Git

Guillaume Latour - Condorcet - 2025

1. Introduction
2. Git en local
3. Git à plusieurs
4. Gitflow
5. Conclusion

C'est quoi git ?

git est un logiciel de gestion de version décentralisé créé par Linus Torvalds en 2005 et sans cesse amélioré depuis

> Chaque seconde de la conférence Linus Torvalds & git mérite votre attention.

C'est quoi la gestion de version ?

Garder un historique des modifications: qui a fait quoi et quand?

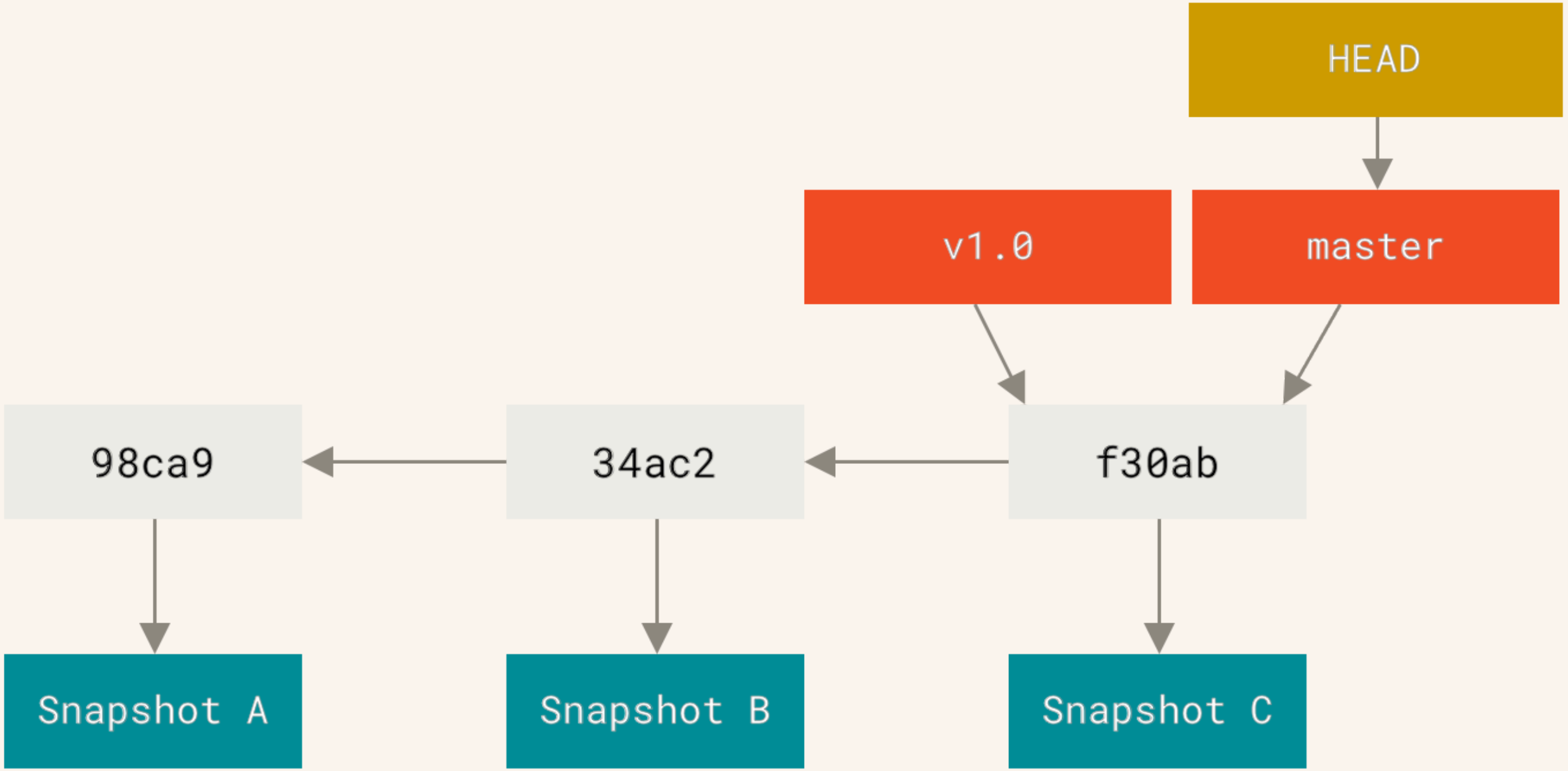
Avant git, la meilleure façon de faire (selon Linus) était d'avoir des archives `tar` avec une nomenclature qui permet de s'y retrouver.

Pourquoi "décentralisé" est important ?

- Il n'y a pas un unique point central qui garde les données
- Il est possible de travailler *offline*
- Aucun endroit n'est plus important que n'importe quel autre
- Pas de problèmes liés au "commit access"

Concepts clefs

- Un ***repository*** est le contenant du projet.
- Un ***commit*** contient un ensemble de modifications apportées au projet. Celles-ci sont datées et l'auteur est identifié sur chacune d'entre elles.
- Une **branche** est un enchainement de commit qui partagent une idée commune.


HEAD
v1.0
master
98ca9
34ac2
f30ab
Snapshot A
Snapshot B
Snapshot C

Les outils

- CLI >>> tout le reste
  - précision
  - performances
  - utilisation dans des scripts
- ya des GUI qui existent...

Installation

Sur la page téléchargement du site git-scm.com, téléchargez et installez git.

```
$ git --version
git version 2.34.1
```

Git en local

Init

On peut se créer un petit projet C# avec

```
dotnet new console -n git_cshonsole
cd git_cshonsole
ls -la # `dir` si vous utilisez un OS inférieur
git init
ls -la
```

Le dossier `.git`

La puissance d'un *repository*, juste pour vous.

```
.
├── branches
├── config
├── description
├── HEAD
├── hooks
├── info
├── objects
└── refs
```

git status

```
$ git status --help
GIT-STATUS(1)                          Git Manual                          GIT-STATUS(1)

NAME
       git-status - Show the working tree status

SYNOPSIS
       git status [<options>...] [--] [<pathspec>...]

DESCRIPTION
       Displays paths that have differences between the index file and the current
       HEAD commit, paths that have differences between the working tree and the
       index file, and paths in the working tree that are not tracked by Git (and are
       not ignored by gitignore(5)). The first are what you would commit by running
       git commit; the second and third are what you could commit by running git add
       before running git commit.
```

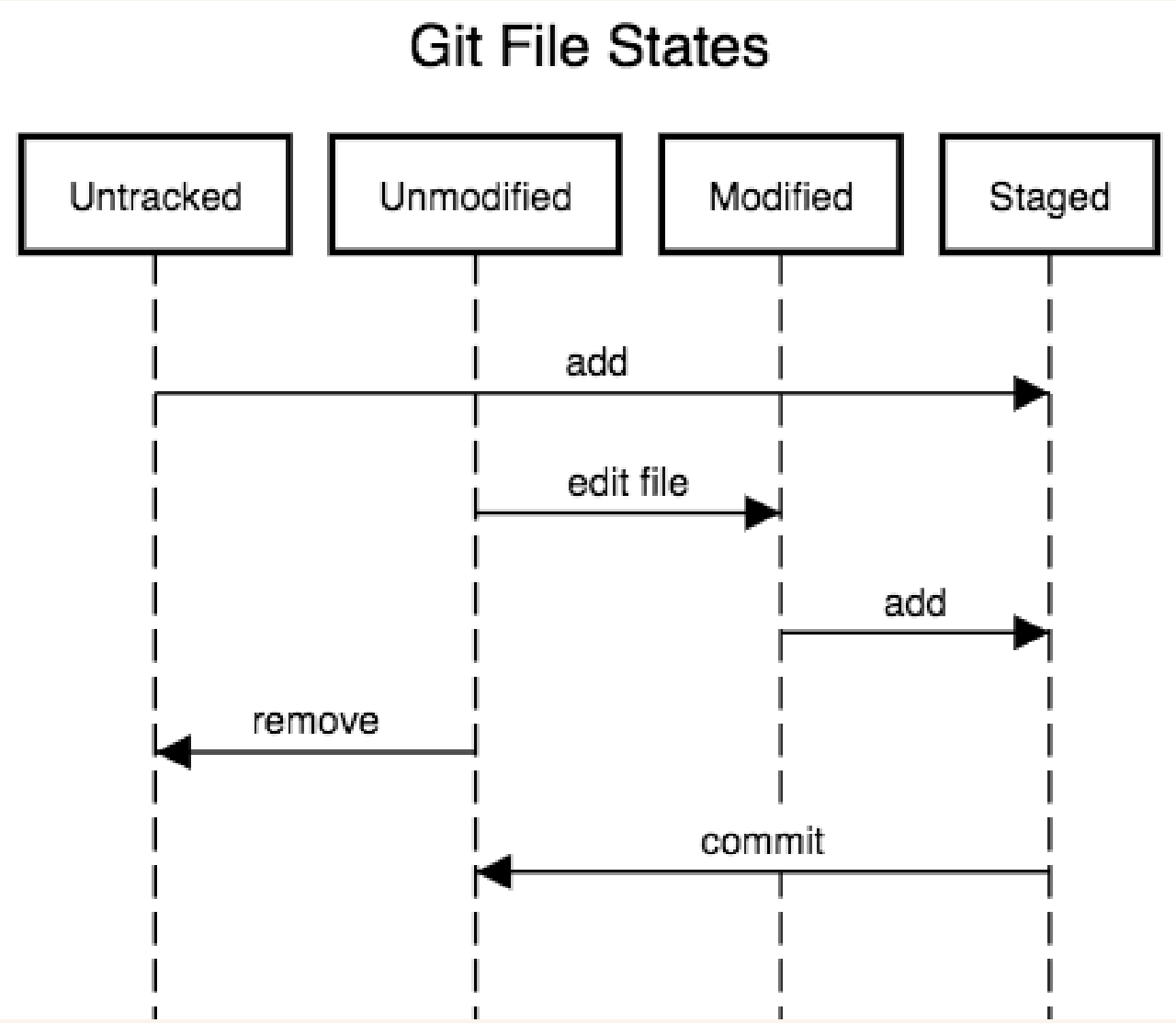
La commande `git status` permet de connaitre l'état de votre espace de travail.

```
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Program.cs
        git_cshonsole.csproj
        obj/

nothing added to commit but untracked files present (use "git add" to track)
```

États de fichier

Les fichiers présents dans un *repository* git se voient assigné un état par git.

Il est alors possible de voir rapidement quels sont les fichiers modifiés ou quels sont les fichiers qui vont être inclus dans le prochain commit.



git add

```
$ git add --help
GIT-ADD(1)                          Git Manual                          GIT-ADD(1)

NAME
       git-add - Add file contents to the index

SYNOPSIS
       git add [--verbose | -v] [--dry-run | -n] [--force | -f] [--interactive | -i] [--
patch | -p]

DESCRIPTION
       This command updates the index using the current content found in the working
       tree, to prepare the content staged for the next commit. It typically adds the
       current content of existing paths as a whole, but with some options it can
       also be used to add content with only part of the changes made to the working
       tree files applied, or remove paths that do not exist in the working tree
       anymore.

       [...]
```

```
$ git add .
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
 new file:   Program.cs
 new file:   git_cshonsole.csproj
 new file:   obj/git_cshonsole.csproj.nuget.dgspec.json
 new file:   obj/git_cshonsole.csproj.nuget.g.props
 new file:   obj/git_cshonsole.csproj.nuget.g.targets
 new file:   obj/project.assets.json
 new file:   obj/project.nuget.cache
```

Pour *unstage* un fichier, utiliser `git rm --cached`

```
$ git rm --cached -r obj # -r pour "recursive"
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
 new file:   Program.cs
 new file:   git_cshonsole.csproj

Untracked files:
  (use "git add <file>..." to include in what will be committed)
 obj/
```

`git commit`

`git commit` permet d'effectuer un snapshot de la version actuelle, en ajoutant le code "staged" dans l'arborescence git.

```
$ git commit -m "init"
[master (root-commit) 776a8f2] init
 2 files changed, 12 insertions(+)
 create mode 100644 Program.cs
 create mode 100644 git_cshonsole.csproj
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
 obj/

nothing added to commit but untracked files present (use "git add" to track)
```

`.gitignore`

Le `.gitignore` est un fichier qui permet à git d'ignorer des fichiers/dossiers. Cela empêche de `commit` des données sensibles, ou d'avoir de la pollution via le `git status`.

```
# Dans `.gitignore`
[Oo]bj/
```

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
 .gitignore

nothing added to commit but untracked files present (use "git add" to track)
```

```
$ git add .gitignore
$ git commit -m "ignore obj folder"
[master 7ddaeac] ignore obj folder
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
$ git status
On branch master
nothing to commit, working tree clean
```

```
$ dotnet build
MSBuild version 17.8.19+c3ade832a for .NET
  Determining projects to restore...
  All projects are up-to-date for restore.
  git_cshonsole -> /tmp/git_cshonsole/bin/Debug/net8.0/git_cshonsole.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:06.79
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
 bin/

nothing added to commit but untracked files present (use "git add" to track)
```

Un `.gitignore` de pro

```
$ curl -sL https://www.toptal.com/developers/gitignore/api/csharp > .gitignore
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
 modified:   .gitignore

no changes added to commit (use "git add" and/or "git commit -a")
```

git diff

```
diff --git a/.gitignore b/.gitignore
index 760f99c..30a404f 100644
--- a/.gitignore
+++ b/.gitignore
@@ -1 +1,402 @@
+# Created by https://www.toptal.com/developers/gitignore/api/csharp
+# Edit at https://www.toptal.com/developers/gitignore?templates=csharp
+
+### Csharp ###
+## Ignore Visual Studio temporary files, build results, and
+## files generated by popular Visual Studio add-ons.
+##
+## Get latest from https://github.com/github/gitignore/blob/main/VisualStudio.gitignore
+
+# User-specific files
+*.rsuser
+[Aa][Rr][Mm]/
+[Aa][Rr][Mm]64/
+bld/
+[Bb]in/
 [Oo]bj/
+[Ll]og/
+[Ll]ogs/
[...]
```

git log

```
$ git log
commit 285f6ae19334a99168aa639bc251f82630d96cc8 (HEAD -> master)
Author: Tazoeur <g0latour@gmail.com>
Date:   Mon Mar 31 17:19:56 2025 +0200

    profesionnal ignore

commit 7ddaeacb1e9ac91288b865666a90046b02f60133
Author: Tazoeur <g0latour@gmail.com>
Date:   Mon Mar 31 16:52:01 2025 +0200

    ignore obj folder

commit 776a8f28ddcc7b7bddcae49e7f1ded8c61418e15
Author: Tazoeur <g0latour@gmail.com>
Date:   Mon Mar 31 16:43:29 2025 +0200

    init
```

`git log` est flexible sur la vue que l'on souhaite avoir

```
$ git log --one-line
285f6ae (HEAD -> master) profesionnal ignore
7ddaeac ignore obj folder
$ git log --all --decorate --oneline --graph # log A DOG
* 285f6ae (HEAD -> master) profesionnal ignore
* 7ddaeac ignore obj folder
* 776a8f2 init
```

Les branches

Créer une branche est très facile avec `git branch`.

```
$ git branch develop
$ git log --all --decorate --oneline --graph
* 285f6ae (HEAD -> master, develop) profesionnal ignore
* 7ddaeac ignore obj folder
* 776a8f2 init
$ git status
On branch master
nothing to commit, working tree clean
```

`git checkout`

Pour se déplacer de branche en branche, on peut utiliser la commande `git checkout`, même si la nouvelle documentation a plutôt l'air de recommander `git switch <branch>`.

```
$ git checkout develop
$ git status
On branch develop
nothing to commit, working tree clean
```

Maintenant, les nouveaux commits vont être ajoutés à la branche sur laquelle on se trouve.

```
$ git log --all --decorate --oneline --graph
* e23387e (HEAD -> develop) feat: say my name at startup
* 285f6ae (master) profesionnal ignore
* 7ddaeac ignore obj folder
* 776a8f2 init
```

HEAD

`HEAD` est un pointeur vers le workspace actuel. On peut se déplacer dans l'arborescence en utilisant l'id des commits

```
$ git checkout 7ddaeac
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 7ddaeac ignore obj folder
```

```
$ git log --all --decorate --oneline --graph
* e23387e (develop) feat: say my name at startup
* 285f6ae (master) profesionnal ignore
* 7ddaeac (HEAD) ignore obj folder
* 776a8f2 init
$ git status
HEAD detached at 7ddaeac
Untracked files:
  (use "git add <file>..." to include in what will be committed)
 bin/

nothing added to commit but untracked files present (use "git add" to track)
```

On peut créer une(des) nouvelle(s) branche(s) ou des tags à l'endroit où on se trouve

```
$ git branch retro-coding
$ git tag v0.0.1
$ git tag amateurism
$ git log --all --decorate --oneline --graph
* e23387e (develop) feat: say my name at startup
* 285f6ae (master) profesionnal ignore
* 7ddaeac (HEAD, tag: v0.0.1, tag: amateurism, retro-coding) ignore obj folder
* 776a8f2 init
```

Différence entre une branche et un tag

Tous les deux sont des concepts utilisé en tant que pointeur de commit.

Une **branche** est dynamique et pointe vers le dernier commit de l'embranchement.

Un **tag** est un pointeur vers un commit précis et n'en changera jamais.

Imaginons maintenant que plusieurs branches aient effectué des modifications dans le projet.

```
$ git log --all --decorate --oneline --graph
* 30f506c (HEAD -> master) feat: using system
| * e23387e (develop) feat: say my name at startup
|/
* 285f6ae profesionnal ignore
* 7ddaeac (tag: v0.0.1, tag: amateurism, retro-coding) ignore obj folder
* 776a8f2 init
```

git merge

```
$ git merge develop
Auto-merging Program.cs
Merge made by the 'ort' strategy.
 Program.cs | 1 +
 1 file changed, 1 insertion(+)
$ git log --all --decorate --oneline --graph
*   d6458d0 (HEAD -> master) Merge branch 'develop'
|\
| * e23387e (develop) feat: say my name at startup
* | 30f506c feat: using system
|/
* 285f6ae profesionnal ignore
* 7ddaeac (tag: v0.0.1, tag: amateurism, retro-coding) ignore obj folder
* 776a8f2 init
```

Exercices

- refaire manuellement tout ce qui a été vu dans les slides
- Learn git branching

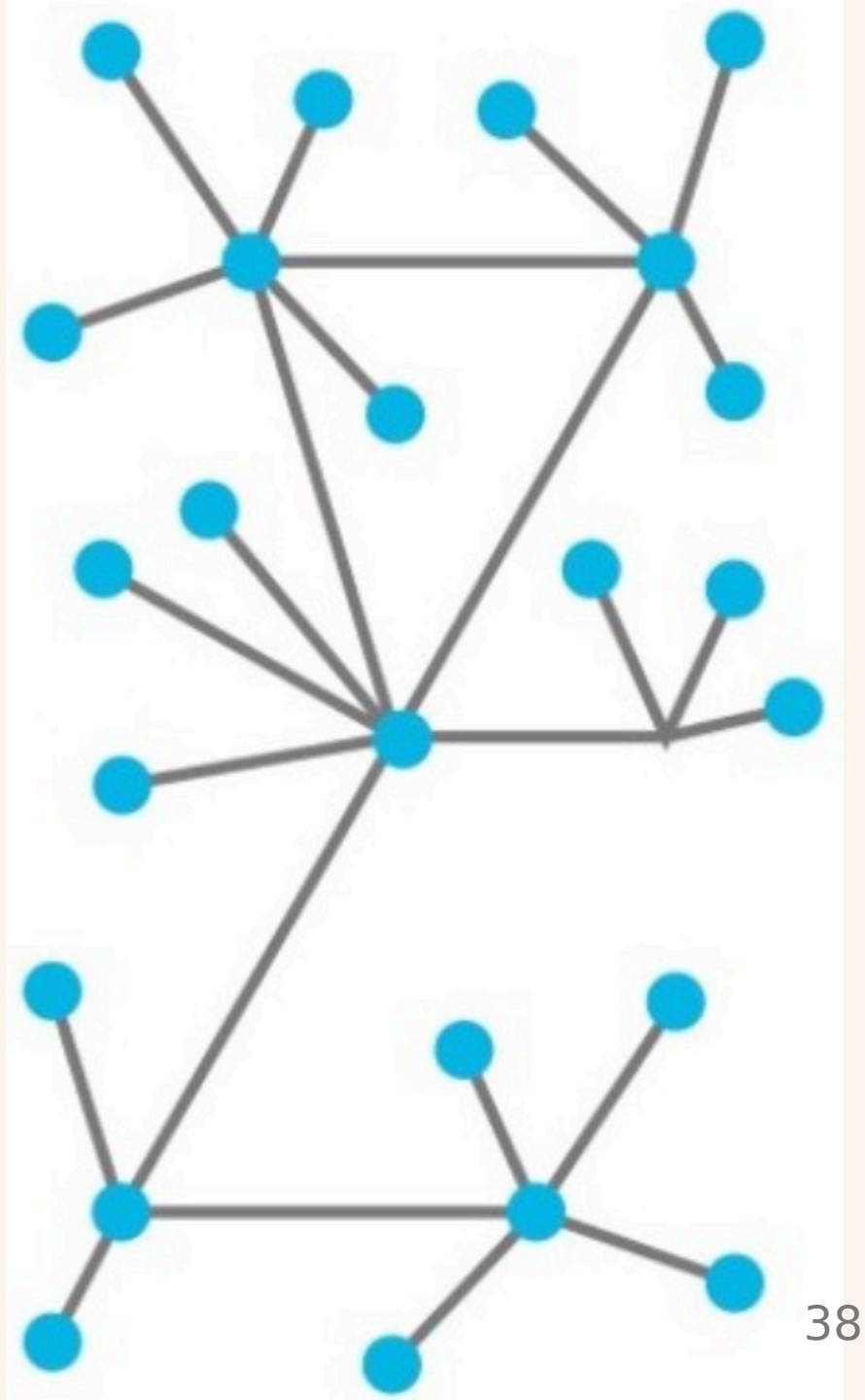
Git à plusieurs

Git est décentralisé

Comme dit précédemment, git est décentralisé.

Cela implique l'existance d'un méchanisme pour partager son *repository* et pour utiliser les autres *repository* qui ont été partagés.

Plusieurs services simplifient cette distribution: github, gitlab, bitbucket, framagit, gitea

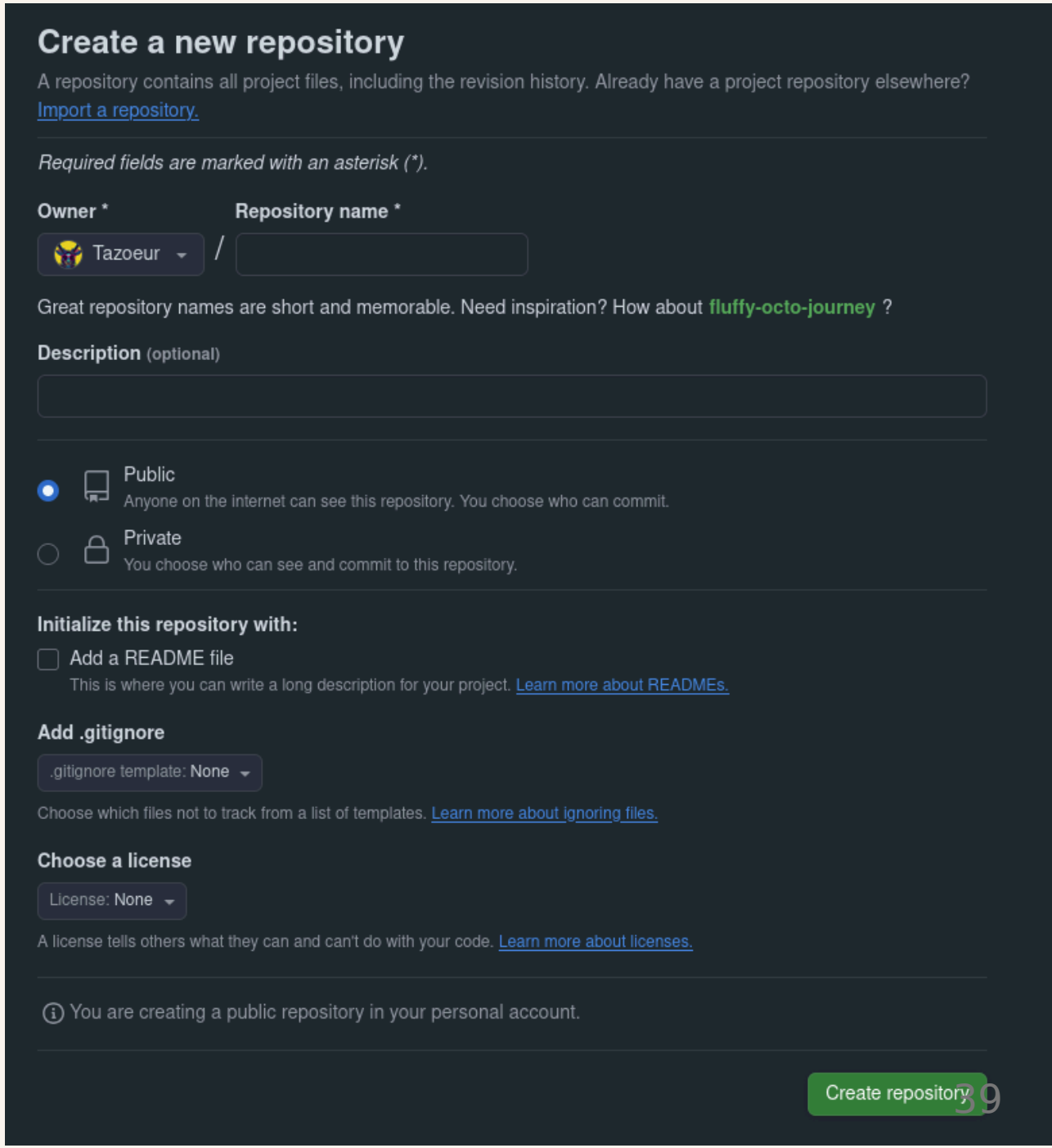


Github

Une fois votre compte créé, vous allez pouvoir créer un nouveau *repository*.

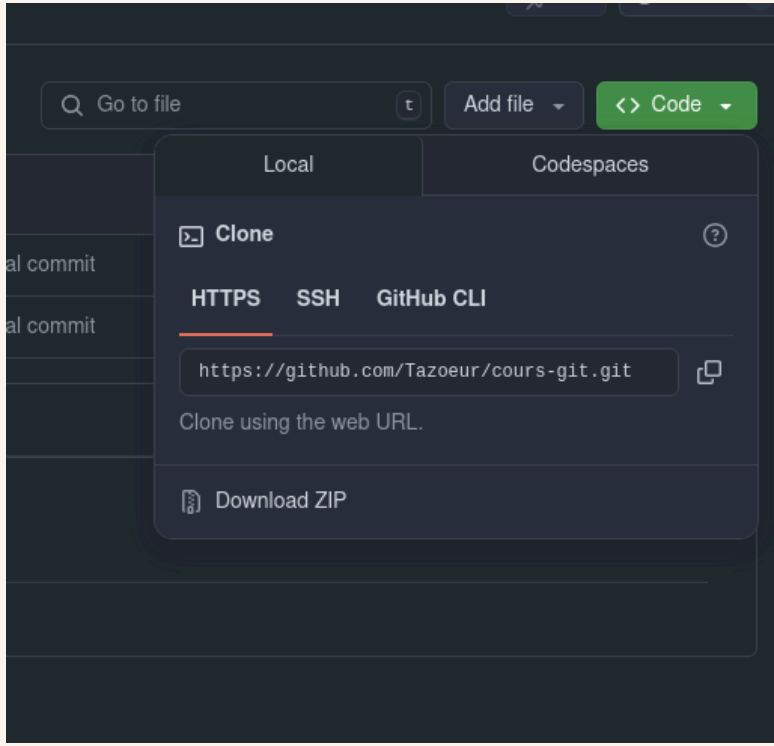
Les fichiers `README.md` sont très importants car ils permettent à quelqu'un qui ne connait pas votre projet d'en saisir les tenants et les aboutissants.

Il ne faut pas négliger la licence lorsqu'un *repo* est publique. Pour de l'aide aller sur choosealicense.com



`git clone`

```
$ git clone git@github.com:Tazoeur/cours-git.git # ou alors https://...
Cloning into 'cours-git'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (4/4), 12.72 KiB | 12.72 MiB/s, done.
```



C'est quoi un *remote*

Avec la création de votre *repo* github, vous obtenez une url.
Cette url vous permet de lier votre *repository* local à ce *repository* distant; on dit que le *repository* github est un *remote*.

Très souvent, les *repos* n'ont qu'un seul remote, et plusieurs personnes `pull` et `push` depuis et vers ce *repo*.

Le nom du remote est souvent `origin`.

```
$ git remote -v
origin  git@github.com:Tazoeur/cours-git.git (fetch)
origin  git@github.com:Tazoeur/cours-git.git (push)
```

`git push`

Imaginons que vous ayez travaillé sur le projet, vous avez *commit* votre travail et vous voulez le partager avec le reste du monde.

Vous allez **pousser** votre travail sur le remote.

```
$ git push origin main
Enumerating objects: 12, done.
Counting objects: 100% (12/12), done.
Delta compression using up to 8 threads
Compressing objects: 100% (11/11), done.
Writing objects: 100% (12/12), 562.99 KiB | 4.94 MiB/s, done.
Total 12 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'main' on GitHub by visiting:
remote:      https://github.com/Tazoeur/cours-git/pull/new/main
remote:
```

`git pull`

Maintenant imaginons que quelqu'un d'autre a travaillé sur le projet, et vous aimeriez récupérer sur votre machine le travail qui a été fait.

```
$ git pull origin main
remote: Enumerating objects: 19, done.
remote: Counting objects: 100% (19/19), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 18 (delta 3), reused 18 (delta 3), pack-reused 0 (from 0)
Unpacking objects: 100% (18/18), 569.84 KiB | 1.54 MiB/s, done.
From github.com:Tazoeur/cours-git
 * branch            main       -> FETCH_HEAD
   658324e..da644a2  main       -> origin/main
Updating 658324e..da644a2
Fast-forward
 images/decentralised.png        | Bin 0 -> 199385 bytes
 images/git-commit-branch.png    | Bin 0 -> 59767 bytes
 images/git-file-states.png      | Bin 0 -> 21550 bytes
 presentation.md                 | 701 ++++++++++++++++++++++
 10 files changed, 701 insertions(+)
 create mode 100644 images/decentralised.png
 create mode 100644 images/github-create-repo.png
 create mode 100644 images/github-git-clone.png
 create mode 100644 presentation.md
```

C'est quoi un *fork*

Un fork est une copie d'un *remote* maintenu par un autre compte github.

Par exemple vous avez *clone* ce cours et l'avez amélioré. Vous aimeriez maintenant que votre travail soit inclu et disponible pour tout le monde.

Mais vous n'avez pas les droits pour pousser sur ce *repository*, vous allez donc créer un *fork*.

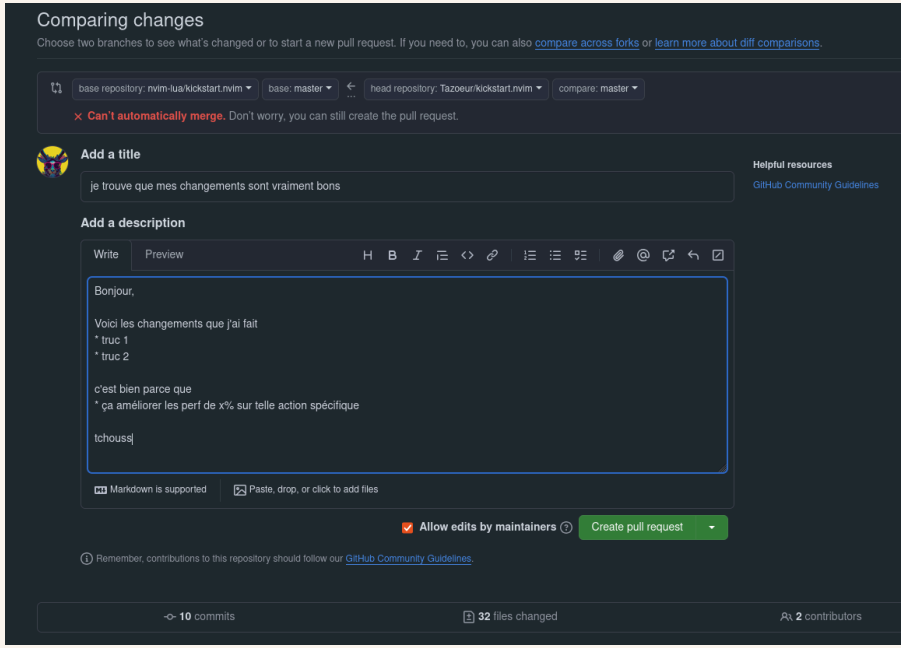
Généralement vous allez *fork* un projet pour y contribuer sans en prendre en charge la maintenance du projet.

Vous allez alors travailler avec plusieurs *remote*, et il vous sera possible de *pull* les modifications faites dans les différents *remotes*.

C'est quoi une *pull request*

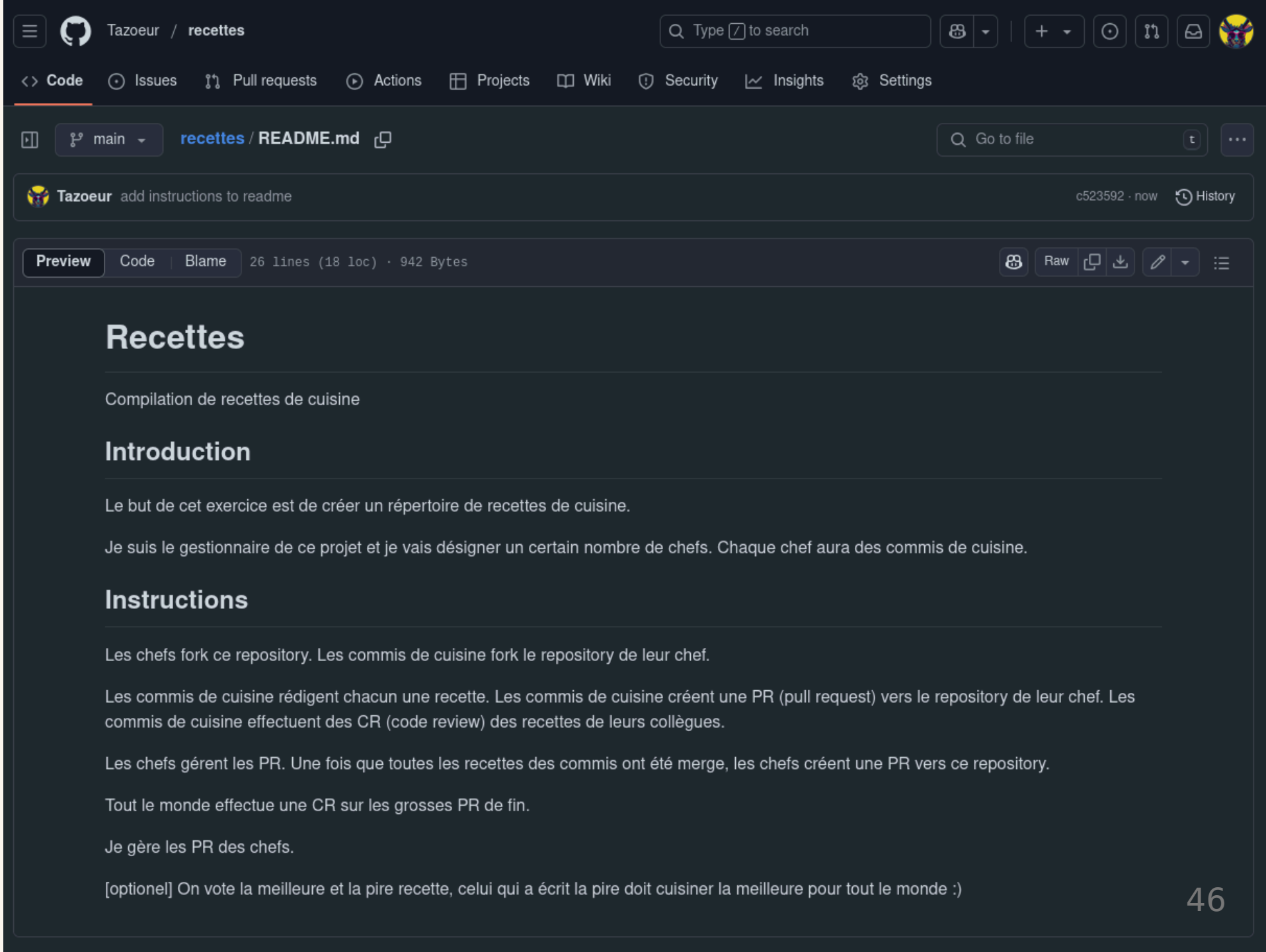
Le processus qui permet de proposer des changements à un auteur d'un repository *forké* s'appelle ***pull request*** et est géré de la même façon qu'une *issue* github.
Donc c'est une page spéciale sur la vue du projet, identifiée via un numéro unique, et sur laquelle il est possible de discuter des modifications qui sont faites.

Dans le processus d'acceptation d'une PR, on trouve généralement une *code review*, c'est à dire une relecture du code soumis dans la PR.



Exercice

lien

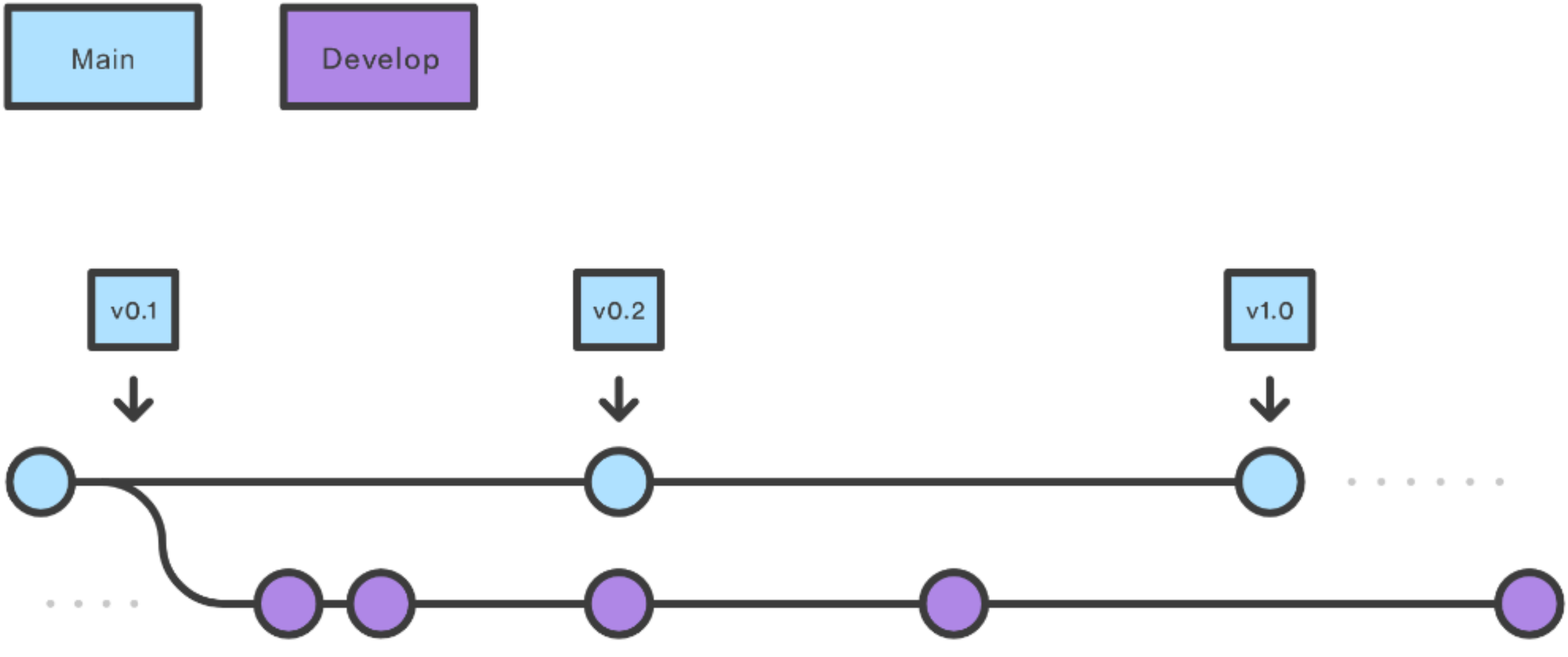


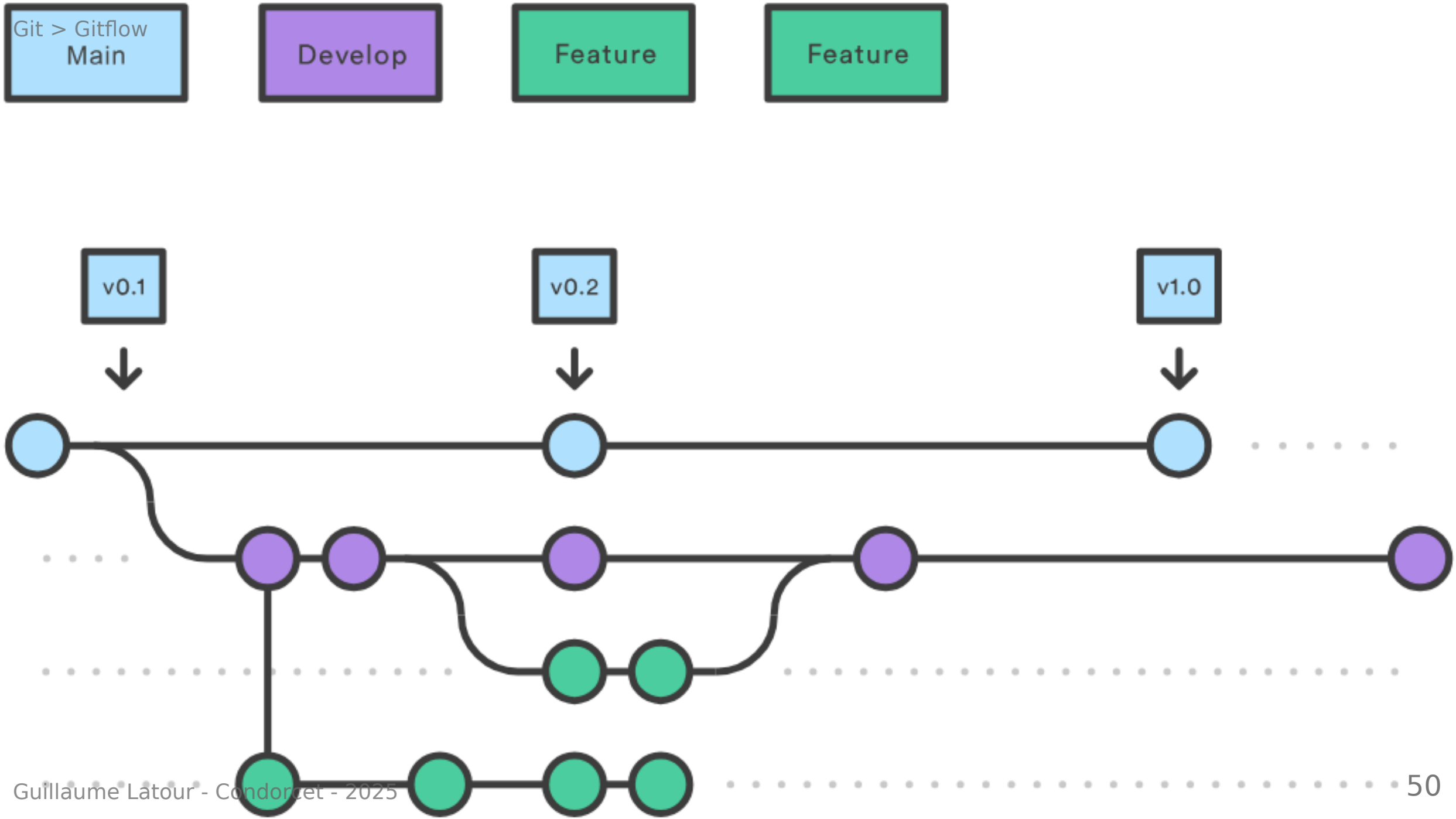
Gitflow

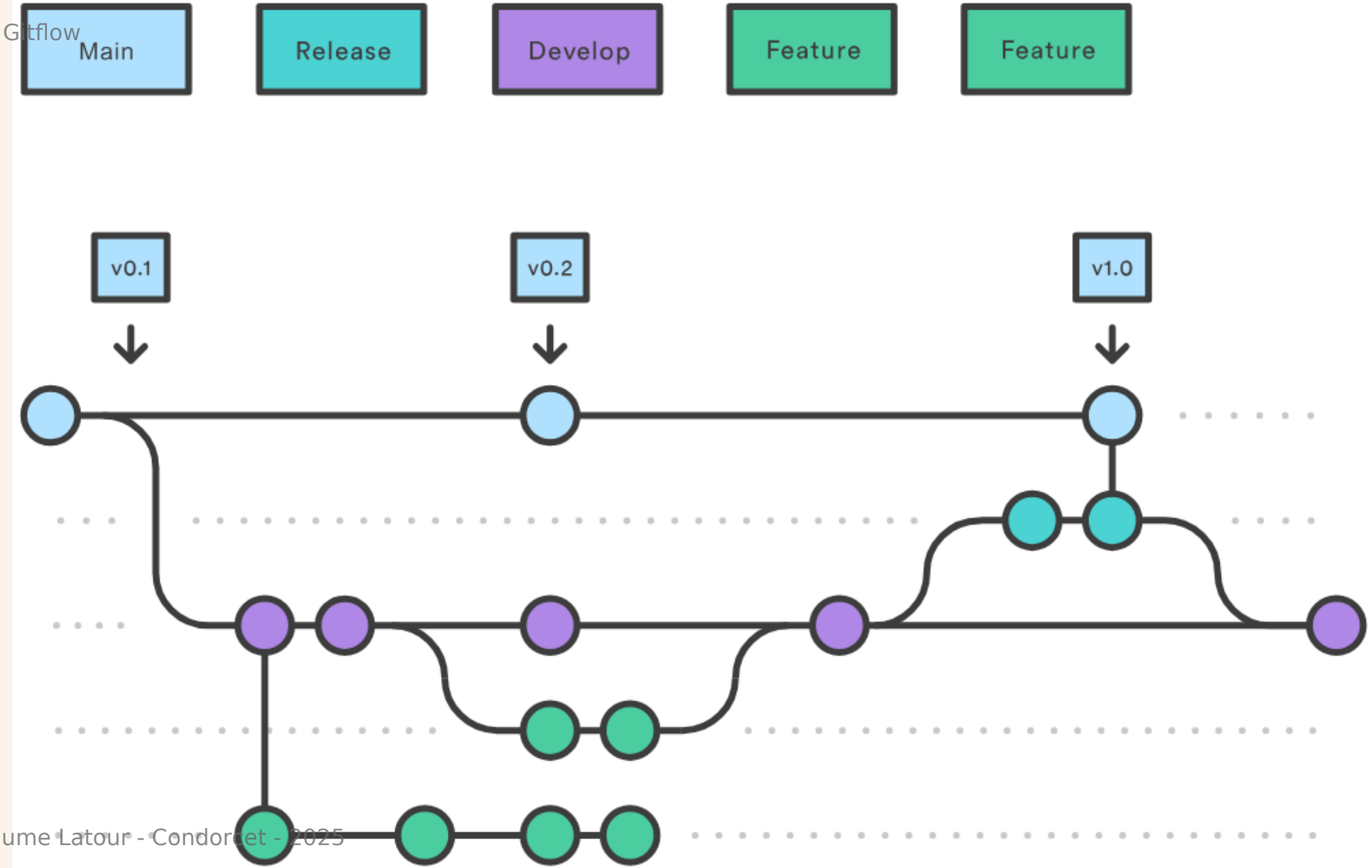
Gitflow

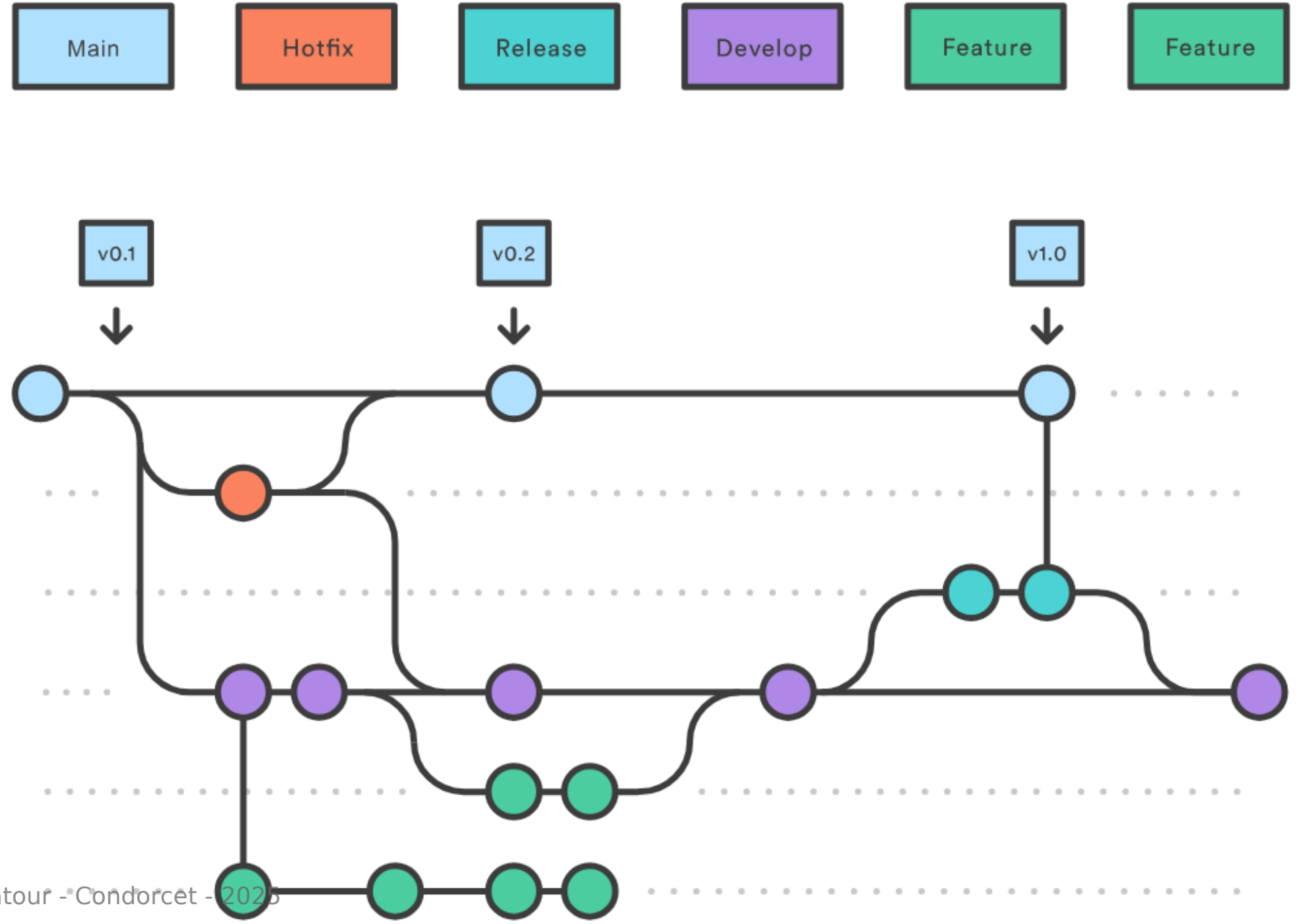
C'est une façon de travailler avec git qui apporte structure et clareté.

On va se baser sur l'explication d'Atlassian.


Main
Develop
v0.1
v0.2
v1.0


Main
Develop
Feature
Feature
v0.1
v0.2
v1.0


Main
Release
Develop
Feature
Feature
v0.1
v0.2
v1.0


Main
Hotfix
Release
Develop
Feature
Feature
v0.1
v0.2
v1.0

Conclusion

Vous savez

- `add` des fichiers modifiés
- `commit` votre travail
- `push` vers un ***remote***
- `pull` depuis un ***remote***
- `status` pour voir l'état de votre ***repository***
- `log` pour voir l'historique des ***commits***
- `checkout` pour naviguer votre `HEAD` vers un ***commit***
- `switch` pour naviguer votre `HEAD` vers une **branche**
- `branch` pour créer des **branches**

Ce qui n'a pas été vu

- `git rebase`
- `git stash`
- `git submodules`
- git hooks
- `git reset`
- résoudre des conflits
- sign with gpg key
- configuration via `~/.gitconfig`
- partial staging
- *cette liste n'est pas exhaustive*

Liens utiles

- pdf git cheatsheet
- oh shit git
- documentation
- jeu git branching