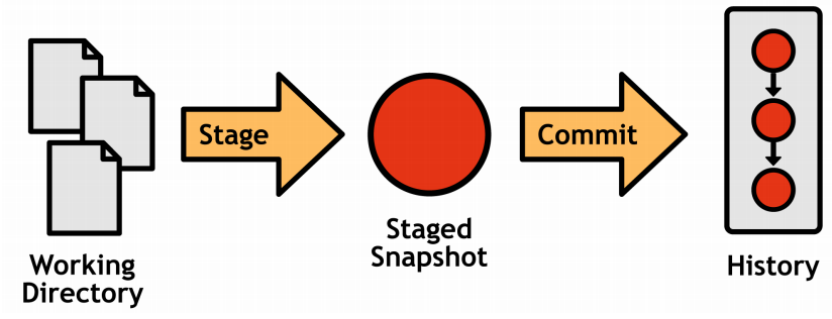
Working Directory



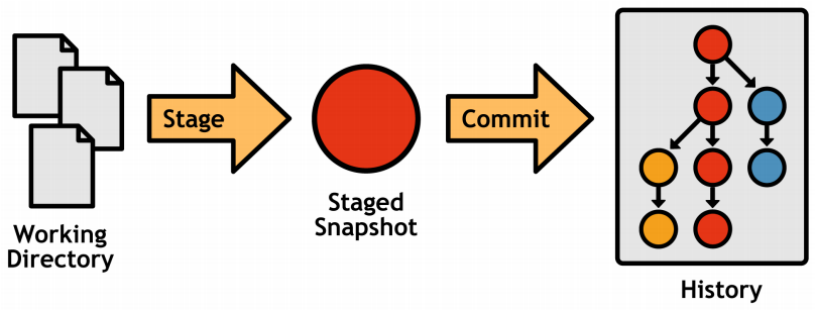


You pick the files that you want to commit, not default “all changed” files.

Committed History



Development Branches



cheap to make, simple to merge, and easy to share

Git branches are not like the branches of centralized version control systems. They are cheap to make, simple to merge, and easy to share, so Git-based developers use branches for everything—from long-running features with several contributors to 5-minute fixes. Many developers only work in dedicated topic branches, leaving the main history branch for public releases.

Demo / Steps

- Initializing repositories
- Cloning repositories
- Add files to staged snapshot
- Inspect the stage: git status
- Commit
- Inspect commit: git log
- Tag a commit: git tag

Tags are simple pointers to commits, and they are incredibly useful for bookmarking important revisions like public releases. The git tag command can be used to create a new tag:

git tag -a v1.0 -m "Stable release“

The -a option tells Git to create an annotated tag, which lets you record a message along with it (specified with -m).

Running the same command without arguments will list your existing tags:

git tag

UNDOING

Undoing

- Great flexibility
- “Everything can be undone”
  - Read “Git Succintly” Chapter 4
  - → only commands mentioned in the slides

Chapter 4 uit “git succintly” is GEEN leerstof (alleen de commando’s uit deze slides)

→ Vermits je vertrekt vanuit een branch voor elke feature/experiment, kan je de branch gemakkelijk verwijderen
→ Alles is ongedaan te maken
→ Gebruik dit hoofdstuk als referentie als je het nodig zou hebben (stage/werk)

Undo modified file, not staged

- git checkout HEAD <<file>>

Undo modified file, staged

- git reset HEAD <<file>>

Undo stage and modified files

- git reset HEAD --hard
- Clean up: get rid of untracked files
  - git clean -f

Undoing commits

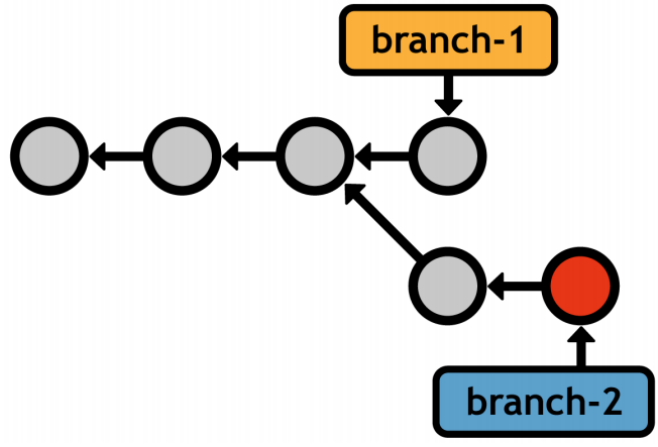
- git reset HEAD~1
  - Creates problematic commits in isolation
- git revert <<commitid>>
  - Creates a new commit with the undoing info

amending

- git commit --amend
- Add the current stage to the previous commit
- Handy for forgotten files

BRANCH, MERGE, REBASE

Branches



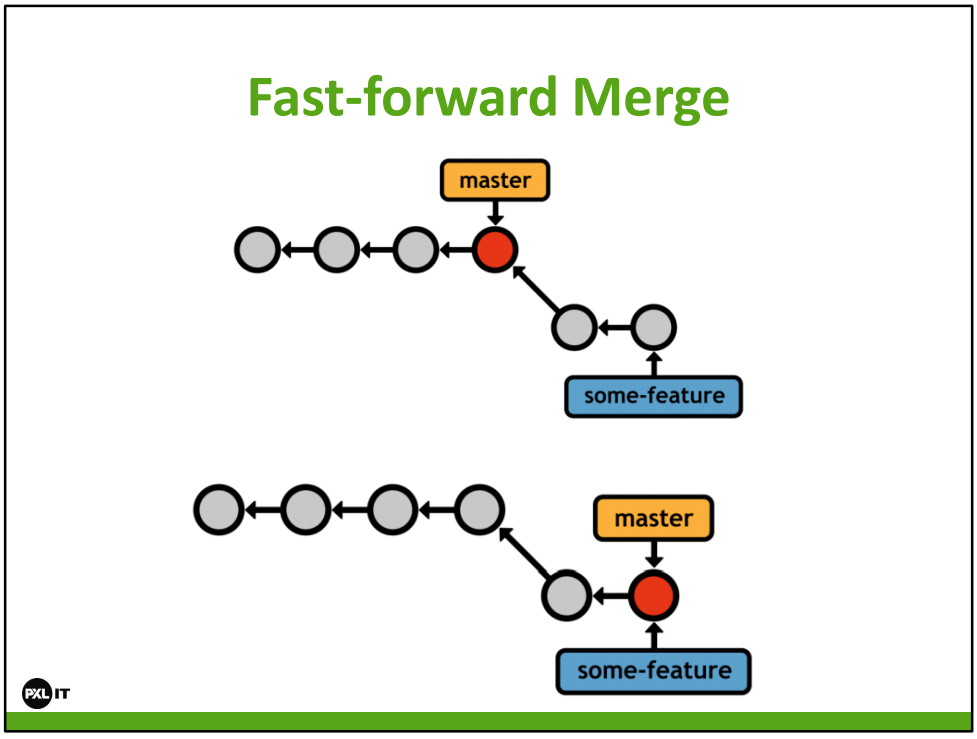
Demo

- Create a branch
- Delete a branch
- Checkout branch (and show in the IDE)

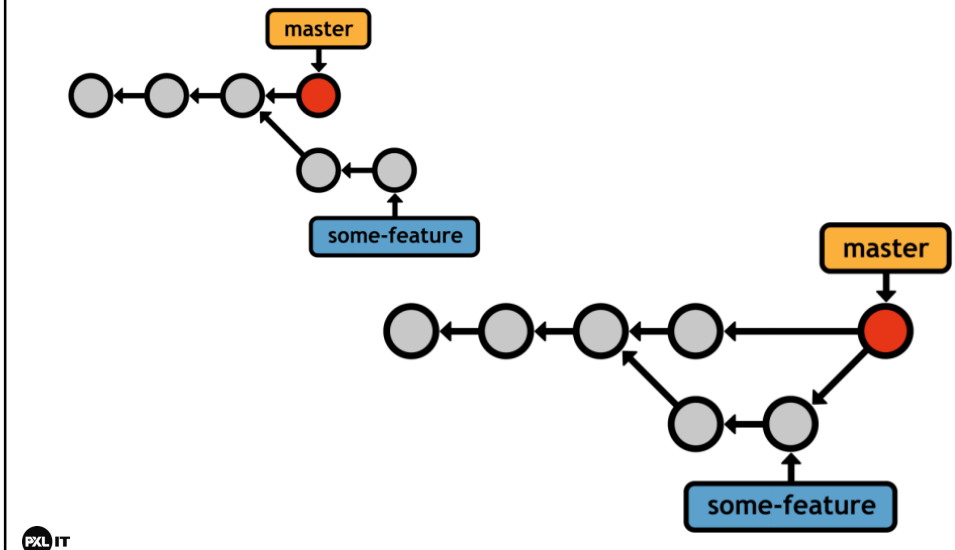
Merge

- Fast forward
- 3 way merge

Fast-forward Merge

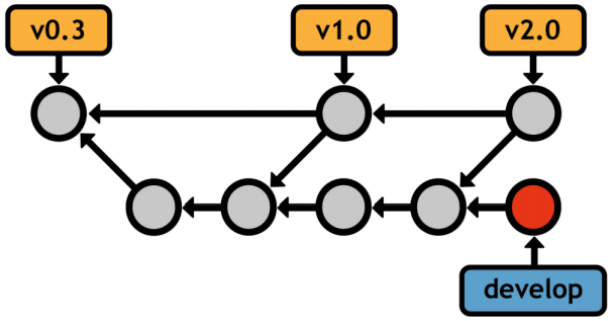


3 way merge



Branching Workflows

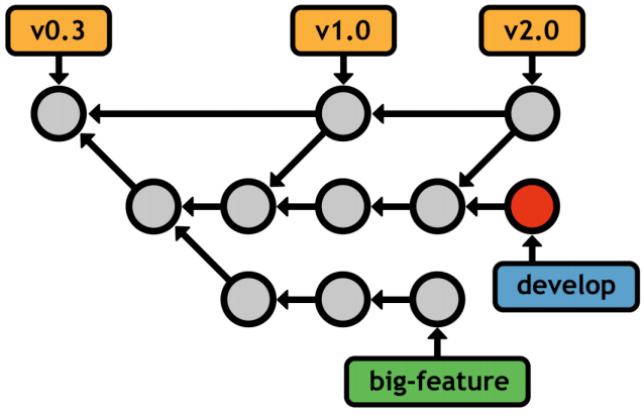
- Permanent branches



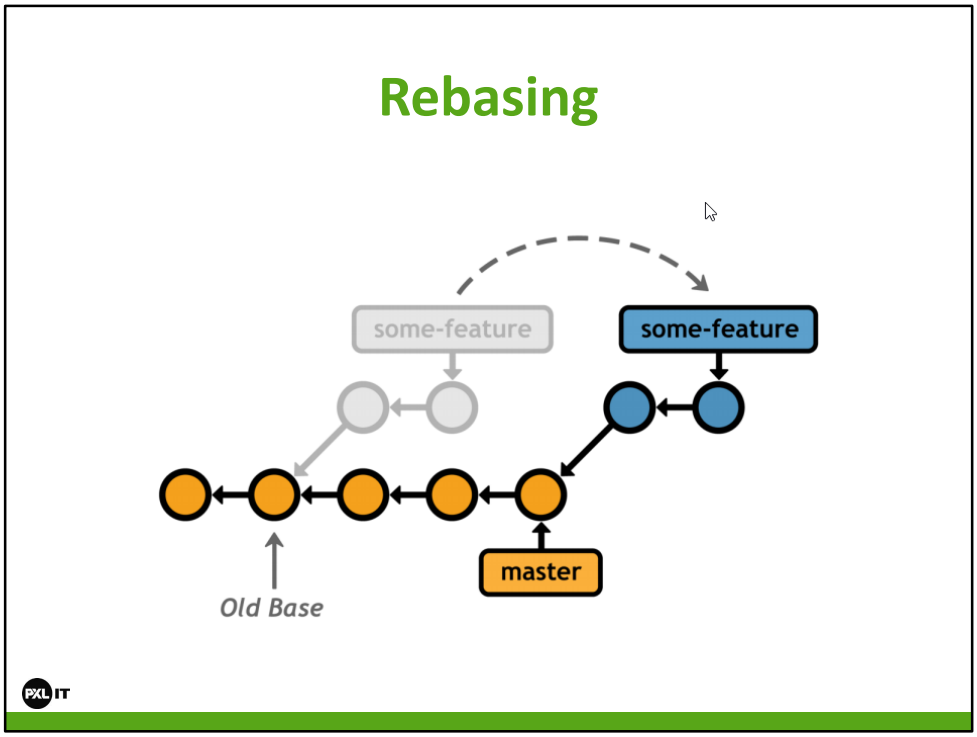
Most developers use master exclusively for stable code. In these workflows, you never commit directly on master—it is only an integration branch for completed features that were built in dedicated topic branches. In addition, many users add a second layer of abstraction in another integration branch (conventionally called develop, though any name will suffice). This frees up the master branch for really stable code (e.g., public commits), and uses develop as an internal integration branch to prepare for a public release

Branching Workflows

- Topic branches



Topic branches generally fall into two categories: **feature** branches and **hotfix** branches. Feature branches are temporary branches that encapsulate a new feature or refactor, protecting the main project from untested code. They typically stem from another feature branch or an integration branch, but not the “super stable” branch.

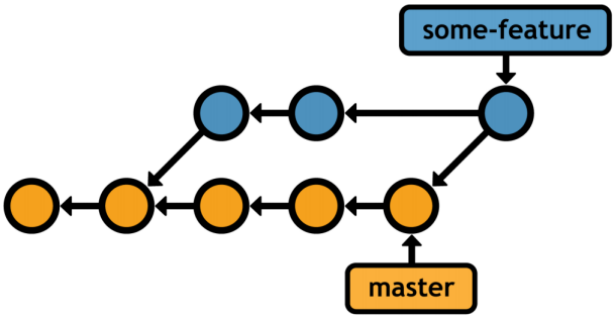


Rebasing is the process of moving a branch to a new base. Git's rebasing capabilities make branches even more flexible by allowing users to manually organize their branches. Like merging, git rebase requires the branch to be checked out and takes the new base as an argument:

git checkout some-feature
git rebase master

After the rebase, the feature branch is a linear extension of master, which is a much cleaner way to integrate changes from one branch to another. Compare this linear history with a merge of master into some-feature, which results in the exact same code base in the final snapshot:

Rebasing vs 3 way commit



Since the history has diverged, Git has to use an extra merge commit to combine the branches. Doing this many times over the course of developing a longrunning feature can result in a very messy history

Golden rule

Never rebase a branch that has been pushed to a public repository.

WORKING REMOTELY

git remote -v

- Demo
- git fetch
  git merge
- git pull
- git push

Since the fetch/merge sequence is such a common occurrence in distributed development, Git provides a pull command as a convenient shortcut:

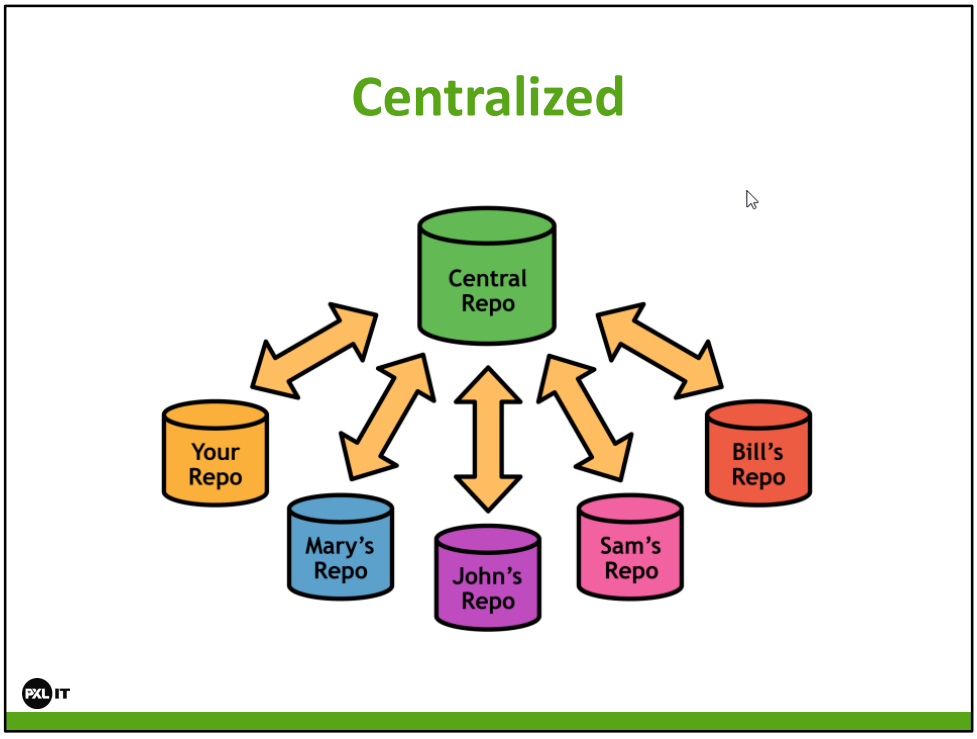
git fetch <remote> [<branch>] → fetch a remote branch (or ALL remote branches if branchname is omitted)

After a fetch, you typically merge the remote work into your own branch.

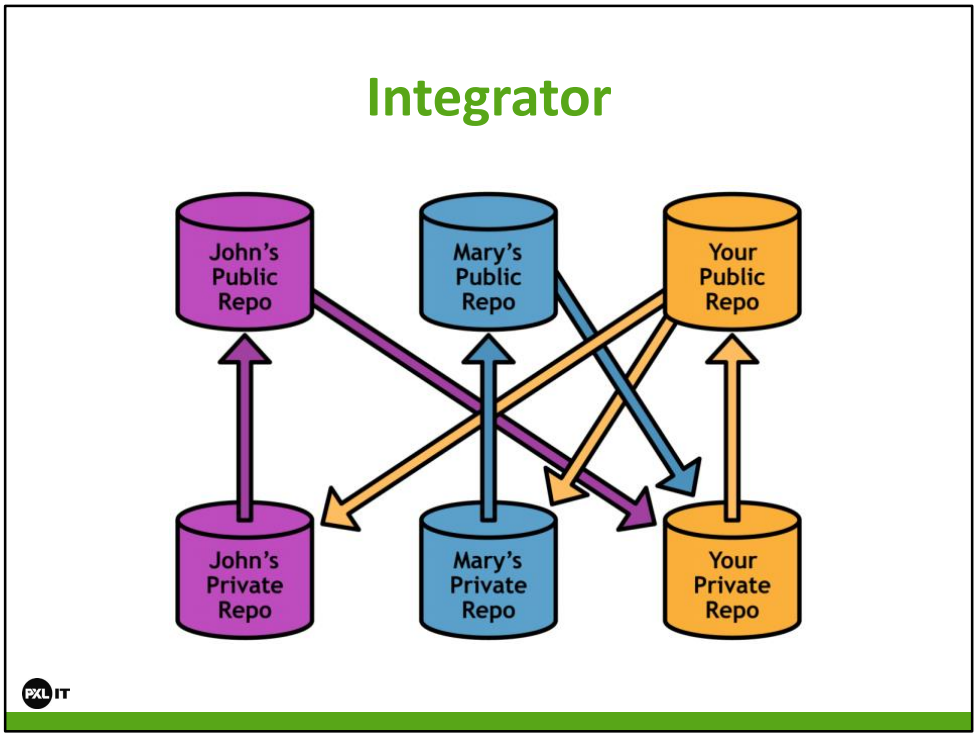
Shortcut for fetch and merge:
git pull

Remote workflow

- Centralized
- Integrator



Individual developers work in their own local repository, which is completely isolated from everyone else. Once they've completed a feature and are ready to share their code, they clean it up, integrate it into their local master, and push it to the central repository (e.g., origin). This also means all developers need SSH access to the central repository.



Note that the team must still agree on a single "official" repository to pull from—otherwise changes would be applied out-of-order and everyone would wind up out-of-sync very quickly. In the above diagram, "Your Public Repo" is the official project. As an integrator, you have to keep track of more remotes than you would in the centralized workflow, but this gives you the freedom and security to incorporate changes from any developer without threatening the stability of the project. In addition, the integrator workflow has no single point-of-access to serve as a choke point for collaboration. In centralized workflows, everyone must be completely up-to-date before publishing changes, but that is not the case in distributed workflows. Again, this is a direct result of the nonlinear development style enabled by Git's branch implementation. These are huge advantages for large open-source projects. Organizing hundreds of developers to work on a single project would not be possible without the security and scalability of distributed collaboration.

Branching strategies

- When to use a (remote) branch
- What's its purpose
- Example: Git Flow
  https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

Git Flow

Master : contains only releases (tagged), official releases
Develop : ongoing work
Feature : branched from Develop, work on new features, merged into develop
Release : Candidates for release, once polished can be merged into Master
Hotfix : User reports error, dev creates hotfix branch from master, merged into master and develop

The overall flow of Gitflow is:
1. A develop branch is created from master
2. A release branch is created from develop
3. Feature branches are created from develop
4. When a feature is complete it is merged into the develop branch
5. When the release branch is done it is merged into develop and master
6. If an issue in master is detected a hotfix branch is created from master
7. Once the hotfix is complete it is merged to both develop and master

Some advanced stuff to try out

- Stash dirty working dir
  https://www.atlassian.com/git/tutorials/saving-changes/git-stash
- Squash commits while rebasing
  http://gitready.com/advanced/2009/02/10/squashing-commits-with-rebase.html

Referenties

- Git Succintly (= leerstof)
- Pro Git (= uitgebreide referentie)
  https://git-scm.com/book/en/v2