

04.



git

Maîtriser git pour un développement structuré et collaboratif

v2.6.0-beta.2

build: release 2.6.0-beta.2

build: build 2.6.0-beta.2

feat: dynamic directive arguments for v-on, v-bind and custom directives (#9372)

origin/dynamic-directive-arguments feat: dynamic args for custom directives

perf: improve scoped slots change detection accuracy (#9371)

test: test cases for v-on/v-bind dynamic arguments

refactor: v-bind dynamic arguments use bind helper

test: fix tests, resolve helper conflict

fix: fix middle modifier

feat: handle dynamic argument for v-bind.sync

origin/slot-optimization perf: improve scoped slots change detection

feat: dynamic directive arguments for v-bind and v-on

refactor: extend dom-props update skip to more all keys except value

fix: fix checkbox event edge case in Firefox

test: fix tests in IE/Edge

refactor: simplify timestamp check

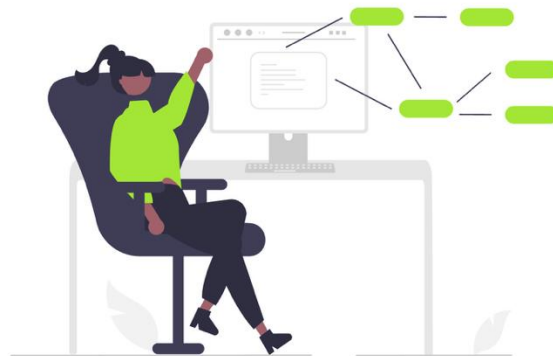
chore: update comment

fix: some edge case for should update to new version

Introduction.

Cette session vous apprendra à :

- Comprendre le fonctionnement de Git et son intérêt dans les projets collaboratifs
- Utiliser les commandes de base et avancées de Git
- Travailler efficacement avec GitHub, SourceTree et les workflows modernes (GitFlow, GitHub Flow)
- Adopter des bonnes pratiques de commit et de revue de code
- Gérer les conflits et maintenir un historique propre et compréhensible



Gestionnaires de versions

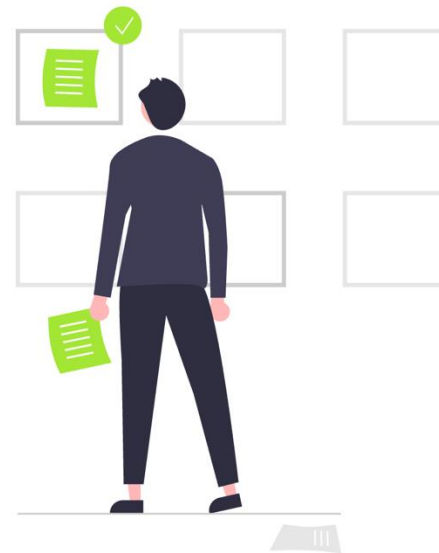
Introduction à git, comparatif avec SVN



Qu'est ce que le versioning ?

Un système de versionnage permet de suivre les modifications d'un projet dans le temps.

- Sauvegarder l'état d'un projet à **chaque étape** importante
- **Travailler à plusieurs** sans perdre d'informations
- Pouvoir **revenir en arrière** en cas de besoin

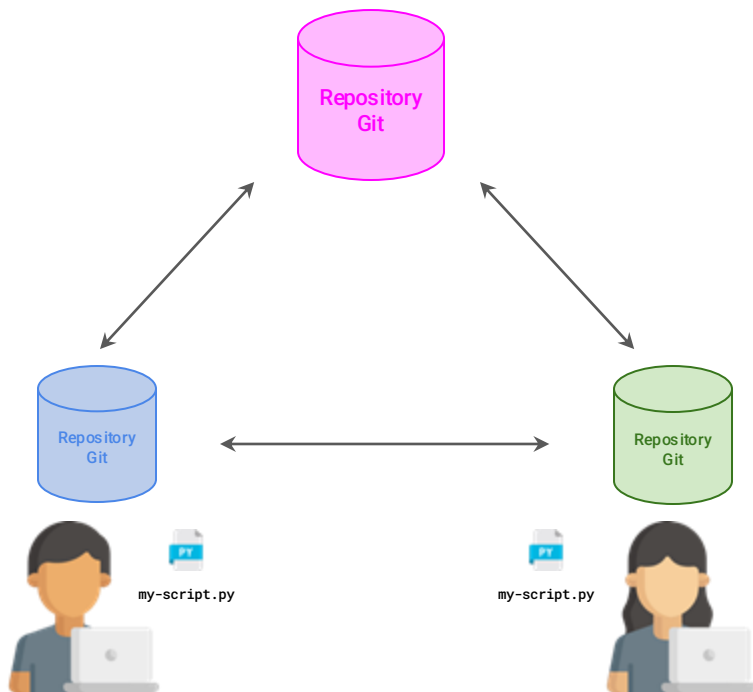


Qu'est ce que Git ?

Système de gestion de versions
décentralisé

Qu'est ce que Git ?

Système de gestion de versions **décentralisé**



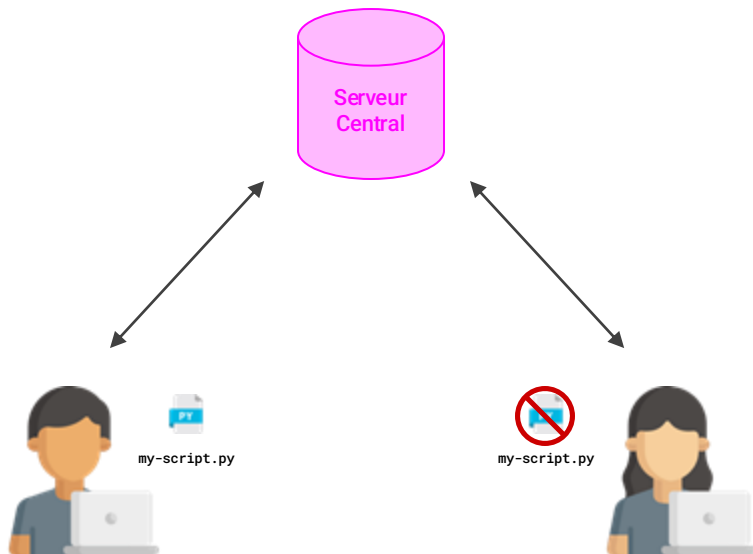
- ✅ Travail collaboratif réel, en **parallèle** avec des **copies locales** pour chaque développeur.
 - ✅ **Fusion des modifications simplifiée** et fiable, Git gère les conflits automatiquement.
 - ✅ Travail **hors ligne possible**, chaque développeur peut committer et consulter l'**historique localement**.
 - ✅ **Indépendance du serveur**, pas de dépendance au serveur pour effectuer des commits ou consulter l'historique.
 - ✅ **Gestion décentralisée**, chaque développeur a une copie complète du dépôt et peut travailler de manière autonome.
- 👉 **En résumé** : Git permet un travail flexible, parallèle et indépendant, avec une gestion efficace des fusions et de l'historique.

Qu'est ce que Git ?









SVN (Subversion)

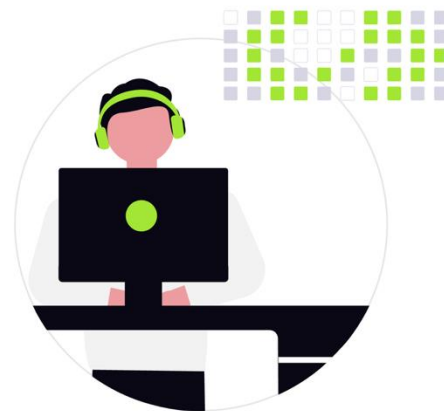
Système de gestion de versions **centralisé**



- ✅ **Premier pas vers la collaboration**, mais **pas de travail réellement en parallèle** (sauf avec risques de conflits).
 - ✅ **Fusion des modifications compliquée, peu fiable et souvent manuelle**, surtout si plusieurs personnes modifient le même fichier.
 - ✅ **Travail hors ligne impossible**, car chaque commit nécessite une connexion au serveur central.
 - ✅ **Dépendance forte au serveur** : si le serveur est indisponible, impossible de commit ou de récupérer les mises à jour.
- 👉 **En résumé** : SVN permet de travailler ensemble, mais avec des **contraintes fortes qui limitent le travail fluide en équipe**.

Pourquoi git est devenu la norme ?

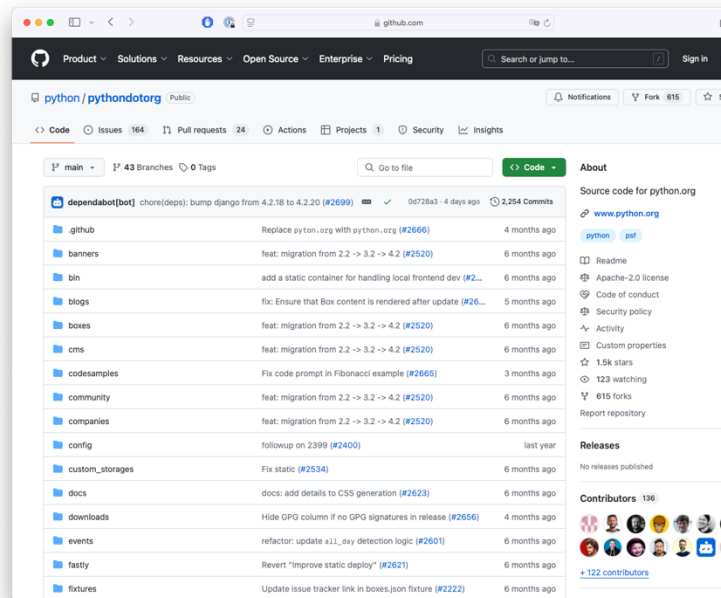
-  **Open-source & gratuit**
Tout le monde peut l'utiliser, l'intégrer, l'adapter. Aucune barrière à l'entrée, que ce soit pour un particulier ou une entreprise.
-  **Ultra-rapide et léger**
Opérations locales sans dépendance réseau (commit, log, diff...), ce qui le rend très efficace même sur des projets volumineux.
-  **Travail distribué**
Chaque collaborateur possède un clone complet du dépôt.
→ Travailler hors-ligne, revenir en arrière, créer des branches sans serveur.
-  **Adopté par tous les grands acteurs**
-  **Intégration CI/CD** (GitHub Actions, GitLab CI, Jenkins...)
-  **Compatibilité avec des outils** de sécurité, d'analyse de code et de gestion de projet



Quid de Github, Gitlab, etc. ?



- **Plateforme de dépôts Git** : elles hébergent des dépôts Git distants, permettant aux équipes de collaborer et de partager du code.
- **Collaboration simplifiée** : elles facilitent la collaboration en ligne, avec des outils pour examiner, discuter et fusionner les modifications.
- **Accès et contrôle** : elles offrent des contrôles d'accès pour gérer les permissions et autoriser ou restreindre l'accès au code source.
- **Suite d'outils complète** : intégration avec des outils DevOps pour automatiser le déploiement et les tests du code, suivi des bugs, suivi de projet...



Git de A à Z.

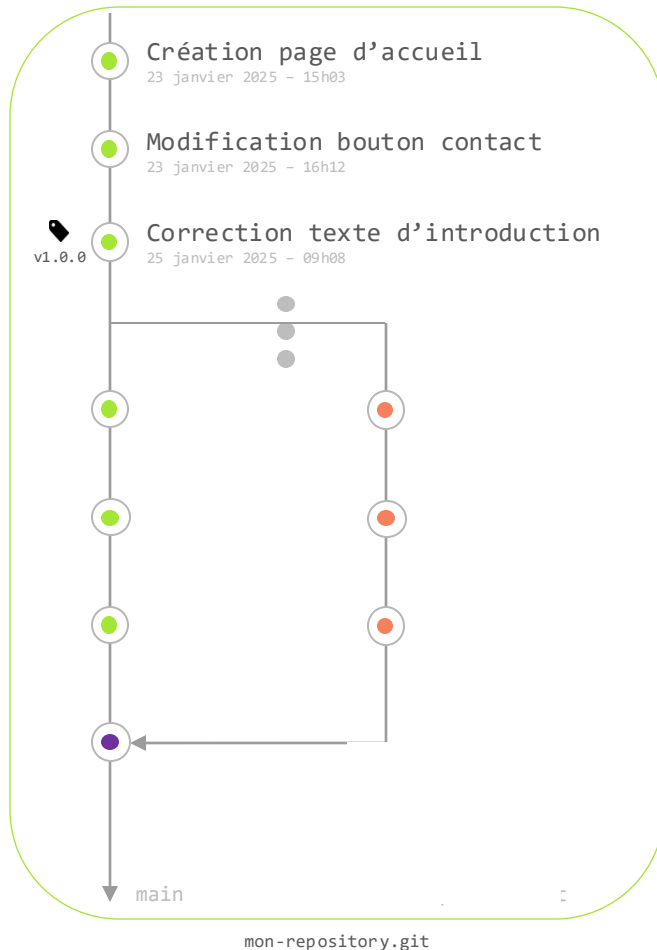
Repository : un dépôt où tout le code et l'historique des versions sont stockés

Commit : un enregistrement de changements apportés au code dans le dépôt

Tag : un marqueur utilisé pour identifier des points spécifiques de l'historique (souvent pour des versions ou des releases)

Branche : une version parallèle du projet permettant de travailler sur des fonctionnalités ou corrections sans affecter le code principal.

Merge : action de fusionner les modifications de deux branches différentes dans une seule.



A person with a beard, wearing a light-colored button-down shirt with the letters 'KRA' on the chest, is seated at a dark wooden table. They are using a pen to draw a technical diagram on a large sheet of paper. The diagram appears to be a schematic or a floor plan with various lines and shapes. The person's left hand is resting on a rolled-up sheet of paper. The background is dark and out of focus, with some blue light visible in the upper right corner.

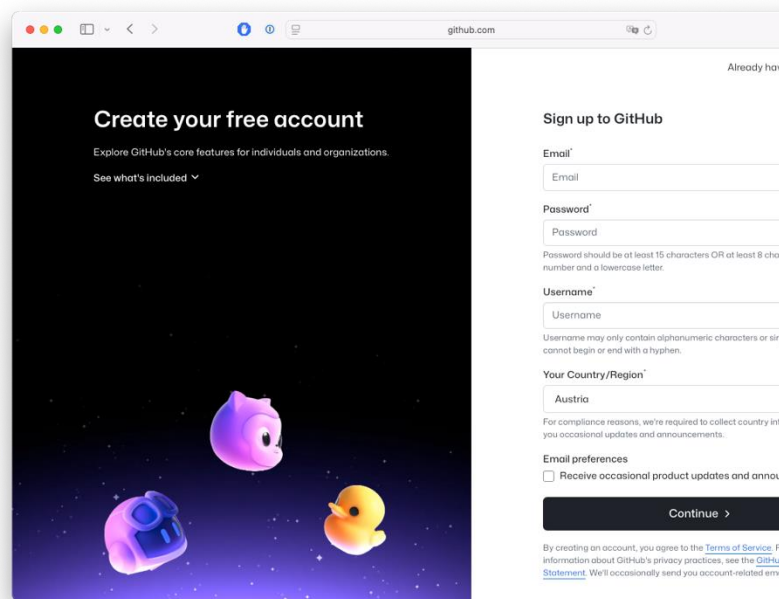
Installation et configuration

*Mise en place de git sur votre système,
installation d'un premier client GUI*

Créer un compte Github.

Créons votre compte sur la plateforme **Github**. C'est gratuit, vous prendrez part à *Github Classroom* pour les exercices et évals et vous vous en servirez dans tout votre parcours.

- Rendez-vous sur : <https://github.com/signup>
- Saisissez votre e-mail (perso)
- Choisissez un mot de passe (pas azerty123 !)
- Choisissez un **username**
→ C'est avec cette information que **vous serez reconnu publiquement** sur Github, incluant les futures entreprises avec lesquelles vous allez travailler.



Create your free account

Explore GitHub's core features for individuals and organizations.

See what's included ▾

Sign up to GitHub

Email*

Email

Password*

Password

Password should be at least 15 characters OR at least 8 characters with at least 1 uppercase letter, 1 lowercase letter, and 1 number.

Username*

Username

Username may only contain alphanumeric characters or hyphens. It cannot begin or end with a hyphen.

Your Country/Region*

Austria

For compliance reasons, we're required to collect country information for occasional updates and announcements.

Email preferences

☐ Receive occasional product updates and announcements

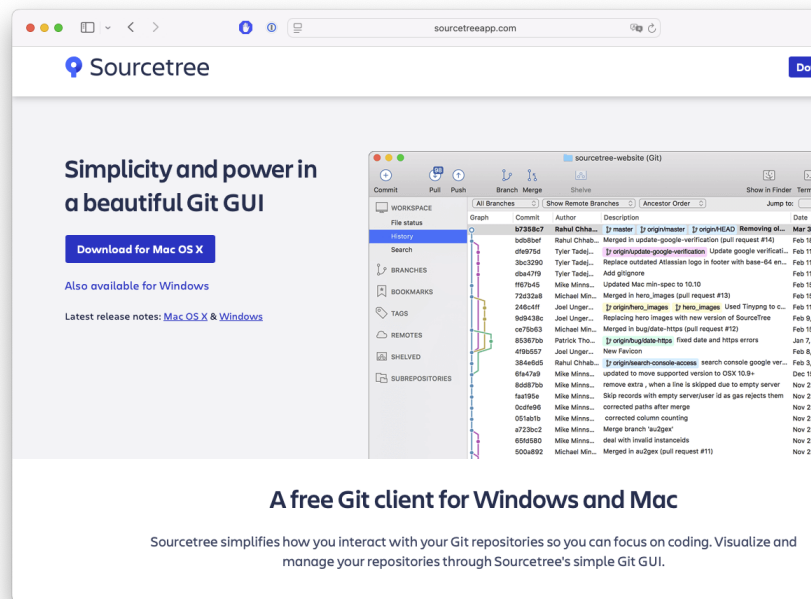
Continue >

By creating an account, you agree to the [Terms of Service](#). For information about GitHub's privacy practices, see the [GitHub Privacy Statement](#). We'll occasionally send you account-related emails.

Installer Sourcetree.

Sourcetree est un **client git** qui vous servira à gérer vos repositories.

- ➔ Rendez-vous sur :
<https://www.sourcetreeapp.com>
- ➔ Téléchargez la dernière version pour votre OS
- ➔ Lors de l'installation, Sourcetree vous proposera de créer un compte Atlassian (l'éditeur de Sourcetree) : vous pouvez l'ignorer.



Clé SSH vs HTTPS : sécurisez vos échanges.

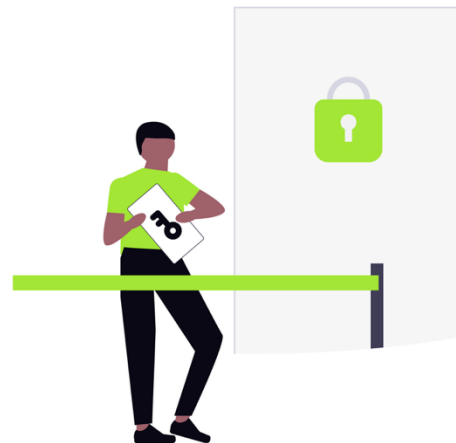
Pour interagir avec un dépôt distant (GitHub, GitLab...), Git utilise une méthode d'authentification. Deux options principales : **HTTPS** ou **SSH**.

🧠 Pourquoi utiliser une clé SSH ?

➡️ 🔒 **Connexion sécurisée sans mot de passe**
Plus besoin de saisir ses identifiants à chaque git push ou pull

➡️ 🧑💻 **Identifie votre machine de façon unique**
git reconnaît votre ordinateur grâce à votre clé publique

➡️ 🚀 **Fluide et automatisable**
Parfait pour les scripts, CI/CD, ou usage fréquent



Clé SSH vs HTTPS : sécurisez vos échanges.

Pour interagir avec un dépôt distant (GitHub, GitLab...), Git utilise une méthode d'authentification. Deux options principales : **HTTPS** ou **SSH**.

 Comment générer une clé SSH ?



Depuis **SourceTree**, aller dans « Actions » > « Ouvrir dans un terminal », puis entrez la commande :



Dans un terminal, entrez la commande :

```
1 ssh-keygen -t ed25519 -C "votre@email.com"
```



Dans les options de **SourceTree** (Outils > Options), onglet « Général », sélectionnez comment client SSH : « **OpenSSH** » puis dans le champ au dessus, choisissez votre clé privée.

Clé SSH vs HTTPS : sécurisez vos échanges.

Pour interagir avec un dépôt distant (GitHub, GitLab...), Git utilise une méthode d'authentification. Deux options principales : **HTTPS** ou **SSH**.

🔑 Comment ajouter ma clé à Github ?

➡ Dans un terminal, entrez la commande :

```
1 cat ~/.ssh/id_ed25519.pub
```

➡ Copiez la clé publique dans votre presse-papier, elle doit être de la forme :

ssh-ed25519 <votre clé> <votre email>

➡ Rendez-vous sur : <https://github.com/settings/keys>

➡ Cliquez sur « New SSH key »

➡ Saisissez un titre, collez votre clé publique dans le champ, puis validez.



Clé SSH vs HTTPS : sécurisez vos échanges.

Pour interagir avec un dépôt distant (GitHub, GitLab...), Git utilise une méthode d'authentification. Deux options principales : **HTTPS** ou **SSH**.

🔑 Testez la clé :

➡ Dans un terminal, entrez la commande :

```
1 ssh -T git@github.com
```



Si votre clé est bien installée, vous recevrez ce message :

*Hi **username**! You've successfully authenticated, but GitHub does not provide shell access.*

Configurez votre git.

Il est important de définir certains paramètres pour votre client git :

```
1 # Configurez votre identité
2 git config --global user.name "Jean Dupont"
3
4 # Configurez votre adresse e-mail
5 git config --global user.email "votre@email.com"
```

Commandes de base

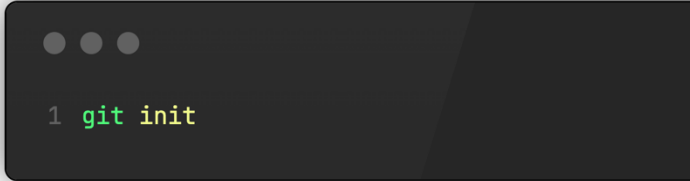
Comment effectuer les actions basiques au sein de git ?



Initialiser un repository.

Un repository peut être créé en local, sur votre machine, sans nécessairement être (pour le moment) lié à un serveur distant.

- La commande ci-contre **va transformer votre répertoire courant en un repository git**
- Un dossier `.git/` (masqué) va être créé pour contenir **l'ensemble de l'historique** de vos modifications
- A partir de cet instant, **vos modifications sont suivies**, même si vous ne décidez d'ajouter un serveur distant que dans quelques jours.

A dark-themed terminal window with three window control buttons (minimize, maximize, close) in the top left corner. The terminal shows a single line of code: `1 git init`.

```
1 git init
```

Cloner un repository.

Lorsque vous récupérez un repository depuis un dépôt distant (e.g. Github), vous **clonez** ce repository sur votre machine.

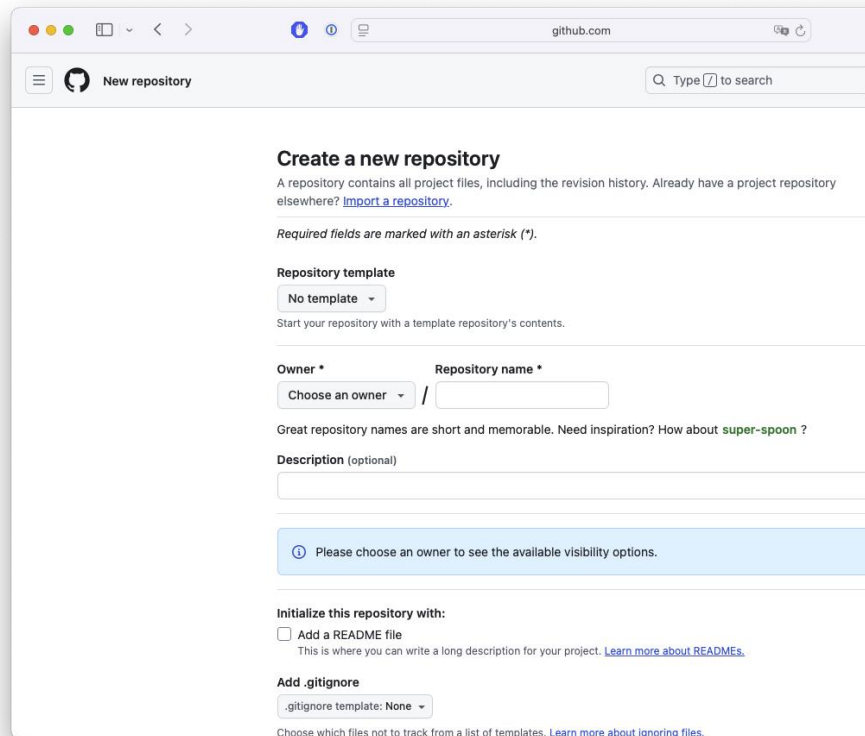
- La commande ci-contre **va créer une copie identique** du repository hébergé sur Github, sur votre machine
- A partir de cet instant, vous pourrez à tout moment synchroniser les modifications des auteurs avec votre copie locale et, si vous en avez les droits, héberger vos modifications.

```
1 git clone git@github.com:python/pythondotorg.git
```

Créer un repository sur Github.

Vous pouvez créer un nouveau repository sur Github. Il sera prêt à accueillir le code de votre projet.

- Sur la page d'accueil, cliquez sur « New » (colonne de gauche)
- Remplissez les informations du formulaire
- Une fois le repository créé, suivez les instructions à l'écran pour le récupérer sur votre machine.



The screenshot shows the 'New repository' page on GitHub. At the top, there's a search bar and a 'New repository' button. The main heading is 'Create a new repository'. Below it, a paragraph explains that a repository contains all project files, including revision history, and provides a link to 'import a repository' if one already exists elsewhere. A note states that required fields are marked with an asterisk (*). The 'Repository template' section has a 'No template' dropdown. Below this, the 'Owner' and 'Repository name' fields are shown, with a 'Choose an owner' dropdown and a text input for the name. A message below these fields says 'Great repository names are short and memorable. Need inspiration? How about super-spoon?'. The 'Description (optional)' field is a text input. A blue banner with an information icon and text says 'Please choose an owner to see the available visibility options.' The 'Initialize this repository with:' section has a checkbox for 'Add a README file' with a link to 'Learn more about READMEs'. Below that, the '.gitignore' section has a dropdown set to 'None' and a link to 'Learn more about ignoring files'.

Les différents états d'un fichier dans Git.

Un fichier dans Git peut passer par plusieurs états : comprendre ces états est essentiel pour bien utiliser Git.



Non suivi (*Untracked*)

Le fichier existe sur votre disque, mais Git ne le suit pas encore

Exemple : vous venez de créer notes.txt



Suivi mais modifié (*Modified*)

Le fichier est suivi, mais vous avez modifié son contenu depuis le dernier commit.



Indexé (*Staged*)

Le fichier est prêt à être enregistré dans l'historique

Vous l'avez ajouté avec `git add`

Il est en attente de commit dans la *Staging Area*



Committé (*Committed*)

Le fichier est officiellement enregistré dans l'historique local

Le commit contient un instantané du fichier, toujours local à ce stade



Pushé (*Pushed*)

Le commit a été envoyé vers un dépôt distant (GitHub, GitLab...)

Le fichier est maintenant visible par vos collaborateurs

Les différents états d'un fichier dans Git.

```
1 # On crée un fichier basique. Le fichier est "Untracked"
2 echo "Hello World!" > README.md
3
4 # On ajoute le fichier à l'index. Il devient "Staged"
5 git add README.md
6
7 # On commit les modifications du fichier dans l'historique local. Il est "Committed"
8 git commit -m "Ma super modification"
9
10 # On envoie les modifications vers le serveur distant. Il est "Pushed"
11 git push
12
13 # On modifie le fichier, il devient "Modified"
14 echo "Hello World Bis!" > README.md
```


L'historique des modifications.

Comprendre ce qui a été modifié, quand, et par qui, est l'un des principaux avantages de Git.

→ Voir la liste des commits

La commande de logs permet d'afficher les commits du plus récent au plus ancien.

Usage: `git log`

Affiche :

- Le **hash** du commit (son identifiant unique)
- Le **nom** de l'auteur
- La **date** de l'heure de création
- Le **message** indiqué par l'auteur

```
git log
commit 8a26eb42adb303f4adc7ef56e300f14c5992aa68 (HEAD -> main, origin/main, origin/HEAD)
Author: Jon Church <me@jonchurch.com>
Date: Thu Dec 12 17:27:03 2024 -0500

    add security.md from afcd5bc (#5946)

commit f299b52f39486275a9e6483b60a410e06520c538 (tag: 4.17.21, origin/4.17)
Author: Benjamin Tan <benjamin@dev.ofcr.se>
Date: Sat Feb 20 23:33:48 2021 +0800

    Bump to v4.17.21

commit c4847ebe7d14540bb28a8b932a9ce1b9ecbfee1a
Author: Michał Lipiński <mylith@gmail.com>
Date: Tue Jan 26 23:17:05 2021 +0100

    Improve performance of `toNumber`, `trim` and `trimEnd` on large input strings

    This prevents potential ReDoS attacks using `_.toNumber` and `_.trim*` as potential attack vectors.

    Closes #5065.
```

```
1 git log --oneline      # Résumé en une ligne par commit
2 git log --graph       # Arborescence des branches
3 git log --author="Nom" # Filtrer par auteur
```

L'historique des modifications.

Comprendre ce qui a été modifié, quand, et par qui, est l'un des principaux avantages de Git.

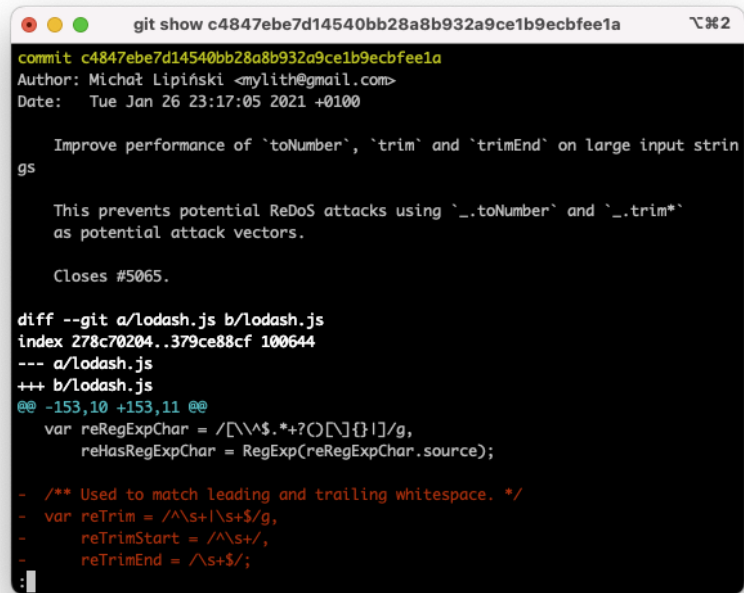
→ Voir les détails d'un commit

Affiche le contenu modifié dans un commit précis.

Usage: `git show <hash>`

Affiche :

- **Les métadonnées** du commit (hash, date, auteur, message)
- **Le diff** entre le commit précédent et ce commit



```
git show c4847ebe7d14540bb28a8b932a9ce1b9ecbfee1a  1%2

commit c4847ebe7d14540bb28a8b932a9ce1b9ecbfee1a
Author: Michał Lipiński <mylith@gmail.com>
Date:   Tue Jan 26 23:17:05 2021 +0100

    Improve performance of `toNumber`, `trim` and `trimEnd` on large input strings

    This prevents potential ReDoS attacks using `_.toNumber` and `_.trim` as potential attack vectors.

    Closes #5065.

diff --git a/lodash.js b/lodash.js
index 278c70204..379ce88cf 100644
--- a/lodash.js
+++ b/lodash.js
@@ -153,10 +153,11 @@
var reRegExpChar = /[\\^$.*+?()[\]{}|]/g,
    reHasRegExpChar = RegExp(reRegExpChar.source);

- /** Used to match leading and trailing whitespace. */
- var reTrim = /^\s+|\s+$/g,
-     reTrimStart = /^\s+/,
-     reTrimEnd = /\s+$/;
:
```

L'historique des modifications.

Comprendre ce qui a été modifié, quand, et par qui, est l'un des principaux avantages de Git.

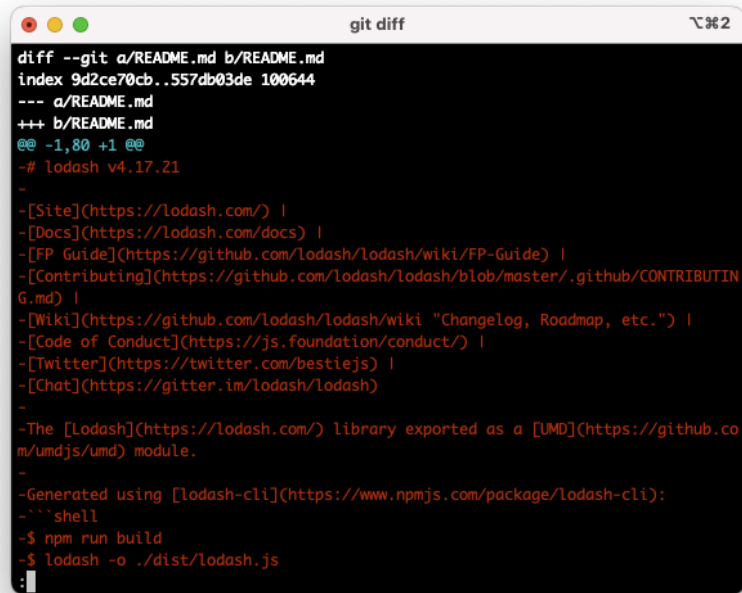
→ Voir ce qui a changé avant de committer

Compare un fichier avec sa dernière version committée.

Usage: `git diff`

Affiche :

→ Un état avant/après du contenu du fichier.



```
diff --git a/README.md b/README.md
index 9d2ce70cb..557db03de 100644
--- a/README.md
+++ b/README.md
@@ -1,80 +1 @@
-# lodash v4.17.21
-
-[Site](https://lodash.com/) |
-[Docs](https://lodash.com/docs) |
-[FP Guide](https://github.com/lodash/lodash/wiki/FP-Guide) |
-[Contributing](https://github.com/lodash/lodash/blob/master/.github/CONTRIBUTING.md) |
-[Wiki](https://github.com/lodash/lodash/wiki "Changelog, Roadmap, etc.") |
-[Code of Conduct](https://js.foundation/conduct/) |
-[Twitter](https://twitter.com/bestiejs) |
-[Chat](https://gitter.im/lodash/lodash)
-
-The [Lodash](https://lodash.com/) library exported as a [UMD](https://github.com/umdjs/umd) module.
-
-Generated using [lodash-cli](https://www.npmjs.com/package/lodash-cli):
-```shell
-$ npm run build
-$ lodash -o ./dist/lodash.js
:
```

Comprendre git blame : Qui a changé quoi ?.

Quand on tombe sur une ligne de code incompréhensible ou un bug, git blame permet d'identifier **qui a modifié cette ligne, quand, et dans quel commit**.

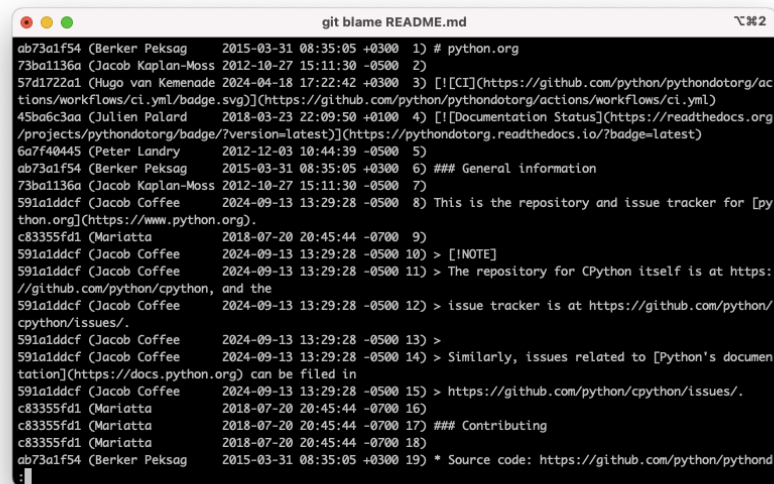
➡ Que fait git blame ?

Affiche, pour chaque ligne d'un fichier :

- ➔ L'auteur du changement
- ➔ Le commit d'origine
- ➔ La date

Très utile pour comprendre l'historique ligne par ligne

⚠ **À éviter : le "blame-shaming"** : Le but de git blame n'est pas d'accuser, mais de comprendre. L'auteur d'un bug a souvent fait de son mieux avec les infos du moment.



```
git blame README.md
ab73a1f54 (Berker Peksag 2015-03-31 08:35:05 +0300 1) # python.org
73ba1136a (Jacob Kaplan-Moss 2012-10-27 15:11:30 -0500 2)
57d1722a1 (Hugo van Kemenade 2024-04-18 17:22:42 +0300 3) [!CI](https://github.com/python/pythonondotorg/actions/workflows/ci.yml/badge.svg)](https://github.com/python/pythonondotorg/actions/workflows/ci.yml)
45ba6c3aa (Julien Palard 2018-03-23 22:09:50 +0100 4) [!Documentation Status](https://readthedocs.org/projects/pythonondotorg/badge/?version=latest)](https://pythonondotorg.readthedocs.io/?badge=latest)
6a7f40445 (Peter Landry 2012-12-03 10:44:39 -0500 5)
ab73a1f54 (Berker Peksag 2015-03-31 08:35:05 +0300 6) ### General information
73ba1136a (Jacob Kaplan-Moss 2012-10-27 15:11:30 -0500 7)
591a1ddcf (Jacob Coffee 2024-09-13 13:29:28 -0500 8) This is the repository and issue tracker for [python.org](https://www.python.org).
c83355fd1 (Mariatta 2018-07-20 20:45:44 -0700 9)
591a1ddcf (Jacob Coffee 2024-09-13 13:29:28 -0500 10) > [!NOTE]
591a1ddcf (Jacob Coffee 2024-09-13 13:29:28 -0500 11) > The repository for CPython itself is at https://github.com/python/cpython, and the
591a1ddcf (Jacob Coffee 2024-09-13 13:29:28 -0500 12) > issue tracker is at https://github.com/python/cpython/issues/.
591a1ddcf (Jacob Coffee 2024-09-13 13:29:28 -0500 13) >
591a1ddcf (Jacob Coffee 2024-09-13 13:29:28 -0500 14) > Similarly, issues related to [Python's documentation](https://docs.python.org) can be filed in
591a1ddcf (Jacob Coffee 2024-09-13 13:29:28 -0500 15) > https://github.com/python/cpython/issues/.
c83355fd1 (Mariatta 2018-07-20 20:45:44 -0700 16)
c83355fd1 (Mariatta 2018-07-20 20:45:44 -0700 17) ### Contributing
c83355fd1 (Mariatta 2018-07-20 20:45:44 -0700 18)
ab73a1f54 (Berker Peksag 2015-03-31 08:35:05 +0300 19) * Source code: https://github.com/python/pythonondotorg
```

```
1 git blame -L 10,20 fichier.py # Blâme uniquement les lignes 10 à 20
2 git blame --since="2 weeks ago" # Limite à une période
```



Qui a fait ça ?

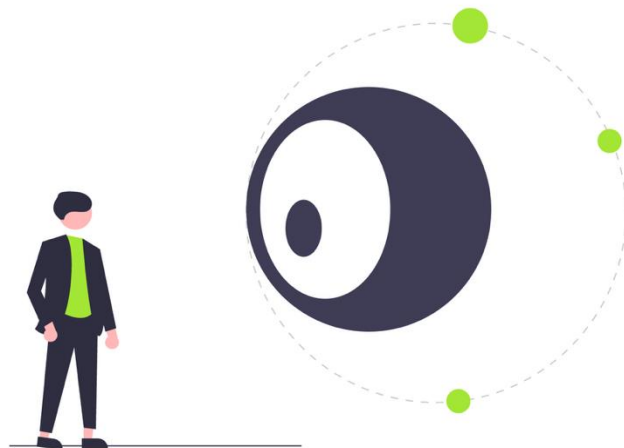
👁️ Clonez le repository :

<https://github.com/python/pythondotorg>

Quelqu'un a récemment ajouté un champ *sponsor_id* dans une des API...

Le fichier est **sponsors/api.py**

- Qui a fait ça ?
- Quand ?
- Dans quel commit ?
- Était-ce lié à un problème ?
- A-t-il modifié autre chose ?



Ignorer des fichiers.

Il n'est pas recommandé de **tout** suivre dans un système de gestion de version. Certains éléments ne doivent pas être committé, ni pushé vers un serveur.

→ **Les configurations** : cela permettrait de retrouver des clés API, des infos sensibles...

→ **Les dépendances** : seul un manifeste (*requirements.txt*) est pushé. Les dépendances ne sont pas suivies.

→ **Les fichiers de debug** : les logs, les fichiers de tests, ne sont pas à inclure dans le suivi de version.

→ **Les fichiers temporaires** de votre système

Pour cela, ils sont ajoutés dans un fichier `.gitignore` sous forme de chemin (1 par ligne).

```
1  ### Flask ###
2  instance/*
3  !instance/.gitignore
4  .webassets-cache
5  .env
6
7  ### Flask.Python Stack ###
8  # Byte-compiled / optimized / DLL files
9  __pycache__/*
10 *.py[co]
11 *$py.class
12
13 # C extensions
14 *.so
15
16 # Distribution / packaging
17 .Python
18 build/
19 develop-eggs/
20 dist/
21 downloads/
22 eggs/
23 .eggs/
24 lib/
25 lib64/
26 parts/
```

→ Générateur de `.gitignore` :
www.gitignore.io

Travailler avec des branches.

Git permet de travailler en parallèle grâce aux branches, sans bloquer le développement principal. Chaque branche est comme une version indépendante du projet.

- Une branche est un **pointeur** vers un commit.
- Elle permet **d'isoler une fonctionnalité**, une **correction**, une **expérimentation**.
- On peut **fusionner** ensuite dans la branche principale (main, develop...).

Commandes utiles :

```
1 git branch # Affiche la liste des branches locales
2 git branch "ma-nouvelle-branche" # Crée une branche sans y basculer
3 git checkout "ma-branche-existante" # Bascule vers une branche existante
4 git checkout -b "ma-nouvelle-branche" # Crée la branche et bascule dessus
```

Travailler avec des branches.

Une fois qu'une fonctionnalité est terminée dans une branche, il faut la fusionner dans la branche principale pour l'intégrer au projet. Pour cela : `git merge`.

→ Le **merge** s'effectue toujours depuis la branche **réceptrice** (ou de **destination**)

→ Merge via la commande :

```
1 git merge "ma-fonctionnalite-terminee"
```


Travailler avec des branches.

3 issues possibles à un merge :

- **Pas de conflits**
 - Git fusionne automatiquement
- **Modifications dans des zones séparées**
 - Git fusionne automatiquement sans intervention
- **Modifications sur les mêmes lignes**
 - Conflit à résoudre manuellement

En cas de conflits :

CONFLICT (content): Merge conflict in app.py

Une fois le conflit résolu :

```
1 git add app.py
2 git commit
```

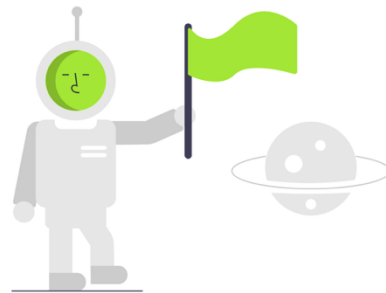
```
13
14 /**
15  * Prints the welcome message
16  */
17 Accept Current Change | Accept Incoming Change | Accept Both Changes | Compar
18 <<<<<<< HEAD (Current Change)
19 function printMessage(showUsage, message) {
20     console.log(message);
21 }
22 =====
23 function printMessage(showUsage, showVersion) {
24     console.log("Welcome To Line Counter");
25     if (showVersion) {
26         console.log("Version: 1.0.0");
27     }
28 }
29 >>>>>> theirs (Incoming Change)
30 if (showUsage) {
31     console.log("Usage: node base.js <file1> <fil
32 }
33 /**
```

φ You, 20 seconds ago Ln 11, Col 26 Spaces: 4 UTF-8 CRLF

Bonnes pratiques de merge.

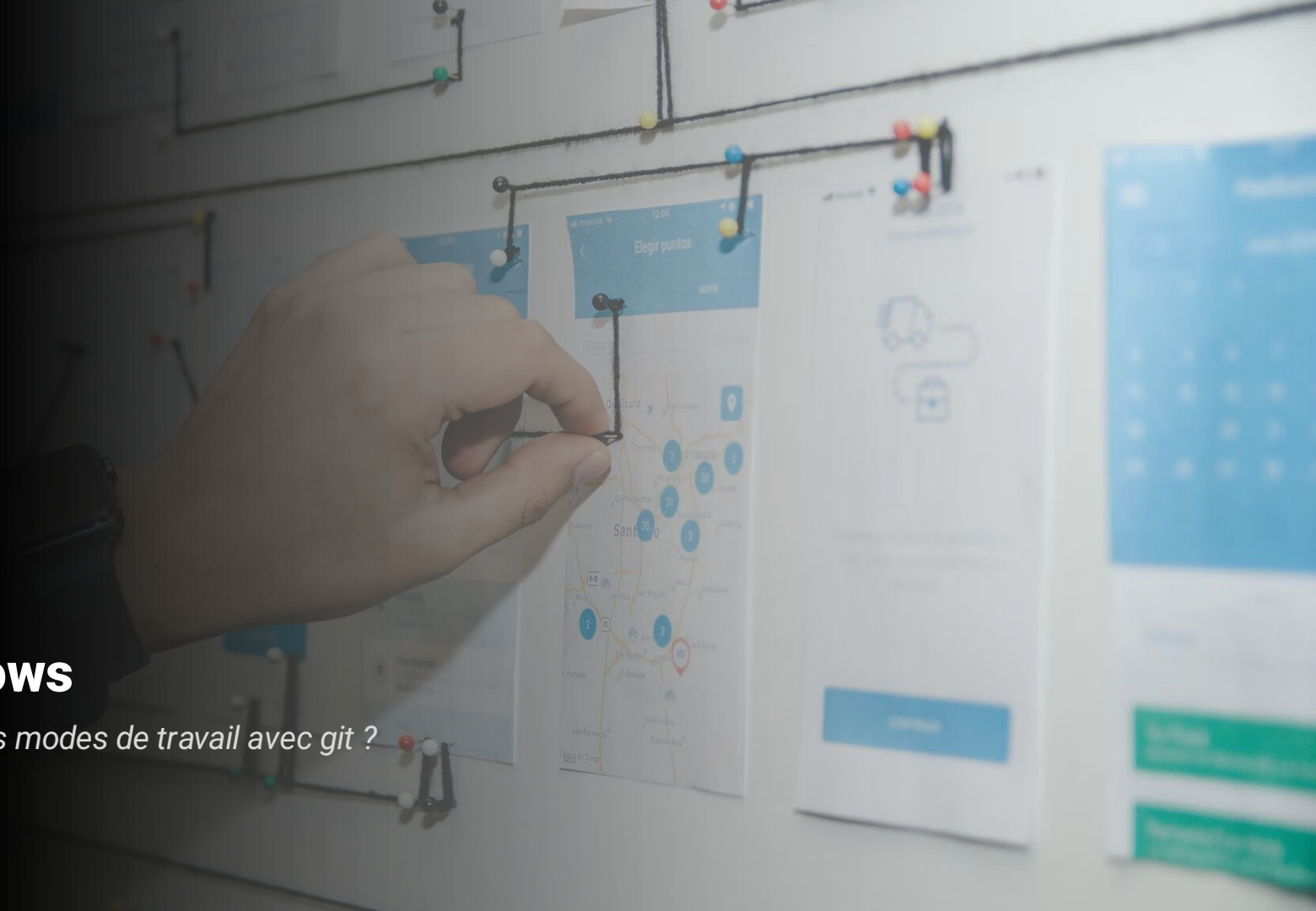
✓ Bonnes pratiques :

- Toujours synchroniser la branche source avant le merge via un `git push`
- Toujours mettre à jour la branche de destination avant le merge (`git pull`)
- Tester après un merge
- Supprimer la branche fusionnée si elle n'est plus utile via :
`git branch -d ma-branche`
- Tant que vous ne poussez rien sur le serveur, tout est en local ! Il n'y a aucun risque de tout casser (tant que vous ne faites pas un `git push`).



Workflows

Quels sont les modes de travail avec git ?



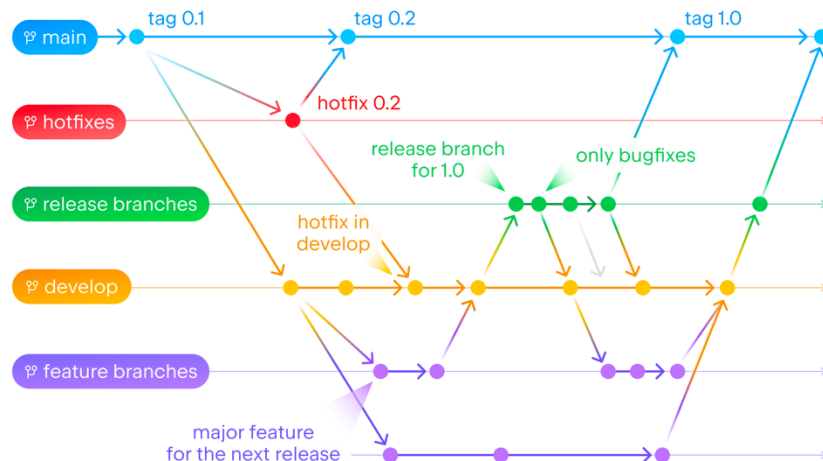
Pourquoi un workflow ?

Un workflow définit **comment** une équipe utilise Git. Il structure la collaboration et **évite les chaos de branches**.

- 📁 **Organiser clairement** le travail en équipe
- 🔧 **Clarifier les étapes** : développement, test, production
- 👉 **Faciliter les revues de code** et l'intégration continue
- 📄 **Standardiser les contributions** avec des **règles claires**
- 🚒 **Réagir rapidement aux bugs en production**

Gitflow.

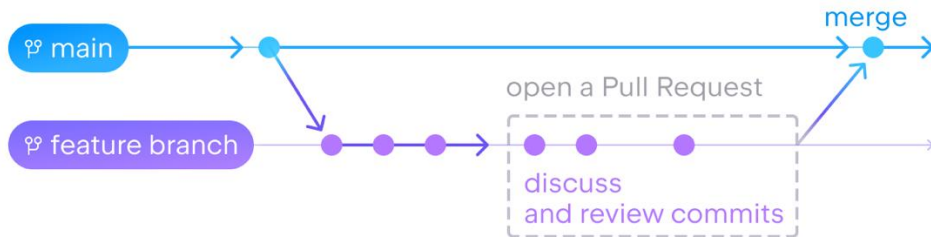
Gitflow propose un **workflow rigoureux**, parfait pour les équipes stables avec des cycles de release définis.



- **main** est uniquement pour la production
- **develop** est pour la version en développement
- Les branches **features** sont créées depuis **develop**
- Les branches **hotfix** sont créées depuis **main**
- Les branches **releases** sont créées depuis **develop**

Githubflow.

GitHub Flow privilégie la **rapidité**, la **CI/CD** et les **PRs**. Il repose sur une seule branche stable (**main**).



- **main** est uniquement pour la production
- Le développement est effectué dans des branches **features** qui sont ensuite mergées directement dans **main**
- Des Pull Requests sont utilisées pour vérifier ce qui va être injecté dans **main** (code review, contrôles qualités...)
- Les releases sont tagguées dans **main**

Trunk based.



- Tous les commits sont effectués dans une même branche, généralement appelée **trunk**
- La branche **trunk** est toujours « production-ready »
- Les développeurs effectuent une batterie de tests avant chaque commits dans **trunk**
- Tous les changements dans **trunk** peuvent être audités après-coup



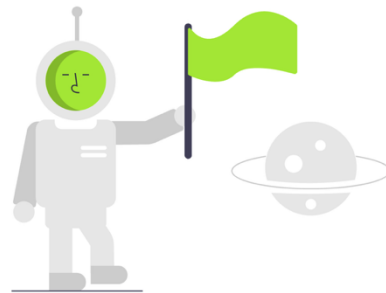
Commits et conventions

Comment bien committer son code ?

Bien committer : les bonnes pratiques.

✓ Un historique propre est un projet sain :

- **Un commit = une idée**
(un bug ? Une feature ? → un objet à la fois)
- **Jamais** de commit « **fourre-tout** »
- Limiter la taille du commit à 10-15 fichiers **maximum**
- Privilégier les commits dits **atomiques** (très peu de modifications)
- **Commits fréquents = historique lisible**
→ Gardez les étapes intermédiaires compréhensibles



Conventional Commits : une convention pour tous.

Les **conventional commits** standardisent la manière d'écrire les messages pour les rendre lisibles et exploitables automatiquement.

`<type>(<scope>) : <message>`

- **Type** : nature du changement (fonctionnalité, bug...)
- **Scope** (optionnel) : la zone du projet concernée
- **Message** : un résumé clair et concis des modifications

Conventional Commits : une convention pour tous.

```
<type>(<scope>) : <message>
```

Une liste pré-définie de types acceptés :

- **feat** : nouvelle fonctionnalité
- **fix** : un correctif
- **refactor** : une réécriture de code sans changement fonctionnel
- **docs** : une modification de la documentation uniquement
- **test** : une modification impactant uniquement les tests (unitaires, E2E...)
- **chore** : une tâche technique (CI, build, ...)
- **style** : formatage, indentation...

Conventional Commits : une convention pour tous.

```
<type>(<scope>) : <message>
```

Le scope indique la zone dans laquelle la modification a lieu.

La nomenclature dépend du projet, par exemple :

- **auth** : on note que la modification est dans la zone de l'authentification
- **users** : une modification étant apportée sur la gestion des utilisateurs
- **payment** : on a modifié quelque chose concernant le paiement

Conventional Commits : une convention pour tous.

`<type>(<scope>) : <message>`

Le message doit être clair, concis mais compréhensible. Par exemple :

- ✓ add new email validation on signup
- ✓ remove unused payment provider

Et non :

- ✗ fix stuff
- ✗ add a lot of elements in various part of the project

Conventional Commits : une convention pour tous.

Quelques exemples à suivre...

`feat(login): add password reset functionality`

`fix(auth): handle token expiration properly`

`docs(readme): update installation instructions`

`refactor(user): simplify avatar upload logic`

`style(ui): format dashboard header CSS`

`test(api): add tests for error handling in /orders route`

`chore(ci): update GitHub Actions to Node 20`

Knowledge Check.

Quel message de commit respecte la convention
Conventional Commits ?

A Added login feature

B feat(login): add login feature

C add login feat

D Fix login



Knowledge Check.

Quel message de commit respecte la convention
Conventional Commits ?

A

Added login feature

B

feat(login): add login feature

C

add login feat

D

Fix login



Knowledge Check.

Lequel de ces types n'est pas valide dans Conventional Commits ?

A fix

B improve

C chore

D refactor



Knowledge Check.

Lequel de ces types n'est pas valide dans Conventional Commits ?

A fix

B improve

C chore

D refactor



Knowledge Check.

Lequel de ces messages serait le plus adapté pour corriger une faute d'orthographe dans un README ?

A `fix: typo in README`

B `docs(readme): correct spelling error`

C `style: change text`

D `refactor: fix text in documentation`



Knowledge Check.

Lequel de ces messages serait le plus adapté pour corriger une faute d'orthographe dans un README ?

A `fix: typo in README`

B `docs(readme): correct spelling error`

C `style: change text`

D `refactor: fix text in documentation`



Knowledge Check.

Quel est le problème avec ce message ?

`chore(ci): update workflow file`

- ☐ A Il manque le type
- ☐ B Le scope n'est pas autorisé
- ☐ C Rien, il est valide
- ☐ D Il ne commence pas par une majuscule



Knowledge Check.

Quel est le problème avec ce message ?

`chore(ci): update workflow file`

- ☐ A Il manque le type
- ☐ B Le scope n'est pas autorisé
- ☒ C Rien, il est valide
- ☐ D Il ne commence pas par une majuscule



Automatisation avec pre-commit.



L'outil **pre-commit** permet d'automatiser des vérifications avant chaque commit. Idéal pour éviter les oublis ou les erreurs triviales.

→ <https://pre-commit.com>

```
1 pip install pre-commit
2 pre-commit install
```

- ✓ Corrige **les erreurs simples** (espaces en fin de ligne, fichiers mal formatés...)
- ✓ Vérifie le **respect de conventions** (ex : nom de commit, formatage)
- ✓ **Gagne du temps** en automatisant les relectures basiques

Commandes avancées

Allons plus loin...

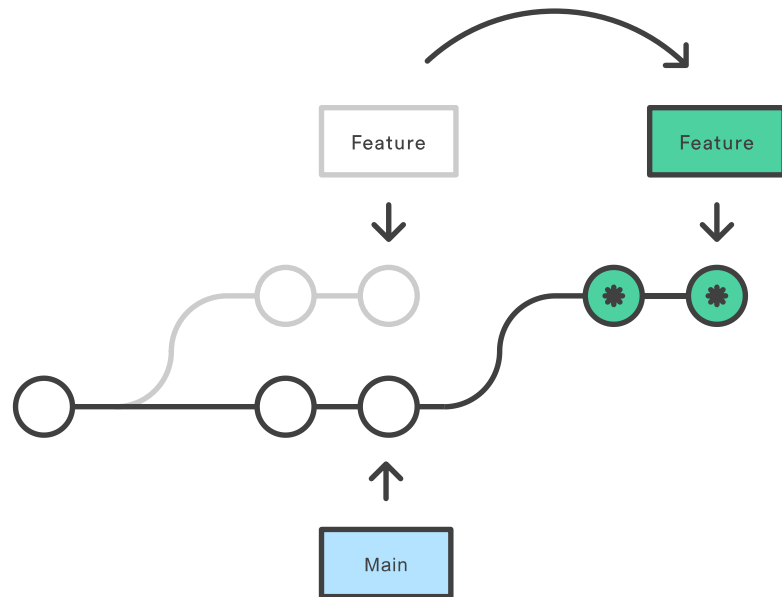


Rebase : nettoyage de l'historique.

git rebase permet de **réécrire l'historique** en le rendant plus propre, linéaire et lisible. Très utile avant de fusionner une branche.

→ Concrètement ?

- Rejoue les commits d'une branche par-dessus une autre
- Permet de réorganiser, renommer, fusionner ou supprimer des commits
- Crée un historique sans commits de merge parasites
- S'exécute depuis la branche à rebaser
- **Nécessite un push forcé !** Cette commande peut être dangereuse si elle n'est pas maîtrisée



```
1 git rebase main
```

Stash : mettre ses modifications de côté.

Besoin de changer de branche mais vous avez du code non terminé ? `git stash` vous permet de sauvegarder temporairement vos modifications.

➡ Concrètement ?

- ➔ Rejoue les commits d'une branche par-dessus une autre
- ➔ Permet de réorganiser, renommer, fusionner ou supprimer des commits
- ➔ Crée un historique sans commits de merge parasites
- ➔ S'exécute depuis la branche à rebaser
- ➔ **Nécessite un push forcé !** Cette commande peut être dangereuse si elle n'est pas maîtrisée

```
1 git stash                # Sauvegarde les modifications (tracked uniquement!)
2 git stash -u             # Sauvegarde les modifications (tracked et untracked)
3 git stash list           # Affiche toutes les sauvegardes en attente
4 git stash pop            # Restaure la dernière sauvegarde de la liste
5 git stash apply stash@{1} # Applique une sauvegarde sans la supprimer
```

Reset & revert : annuler une erreur.

Vous avez commis une erreur ? Git vous offre plusieurs outils pour **annuler proprement** selon le contexte.

➡ `git reset <option> <commit>` – **revenir en arrière localement**

- ➔ L'option `--soft` permet de revenir à un commit en gardant les modifications en « staged »
- ➔ L'option `--mixed` (par défaut) retire les commits mais garde les modifications dans l'espace de travail (fichier « modified »)
- ➔ L'option `--hard` supprime les commits et les modifications (**irréversible !**)

➡ `git revert <commit>` – **annuler un commit partagé**

- ➔ Crée un nouveau commit inverse de celui spécifié
- ➔ Utile pour annuler un bug sans casser l'historique partagé