

# Manipulation de données

Gestion de versions avec Git

Xavier Gendre 

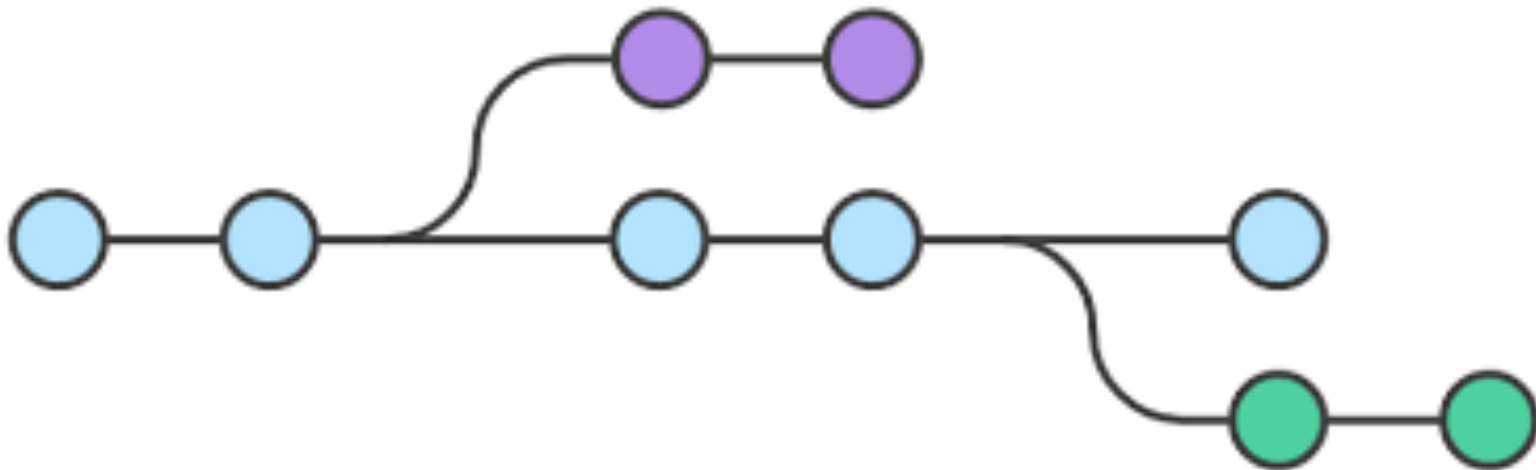
# Système de gestion de version

Objectifs principaux :

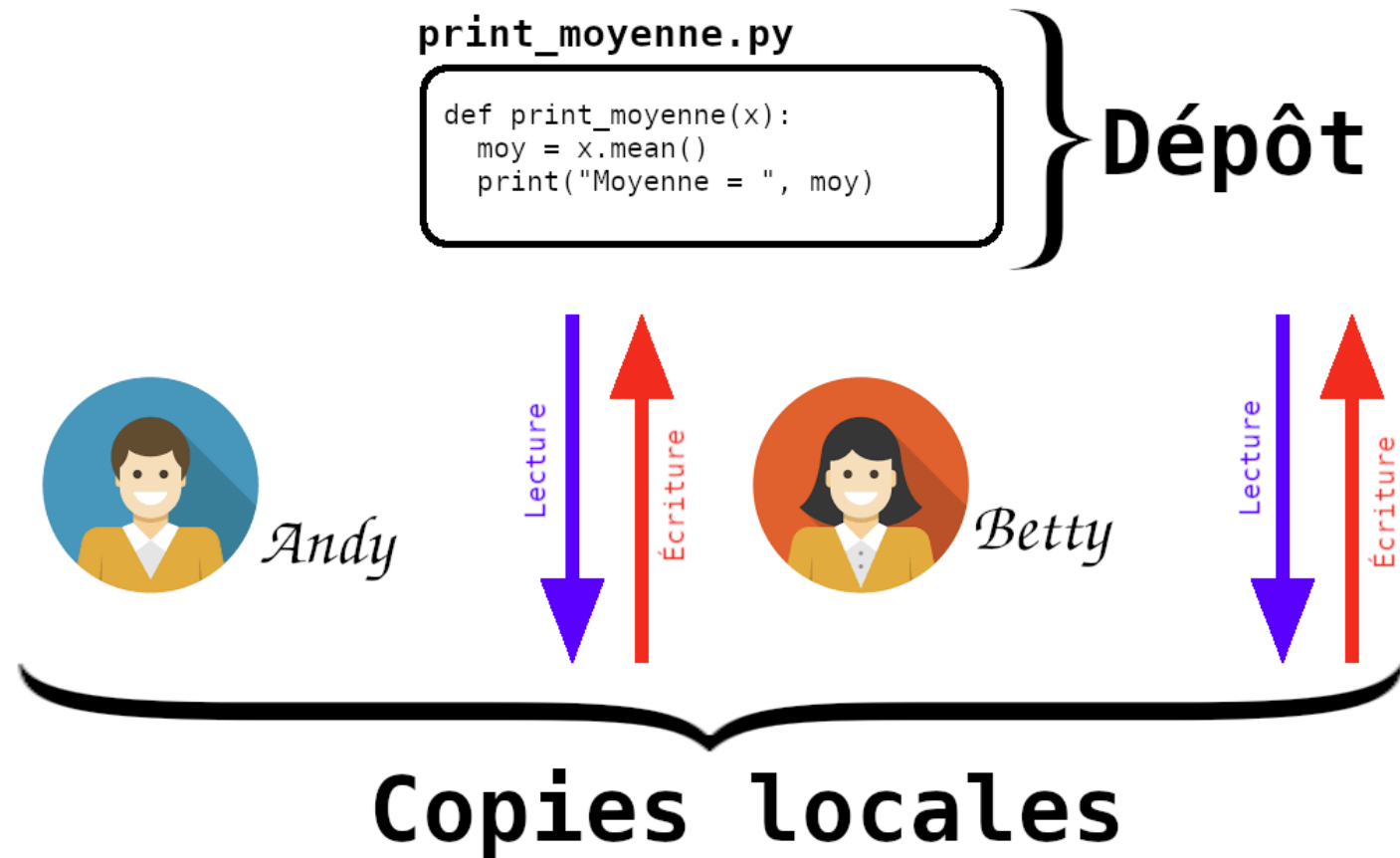
- Travailler à plusieurs
  - Partage du code
  - Gestion des modifications
  - Mutualisation
- Historique des changements
  - Retour en arrière
  - Différences entre 2 versions d'un fichier
  - Qui a modifié un fichier en dernier ?

## Autres objectifs de la gestion de version :

- Gestion des **branches** : mener en parallèle plusieurs versions (stable, dev, ...)
- Utilisation de **tags** : donner un nom explicite à une version pour y accéder facilement
- Sécurité : intégrité, confidentialité, ...



# Principe général



Lecture et écriture via le système de gestion de version.

# Principe général - Problème

`print_moyenne.py`

```
def print_moyenne(x):  
    moy = x.mean()  
    print("Moyenne = ", moy)
```



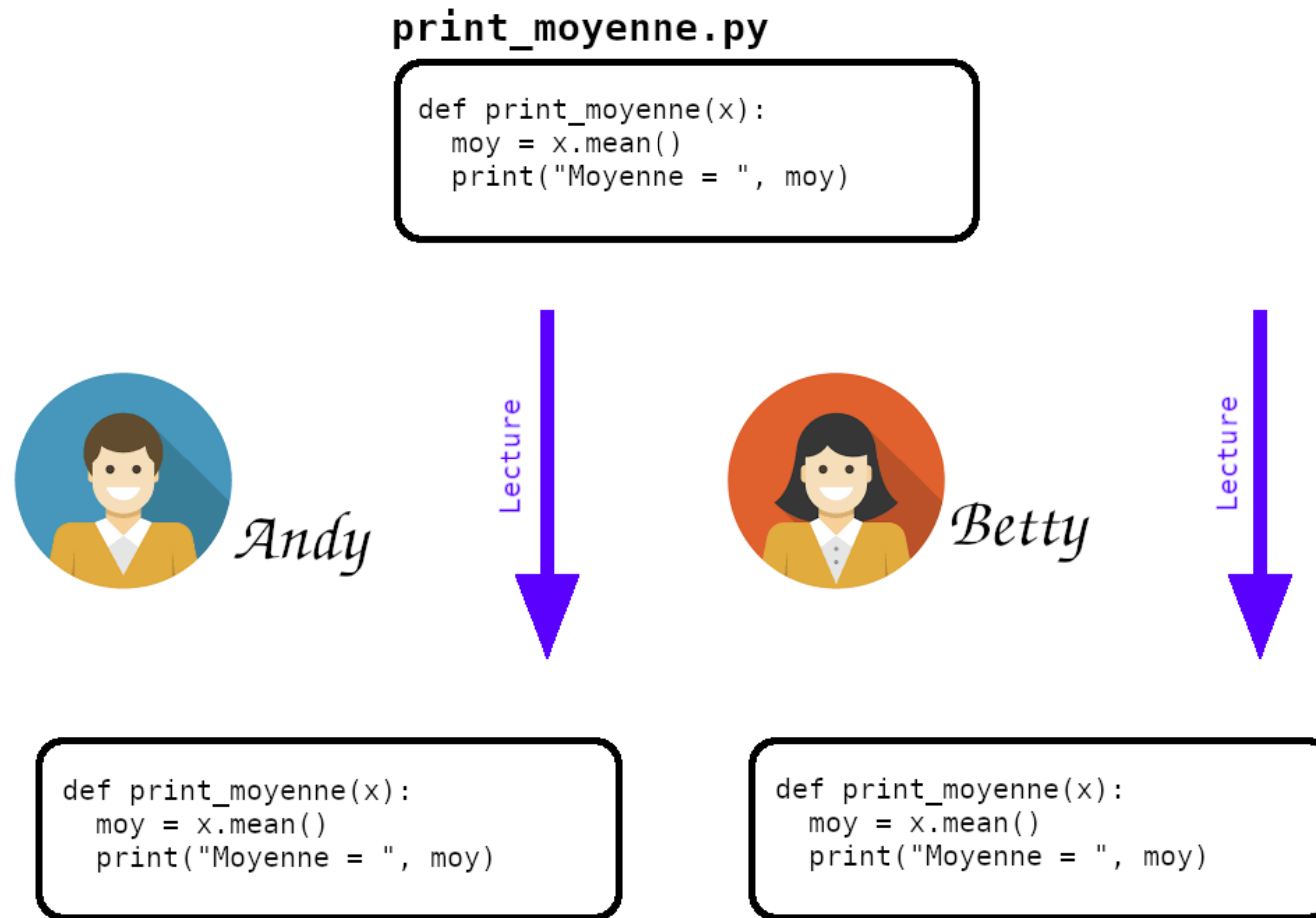
*Andy*



*Betty*

Andy et Betty veulent accéder au **même fichier** du dépôt.

# Principe général - Problème



Andy et Betty **copient** le fichier chez eux.

# Principe général - Problème

**print\_moyenne.py**

```
def print_moyenne(x):  
    moy = x.mean()  
    print("Moyenne = ", moy)
```



*Andy*



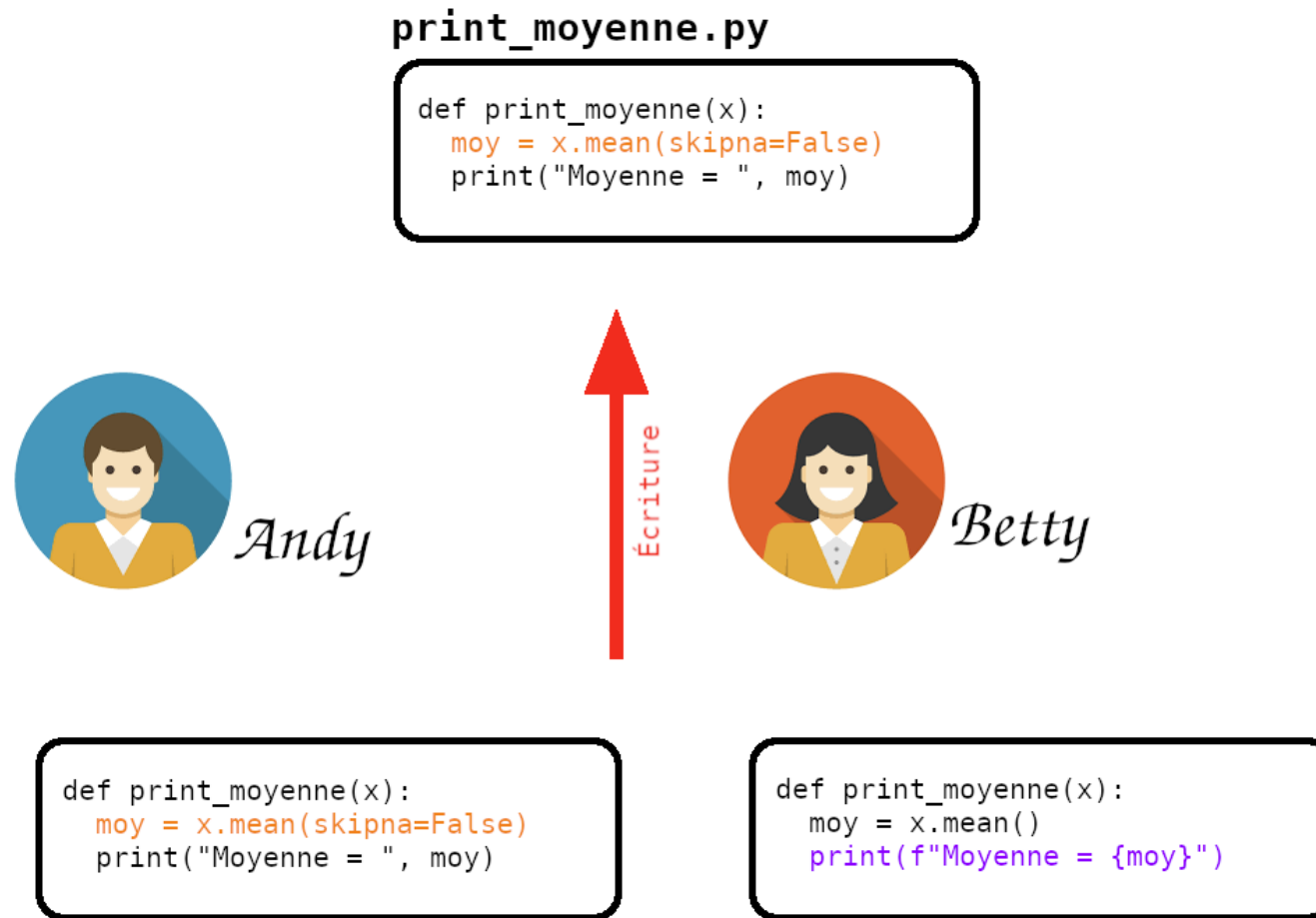
*Betty*

```
def print_moyenne(x):  
    moy = x.mean(skipna=False)  
    print("Moyenne = ", moy)
```

```
def print_moyenne(x):  
    moy = x.mean()  
    print(f"Moyenne = {moy}")
```

Andy et Betty font **chacun des modifications**.

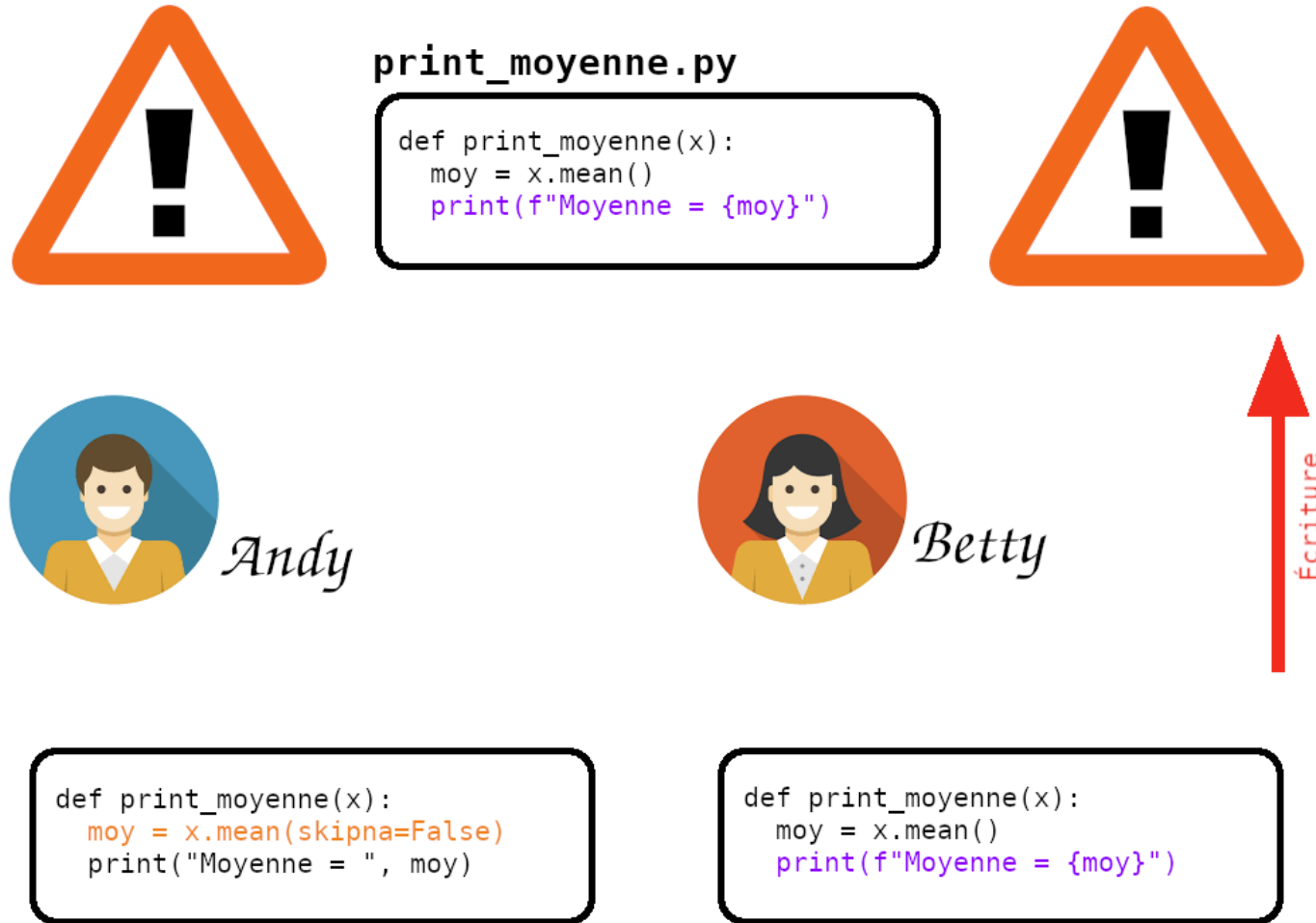
# Principe général - Problème



Andy écrit sur le dépôt.



# Principe général - Problème



Si Betty écrit sur le dépôt, elle écrase la version de Andy !

# Principe général - Solution

`print_moyenne.py`

```
def print_moyenne(x):  
    moy = x.mean(skipna=False)  
    print("Moyenne = ", moy)
```



*Andy*



*Betty*



```
def print_moyenne(x):  
    moy = x.mean(skipna=False)  
    print("Moyenne = ", moy)
```

```
def print_moyenne(x):  
    moy = x.mean()  
    print(f"Moyenne = {moy}")
```

Betty ne peut pas écrire sur le dépôt car elle n'est **pas à jour**.

# Principe général - Solution

**print\_moyenne.py**

```
def print_moyenne(x):  
    moy = x.mean(skipna=False)  
    print("Moyenne = ", moy)
```



*Andy*

```
def print_moyenne(x):  
    moy = x.mean(skipna=False)  
    print("Moyenne = ", moy)
```



*Betty*

```
def print_moyenne(x):  
    moy = x.mean()  
    print(f"Moyenne = {moy}")  
  
def moy(x):  
    print("Moyenne = ", moy)
```

Lecture



Betty se met à jour sans perdre ses modifications.

# Principe général - Solution

`print_moyenne.py`

```
def print_moyenne(x):  
    moy = x.mean(skipna=False)  
    print("Moyenne = ", moy)
```



*Andy*



*Betty*

```
def print_moyenne(x):  
    moy = x.mean(skipna=False)  
    print("Moyenne = ", moy)
```

```
def print_moyenne(x):  
    moy = x.mean(skipna=False)  
    print(f"Moyenne = {moy}")
```

Betty fusionne la version du dépôt avec la sienne.

# Principe général - Solution

`print_moyenne.py`

```
def print_moyenne(x):  
    moy = x.mean(skipna=False)  
    print(f"Moyenne = {moy}")
```



*Andy*



*Betty*

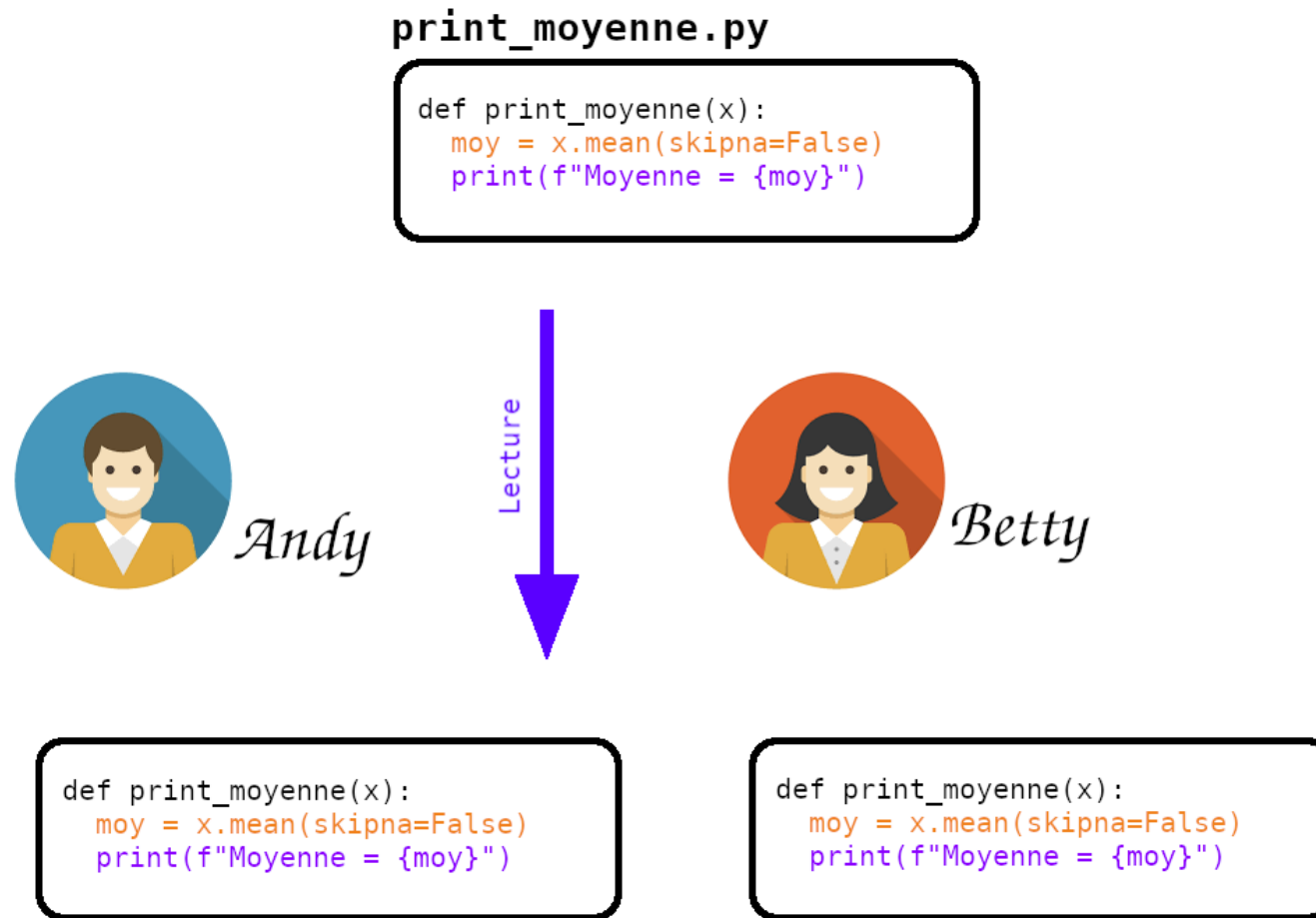


```
def print_moyenne(x):  
    moy = x.mean(skipna=False)  
    print("Moyenne = ", moy)
```

```
def print_moyenne(x):  
    moy = x.mean(skipna=False)  
    print(f"Moyenne = {moy}")
```

Betty peut écrire cette nouvelle version sur le dépôt.

# Principe général - Solution



Andy récupère la nouvelle version.

# Principe général - Bilan

Un système de gestion de version gère un mécanisme de **lecture-fusion-écriture** :

- Les accès (lecture et écriture) se font via le système de gestion de version.
- Le système de gestion de version conserve l'historique.
- La fusion automatique n'est possible que si
  - elle concerne un fichier texte (utilisation de **diff**),
  - les modifications ne touchent pas à la même chose.

# Git

Git est un logiciel de gestion de version (décentralisé) créé en 2005 par Linus Torvalds.

C'est un **logiciel libre** distribué sous licence GNU GPLv2.

- Git est multi-système (Linux, Mac, Windows, ...)
- Git est très utilisé (documentation, forum, ...)
- Git est sécurisé (protocoles HTTPS, SSH, ...)





# Git - Vocabulaire

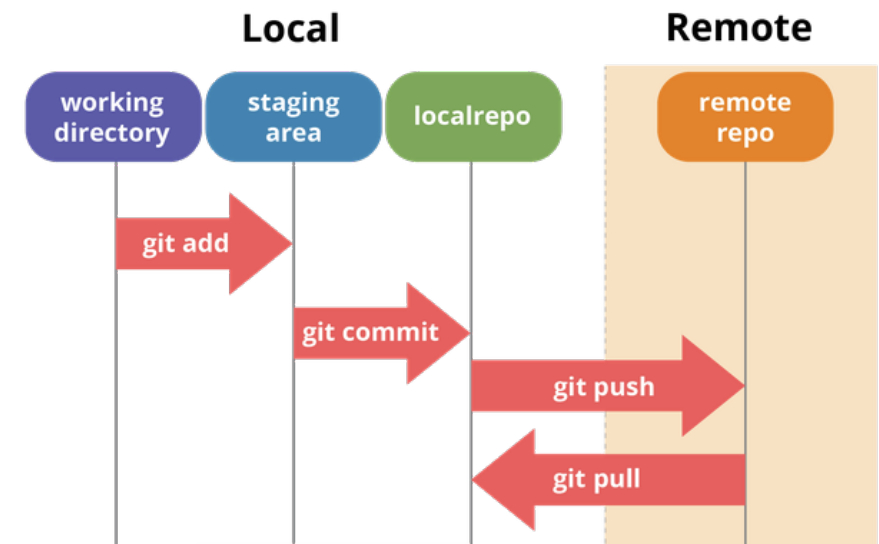
**add** Ajoute un élément à la zone d'index (*staging area*), cet élément est dit **staged**.

**commit** Valide les modifications dans la zone d'index.

**push** Pousse les modifications vers le dépôt distant.

**pull** Tire les modifications du dépôt distant vers le dépôt local.

Un fichier peut être **modifié, indexé et validé**.



# Git - Fichiers ignorés

Par défaut, un nouveau fichier ne fait pas partie de la zone d'index. Son état est dit **Untracked** et un tel fichier **n'est jamais poussé** sur le dépôt distant.

Le système de suivi de Git (**status**) signale ces fichiers.

De tels fichiers sont parfois nécessaires (fichiers temporaires, fichiers liés à la session en cours, ...). Pour signifier à Git qu'ils peuvent être ignorés, ils doivent apparaître dans un fichier spécial du dépôt appelé **.gitignore**.

```
1 __pycache__/  
2 *.py[.cod]
```

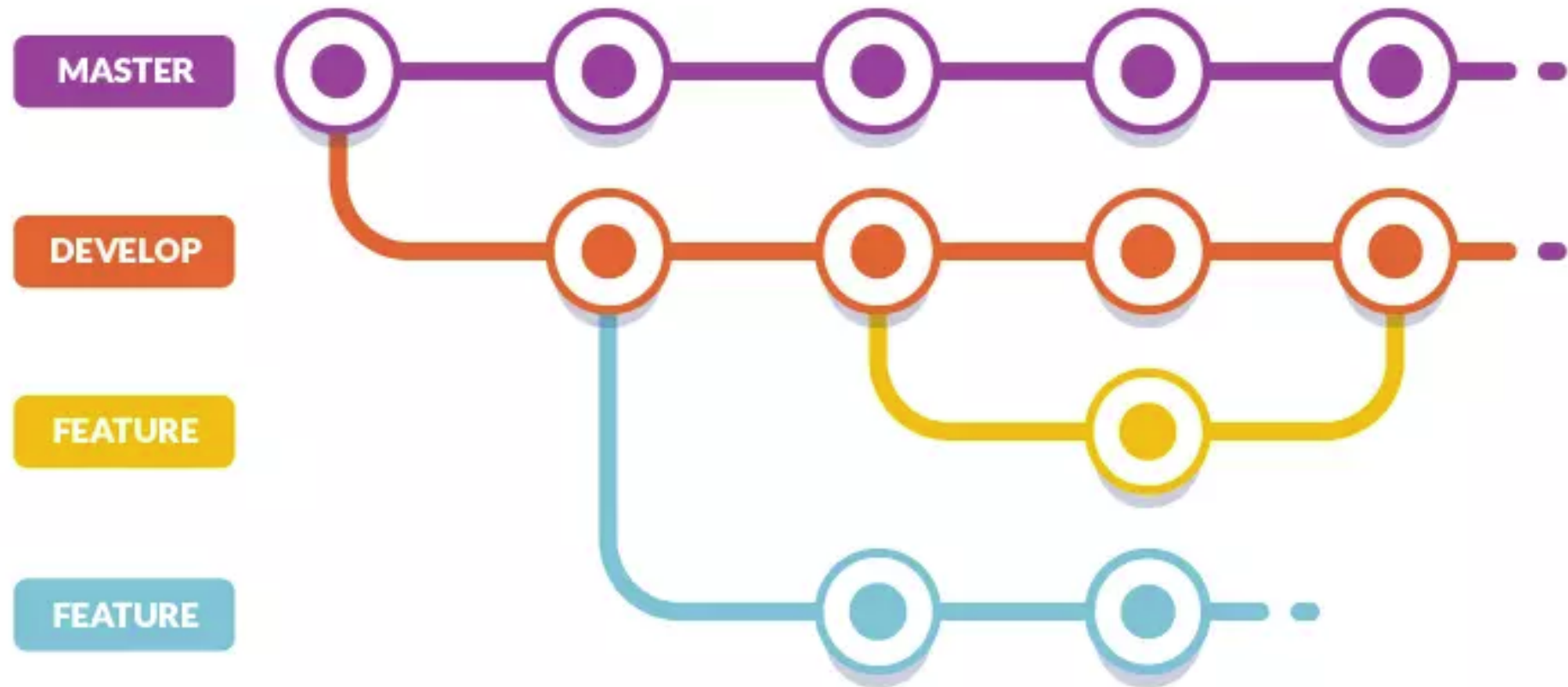
# Git - Les branches

Une **branche** est une version du projet qui suit son propre développement (avec son historique dans le système de gestion de version) et qui peut être **fusionnée** avec d'autres branches.

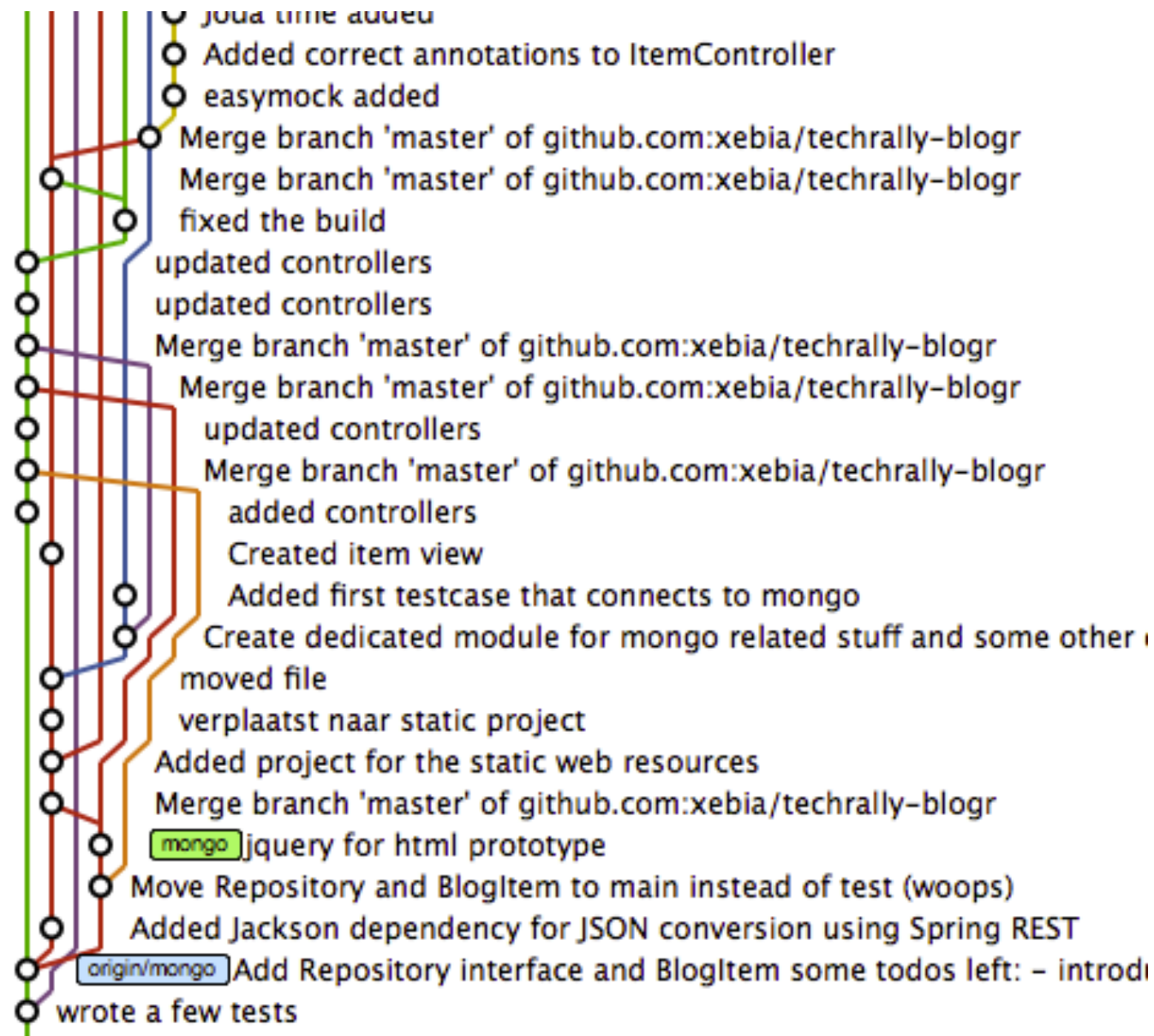
Exemples classiques :

- Branche **main** (ou **master**) : version stable du projet qui peut être utilisée en production.
- Branche **dev** : version de développement, instable par définition mais plus avancée que la version stable du projet.
- Branche **bug42** : version dédiée à la correction d'un bug.

# Git - Les branches (Cas simple)



# Git - Les branches (La réalité...)



# Git - Vocabulaire des branches

`branch test` Crée une nouvelle branche `test` dans le dépôt local.

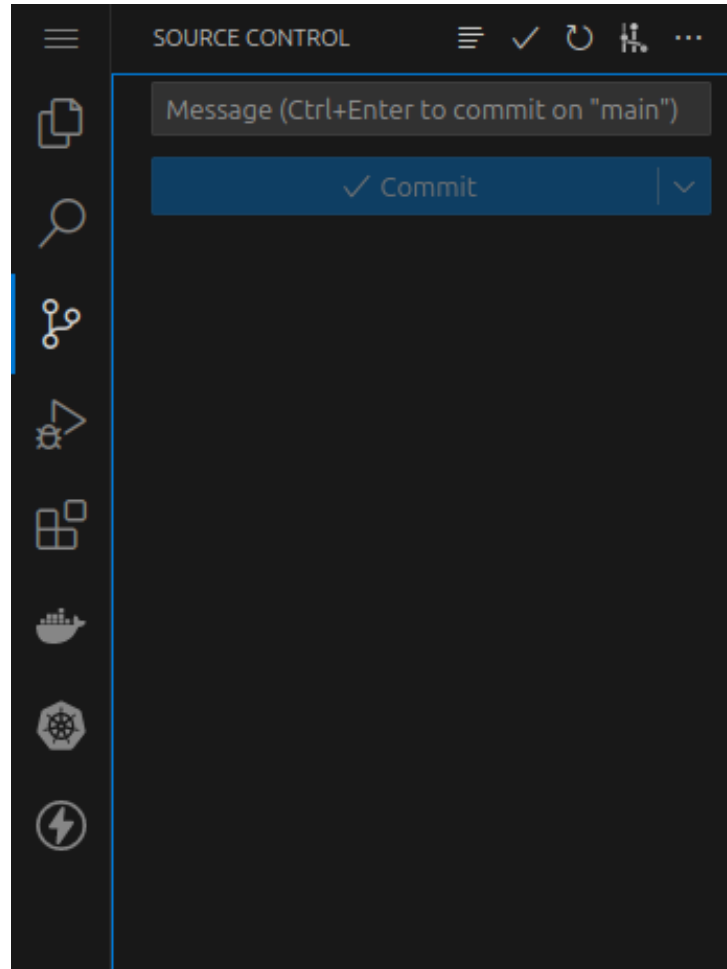
`checkout test` Bascule vers la branche `test` du dépôt local.

`merge test` Fusionne la branche `test` du dépôt local avec la branche courante.

**Une fusion peut être bloquée en cas de conflit entre les modifications. Git offre des outils pour gérer cela mais ce travail reste essentiellement manuel.**

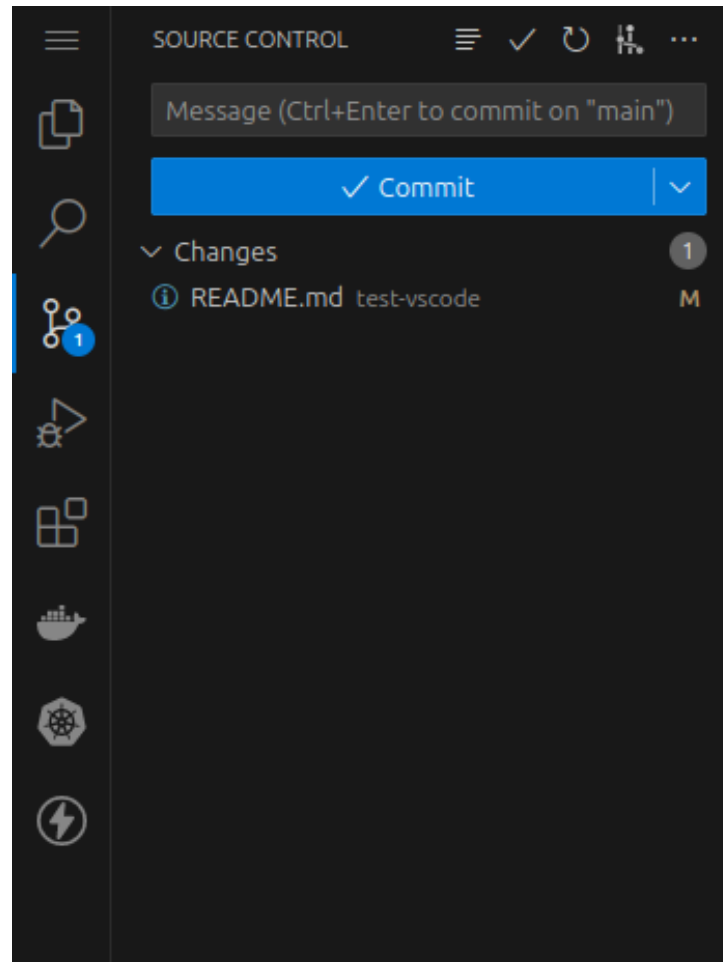
(Il est bien entendu possible de pousser une branche sur le dépôt distant.)

# Git et VSCode



Quand le projet est suivi par Git, un menu *Source Control* est disponible dans la barre de gauche.

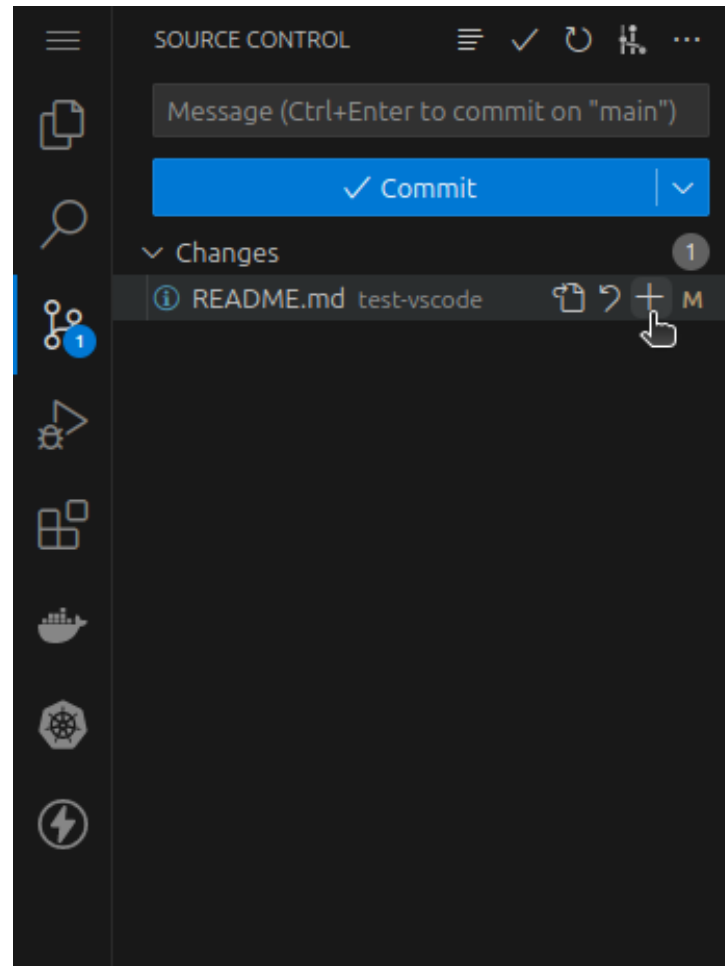
# Git et VSCode



Lorsque un fichier est modifié (**M**), supprimé (**D**), renommé (**R**), non suivi (**U**), ..., il apparaît dans la liste.

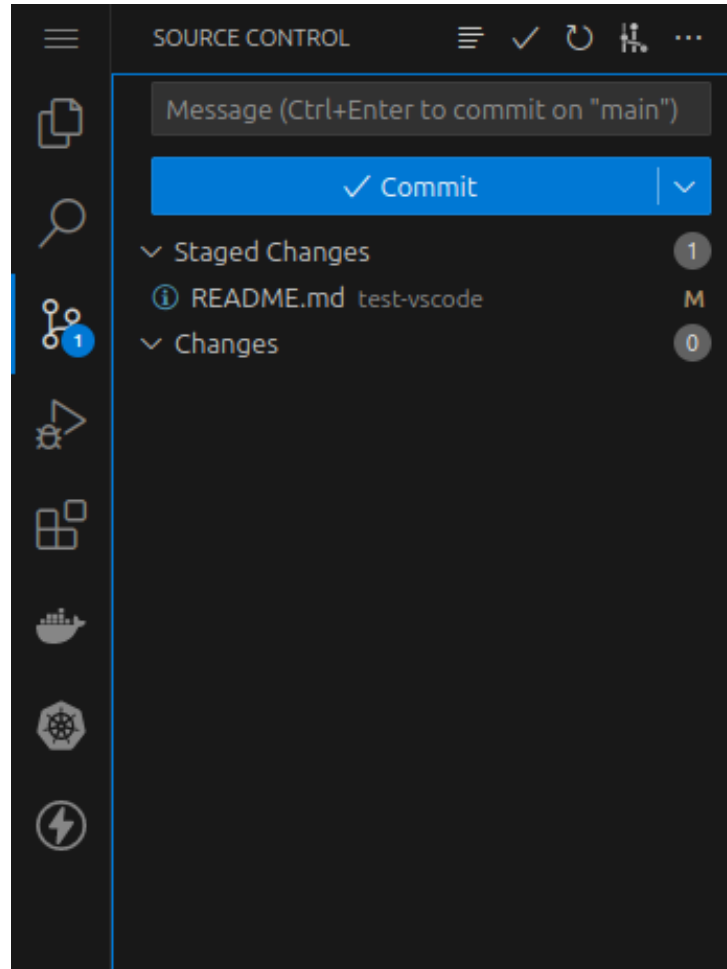


# Git et VSCode



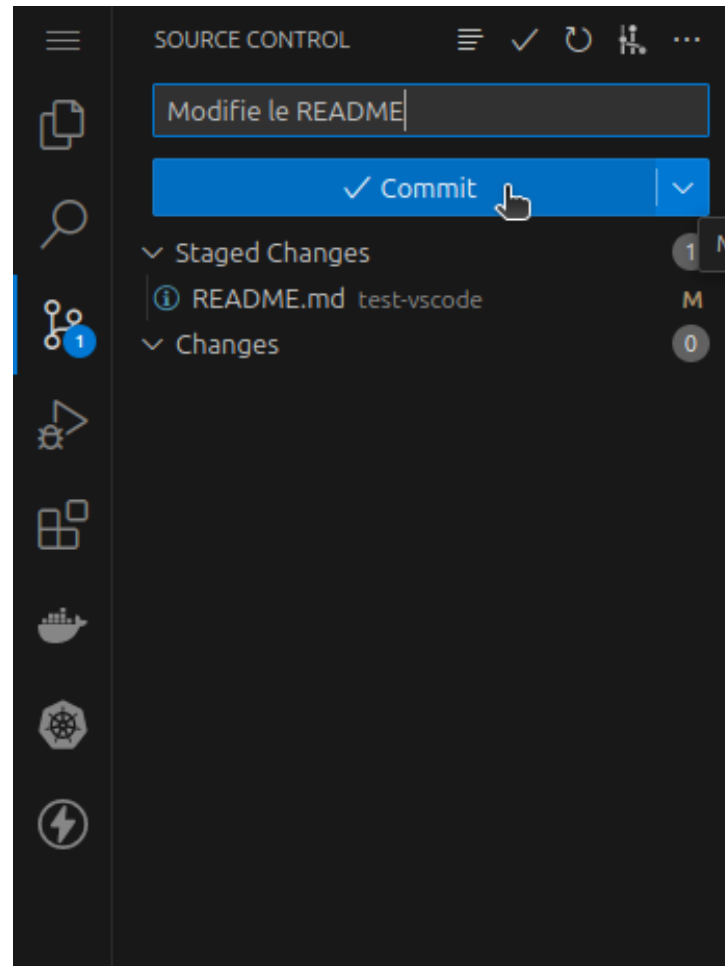
Un ou plusieurs changements peuvent être ajoutés à la *zone d'index* grâce au bouton **+** associé.

# Git et VSCode



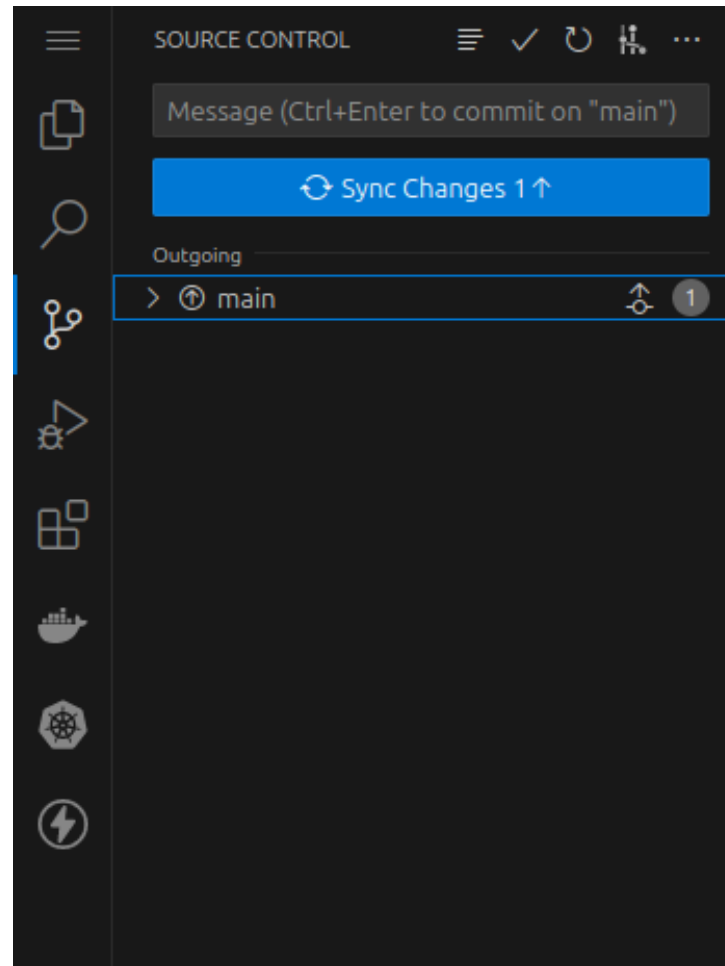
Les changements sont maintenant *staged* et ils peuvent constituer un *commit*, *i.e.* une étape de notre projet.

# Git et VSCode



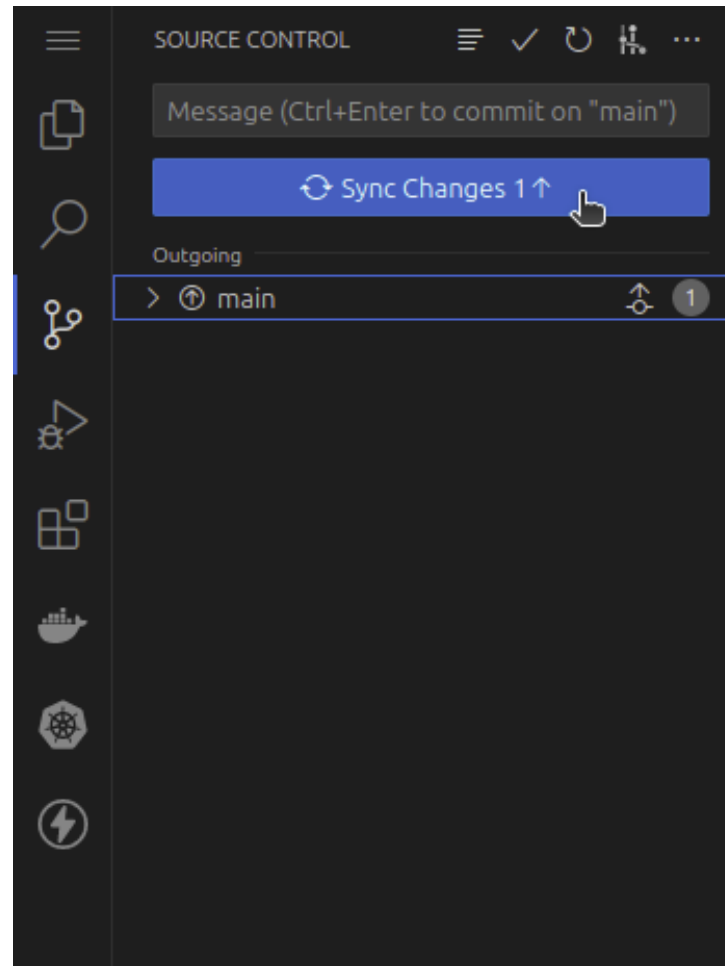
Avec une brève description pour faciliter la maintenance du projet, le commit est créé avec le bouton **Commit**.

# Git et VSCode



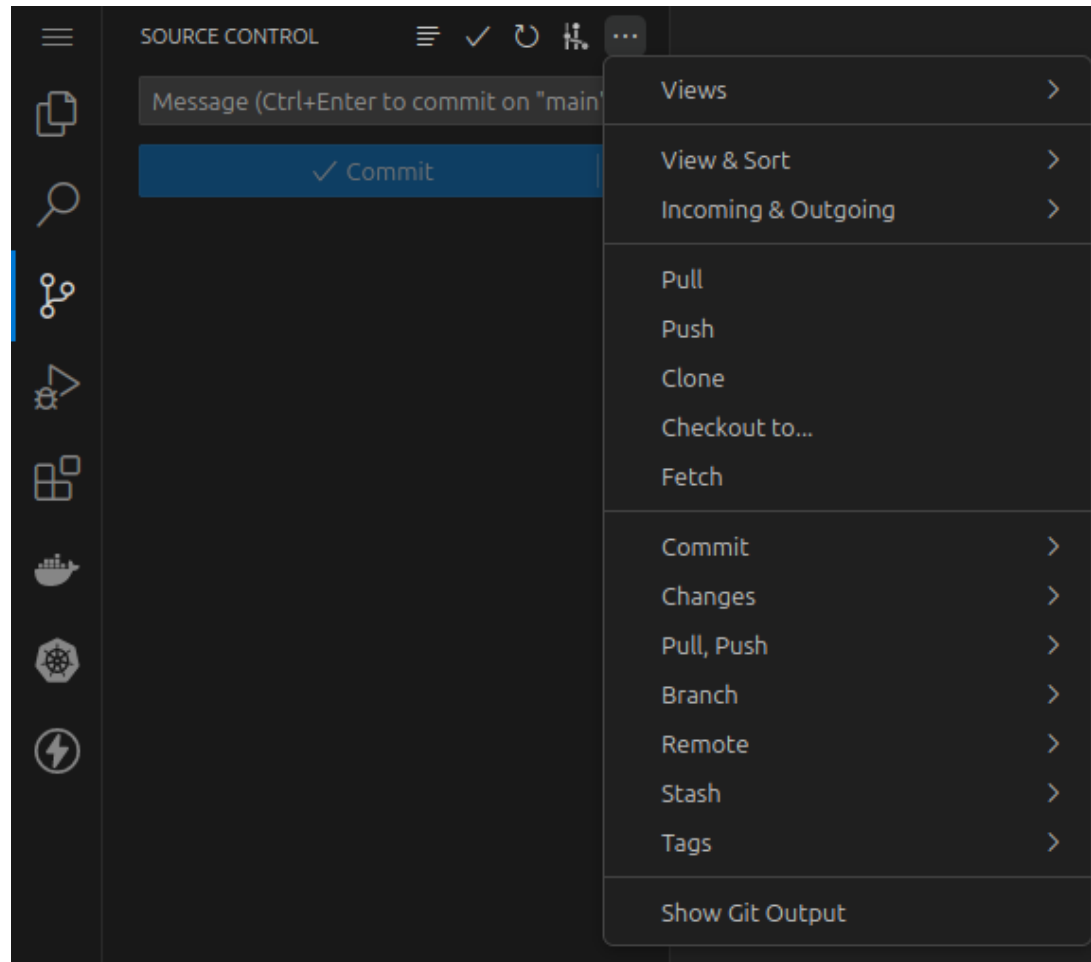
Un commit a été ajouté à la branche `main` dans cet exemple dont le détail apparaît dans la zone inférieure.

# Git et VSCode



Si le projet est stocké dans un *dépôt distant*, les changements peuvent être poussés (**push**) avec **Sync Changes**.

# Git et VSCode



Au-delà de ces opérations courantes, le menu . . . offre bien d'autres possibilités ([pull](#), [branch](#), ...).

# Git et VSCode - Ligne de commande

Si les outils mis à disposition ne suffisent pas, il est possible d'interagir via la *ligne de commande* dans un terminal accessible via le raccourci **Ctrl+J**.

Ces manipulations sont destinées à un **usage plus avancé**.

Par exemple, pour faire un **commit**, Git a besoin de savoir qui nous sommes :

```
1 git config --global user.name "Votre Nom"
2 git config --global user.email "votre@mail.net"
```

Ces informations seront particulièrement utiles pour les projets **hébergés à distance** (voir la configuration Git de VSCode avec Onyxia dans un instant).

# Moralité

L'utilisation d'un système de gestion de version est :

- **indispensable** pour travailler à plusieurs,
- **sécurisant** même lorsqu'on est seul sur le projet,
- **simple** car l'effort à fournir est négligeable.

**Ne pas utiliser un système de gestion de version est une faute professionnelle.**

Pour aller plus loin : `tag`, `revert`, `rebase`, ...



# GitHub

GitHub est un **service web d'hébergement** et de **gestion** de dépôts distants basés sur le logiciel Git.

GitHub propose des comptes gratuits et une offre payantes pour des usages plus avancés.

Un dépôt peut être **public** ou **privé**, GitHub permet le contrôle des accès par les utilisateurs.

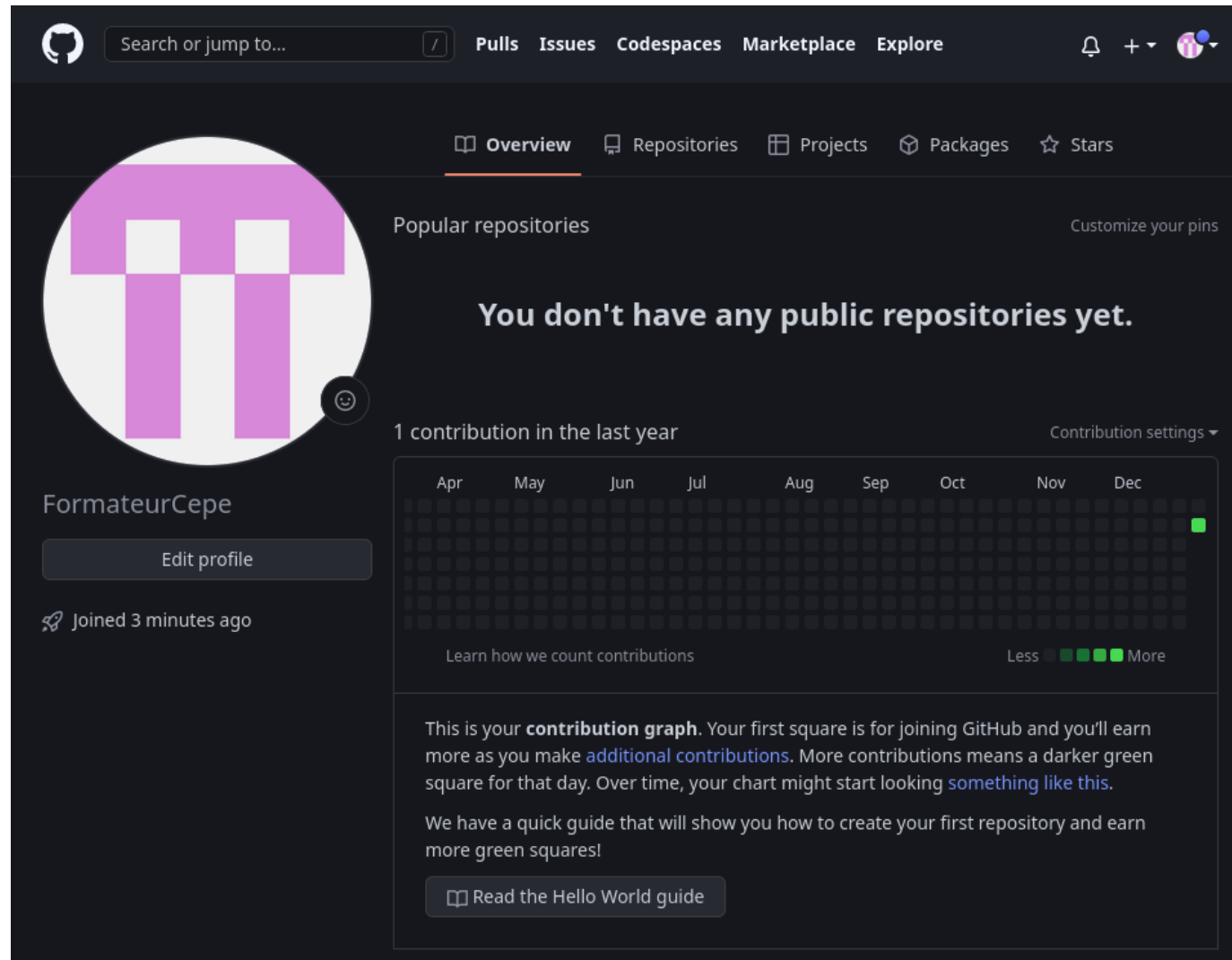
GitHub offre aussi de nombreux services : suivi de bugs, forum, wiki, pages web, ...

GitHub a été racheté par Microsoft en 2018 pour 7,5 milliards \$.



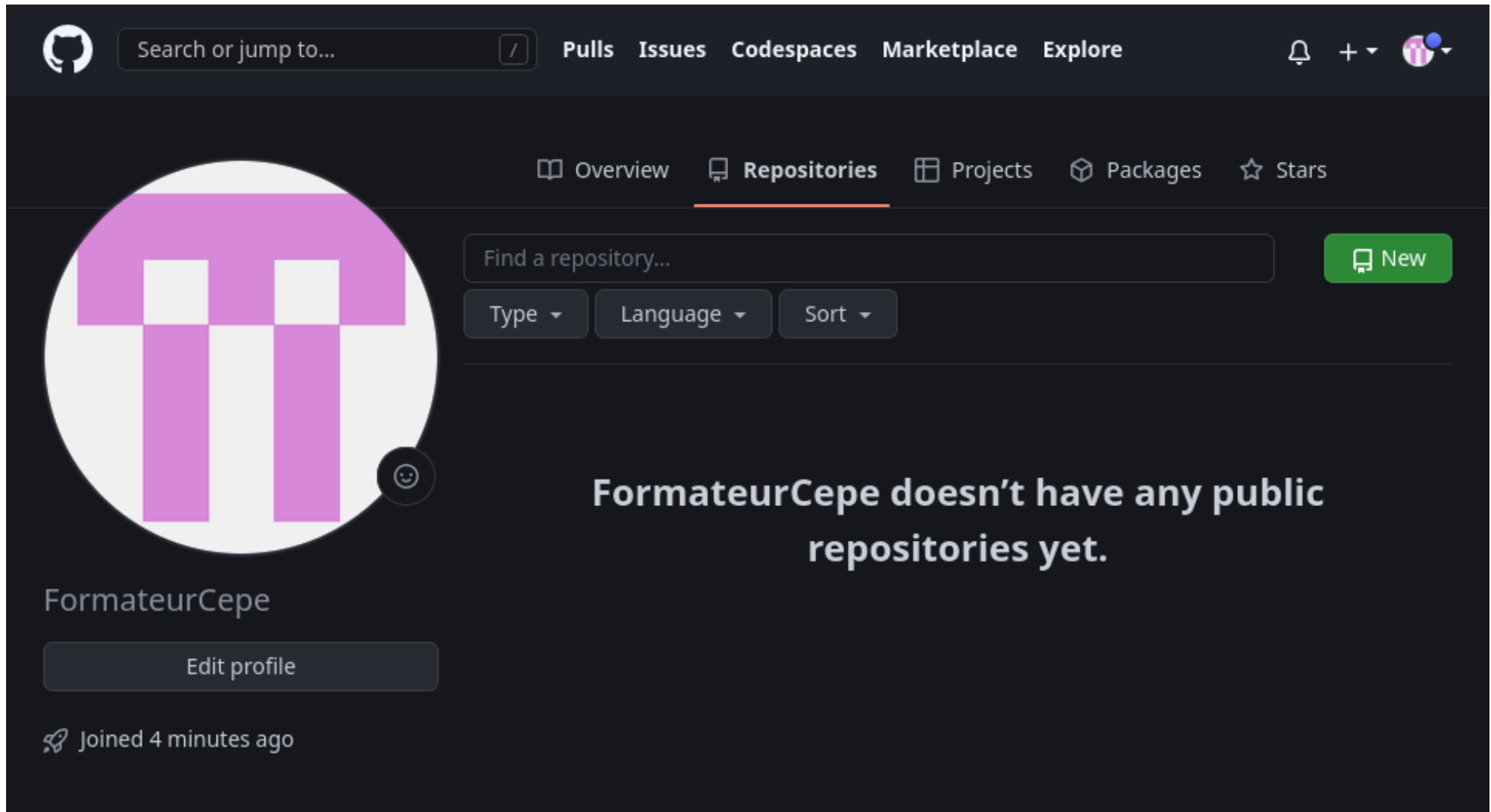
# Profil GitHub

Une fois connecté, vous découvrez l'interface.



# Héberger ses dépôts sur GitHub

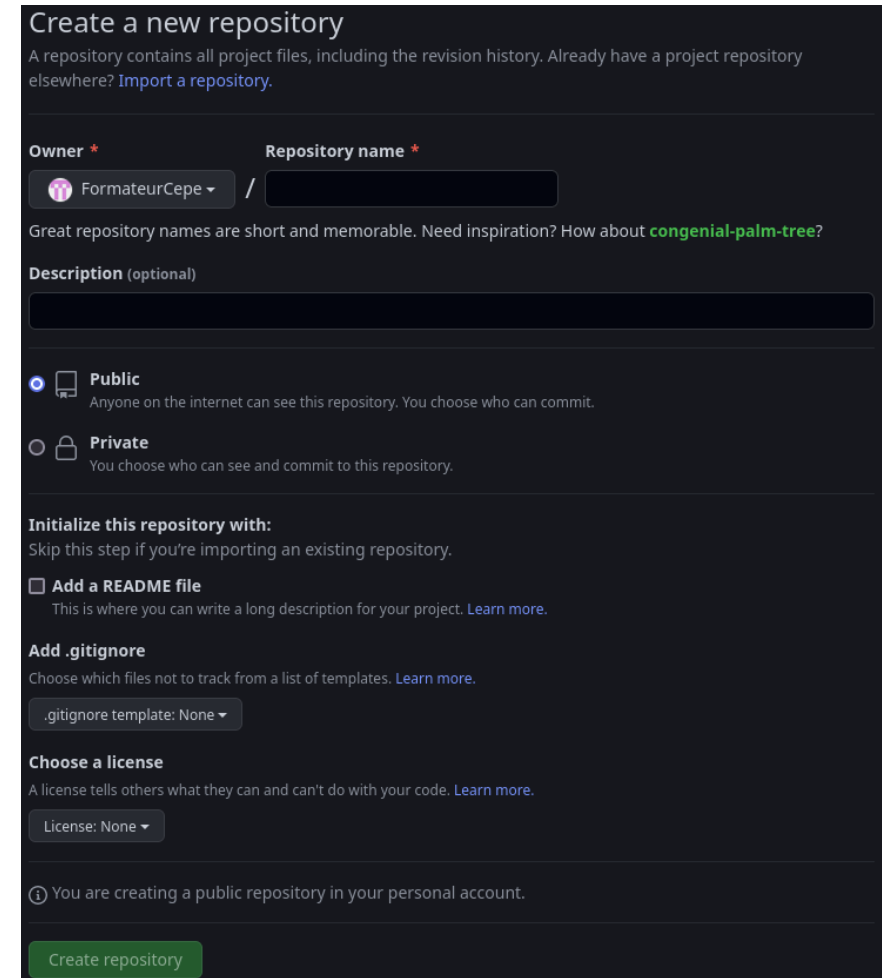
Vous pouvez voir vos dépôts ou en créer de nouveaux.



# Créer un dépôt sur GitHub

Interface de création pour :

- donner un **nom** à votre dépôt,
- le déclarer **public** ou non,
- créer certains **fichiers utiles** : le fichier **README** initialise le dépôt,
- choisir une **licence** (libre, naturellement).



The screenshot shows the GitHub 'Create a new repository' page. At the top, it says 'Create a new repository' and provides a brief explanation: 'A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)'

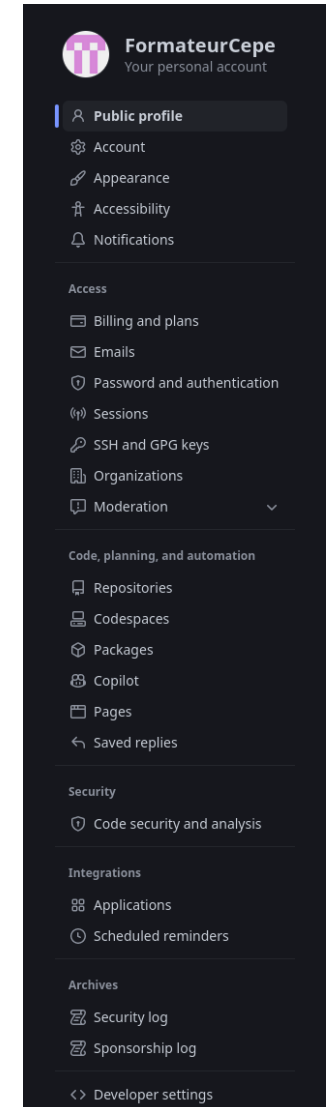
The form includes the following sections:

- Owner \***: A dropdown menu showing 'FormateurCepe'.
- Repository name \***: An empty text input field.
- A note: 'Great repository names are short and memorable. Need inspiration? How about [congenial-palm-tree?](#)'
- Description (optional)**: An empty text input field.
- Visibility**: Two radio buttons. 'Public' is selected, with the description 'Anyone on the internet can see this repository. You choose who can commit.' The 'Private' option is unselected, with the description 'You choose who can see and commit to this repository.'
- Initialize this repository with:** A section with the instruction 'Skip this step if you're importing an existing repository.' It includes an unchecked checkbox for 'Add a README file' with the subtext 'This is where you can write a long description for your project. [Learn more.](#)'
- Add .gitignore**: A section with the instruction 'Choose which files not to track from a list of templates. [Learn more.](#)' and a dropdown menu currently set to '.gitignore template: None'.
- Choose a license**: A section with the instruction 'A license tells others what they can and can't do with your code. [Learn more.](#)' and a dropdown menu currently set to 'License: None'.
- A warning icon and text: 'You are creating a public repository in your personal account.'
- A green 'Create repository' button at the bottom.

# Paramètres de son compte GitHub

Vous pouvez configurer de nombreux paramètres dans l'interface :

- vos informations personnelles (l'**adresse mail** est la plus importante car elle apparaît dans chaque commit),
- l'apparence de l'interface (il faudra choisir son côté de la Force),
- votre **clé SSH** (obligatoire pour pouvoir pousser des modifications en l'absence de *token*), ...

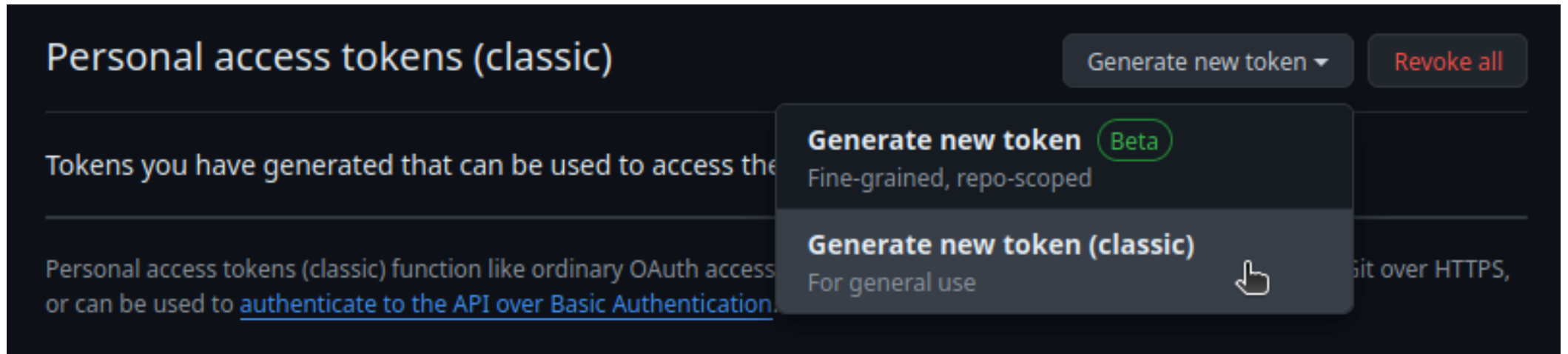




# Obtenir un token pour GitHub

Un *token* (ou jeton d'accès) est un mécanisme d'identification sécurisé pour interagir de manière fluide avec vos dépôts. Une alternative est l'utilisation d'une clé SSH.

La [page de gestion des tokens](#) permet de générer un nouveau token. Nous utiliserons le format *classique*.



# Obtenir un token pour GitHub

New personal access token (classic)

Personal access tokens (classic) function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

**Note**

Onyxia

What's this token for?

**Expiration \***

30 days The token will expire on Fri, Jul 19 2024

**Select scopes**

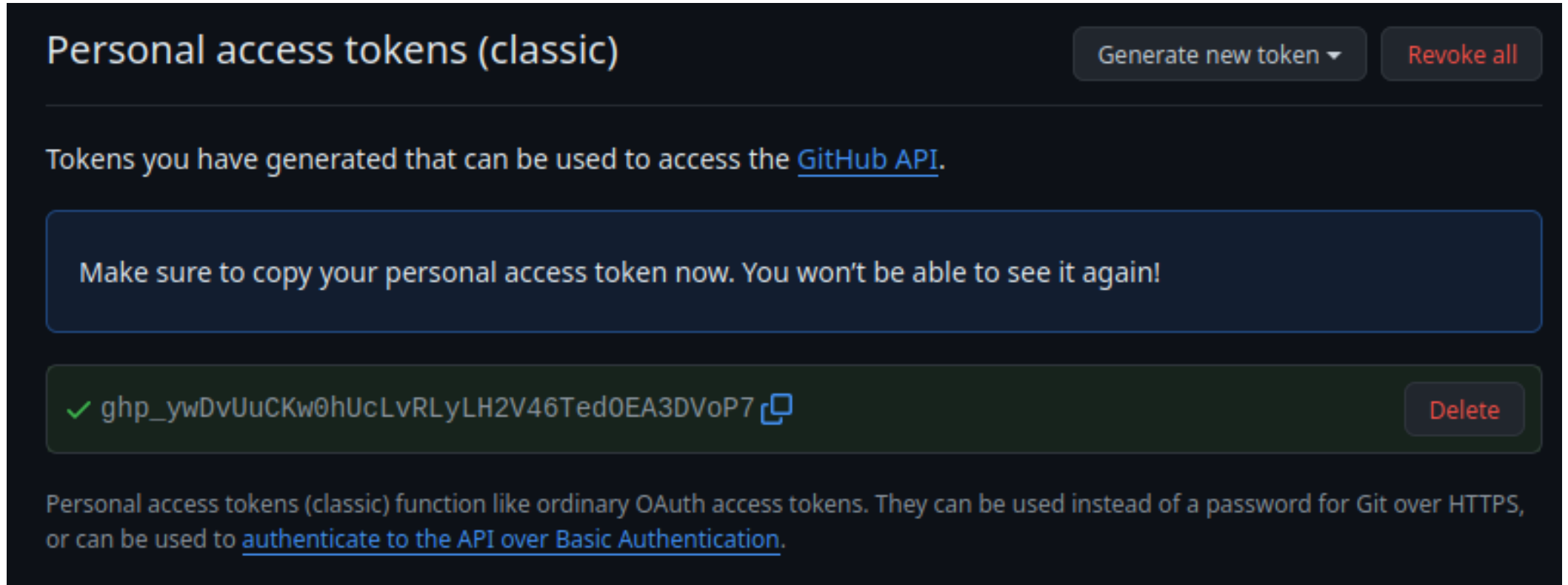
Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

<input checked="" type="checkbox"/> <b>repo</b>	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations
<input checked="" type="checkbox"/> security_events	Read and write security events
<input type="checkbox"/> <b>workflow</b>	Update GitHub Action workflows
<input type="checkbox"/> <b>write:packages</b>	Upload packages to GitHub Package Registry
<input type="checkbox"/> read:packages	Download packages from GitHub Package Registry

- Donner un nom à votre token.
- Définir une durée de validité (par sécurité).
- Limiter la portée à **repo** uniquement pour éviter les problèmes en cas de fuite.



# Obtenir un token pour GitHub




The screenshot shows the GitHub 'Personal access tokens (classic)' interface. At the top, there's a title 'Personal access tokens (classic)' and two buttons: 'Generate new token' and 'Revoke all'. Below this, a message states: 'Tokens you have generated that can be used to access the [GitHub API](#).' A prominent warning box says: 'Make sure to copy your personal access token now. You won't be able to see it again!'. Below the warning, a green bar displays a newly generated token: 'ghp\_ywDvUuCKw0hUcLvRLyLH2V46Ted0EA3DVoP7' with a copy icon. A 'Delete' button is next to the token. At the bottom, a note explains: 'Personal access tokens (classic) function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).'

Personal access tokens (classic)

Generate new token ▼ Revoke all

Tokens you have generated that can be used to access the [GitHub API](#).

Make sure to copy your personal access token now. You won't be able to see it again!

✓ ghp\_ywDvUuCKw0hUcLvRLyLH2V46Ted0EA3DVoP7 

Delete

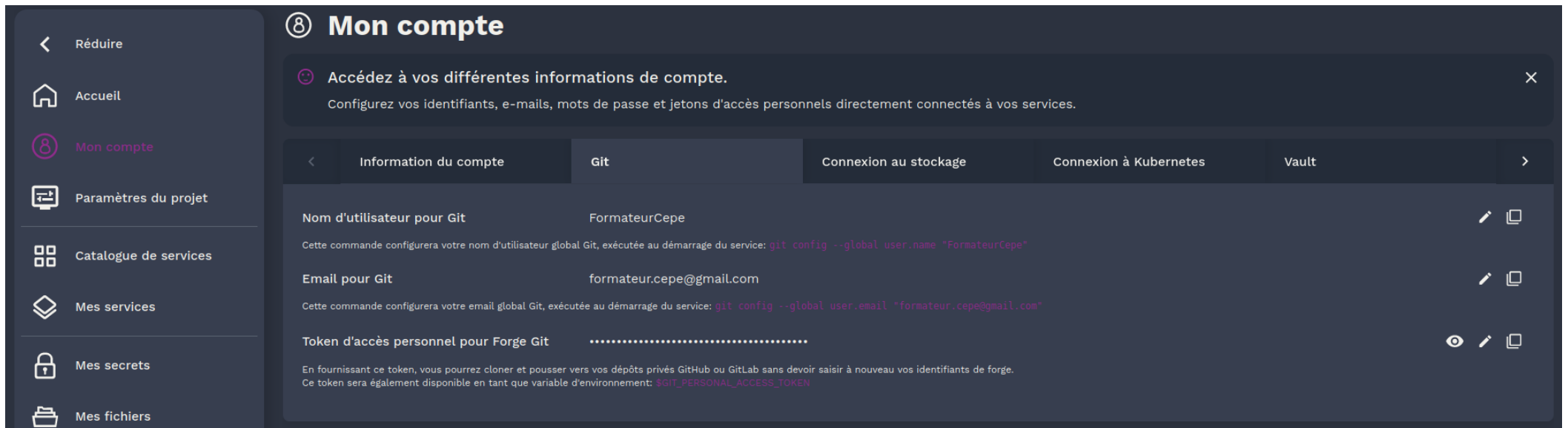
Personal access tokens (classic) function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Le token est créé ! Il doit être copié et stocké en lieu sûr (par exemple, sur Onyxia) car il ne pourra plus être affiché.

# Configuration Git avec Onyxia

Par défaut, un compte sur la forge **GitGenes** a été créé mais nous utiliserons *GitHub* dans la suite (le même principe reste valable pour d'autres forges basées sur Git).

Une configuration **globale** du nom, du mail et du token est possible depuis l'onglet *Git* du menu *Mon compte* de Onyxia.



The screenshot shows the 'Mon compte' (My account) page in the Onyxia interface. The left sidebar contains navigation links: Réduire, Accueil, Mon compte (active), Paramètres du projet, Catalogue de services, Mes services, Mes secrets, and Mes fichiers. The main content area is titled 'Mon compte' and features a sub-header 'Accédez à vos différentes informations de compte.' with a close button. Below this is a tabbed interface with five tabs: Information du compte, Git (active), Connexion au stockage, Connexion à Kubernetes, and Vault. The 'Git' tab displays three configuration items: 'Nom d'utilisateur pour Git' (FormateurCepe), 'Email pour Git' (formateur.cepe@gmail.com), and 'Token d'accès personnel pour Forge Git' (masked with dots). Each item has a description of the Git command it configures and icons for editing or copying. The token section also includes a note about its use for cloning and pushing to private repositories.

**Mon compte**

Accédez à vos différentes informations de compte.  
Configurez vos identifiants, e-mails, mots de passe et jetons d'accès personnels directement connectés à vos services.

Information du compte | **Git** | Connexion au stockage | Connexion à Kubernetes | Vault

**Nom d'utilisateur pour Git** FormateurCepe  
Cette commande configurera votre nom d'utilisateur global Git, exécutée au démarrage du service: `git config --global user.name "FormateurCepe"`

**Email pour Git** formateur.cepe@gmail.com  
Cette commande configurera votre email global Git, exécutée au démarrage du service: `git config --global user.email "formateur.cepe@gmail.com"`

**Token d'accès personnel pour Forge Git** .....  
En fournissant ce token, vous pourrez cloner et pousser vers vos dépôts privés GitHub ou GitLab sans devoir saisir à nouveau vos identifiants de forge.  
Ce token sera également disponible en tant que variable d'environnement: `$GIT_PERSONAL_ACCESS_TOKEN`


Une configuration **locale** avec l'adresse du dépôt (et la branche en option) est disponible à la création du service.

Créer votre propre service

Réinitialiser les configurations

Copier l'URL de lancement automatique

Enregistrer la configuration

 Vscode-python

Nom personnalisé

vscode-python

Version ?

1.11.35

Annuler

Lancer

Configuration Vscode-python

^

<

Git

Discovery

Service

Persistence

>

Git user configuration

☒ Enabled

Add git config inside your environment

Name

FormateurCepe

Email


formateur.cepe@gmail.com

Cache

0

Token

.....



Repository

https://github.com/FormateurCepe/mon\_depot.git

Branch

Branch automatically checked out

**À vous de jouer !**