

# Approfondissement Git et GitHub

mercredi 14 septembre 2022 11:38

Lien du cours : <https://openclassrooms.com/fr/courses/7688581-devenez-un-expert-de-git-et-github>

## Contribuez à votre premier projet open source

Notre point de départ : nous apprêter le projet **Open Transport** (projet OpenSource utilisé dans le cours OpenClassroom) La question est donc : **comment intégrer une communauté open source ?**

Il n'y a évidemment **pas de réponse absolue**, dans la mesure où chaque communauté peut avoir ses spécificités. Néanmoins, on retrouve souvent des éléments similaires d'un projet à l'autre, comme un **fichier README**, des **directives de collaboration**, un **code de conduite** ou encore une **licence**.

*Vous entendrez d'ailleurs souvent parler de **guidelines**, **code of conduct** ou **contributing rules**.*

Le projet **Open Transport** est disponible sur GitHub à cette adresse :

<https://github.com/OpenClassrooms-Student-Center/7688581-Expert-Git-GitHub>.

Il est important de consulter les éléments suivants avant toute chose :

- le **fichier README** qui donne les **informations générales** sur le projet
- le fichier **CONTRIBUTING** pour les **directives de collaboration**
- le fichier **CODE OF CONDUCT** pour les **règles de conduite** au sein du projet.

## Mettez en place votre environnement

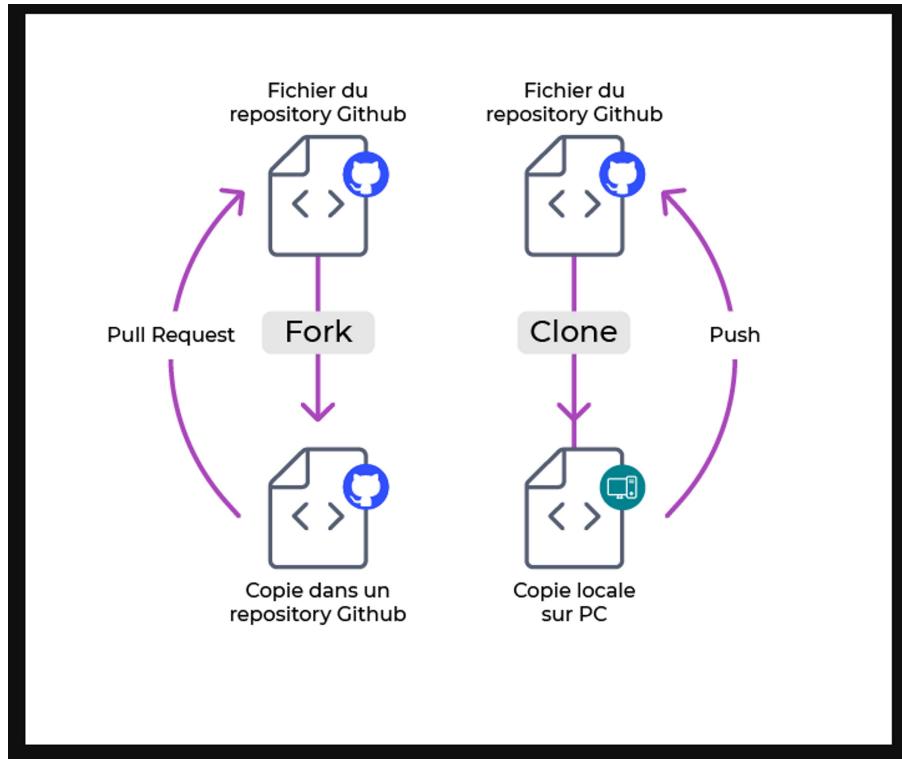
Dans le cadre de ce cours, il est nécessaire d'avoir à disposition :

- un **navigateur** comme Google Chrome pour accéder à GitHub
- une **installation en local** de **Git**
- un **éditeur de texte** comme **Sublime Text** ou **VSCode**
- un **compte personnel GitHub**.

Une fois que tout est prêt, une étape importante nous attend : **Forker** notre projet

**Forker** : **Forker** signifie **faire une copie d'un projet**, de manière à **travailler dessus sans impacter l'original**.

Fondamentalement, **un fork et un clone sont similaires** : tous deux sont des **copies du projet**. Mais il y a bien une différence dans leur lien avec le projet originel. **Un clone maintient un lien avec le projet**. Par conséquent, si vous faites un **git push**, la **modification sera (uniquement) envoyée au projet originel**. En revanche, le **fork est complètement isolé du projet originel**. Pour envoyer une modification au projet originel, vous devez **soumettre une pull request**.

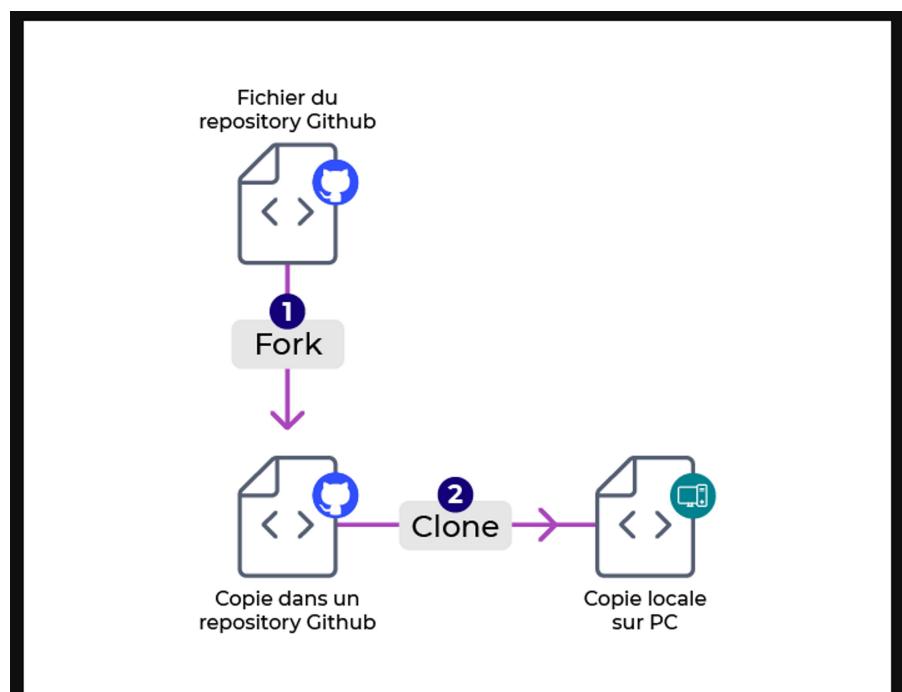


Autre différence importante : **forker un repository GitHub crée un autre repository GitHub**. Alors que **cloner un repository GitHub crée seulement une copie locale sur votre ordinateur**.

*Et si je veux travailler en local sur un fork ?*

La réponse est simplement de **cloner en local le fork** qui est sur votre repository GitHub. Vous aurez donc 2 actions à réaliser :

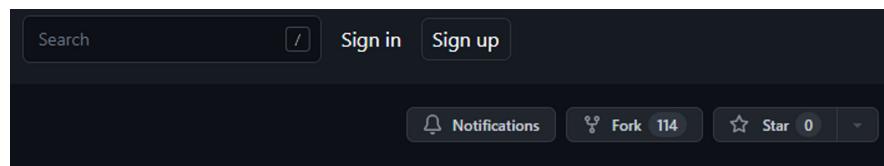
- **Forker le projet** – ce qui aura pour conséquence de **duplicer le projet** en créant un **nouveau repo dans votre compte GitHub**.
- **Cloner le nouveau repository** créé dans votre compte GitHub – ce qui va **créer une copie locale du fork sur votre ordinateur**.



## Apportez votre première contribution

Pour Forker le projet, il faut effectuer les manipulations dans un 1er temps sur le GitHub :

- Sur la page du projet, cliquer sur le bouton "Fork" en haut à droite



**Create a new fork**

A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. [View existing forks.](#)

**Owner \***      **Repository name \***

 VelStack  / 7688581-Expert-Git-GitHub ✓

By default, forks are named the same as their parent repository. You can customize the name to distinguish it further.

**Description (optional)**

**Copy the main branch only**  
Contribute back to OpenClassrooms-Student-Center/7688581-Expert-Git-GitHub by adding your own branch. [Learn more.](#)

ⓘ You are creating a fork in your personal account.

**Create fork**

*Normalement votre compte doit être affiché pour pouvoir fork le projet*

Si tout est bon, cliqué sur "**create fork**"

Une fois le projet créé, nous constatons que nous revenons sur notre compte avec le nouveau projet, avec l'indication que le projet viens d'un fork OpenClassroom.



Nous allons récupérer le lien du projet et en créer un clone sur notre PC avec la commande **git clone**

```
aresv@MSI MINGW64 ~/documents/projetOpenClassroom
$ git clone https://github.com/VelStack/7688581-Expert-Git-GitHub.git
Cloning into '7688581-Expert-Git-GitHub'...
remote: Enumerating objects: 15, done.
remote: Counting objects: 100% (15/15), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 15 (delta 6), reused 11 (delta 5), pack-reused 0
Receiving objects: 100% (15/15), 17.66 KiB | 1.96 MiB/s, done.
Resolving deltas: 100% (6/6), done.
```

Pour permettre à tous de contribuer, Open Transport utilise la fonctionnalité **GitHub Issues**.

**Les issues permettent d'identifier une tâche à réaliser sur le projet.** Elles peuvent représenter de **nouvelles fonctionnalités, des corrections, des problèmes à résoudre**, etc. Elles sont généralement **hiérarchisées**, par exemple par **type**, par **importance**, par **thématische**, etc.

Ainsi, une façon de commencer à contribuer est de lire les issues du projet et d'en choisir une à notre portée technique.  
Voici les issues du projet OpenTransport :

[la liste des issues](#)

Je vous propose de traiter la première issue, qui correspond à une erreur de texte dans le fichier README.md :

Texte en anglais dans le fichier README.md #1

Open romainsessa opened this issue on 16 Mar · 0 comments

romainsessa commented on 16 Mar

Collaborator

Le fichier REAME.md contient une sous section nommée "Deployment".  
Le langue pour le fichier est le français, il faut donc corriger le texte en "Déploiement".

Notre plan d'attaque est le suivant :

- **Modifier en local** le fichier README.md.
- Faire un **git add**, un **git commit** (avec le numéro de l'issue dans le message, cf. les règles de contribution) et un **git push** pour **envoyer la modification sur notre repository distant (le fork)**.
- Créer une **pull request** via l'interface graphique de GitHub.

```
aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/7688581-Expert-Git-Github (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

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:   README.md

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

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/7688581-Expert-Git-Github (main)
$ git add README.md

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/7688581-Expert-Git-Github (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   README.md

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/7688581-Expert-Git-Github (main)
$ git commit -m "Issue N°1 : Changement du mot Deployment par Déploiement dans le README"
[main 5cf905a] Issue N°1 : Changement du mot Deployment par Déploiement dans le README
 1 file changed, 1 insertion(+), 1 deletion(-)

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/7688581-Expert-Git-Github (main)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

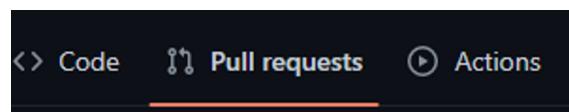
```

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 358 bytes | 358.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/VelStack/7688581-Expert-Git-GitHub.git
  273b379..5cf905a main -> main

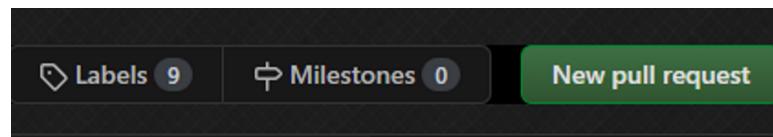
```

Concernant l'étape 3 :

Premièrement rendez-vous sur l'onglet **Pull Request** de votre **repository GitHub** :



Il faut cliquer sur le bouton "New Pull Request" qui apparaît sur la page



Vous devriez arriver sur cette page :

The screenshot shows the 'Comparing changes' page on GitHub. At the top, it says 'base repository: OpenClassrooms-Student-Cent...' and 'head repository: VelStack/7688581-Expert-Git-G...'. It indicates that the branches can be automatically merged. Below this, there's a section for discussing changes with others and a 'Create pull request' button. The main area shows a single commit made on Sep 14, 2022, by 'VelStack'. The commit message is 'Issue N°1 : Changement du mot Deployment par Déploiement dans le README'. The diff view shows a change in the README.md file where line 36 was modified from '- ## Deployment' to '+ ## Deploiement'. The commit also includes the message 'Voici les étapes à suivre pour déployer en production :'.

Notons avec intérêt que le premier encart gris clair indique bien que la **pull request va vers le repository de base** (lire la **ligne de droite à gauche, car la flèche est <-**), et qu'il y a la **mention "Able to merge"**. Tous les signaux sont au vert, il suffit désormais d'appuyer sur le bouton "Create Pull Request"

# Issue N°1 : Changement du mot Deployment par Déploiement README #150

The screenshot shows a GitHub pull request page. At the top, there's a green button labeled "Open" and a message: "VelStack wants to merge 1 commit into OpenClassrooms-Student-Center:main from VelStack:main". Below this, there are tabs for "Conversation" (0), "Commits" (1), "Checks" (0), and "Files changed" (1). A comment from "VelStack" is visible, stating "Changement du mot Deployment par Déploiement dans le README". Below the comment, the pull request summary is shown: "Issue N°1 : Changement du mot Deployment par Déploiement dans le README" and the commit hash "5cf905a". A note at the bottom says "Add more commits by pushing to the main branch on VelStack/7688581-Expert-Git-GitHub." A green icon with a checkmark and the text "This branch has no conflicts with the base branch" is present, followed by the subtext "Only those with write access to this repository can merge pull requests."

Comme le précisent les règles de contribution, votre pull request sera ensuite revue et traitée (sauf dans ce cours).

*Je vous propose de pratiquer une nouvelle fois en traitant l'issue n° 2 !*

*Les étapes sont les mêmes que dans l'exemple précédent :*

*Modifier en local le fichier.*

*Faire un git add, un git commit et un git push pour envoyer la modification sur notre repository distant (le fork).*

*Créer une pull request via l'interface graphique de GitHub.*

*C'est parti, à vous de jouer !*

```
aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

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: README.md

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

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git add README.md

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified: README.md
```

```

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/7688581-Expert-Git-Github (main)
$ git commit -m "Issue n°2 : rajout du nom de l'auteur du projet"
[main 0a4acc9] Issue n°2 : rajout du nom de l'auteur du projet
 1 file changed, 1 insertion(+), 1 deletion(-)

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/7688581-Expert-Git-Github (main)
$ git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 326 bytes | 326.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/VelStack/7688581-Expert-Git-Github.git
  5cf905a..0a4acc9 main -> main

```

En résumé :

- Avant de contribuer à un projet open source, prenez connaissance de ses **directives de collaboration**.
- Pour contribuer à un projet open source, vous devez le **forker** puis **cloner** le nouveau repository sur votre poste de travail.
- Identifiez votre périmètre d'action en consultant les **issues du projet**.
- **Proposez des modifications** au projet en réalisant des **pull requests**.

## Résolvez des conflits avec Git

Nous allons voir comment résoudre des conflits avec Git

Le travail collaboratif permet à plusieurs développeurs de manipuler les fichiers du projet. Mais que se passe-t-il si 2 développeurs modifient la même ligne d'un fichier ? Comment Git choisit-il la modification à retenir ?

Il va vous demander de faire ce choix vous-même. Comment ? En créant **un conflit**

Prenons l'exemple suivant : vous poussez votre mise à jour sur le repository GitHub distant. Le push est **refusé** car vous n'êtes **plus à jour avec ce repository GitHub distant**. En effet, d'autres commits ont été réalisés depuis votre dernière **synchronisation**.

Pour mettre à jour votre projet, vous exécutez la commande **git pull**. Cette dernière va récupérer les derniers commits puis faire un **merge** avec votre copie locale.

Un **conflit** est susceptible d'apparaître **pendant le merge** avec la copie local si un élément du fichier a été modifié pendant ce temps par un autre collaborateur.

*Le conflit existe uniquement chez la personne qui déclenche le merge. Autrement dit, votre collaborateur n'a pas conscience de l'existence du problème. Maintenant que vous savez cela, vous comprenez peut-être mieux pourquoi c'est parfois la "course au push" dans un projet, car c'est au second de résoudre les conflits.*

D'autre part, comme vous l'avez peut-être noté dans ce scénario, c'est l'**opération de merge qui a déclenché le conflit**. Vous pouvez donc déduire qu'**un merge entre 2 branches peut également créer des conflits**.

Pour simuler ce conflit sur votre poste de travail, suivez les étapes suivantes :

- git pull
- modifier en local le fichier README.md
- git add README.md

- git commit -m "#3 Ajout d'un prérequis"
- modifier via l'interface web de GitHub le fichier README.md (avec un texte différent de la modification locale) ;
- git push
- git pull.

```
aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git push
To https://github.com/VelStack/7688581-Expert-Git-GitHub.git
! [rejected]      main -> main (fetch first)
error: failed to push some refs to 'https://github.com/VelStack/7688581-Expert-Git-GitHub.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.

aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 689 bytes | 98.00 KiB/s, done.
From https://github.com/VelStack/7688581-Expert-Git-GitHub
  0a4acc9..0611fcf main      -> origin/main
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

Les 3 dernières lignes sont très claires : **le merge du fichier README.md a échoué**, et un “Merge conflict” a été créé.

Pour en savoir plus sur ce problème, utilisons la commande **git status**.

**git status** est une commande précieuse car elle nous informe de l'état de notre repository Git local. Si quelque chose ne va pas, cette commande saura nous le dire.

```
aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git
$ git status
On branch main
Your branch and 'origin/main' have diverged,
and have 1 and 1 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified: README.md

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

La phrase “**You have unmerged paths.**” signifie qu’**une opération de merge n'a pas pu aboutir**. Git nous propose ensuite 2 solutions :

- (fix conflicts and run “git commit”) : résoudre le conflit.
- (use “git merge –abort to abort the merge”) : annuler le merge, ce qui retire le conflit mais ne nous permet pas pour autant de pusher notre fichier README.md.

Si nous choisissons la solution 1, alors la suite nous intéresse particulièrement car cela **nous indique le ou les fichiers à corriger**.

```
(use "git add <file>..." to mark resolution)
  both modified: README.md
```

Ce texte nous informe que le fichier README.md a été modifié par 2 personnes. Suite à cela, sachez que **Git a modifié le fichier pour y écrire le détail du conflit.**

## Résolvez votre conflit

Nous avons 3 possibilités :

- Conserver la version locale aux dépens de la version distante.
- Conserver la version distante aux dépens de la version locale.
- Modifier le fichier en conflit et en faire une 3e version qui sera ensuite conservée.

**Il n'existe aucune réponse absolue à cette question ! Tout dépend de la situation dans laquelle vous vous trouvez.**  
Cela vous demandera peut-être d'échanger au préalable avec le collaborateur qui a travaillé sur le même fichier que vous.

Ceci dit, il vous sera certainement utile de comprendre le conflit au sein du fichier pour prendre une décision.

Voici une capture d'écran du contenu du fichier :

The screenshot shows a terminal window with a dark theme. The title bar says 'README.md !'. The window displays a GitHub pull request merge conflict. The code is as follows:

```
1 # Open Transport
2
3 Application web pour covoiturage.
4
5 ## Getting Started
6
7 Ces instructions permettent d'executer une copie du projet en local sur votre poste de travail pour
8 le développement et les tests. Référez-vous à la section "Déploiement" pour les étapes à suivre pour
9 déployer le projet en production.
10
11 Pour executer en local le projet Open Transport, vous devez au préalable installer :
12
13
14 Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
15 <<<<< HEAD (Current Change)
16 Java
17 =====
18 Java version 8 obligatoirement
19 >>>>> 0611fcb1df341b04a7857b0c96ad97c338537b38 (Incoming Change)
20
21
22 ## Installation
23
24 Voici les étapes à suivre pour avoir un environnement de développement et de test opérationnel :
25
26
```

The conflict is located between lines 14 and 18, where the 'Java' line from the incoming change is inserted before the 'Java' line from the current change. The status bar at the bottom of the terminal shows the commit hash '0611fcb1df341b04a7857b0c96ad97c338537b38' and '(Incoming Change)'.

Décryptons :

- Les chaînes de caractères <<<<< et >>>>> balisent la zone de conflit
- La première partie du texte qui suit le mot HEAD correspond à la modification locale
- La chaîne ===== indique la fin de cette modification, et il s'ensuit la modification distante en conflit

- Le texte **0611fcb1df341b04a7857b0c96ad97c338537b38** correspond au numéro d'identification unique du commit d'où provient le texte en conflit.

Nous devons donc désormais choisir entre :

- **conserver le texte : Java**
- **conserver le texte : Java version 8 obligatoirement**
- **faire une troisième version qui pourrait être un mix des deux premières options.**

Maintenant, l'important est que vous sachiez techniquement quelles commandes utiliser pour chacune de ses situations. C'est ce que nous allons voir maintenant !

### Utilisez la commande Git adaptée

- **Pour conserver la version distante et écraser la version local**, il faut utiliser la commande : **git checkout --theirs README.md**
- **Pour conserver la version locale et écraser la du dépôt distant**, il faut utiliser la commande : **git checkout --ours README.md**

*La commande **git checkout** est majoritairement utilisée pour basculer d'une branche à une autre. Lors d'un **conflict**, elle nous permet également de restaurer un fichier, et c'est l'**option --ours ou --theirs** qui permet de déterminer quelle version on veut restaurer.*

- Pour corriger le conflit manuellement, il faut ouvrir le fichier et modifier son contenu. Cette modification implique d'enlever les lignes ajoutées automatiquement par Git, et de conserver le texte de notre choix.

Suite au passage d'une des 3 commandes, il faudra à nouveau utiliser les commandes **git add**, **git commit** et **git push** pour refaire un push sur le dépôt distant.

En résumé :

- Git crée un conflit lorsqu'il ne sait pas quelle version d'un fichier il doit conserver. Cela arrive quand un fichier a été **modifié au même endroit par 2 sources distinctes**
- **git status** permet de **comprendre la nature du conflit** : c'est votre **point de départ** pour le résoudre.

À vous de choisir la solution qui vous convient le mieux pour résoudre votre conflit :

- **conserver la version distante**
- **conserver la version locale**
- **faire une version adaptée à la circonstance.**

### Corrigez l'historique du projet au fil de vos développements

Nous allons voir comment **soigner l'historique** de notre projet Git :

Un **repository Git** possède une mémoire, ce qui est une bonne nouvelle pour nous : La mémoire du repository nous permet d'être plus performants dans la gestion de nos projets. C'est ce que nous allons découvrir dans ce chapitre.

La mémoire du repository Git est nommée **historique**. Ce dernier contient la liste des **commits réalisés**.

*Chaque commit regroupe un ensemble d'informations (comme la date, l'auteur, le message, etc.). Il conserve également l'ID du commit précédent. Grâce à cela, Git peut retracer la chaîne des commits et fournir l'historique.*

Vous pouvez visualiser l'historique avec la commande **git log**. Voici un exemple de résultat pour cette commande :

```
$ git log
commit c3ca7e9f03c832b1a101a9d15196df9e86152f58 (HEAD -> main, origin/main)
Author: Romain Sessa <19792764+romainsessa@users.noreply.github.com>
Date:   Wed Mar 16 13:54:06 2022 +0100

    Update CONTRIBUTING.md

commit cba9b3cdf34e868bd7a938629cac11659f41f33f
Author: Romain Sessa <19792764+romainsessa@users.noreply.github.com>
Date:   Wed Mar 16 13:52:58 2022 +0100

    Update CONTRIBUTING.md

commit 81e203e7acb086b2b3a1ce0f01afe73894f9a170
Author: romainsessa <romainsessa@gmail.com>
Date:   Wed Mar 16 11:53:31 2022 +0100

    Fichiers de documentation : README, CONTRIBUTING, CODE OF CONDUCT, LICENCE
```

Pourquoi vouloir **corriger l'historique** ? Pour **corriger une erreur** ou avoir un **historique de commits le plus compréhensible possible** par exemple.

(*pour sortir du git log, il faut faire un !*)

## Faire une correction simple sur l'historique du projet

Commençons avec un cas simple : alors que vous venez de résoudre une issue et que vous avez fièrement committé le résultat de votre travail, vous vous rendez compte que **vous avez oublié un fichier**.

Comment feriez-vous pour corriger cela ?

- *Facile, git add du fichier manquant puis git commit !*

Certes, votre solution vous permettra d'obtenir le résultat souhaité, mais elle aura une **conséquence fâcheuse** : votre **historique contiendra 1 commit supplémentaire qui, fondamentalement, n'apporte rien**. En d'autres termes : **cette solution va polluer votre historique**.

- *Et alors, quel est le problème ?*

Imaginez cette situation à l'échelle de grands projets avec de nombreux collaborateurs : **l'historique deviendrait de plus en plus illisible**, et nous perdrons l'avantage d'une **träçabilité claire** des opérations sur le projet.

Git nous offre une solution simple avec l'option **--amend** de la commande **git commit** pour éviter cette situation. Voilà comment vous en servir :

- **modification de 2 fichiers** : README.md et CONTRIBUTING.md

- git add README.md
- git commit -m "Mise à jour de README.md et CONTRIBUTING.md".

```
aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git add README.md

aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git commit -m "Mise à jour des fichiers README.md et CONTRIBUTING.md"
[main 2482500] Mise à jour des fichiers README.md et CONTRIBUTING.md
  1 file changed, 1 insertion(+), 1 deletion(-)
```

On réalise notre erreur, nous avons **oublié** le fichier **CONTRIBUTING.md** :

- git add CONTRIBUTING.md
- git commit --amend

La commande **git commit --amend** implique la **mise à jour du message du commit**. Si vous ne souhaitez pas le modifier, ajoutez l'option **--no-edit**.

```
aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git commit --amend --no-edit
[main 7f91646] Mise à jour des fichiers README.md et CONTRIBUTING.md
  Date: Thu Sep 15 12:12:40 2022 +0200
  2 files changed, 2 insertions(+), 2 deletions(-)
```

## Modifiez précisément l'historique du projet

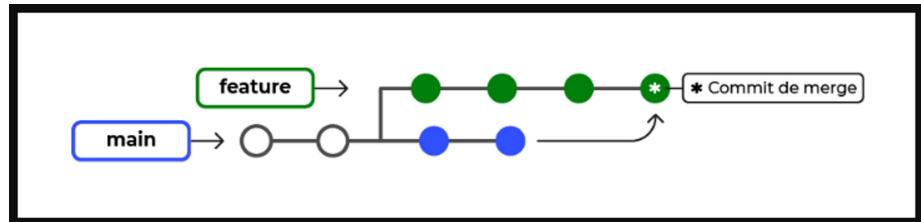
Attaquons-nous à un autre scénario ! En bon contributeur, vous ne travaillez pas sur la **branche main**, vous avez créé une branche nommée **feature** pour **modifier les fichiers**. Vous avez prévu de **merger feature sur main une fois votre contribution réalisée**.

Mais voilà, un **problème majeur** a été détecté par votre équipe sur la branche **main**, et il est résolu dans la foulée par **un collaborateur**. Vous pensiez que votre branche avait comme **point de départ** le dernier commit de la branche **main** mais ce n'est plus le cas ! Qui plus est, il **vous faut intégrer le correctif appliqué à la branche main à votre branche**.

Comment feriez-vous ?

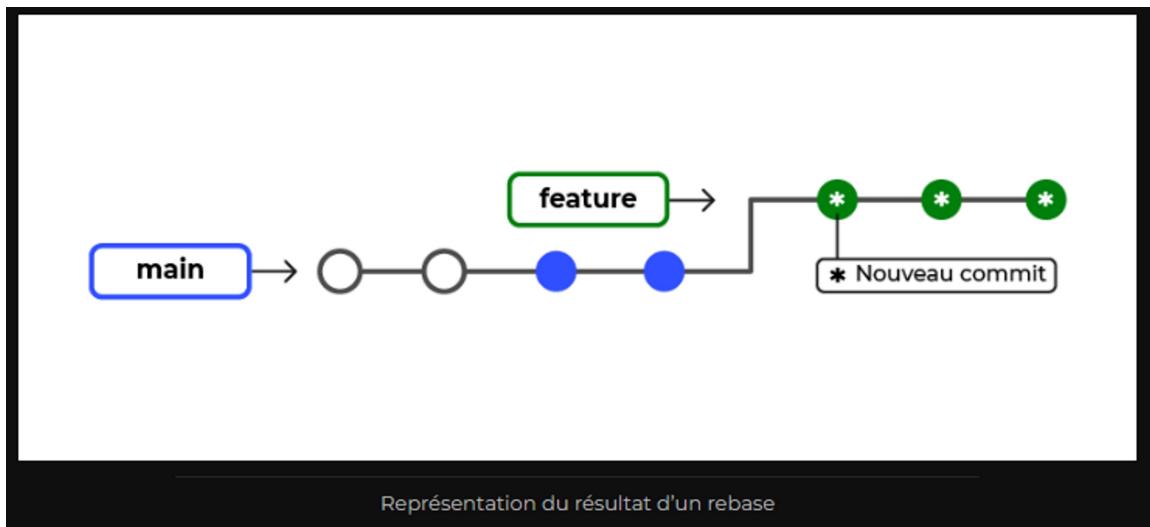
- *je me place sur ma branche, puis je fais un git merge main.*

Intéressant, vous allez en effet **intégrer le correctif à votre branche à travers un commit de merge**, mais ça a l'inconvénient de **polluer la lisibilité de l'historique**.



Une autre solution existe : la commande **git rebase**

Cette commande permet de **réécrire l'historique de la branche que l'on rebase**. Cela signifie que **cette branche est déplacée à la pointe de la branche main**, comme le montre l'image ci-dessous :



Pour illustrer cet exemple, nous allons appliquer les commandes suivantes :

- **Git status** pour vérifier que nous sommes dans la branche main
- **Git checkout -b feature** pour créer une nouvelle branche
- **Echo feature > feature.txt** pour créer un nouveau fichier (bien s'assurer qu'on est dans la branche **feature**)
- **Git add feature.txt** pour mettre notre fichier nouvellement créé dans la zone de l'index
- **Git commit -m "feature.txt ajouté"** pour créer un commit du nouveau fichier

```
aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-Github (main)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-Github (main)
$ git checkout -b feature
Switched to a new branch 'feature'

aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-Github (feature)
$ echo feature > feature.txt

aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-Github (feature)
$ git add feature.txt
warning: LF will be replaced by CRLF in feature.txt.
The file will have its original line endings in your working directory

aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-Github (feature)
$ git status
On branch feature
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   feature.txt

aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-Github (feature)
$ git commit -m "feature.txt ajouté"
[feature 6b5aec0] feature.txt ajouté
 1 file changed, 1 insertion(+)
 create mode 100644 feature.txt

aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-Github (feature)
$ ...
```

Une fois ce commit effectué, nous allons à nouveau nous placer sur la branche main avec la commande **git checkout main**

```
aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (feature)
$ git checkout main
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$
```

Nous allons créer un nouveau commit dans la branche main pour effectuer notre mise en situation, pour cela, nous allons effectuer les commandes suivantes :

- Echo "main" > main.txt
- Git add main.txt
- Git commit -m "main.txt ajouté"

```
aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ echo "main" > main.txt

aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git add main.txt
warning: LF will be replaced by CRLF in main.txt.
The file will have its original line endings in your working directory

aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git commit -m "main.txt ajouté"
[main 67eb580] main.txt ajouté
 1 file changed, 1 insertion(+)
 create mode 100644 main.txt
```

La branche main contient maintenant un commit qui n'existe pas dans la branche feature. Nous souhaitons intégrer ce commit à la branche feature sans polluer l'historique : pour ce faire, il faut repasser sur la branche feature avec la commande **git checkout feature**

```
aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git checkout feature
Switched to branch 'feature'

aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (feature)
$
```

Nous allons effectuer l'opération de rebase avec la commande **git rebase main**

```
aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (feature)
$ git rebase main
Successfully rebased and updated refs/heads/feature.
```

Nous venons de déplacer les **commit de la branche feature** à la **pointe des commit de la branche main**.

Grâce à cette opération, nous allons voir dans le **commit du fichier "main.txt"** est bien visible dans l'**historique de la branche feature** et nous verrons qu'il y a aussi le **fichier feature.txt**

```

laresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (feature)
$ git log
commit 0b53d5d601494b1b723ec1986eff6b8bd5a87cc7 (HEAD -> feature)
Author: Ve1Stack <chaussoy.benoit@gmail.com>
Date:   Thu Sep 15 12:34:25 2022 +0200

    feature.txt ajouté

commit 67eb580eff7792752d558ab972c08bd8f9fbcl12 (main)
Author: Ve1Stack <chaussoy.benoit@gmail.com>
Date:   Thu Sep 15 12:39:23 2022 +0200

    main.txt ajouté

commit 7f91646ccae5e0efalce236d05436b60cd73ebc
Author: Ve1Stack <chaussoy.benoit@gmail.com>
Date:   Thu Sep 15 12:12:40 2022 +0200

    Mise à jour des fichiers README.md et CONTRIBUTING.md

```

**Le dernier commit correspond au feature.txt et l'avant dernier correspond au main.txt** : nous sommes donc dans l'**ordre inverse de la chronologie des opérations car le commit de la branche feature a été déplacé à la pointe des commit de la branche main.**

Retenons qu'une fois positionné sur la branche feature, il ne vous reste plus qu'à exécuter **git rebase main**.

**Veillez à respecter cette règle d'or lorsque vous utilisez la commande rebase : Ne réalisez jamais un rebase d'une branche publique (c'est-à-dire, une branche utilisée par plusieurs personnes).**

Envie d'aller encore plus loin ? Il existe la fonctionnalité de **rebase interactif qui permet de modifier les commits de la branche rebasée**. Nous ne traiterons pas ce cas ensemble, mais la documentation est disponible à cette adresse : <https://git-scm.com/book/fr/v2/Utilitaires-Git-R%C3%A9CRIRE-L%20HISTORIQUE>

## Localisez et corrigez un bug sans impacter la tâche courante

*Le dernier scénario que nous traiterons dans ce chapitre est assez fréquent pour les développeurs. Vous êtes en plein développement d'une nouvelle fonctionnalité quand tout à coup, on vous informe d'un bug à résoudre en haute priorité. Ce bug doit être résolu sur la branche que vous utilisez.*

Quelles sont les étapes à réaliser dans cette situation :

Voici la méthode :

- Mettre de côté le développement en cours.
- Identifier le commit qui est à l'origine du bug.
- Annuler ce commit.
- Reprendre le développement qui a été mis de côté.

### Mettre de côté le développement en cours

Pour mettre nos fichiers de côté, il faut utiliser la commande **git stash**

```

$ git status
on branch develop
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified: contributeur.html
    modified: documentation.html
    modified: index.html

```

Récapitulons les commandes utilisées :

```
romai@PC-DELL-ROMAIN MINGW64 /c/workspace/dev/openclassrooms/expert_git_github/fork/7688581-Expert-Git-GitHub/src (develop)
$ git stash
Saved working directory and index state WIP on develop: 54a79e4 modification de la page contributeur
```

Pour vérifier que la commande **git stash** a bien fonctionné, on peut utiliser la commande **git status**

```
romai@PC-DELL-ROMAIN MINGW64 /c/workspace/dev/openclassrooms/expert_git_github/fork/7688581-Expert-Git-GitHub/src (develop)
$ git status
On branch develop
nothing to commit, working tree clean
```

### Identifier le commit qui est à l'origine du bug

Le problème à résoudre est une suppression non désiré du fichier contact.html, il faut trouver le commit à l'origine de cette erreur. Pour cela nous allons utiliser la commande **git bisect** qui va nous placer sur différents commit que nous allons vérifier avec la commande UNIX **ls -la**

- Si le fichier est présent, il faut taper **git bisect good**
- S'il n'est pas présent, il faut taper la commande **git bisect bad**
- Pour commencer la procédure de vérification, il faut impérativement taper la commande **git bisect start**

```
romai@PC-DELL-ROMAIN MINGW64
$ git bisect start
```

Il faut à présent indiquer un commit erroné, il faut utiliser la commande **git bisect bad HEAD** qui correspond à la pointe de la branche

```
romai@PC-DELL-ROMAIN MINGW64
$ git bisect bad HEAD
```

Nous devons fournir un commit valide : il faut afficher notre historique avec **git log**

```
commit 30f37015e35aecc80e0855b357a8f9b7db171849
Author: romainsessa <romainsessa@gmail.com>
Date:   Wed Apr 27 18:10:57 2022 +0200

    Ajout de la page d'index et de la page de contact

commit c3ca7e9f03c832b1a101a9d15196df9e86152f58 (origin/main, origin/HEAD, main)
Author: Romain Sessa <19792764+romainsessa@users.noreply.github.com>
Date:   Wed Mar 16 13:54:06 2022 +0100
```

Prenons celui datant du 16, il faut copier son id et taper **git bisect good id du commit**

```
romai@PC-DELL-ROMAIN MINGW64 /c/workspace/dev/openclassrooms/expert_git_github/f
$ git bisect good c3ca7e9f03c832b1a101a9d15196df9e86152f58
Bisecting: 3 revisions left to test after this (roughly 2 steps)
[04cf5764c1a526bc10e7f69ca50b55d01086c281] ajout de la page de documentation
```

Nous sommes placé sur différents commit, nous devons vérifier si notre fichier contact.html est présent avec la commande **ls -la**

```
romai@PC-DELL-ROMAIN MINGW64 /c/workspace/dev/openclassrooms
$ ls -la
total 11
drwxr-xr-x 1 romai 197609 0 Apr 27 18:40 .
drwxr-xr-x 1 romai 197609 0 Apr 27 18:09 ../
-rw-r--r-- 1 romai 197609 14 Apr 27 18:40 contact.html
-rw-r--r-- 1 romai 197609 20 Apr 27 18:40 documentation.html
-rw-r--r-- 1 romai 197609 24 Apr 27 18:40 index.html
```

Ici notre commit est valide, nous pouvons indiquer, en nous replaçant à la racine du dossier, que ce commit est good avec la commande **git bisect good**

```
romai@PC-DELL-ROMAIN MINGW64 /c/
$ cd ..

romai@PC-DELL-ROMAIN MINGW64 /c/
$ git bisect good
```

Nous sommes placé sur un autre commit, effectuons la même opération : ls -la dans le dossier concerné

```
romai@PC-DELL-ROMAIN MINGW64 /c/workspace/dev/openclassrooms
$ ls -la src/
total 10
drwxr-xr-x 1 romai 197609 0 Apr 27 18:40 .
drwxr-xr-x 1 romai 197609 0 Apr 27 18:09 ../
-rw-r--r-- 1 romai 197609 53 Apr 27 18:40 documentation.html
-rw-r--r-- 1 romai 197609 47 Apr 27 18:40 index.html
```

Notre fichier n'est pas présent, il faut taper **git bisect bad**

En effectuant ces opérations, le process va pouvoir identifier le commit qui pose problème : ici c'était le commit suivant qui avait enregistré la suppression du fichier.

```
romai@PC-DELL-ROMAIN MINGW64 /c/workspace/dev/openclassrooms/expe
$ git bisect bad
c85c9a7461722bb34fd818ecf89d47d8258d04f6 is the first bad commit
commit c85c9a7461722bb34fd818ecf89d47d8258d04f6
Author: romainsessa <romainsessa@gmail.com>
Date:   Wed Apr 27 18:13:57 2022 +0200

    mis à jour de la page de contact

src/contact.html | 1 -
1 file changed, 1 deletion(-)
delete mode 100644 src/contact.html
```

Une fois identifié, il faut récupérer l'ID du commit et ensuite interrompre le processus **git bisect** avec **git bisect reset**

### Annuler ce commit

Nous allons maintenant **revert** le commit incriminé pour annuler son effet avec la commande **git revert ID du commit**

```
romai@PC-DELL-ROMAIN MINGW64 /c/workspace/dev/openclassrooms
$ git revert c85c9a7461722bb34fd818ecf89d47d8258d04f6
[develop e9d6974] Revert "mis à jour de la page de contact"
 1 file changed, 1 insertion(+)
  create mode 100644 src/contact.html
```

## Reprendre le développement qui a été mis de côté

Pour récupérer le travail mis de côté : **git stash pop**

```
romai@PC-DELL-ROMAIN MINGW64 /c/workspace/dev/openclassrooms/expert_git
$ git stash pop
On branch develop
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:   src/contributeur.html
    modified:   src/documentation.html
    modified:   src/index.html

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (73caa6358e62cdcb42fdaeb0eae040f596752f73)
```

Pour récapituler les commandes utilisées dans cet exemple

- **git stash** permet de mettre de côté le développement en cours
- **git stash pop** permet de récupérer ce qui a été stashé
- **git bisect** (et ses options start, good, bad, reset) sert à **identifier un commit problématique** en parcourant l'historique
- **git revert** permet de créer un **nouveau commit qui contient l'inverse du commit concerné, ce qui revient à annuler l'effet de ce dernier.**

Il est important de vérifier si toutes les branches ont bien été importées :

Dès le clonage d'un dépôt : **git remote add nom de la branche à importer - lien du dépôt**

```
aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git remote add p1c4 https://github.com/OpenClassrooms-Student-Center/7688581-Expert-Git-GitHub.git
```

Toutefois, cette branche ne sera pas importée en local : il faudra faire la commande :

```
aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git fetch https://github.com/OpenClassrooms-Student-Center/7688581-Expert-Git-GitHub.git p1c4:p1c4

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git branch
* main
  p1c4
```

## En résumé

- **git commit --amend** permet de modifier le dernier commit.
- **git rebase** permet de réécrire l'historique d'une branche.
- **git stash** permet de mettre de côté le développement en cours.
- **git stash pop** permet de récupérer le développement mis de côté.

- **git bisect** permet d'identifier un commit problématique en parcourant l'historique.
- **git revert** permet d'annuler un commit.

## Structurer la collaboration grâce à GitFlow

Git et GitHub sont des **outils clés** pour **collaborer** sur un projet. Pour Open Transport, ce sont également de vrais atouts pour intégrer de nouveaux collaborateurs car c'est un outil connu et maîtrisé par la plupart d'entre eux. Mais attention, **cela ne garantit pas une collaboration efficace** :

Il faut simplement garder en tête que, **même si 2 personnes ont un outil en commun, cela ne garantit pas automatiquement une collaboration efficace** :

**Vos collaborateurs maîtrisent l'outil**, mais ils ont peut-être chacun leurs **propres habitudes** :

- quant à la façon de **nommer les branches**, et dans quel cas les **créer ou les merge**
- quant au **niveau de détail des informations à fournir dans les messages de commit**
- quant à la **catégorisation des tâches**
- Etc...

Pour mettre tout le monde d'accord, il existe des **workflows**

## Découvrez les workflows

**Workflow** : il s'agit d'**instructions ou de recommandations** à suivre pour utiliser Git et GitHub de façon efficace. C'est, en quelque sorte, **une ligne de conduite pour tous les collaborateurs**.

Git en lui-même ne pousse pas à utiliser un workflow en particulier, car il se veut le **plus flexible possible**. Ce sont donc **les collaborateurs qui décident du workflow à appliquer**. Deux options se présentent à eux : **créer pour chaque projet son propre workflow**, ou bien **réutiliser des workflows** qui ont fait leurs preuves. Nous allons voir les 5 **workflow** les plus connus d'entre eux :

- **Le workflow centralisé (Centralized Workflow)** : Le workflow centralisé est la méthode la plus simple pour utiliser Git. Son principe ? **Tous les collaborateurs utilisent la même branche**. Ainsi, on crée d'abord un repository centralisé, qui est ensuite cloné par tous les collaborateurs. Puis, on alimente ce repository par une suite de git pull / git push. Les conflits sont gérés sur la branche main utilisée par tous les collaborateurs.
- **Le workflow de création de branches de fonctionnalités (Feature Branch Workflow)** : Son principe ? **Pour chaque nouvelle fonctionnalité, une branche est créée. La branche main ne contient pas de fonctionnalités en cours de développement**. Dans la pratique, on crée un dépôt centralisé, qui est ensuite cloné par tous les collaborateurs. Ensuite, pour chaque fonctionnalité, on crée une nouvelle branche. Lorsqu'une fonctionnalité est terminée, la branche de fonctionnalité doit être mergée sur la branche main via une pull request.
- **Le workflow basé sur le tronc (Trunk Based Development)** : En réponse à la montée en puissance des techniques d'intégration continue et de développement continu, ce workflow est de plus en plus utilisé.

Son principe ? **Les opérations de merge sur la branche principale** (nommée Trunk) sont à haute fréquence, tout en permettant au trunk d'être toujours opérationnel. On s'inspire ici des **principes agiles** et de la **volonté d'itérer rapidement**.

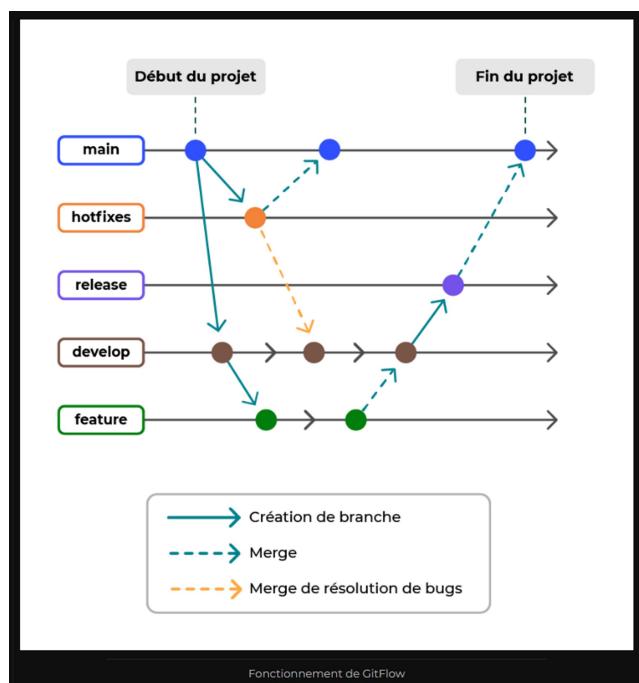
Son principal atout est de résoudre certains inconvenients de GitFlow tels que les **branches de fonctionnalités** qui s'éternisent dans le temps, ou la **complexité** qui en découle lors du merge.

- **Le workflow GitFlow** : C'est une évolution du workflow de création de branches de fonctionnalités présentée par **Vincent Driessen** en **2010** dans l'article (en anglais) "[A successful Git branching model](#)". Son principe ? **Définir encore plus précisément les rôles des branches et les interactions entre elles**.
- **Le workflow de Duplication (Forking Workflow)** : Ce workflow diffère des précédents dans la mesure où il peut être **combiné** avec ceux-ci.

Son principe ? Le **gestionnaire du projet** crée un **dépôt officiel du projet**. Chaque **collaborateur** crée ensuite un **double (fork)** côté serveur de ce dépôt officiel. Puis, il crée un **dépôt local (clone)** sur la base du fork. Pour contribuer au dépôt officiel, les collaborateurs utilisent **des pull requests**. C'est ensuite le **gestionnaire du projet** qui **validera ou non les contributions**.

le workflow de duplication est généralement utilisé dans les **collaborations open source**. (Ce cours est un exemple du workflow de duplication).

Dans ce chapitre, nous allons découvrir plus amplement **GitFlow**, qui est un workflow très répandu aujourd'hui. Commençons par analyser son fonctionnement grâce au schéma ci-dessous :



GitFlow repose sur le **workflow de création de branches de fonctionnalités**. Ainsi, on dispose d'**une branche par fonctionnalité** (de type **feature**). Mais il introduit également :

- **une branche**, nommée **develop**, qui sert à **intégrer les branches de fonctionnalités**
- des **branches de livraison**, de type **release**, qui vont **regrouper un ensemble de fonctionnalités identifiées comme finalisées**
- la **branche main**, qui contient **uniquement du code provenant des branches de release**
- des **branches dédiées à la résolution des bugs**, de type **hotfix**, pour traiter des bugs détectés dans la branche main.

**GitFlow impose une procédure de travail contraignante.** Ce qui est un **vrai avantage** dans les **situations suivantes** :

- vous êtes le **gestionnaire d'un projet open source** et vous souhaitez **contrôler efficacement les contributions**
- vous **travaillez avec de nombreux développeurs juniors** qui ont besoin d'être cadrés
- vous cherchez à **limiter les risques de corruption** sur le **code d'un projet d'envergure**.

D'un autre côté, cette **procédure de travail contraignante** peut devenir un **réel inconvénient** dans les situations suivantes où vous travaillez :

- dans une **startup qui a une volonté d'aller très vite** dans le **développement du produit**
- avec des **développeurs expérimentées**
- dans un projet où l'**agilité** et les **itérations rapides** sont **poussées à l'extrême**.

Dans **ces différentes situations**, **GitFlow devient un inconvénient**, car il **induit des latences dans la gestion des branches** qui peuvent **freiner l'avancée du projet**.

### Utilisez les commandes GitFlows

L'un des avantages de GitFlow est que vous pouvez faciliter sa mise en place et, surtout, y ajouter des commandes, en utilisant un plug-in. Faisons une démonstration :

*Si vous avez installé **Git for Windows**, le plugin **Git Flow** est embarqué. Si vous utilisez Mac OSX ou Linux, je vous encourage à consulter la documentation officielle pour installer le plug-in.*

Nous devons créer un nouveau dossier pour créer un nouveau repository git : nous l'appellerons **GitFlow**

```
aresv@MSI MINGW64 ~/Documents/projetOpenClassroom
$ mkdir gitflow

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom
$ cd gitflow

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/gitflow
$ ..
```

Pour initier notre projet gitflow, il faudra taper la commande : **git flow init**

```
aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/gitflow
$ git flow init
Initialized empty Git repository in C:/Users/aresv/Documents/projetOpenClassroom/gitflow/.git/
No branches exist yet. Base branches must be created now.
Branch name for production releases: [master] ..
```

Suite à cette commande, il va nous demander **diverses informations** : **le nom de la branche master** (que j'appelerais **main**), nous laisserons les autres valeurs par défaut, il suffit maintenant d'appuyer sur entrée jusqu'à la fin des questions.

```
aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/gitflow
$ git flow init
Initialized empty Git repository in C:/Users/aresv/Documents/projetOpenClassroom/.git/
No branches exist yet. Base branches must be created now.
Branch name for production releases: [master] main
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
Hooks and filters directory? [C:/Users/aresv/Documents/projetOpenClassroom/gitflow/.git/hooks]

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/gitflow (develop)
$
```

Vérifions les branches avec une commande **git branch -v**

```
aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/gitflow (develop)
$ git branch -v
* develop 386542c Initial commit
  main     386542c Initial commit
```

Nous avons **deux branches** : la **branche main** qui aura le **code de production** et la **branche dévelop** qui va **accueillir le code des branches de fonctionnalité**.

Créons une branche de fonctionnalité avec la commande **git flow feature start fonctionnalite1**

```
aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/gitflow (develop)
$ git flow feature start fonctionnalite1
Switched to a new branch 'feature/fonctionnalite1'

Summary of actions:
- A new branch 'feature/fonctionnalite1' was created, based on 'develop'
- You are now on branch 'feature/fonctionnalite1'

Now, start committing on your feature. When done, use:
  git flow feature finish fonctionnalite1

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/gitflow (feature/fonctionnalite1)
```

Nous sommes automatiquement placé sur la **nouvelle branche**, profitons-en pour créer un fichier et le committer

```
aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/gitflow (feature/fonctionnalite1)
$ git add test.txt
warning: LF will be replaced by CRLF in test.txt.
The file will have its original line endings in your working directory

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/gitflow (feature/fonctionnalite1)
$ git commit -m "test.txt ajouté"
[feature/fonctionnalite1 46d5b02] test.txt ajouté
 1 file changed, 1 insertion(+)
 create mode 100644 test.txt
```

Rajoutons le maintenant à la branche **develop** avec la commande **git flow feature finish fonctionnalite1**

La **branche fonctionnalite1** sera merger avec la branche **develop** puis supprimé

```

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/gitflow (feature/fonctionnalite1)
$ git flow feature finish fonctionnalite1
Switched to branch 'develop'
Updating 386542c..46d5b02
Fast-forward
 test.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 test.txt
Deleted branch feature/fonctionnalite1 (was 46d5b02).

Summary of actions:
- The feature branch 'feature/fonctionnalite1' was merged into 'develop'
- Feature branch 'feature/fonctionnalite1' has been locally deleted
- You are now on branch 'develop'

```

Pour **envoyer en production** une **feature terminé**, il faut la **réalise** avec une nouvelle commande : `git flow release start '0.0.1'`

```

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/gitflow (develop)
$ git flow release start '0.0.1'
Switched to a new branch 'release/0.0.1'

Summary of actions:
- A new branch 'release/0.0.1' was created, based on 'develop'
- You are now on branch 'release/0.0.1'

Follow-up actions:
- Bump the version number now!
- Start committing last-minute fixes in preparing your release
- When done, run:

    git flow release finish '0.0.1'

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/gitflow (release/0.0.1)
$ ...

```

Cette branche de release nous permet de faire d'ultime modification avant un envoi en production : ici il n'y en a pas, nous pouvons directement l'envoyer en production avec la commande : `git flow release finish '0.0.1'`

Il faudra répondre à divers réponse puis valider, nous pouvons vérifier le merge avec la commande `ls -la`

```

aresv@MSI MINGW64 ~/Documents/projetOpenClassroom/gitflow (main)
$ ls -la
total 9
drwxr-xr-x 1 aresv 197609 0 Sep 17 19:31 .
drwxr-xr-x 1 aresv 197609 0 Sep 17 19:06 ..
drwxr-xr-x 1 aresv 197609 0 Sep 17 19:32 .git/
-rw-r--r-- 1 aresv 197609 6 Sep 17 19:31 test.txt

```

Notre fichier est bien présent, notre modification a été envoyé en production

Récapitulons ce qui a été réalisé :

- Nous avons initialisé notre projet avec **git flow init**. Cela a eu pour conséquence non seulement de créer la branche main comme nous en avions l'habitude, mais également la branche develop, et de nous positionner sur cette dernière.
- Nous avons créé une nouvelle branche de fonctionnalité avec **git flow feature start fonctionnalite1**. La branche fonctionnalite1 a été créée à partir de develop.
- Nous avons réalisé un commit sur fonctionnalite1, puis exécuté **git flow feature finish fonctionnalite1**. Ainsi la branche fonctionnalite1 a été merge sur develop.

- Enfin nous avons créé la release avec **git flow release start 0.0.1**, et nous l'avons immédiatement intégrée à la branche main avec **git flow release finish '0.0.1'**.

Je vous conseille de **créer une pull request de votre travail entre les étapes 2 et 3**, de manière à ce qu'il soit **validé par qui de droit sur le repository distant**

En résumé :

- **Les workflows Git** permettent de **normaliser l'utilisation de Git au sein d'une équipe ou d'un groupe de collaborateurs.**
- Les principaux workflows sont :
  - o **le workflow centralisé**
  - o **le workflow basé sur la création de branches de fonctionnalités**
  - o **le workflow basé sur le tronc**
  - o **le workflow de duplication**
  - o **le GitFlow.**
- **GitFlow** est un **workflow répandu** qui étend le **workflow basé sur la création de branches de fonctionnalités**, en **normalisant les branches et les interactions entre elles.**
- **Un plugin GitFlow** donne **accès à des commandes facilitant la mise en œuvre de GitFlow.**

## Automatisez des traitements sur vos projets Git grâce aux hooks

Git possède un moyen pour **automatiser des contrôles et traitements** lors des **opérations Git : les Hooks**

**Hooks** : Un **hook** est un **script personnalisé** qui peut être lancé lors de certaines actions. Ce script peut ainsi effectuer des **opérations automatiquement** (plus besoin de répéter 100 fois la même chose !). Ça peut vous être très utile si, par exemple, vous voulez vérifier le contenu du message de commit.

*La documentation française de Git utilise la traduction **crochet** ; cependant, je fais le choix de garder le mot anglais **hook** car c'est bien ce dernier qui est utilisé dans la pratique.*

## Découvrez le fonctionnement des hooks

En premier lieu, il est important de comprendre que les **hooks** peuvent se situer **côté client ou côté serveur** :

- le **côté client** correspond à ce qui se passe sur le **repository Git** qui est sur **votre poste de travail**
- le **côté serveur** correspond à ce qui se passe sur le **repository Git distant, hébergé** sur **GitHub** par exemple.

Ainsi, les **hooks côté client** permettent de **contrôler ou d'ajouter des traitements jusqu'au moment où les fichiers sont envoyés sur le repository distant**. Les **hooks côté serveur** permettront de **contrôler ou d'ajouter des traitements lorsque des fichiers arriveront sur le serveur** (par exemple à la suite d'un git push).

*Attention, nous n'avons pas toujours la main sur le serveur. Il est possible que nous ne puissions pas manipuler les hooks des repositories distants, ou alors selon les limites de ce que nous permet l'hébergeur.*

La bonne nouvelle pour découvrir les **hooks** est que Git nous a généreusement fourni un ensemble d'exemples : vous les trouverez dans le répertoire **.git/hooks** accessible via le **répertoire racine de votre repository**.

```
aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub/.git
/hooks (GIT_DIR!)
$ ls -la
total 53
drwxr-xr-x 1 aresv 197609 0 Sep 15 15:52 .
drwxr-xr-x 1 aresv 197609 0 Sep 17 19:47 ../
-rw xr-xr-x 1 aresv 197609 478 Sep 15 15:52 applypatch-msg.sample*
-rw xr-xr-x 1 aresv 197609 896 Sep 15 15:52 commit-msg.sample*
-rw xr-xr-x 1 aresv 197609 4655 Sep 15 15:52 fsmonitor-watchman.sample*
-rw xr-xr-x 1 aresv 197609 189 Sep 15 15:52 post-update.sample*
-rw xr-xr-x 1 aresv 197609 424 Sep 15 15:52 pre-applypatch.sample*
-rw xr-xr-x 1 aresv 197609 1643 Sep 15 15:52 pre-commit.sample*
-rw xr-xr-x 1 aresv 197609 416 Sep 15 15:52 pre-merge-commit.sample*
-rw xr-xr-x 1 aresv 197609 1374 Sep 15 15:52 pre-push.sample*
-rw xr-xr-x 1 aresv 197609 4898 Sep 15 15:52 pre-rebase.sample*
-rw xr-xr-x 1 aresv 197609 544 Sep 15 15:52 pre-receive.sample*
-rw xr-xr-x 1 aresv 197609 1492 Sep 15 15:52 prepare-commit-msg.sample*
-rw xr-xr-x 1 aresv 197609 2783 Sep 15 15:52 push-to-checkout.sample*
-rw xr-xr-x 1 aresv 197609 3650 Sep 15 15:52 update.sample*
```

Chaque fichier correspond à un **hook utilisable**. Il est intéressant de s'arrêter sur les préfixes **pre** et **post**. Ils permettent de définir si le **hook s'exécute avant ou après une opération**.

De ce fait, les **hooks préfixés** par le mot “**pre**” peuvent être **bloquants** pour l’**opération ciblée**. Bien souvent, ils servent à effectuer des contrôles. Par exemple, dans la liste des hooks disponibles, il y en a un nommé “**pre-commit.sample**”. Le contenu de ce hook sera déclenché lors d’un **commit**, et l’exécution a lieu avant le **commit**.

En revanche, les **hooks préfixés** par le mot “**post**” ne seront pas bloquants. Ils s’exécutent après que l’**opération ciblée** a été réalisée avec succès. Ils nous sont donc très utiles pour appliquer des traitements automatiques, par exemple pour appliquer le **workflow Git** défini sur notre projet.

Le contenu du répertoire **.git/hooks** n’est pas cloné lorsque vous faites un **git clone**. Ainsi, il est nécessaire de mettre en place une procédure pour que tous vos collaborateurs récupèrent et appliquent les mêmes hooks.

## Mettre en place un Hook

Notre objectif est de **mettre en place un hook qui contrôlera le message des commits**. Si ce message contient le **numéro de l’issue** alors c’est gagné, sinon le commit est refusé.

Pour ce faire, nous allons adapter un **hook** existant : **commit-msg.sample**.

Pour ce faire, il faut se rendre dans le dossier **.git/hooks** et utiliser un éditeur de texte pour modifier notre fichier

```

commit-msg.sample
C:\> Users > aresv > Documents > projetOpenClassroom > 7688581-Expert-Git-GitHub >.git > hooks > $ commit-msg.sample
1  #!/bin/sh
2  #
3  # An example hook script to check the commit log message.
4  #
5  # Called by "git commit" with one argument, the name of the file
6  # that has the commit message. The hook should exit with non-zero
7  # status after issuing an appropriate message if it wants to stop the
8  # commit. The hook is allowed to edit the commit message file.
9  #
10 # To enable this hook, rename this file to "commit-msg".
11 #
12 # Uncomment the below to add a Signed-off-by line to the message.
13 # Doing this in a hook is a bad idea in general, but the prepare-commit-msg
14 # hook is more suited to it.
15 #
16 # SOB=$(git var GIT_AUTHOR_IDENT | sed -n 's/^(\.*>\).*/$&/Signed-off-by: \1/p')
17 # grep -qs "$SOB" "$1" || echo "$SOB" >> "$1"
18 #
19 # This example catches duplicate Signed-off-by lines.
20 #
21 test "" = "$(grep '^Signed-off-by: '$1" |
22 | sort | uniq -c | sed -e '/^[\ ]*[ ]*\d*/d')" || {
23 echo >2 Duplicate Signed-off-by lines.
24 exit 1
25 }

```

Pour ce cours, il faut remplacer le code utilisé dans le fichier par ce dernier :

```

LINE_READ=0
REGEX="^(#)([0-9]+)"
while read line;
do
if [[ "$LINE_READ" == "0" ]]; then
if ! [[ $line =~ $REGEX ]]; then
echo -e "Votre message de commit ne commence pas par le numéro de l'issue précédée du caractère #"
exit 1
fi
LINE_READ=$((LINE_READ + 1))
fi
done < $1

```

**La variable REGEX contient la regex vérifiant que la chaîne de caractères commence avec un # suivi d'un ou plusieurs chiffres.**

**La boucle while permet de parcourir les lignes du message de commit. Si la première ligne ne correspond pas à la regex, on sort avec un status 1, ce qui a pour conséquence de stopper le commit.**

Une fois la manipulation effectuée, vous pouvez sauvegarder et fermer le fichier.

Il faut maintenant supprimer l'extension du fichier (supprimer le .sample)

Nom	Modifié le	Type	Taille
applypatch-msg	15/09/2022 15:52	Fichier SAMPLE	1 Ko
commit-msg	18/09/2022 19:50	Fichier	2 Ko

En le renommant, ce **hooks sera maintenant pris en considération par Git** lors de l'exécution de la commande **git commit**

Nous allons maintenant créer un nouveau fichier pour effectuer un commit

```
aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ echo "temp" > temp.txt

aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git add temp.txt
warning: LF will be replaced by CRLF in temp.txt.
The file will have its original line endings in your working directory

aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git commit -m "Rajout du fichier temps"
Votre message de commit ne commence pas par le numéro de l'issue précédée du caractère #
```

Ici, nous testons le commit sans avoir précédé notre message par le caractère #, notre commit est donc **refusé**

Essayons en rajoutant un #1

```
aresv@MSI MINGW64 ~/documents/projetOpenClassroom/7688581-Expert-Git-GitHub (main)
$ git commit -m "#1 Rajout du fichier temps"
[main 288a5ab] #1 Rajout du fichier temps
 1 file changed, 1 insertion(+)
   create mode 100644 temp.txt
```

Récapitulons les étapes :

- J'ai modifié le contenu du fichier **commit-msg.sample** pour y ajouter le code ci-dessus
- J'ai renommé **commit-msg.sample** en **commit-msg** pour activer ce hook
- J'ai testé le bon fonctionnement du hook avec un commit sans le numéro d'issue, et un autre avec le numéro d'issue.

Finalement, d'un point de vue purement Git, la fonctionnalité des hooks est assez simple. La complexité réside plutôt dans le fait qu'écrire un hook requiert des compétences de scripting. Cela vous permettra d'apporter une vraie plus-value à vos projets. Aussi, je vous conseille de développer ces compétences ou de vous rapprocher des administrateurs système (#LanceurDeSort) de votre entreprise pour tirer pleinement profit des hooks.

En résumé :

- Les **hooks** permettent de lancer des scripts personnalisés, et ainsi d'effectuer des contrôles et des traitements lors des opérations Git.
- Les **hooks côté client** sont les scripts exécutés sur votre repository local.
- Les **hooks côté serveur** sont les scripts exécutés sur le repository distant.
- Git fournit un ensemble de hooks utilisables et adaptables dans le répertoire `.git/hooks`.