Git snorkeling

a shallow dive in the stupid content tracker

Credit: ChatGPT

Credit: Gemini

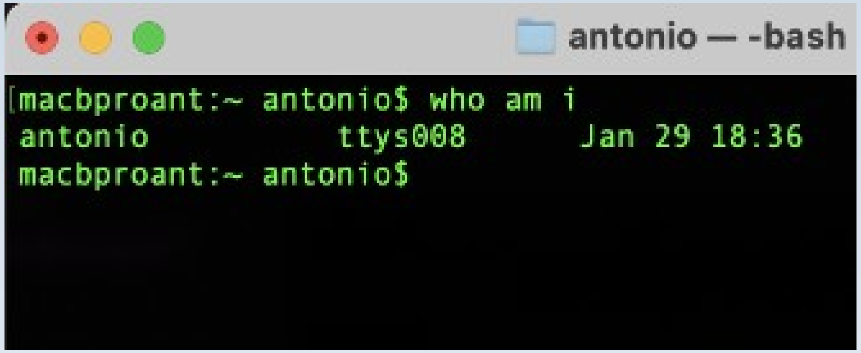
Git

```
Git(1)                    Git Manual                    Git(1)

NAME
     git – the stupid content tracker
SYNOPSIS
     git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
         [--exec-path[=<path>]] [--html-path] …
     ...
```

Who am i

- Antonio Chiurla - Ant1
  - Programmino s.n.c. di Chiurla & C
  - IT Consultant
  - Passioned of:
    - Software Troubleshooting & Performances
    - Java / C / Python / SQL / ...
    - Linux / Docker / vi (vim) / bash
    - ESP32 / Arduino
    - Audiobooks
  - antoniochiurla@gmail.com
  - antoniochiurla@programmino.it



```
[macbproant:~ antonio$ who am i
 antonio            ttys008       Jan 29 18:36
 macbproant:~ antonio$
```

Git - Agenda

- Overview / History
- Common commands
- Staging Area interaction
- Branch
- Log
- Merge / Rebase / Cherry-pick
- Conflict management
- Undo
- Branching models
- Under the hood
- Advanced

Git – interest scores

- 11% - 1 - History
- 11% - 1 - Overview
- 0% - 0 - Index interaction
- 11% - 1 - Branch
- 0% - 0 - Log
- 68% - 6 - Merge / Rebase / Cherry-pick
- 22% - 2 - Undo
- 22% - 2 - Conflict management
- 11% - 1 - Under the hood
- 55% - 5 - Advanced

Git – History

- 2005
- Linus Torvalds
- Instead of BitKeeper (license revoked)...
- Name (unpleasant person)

Git – Overview

- Standard de facto (primary VCS for 95%)
- Rapid branching/merging
- Distributed version control system
- Local copy of entire repository
- Many tasks are available off-line

Git – Common commands

clone

init

status

remote

fetch

push

add

commit

merge

Editor environment variables

- GIT_EDITOR
- EDITOR

Git – Index (staging area)

- The role
  - The index, or staging area, is an intermediate state that allows you to build up a commit piece by piece. It stores information about what will go into your next commit, letting you control the granularity of your changes.
    - https://www.geeksforgeeks.org/how-git-works-under-the-hood/#the-role-of-the-index-staging-area
- Add file(s) to index
  - git add <file...>
  - git add -A
- Undo add
  - git rm –cached <file...>
  - git restore –staged <file...>
  - git reset -- <file...>

Commit

The commit is the smaller entity of the history of the changes

commit [-m “<Comment>”]

Git – Branch

- List
  - git branch –list [<pattern>...]
  - git branch [--all]
- Creation
  - git branch <new_branch> [-u <name>/<branch>]
  - git checkout -b <new_branch>
  - git switch -c <new_branch>
- Switch
  - git checkout <branch>
  - git switch <branch>
- Delete
  - git branch -d <branch>

The **master** branch was the default branch, replaced by **main** starting at Jun 2020

Log

- git log
- git log <branch>
- git log <branch>   ^<branch>
- git log <commit>..<commit>
- git log <commit>…<commit>
- git log --left-right <commit>…<commit>
- git log -L<line>,<line>:<file>
- git log -L/<start>/,/<end>/:file
- git log -L:<function>:<file>

- --all
- --oneline
- --graph
- --patch
- --pretty[=(oneline|short|medium|full|...)]

Commit ref

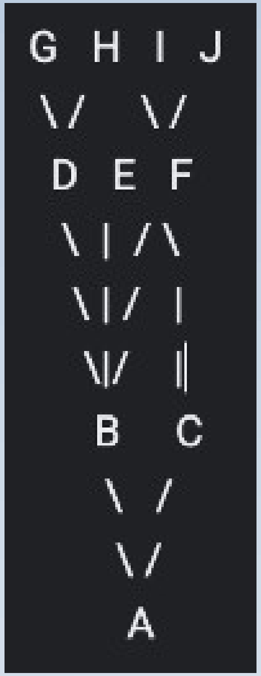
- <sha1>
- <first-characters-of-sha1>
- [<refname>]@{<date>}, e.g. master@{yesterday}, HEAD@{5 minutes ago}, HEAD@{2025-01-20}
- <refname>@{1} – previous timing reference
- HEAD^ - first parent
- HEAD^2 – second parent (in case of merge)
- HEAD~<n> - refers to the nth parent commit

Commit ref

ORIG_HEAD

<rev>^1 = <rev>^ First parent

<rev>~3 = <rev>^^^ =. <rev>^1^1^1



```
G   H   I   J
 \ /     \ /
  D   E   F
   \  |  / \
    \ | /   |
     \|/    ||
      B     C
       \   /
        \ /
         A
```

A = = A^0

B = A^ = A^1 = A~1

C = A^2

D = A^^ = A^1^1 = A~2

E = B^2 = A^^2

F = B^3 = A^^3

G = A^^^ = A^1^1^1 = A~3

H = D^2 = B^^2 = A^^^2 = A~2^2

I = F^ = B^3^ = A^^3^

J = F^2 = B^3^2 = A^^3^2

Merge

git merge

git merge <ref>

Incorporates changes from the named commits (since the time their histories diverged from the current branch) into the current branch

git merge –no-ff / --ff-only

git merge –squash

git merge --autostash

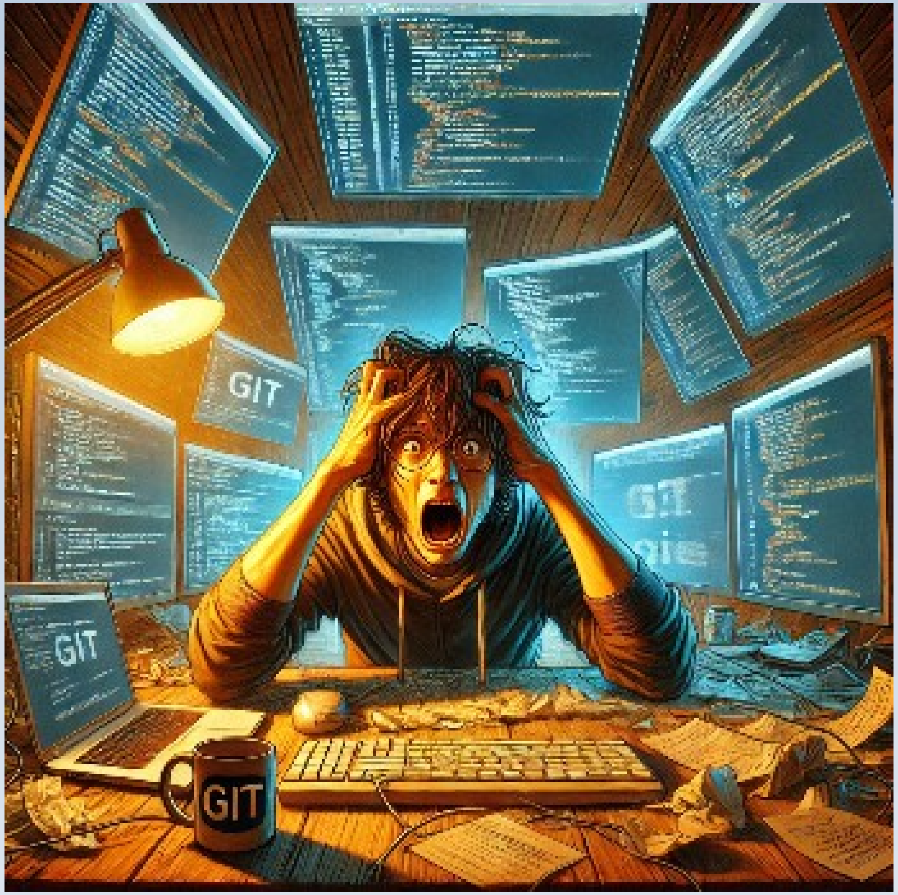
git merge (--continue | --abort | --quit)

Conflict management

When you merge two branches in Git, conflicts can arise if changes have been made to the same lines of code in both branches. In such cases, Git is unable to automatically determine which changes to keep and which to discard, and you will need to manually resolve the conflicts.If there are conflicts, Git will notify you that there are conflicts and tell you which files have conflicts. See the conflict in detail using git status.

Edit the file to keep the changes that you want to keep and remove the ones you want to discard. Once you have resolved all conflicts in a file, remove the conflict markers. Once you have resolved all conflicts, save the changes and add the files to the staging area using the command git add <file>. Then commit your changes using git commit -m "message". Finally, push the changes to the remote my-branch branch using the command git push origin my-branch. This should resolve your issue.

https://stackoverflow.com/a/75863335/2750723



Credit: ChatGPT

Conflict management

```
$ git checkout main

$ git merge featureXY

$ git status

$ # edit filename1.c and remove markers

$ git add filename1.c

$ git commit -m "merged featureXY"

$ git merge continue
```

```
$ ...

$ git merge ; git status

$ git checkout --ours filename1.c

$ git add filename1.c

$ git commit -m "using ours"

$ git checkout --theirs filename2.c

$ git add filename2.c

$ git commit -m "using theirs"

$ git merge continue
```

```
$ ...

$ git push; git status

$ git rebase

$ git push
```

Rebase

- **The golden rule of git rebase is to never use it on public branches.**
- **theirs** and **ours** are swapped

Rebase

- git rebase
- git rebase –onto <branch>
- git rebase (--continue | --skip | --abort | --quit)
- git rebase –interactive <base>
- git rebase -C<n>

Rebase

Rebase interactive

```
git rebase -i ( <after-this-commit> | HEAD~<commits-number> )

pick
edit
reword
drop
```

Editor environment variables

- GIT_SEQUENCE_EDITOR
- GIT_EDITOR
- EDITOR

Rebase

Recovering from upstream rebase

The easy case https://git-scm.com/docs/git-rebase#_the_hard_case
Git knows to skip changes that are already present in the new upstream

```
git rebase <base-branch>
```

The hard case https://git-scm.com/docs/git-rebase#_the_hard_case
The <base-branch> (the new one) contains changes do not exactly correspond to the changes before the rebase

```
git rebase --onto <base-branch> <base-branch>@{1}
```

Cherry-Pick

- git cherry-pick <commit>…
- git cherry-pick (--continue | --skip | --abort | --quit)

```
$ git switch -c private2.6.14 v2.6.14
$ # some edit/compile/test/commit…
$ git checkout main
$ git cherry-pick v2.6.14..private2.6.14
```

Pull request

```
git request-pull [-p] <start> <URL> [<end>]
```

A text will be generated with instructions to integrate changes

Undo

- git reset [--soft | --mixed | --hard] <commit>
- git reset <commit> # default --mixed
- git reset -- <file...>
- git revert <commit>…
- git revert (--continue | --skip | --abort | --quit)

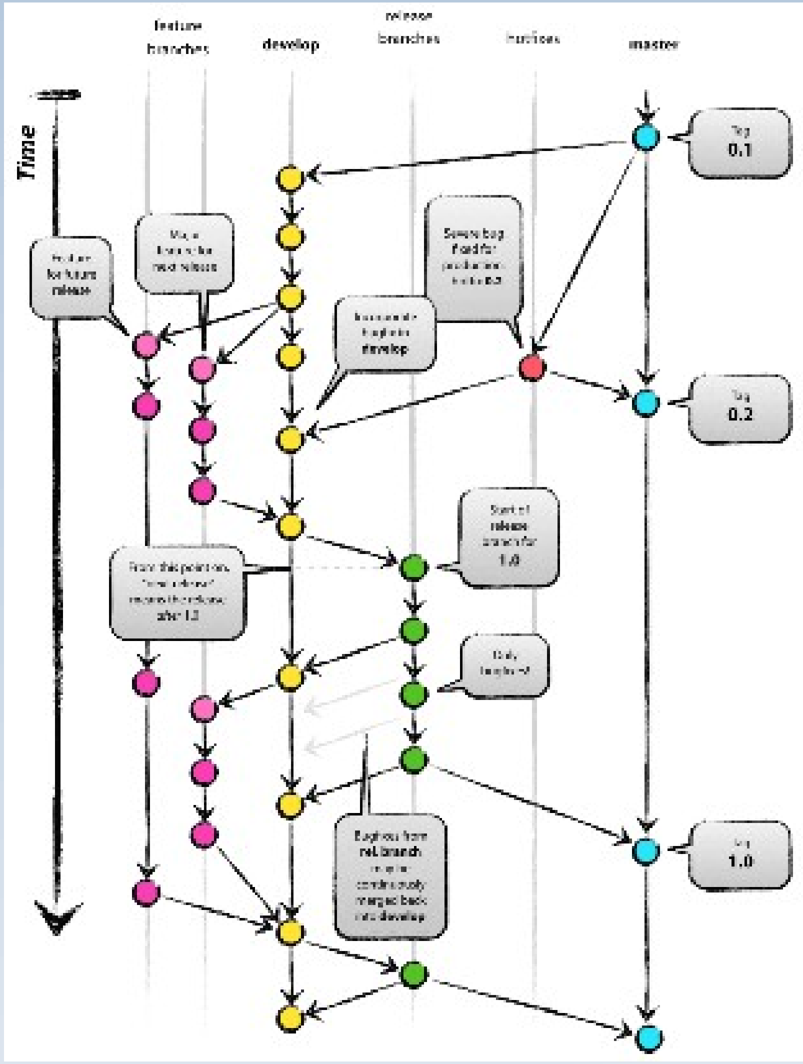
Branching models

Gitflow

Pros:
- Well-suited for large teams and aligning work across multiple teams.
- Effective handling of multiple product versions.
- Clear responsibilities for each branch.
- Allows for easy navigation of production versions through tags.

Cons:
- Complexity due to numerous branches, potentially leading to merge conflicts.
- Development and release frequency may be slower due to multi-step process.
- Requires team consensus and commitment to adhere to the strategy.

https://nvie.com/posts/a-successful-git-branching-model/

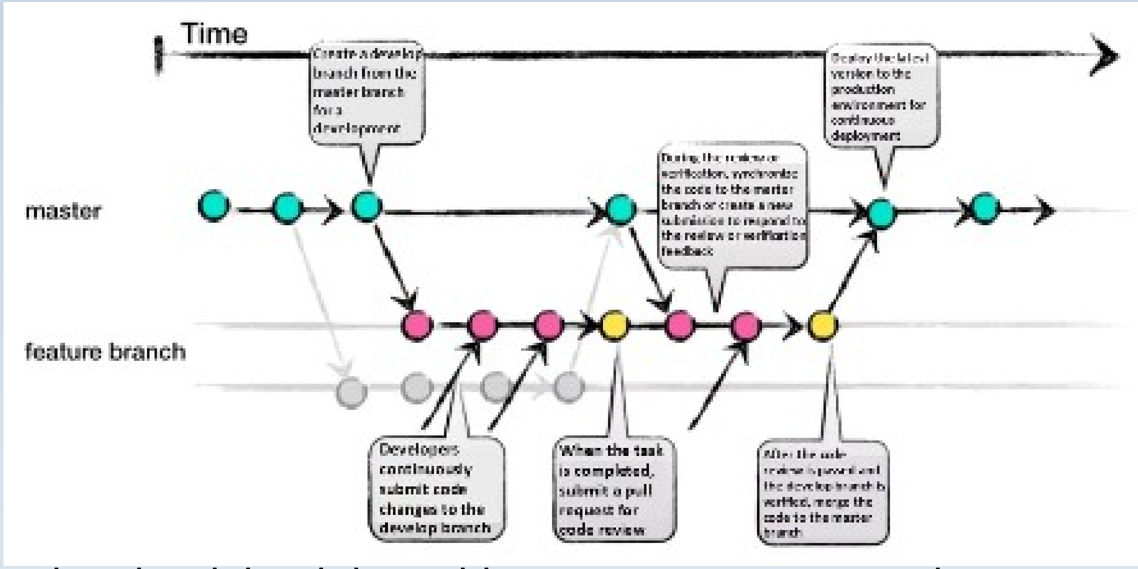

feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Branching models

Github-flow

Pros:
- Faster feedback cycles and shorter production cycles.
- Ideal for asynchronous work in smaller teams.
- Agile and easier to comprehend compared to Git-Flow.

Cons:
- Merging a feature branch implies it is production-ready, potentially introducing bugs without proper testing and a robust CI/CD process.
- Long-living branches can complicate the process.
- Challenging to scale for larger teams due to increased merge conflicts.
- Supporting multiple release versions concurrently is difficult.


Time
Create a develop branch from the master branch for a development
Deploy the latest version to the production environment for continuous deployment
master
During the review or verification, synchronize the code to the master branch or create a new submission to respond to the review or verification feedback
feature branch
Developers continuously submit code changes to the develop branch
When the task is completed, submit a pull request for code review
After the code review is passed and the develop branch is verified, merge the code to the master branch

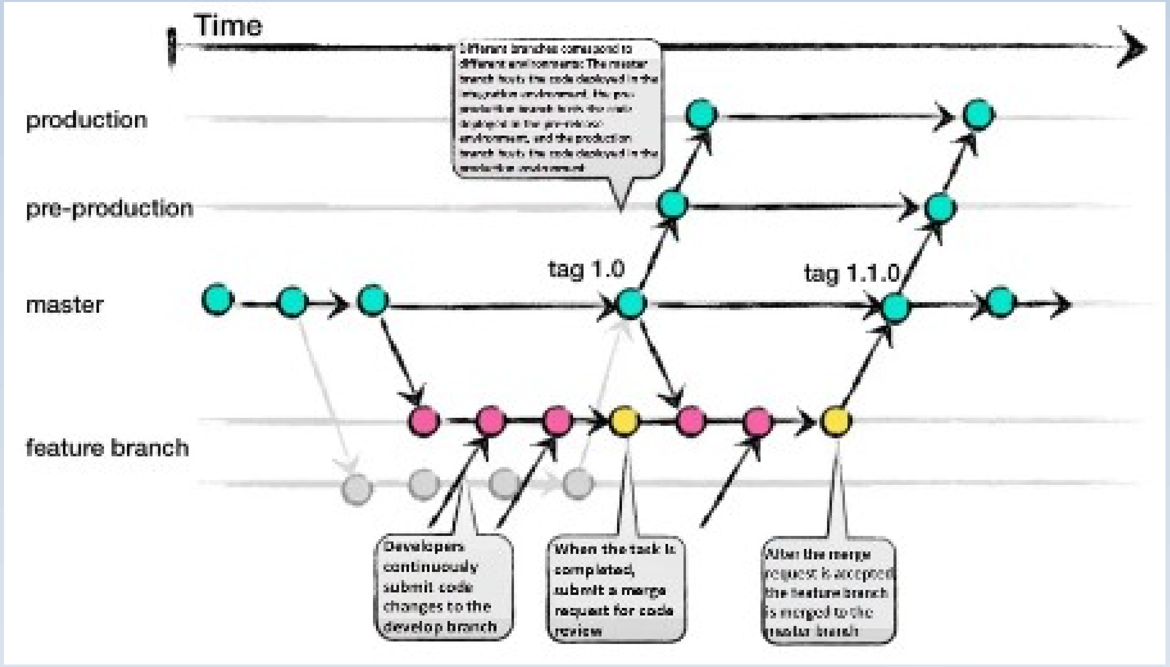
https://medium.com/@sreekanth.thummala/choosing-the-right-git-branching-strategy-a-comparative-analysis-f5e635443423

Branching models

Gitlab-flow

Pros:
- Can handle multiple release versions or stages effectively.
- Simpler than Git-Flow.
- Focuses on quality with a lean approach.

Cons:
- Complexity increases when maintaining multiple versions.
- More intricate compared to GitHub-Flow.


Time
Different branches correspond to different environments: The master branch hosts the code deployed in the integration environment, the pre-production branch hosts the code deployed in the pre-release environment, and the production branch hosts the code deployed in the production environment
production
pre-production
tag 1.0
tag 1.1.0
master
feature branch
Developers continuously submit code changes to the develop branch
When the task is completed, submit a merge request for code review
After the merge request is accepted the feature branch is merged to the master branch

https://medium.com/@sreekanth.thummala/choosing-the-right-git-branching-strategy-a-comparative-analysis-f5e635443423

Branching models

Trunk based development

Pros:

- Encourages DevOps and unit testing best practices.

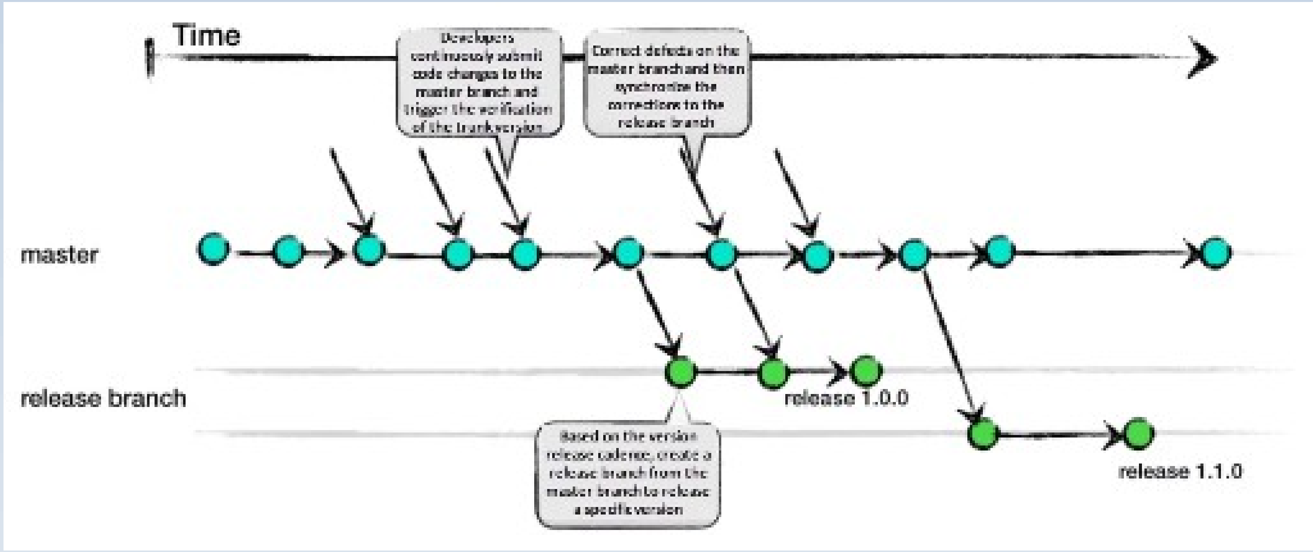
- Enhances collaboration and reduces merge conflicts.

- Allows for quick releases.

Cons:

- Requires an experienced team that can slice features appropriately for regular integration.

- Relies on strong CI/CD practices to maintain stability.


Time
Developers continuously submit code changes to the master branch and trigger the verification of the trunk version
Correct defects on the master branch and then synchronize the corrections to the release branch
master
release branch
release 1.0.0
Based on the version release cadence, create a release branch from the master branch to release a specific version
release 1.1.0

https://medium.com/@sreekanth.thummala/choosing-the-right-git-branching-strategy-a-comparative-analysis-f5e635443423

Under the hood – Local repository

- **.git** local repository
- **.git/objects** contains up to 256 directories of objects
- **.git/refs/heads** references for all local branches
- **.git/refs/remotes** a directory of refs for each remote
- **.git/refs/tags** contains all tags
- **.git/HEAD** reference to current branch

Under the hood – Git Object Model

- Blobs (z-lib compressed)
- Trees
- Commits
- Every object is identified by a 40-characters SHA-1 hash

Under the hood – Commit actions

- Creating blobs for all added/modified files
- Building tree objects
- Creating the commit object with parent references, comment, date, author and committer

Under the hood – Branches/Tags

- Tags
  - Pointers to commits
- Branches
  - Pointers to commits
  - Move forward as new commits are added

Under the hood – Fetch and Push

- Synchronization with remote repositories
- Fetch – download all new objects from remote
- Push
  - Check if local repository is up-to-date
  - upload new changes about a local branch to remote

Under the hood – Garbage collection

- Unreferenced objects due by branches deletion
- git gc
- Reflog and Expiry

Advanced - Submodules

- Pointers to repository at a specific commit
- Clone a submodule
  - git submodule add <git-repo-url>
  - git submodule init
  - Edit .gitmodules
  - git add .gitmodules
  - git commit
  - When you switch branch
    - git submodule update --remote
- git clone –recursive <git-repo-url>

.gitmodules

```
…
[submodule
“<modulename>”]
    path = <modulename>
    url = <submodule-repo-
url>
    branch = <branch>
...
```

Advanced - Worktrees

- Start a quick change in other branch
  - git worktree add <dest_path> <branch>
  - git worktree add -b <new_branch> <dest_path>
  - git worktree list
  - git worktree remove <worktree_name>

During list of branches:
any branches checked out in linked worktrees will be highlighted in cyan and marked with a plus sign

Advanced - Upstreams

- git remote add <name> <repo_url>
- git remote remove <name>
- git remote get-url --all
- git remote set-url --add --push <name> <url>
- git push –set-upstream <name> <branch>

Advanced – Depth clone

- Download last <n> commits of branches
  - git clone --depth <n> <url>
  - git clone --depth <n> --no-single-branch <url>
  - git clone --depth <n> --single-branch -b <branch> <url>

Commit like a patch

- When git applies a commit, it extract the differences with the previous snapshot, generates the changes and tries to apply them to the file in the worktree
- If the context of individual changes does not match, it generates a conflict

1 cent tips

- During normal work, commit often
- A commit must represent one and only one change
- The commit comment must describe the change
- Before pushing, review the history evaluating the expresiveness of each commit and rebase

Thank you!

Grazie!

Java User Group Genova

ESPLACE

Q&A

...