



Formation Git et Gitlab

Apprendre à utiliser le gestionnaire de versions Git et la forge logicielle Gitlab en tant qu'utilisateur de R et RStudio

Sommaire

- 1 Pourquoi Git ?**
- 2 Les bases de Git**
- 3 Les zones de Git**
- 4 Travailler avec une forge**
- 5 Les branches**
- 6 Pour en savoir plus**



0.1 Avant-propos

Ce diaporama de formation a été rédigé dans le but d'être le support visuel des formations dispensées au [MASAF](#).

Ces formations s'adressent à des agents **qui découvrent le contrôle de version**.

Elles sont données en présentiel sur une durée **d'une journée** à partir de juin 2024.

Champ couvert par cette formation

Ce support ne couvre pas tous les aspects du contrôle de version ni toutes les commandes Git.

De même, il est orienté pour être utile aux agents du SSM MASAF et se concentre donc sur une utilisation de R via [RStudio](#) qui est mise à disposition des agents sur la plateforme interne Cerise basée sur RStudio Workbench.

L'utilisation avec d'autres IDE tels que [VS Code](#) ou encore [Github Desktop](#) n'est donc incluse.

Enfin, en tant que forge interne du MASAF, les fonctionnalités principales de [Gitlab](#) sont présentées dans ce support. Même si la majorité des concepts de base sont équivalents, [Github](#) n'est pas abordé dans ce diaporama de formation.



1 Pourquoi Git ?





1.1 Le versioning

Principe général :

Le versioning consiste à conserver la version d'une entité logicielle quelconque de façon à pouvoir la retrouver facilement même après l'apparition et la mise en place de versions plus récentes.

Sans outil de versioning

Exemple de ce que l'on peut trouver sur un server :

	Nom
	...
■	Abattage-R_1.1
■	Abattage-R_2.0.1
■	Abattage-R_2.1
■	Abattage-R_1.2
■	Abattage-R_2.0.2
■	Abattage-R_2.2
■	Abattage-R_2.0.3

1.2 Avantages du versioning

Le versioning permet de :

- Obtenir de **la traçabilité** : on sait qui a changé quoi, quand, comment et pourquoi
- **Travailler collectivement** sur les mêmes programmes **en même temps**
- **L'expérimentation** lors des développements sans risques (à l'aide des branches)
- Faire de la **relecture** (revues de codes...)
- **Revenir en arrière**

Avec une forge, il permet en plus de :

- D'archiver le code
- D'apporter de la visibilité à vos projets

=> Un usage qui se justifie même quand on est tout seul sur un projet !



1.3 Git

Parmi les logiciels de versioning existants, Git est le plus populaire depuis quelques années :

What are the primary version control systems you use?



Source : [Stackoverflow](#)

1.4 À propos de Git

Git est un logiciel libre de versioning.

Créé en 2005 par [Linus Torvalds](#) et utilisé pour le code source de Linux.

- Codé en C
- Plus de 1 200 000 commits en mars 2024 !



Source : Wikipédia

⋮



2 Les bases de Git



2.1 Qu'est-ce qu'on versionne ?

On versionne les fichiers de type texte

Par exemple :

- Les programmes R, Python, SAS...
- La documentation au format texte, Markdown...
- Les fichiers quarto, Rmarkdown...
- Les fichiers de configuration de type yaml par exemple...

Éventuellement, on peut aussi versionner **des images** si on en a besoin dans une application ou une documentation.

2.2 Qu'est-ce qu'on NE versionne PAS ? (1/2)

TOUT LE RESTE :)

C'est-à-dire notamment les fichiers tableurs, de traitements de texte, les pdf, les diaporamas de type powerpoint ou impress, les fichiers spécifiques aux projets R...

Pour se faciliter la tâche, on utilise un fichier spécifique `.gitignore` situé le plus souvent à la racine des projets.

Il s'agit d'un fichier texte que vous devez éditer, qui liste les fichiers et dossiers (sous forme d'expressions régulières) à ne pas versionner.

- une ligne = une règle ;
- on peut ignorer :
 - des fichiers (exemple : `donnees.rds`)
 - des dossiers (exemple : `data/`)
 - des extensions (exemple : `*.xls`)
 - ...

2.3 Qu'est-ce qu'on NE versionne PAS ? (2/2)

Si vous utilisez le package R `gitssp` mis à disposition des agents du MASA, voici ce qui est exclu par défaut avec la fonction `gitssp::creer_gitignore()` :

```
.Rproj.user
.Rhistory
.RData
.Renvironment
.Ruserdata

/* Les fichiers avec ces extensions
*.xls
*.xlsx
*.ods
*.pdf
*.docx
*.odt
*.ppt
...
```

2.4 Comment utiliser Git ?

3 façons seront présentées dans ce support :

-  Via l'interface visuelle de l'IDE RStudio

Pour les commandes les plus courantes du quotidien

-  Via les commandes du terminal

Pour les commandes plus avancées

-  Via le package R `gitssp`

Pour simplifier l'articulation avec les dépôts distants



2.5 Installer Git

- **Si vous travaillez avec Cerise**, Git est déjà installé.
=> Vous pouvez directement commencer à l'utiliser. (il vous faudra simplement régler votre configuration pour faire en sorte de dialoguer avec la forge interne Gitlab => voir [ici](#))
- **Si vous travaillez en local**, consultez [cette procédure interne au MASAF](#)

2.6 Git avec RStudio

Pré-requis : pour pouvoir utiliser correctement Git avec l'IDE RStudio, il convient d'utiliser le mode projet.

Comment savoir si un projet R est versionné avec Git ?

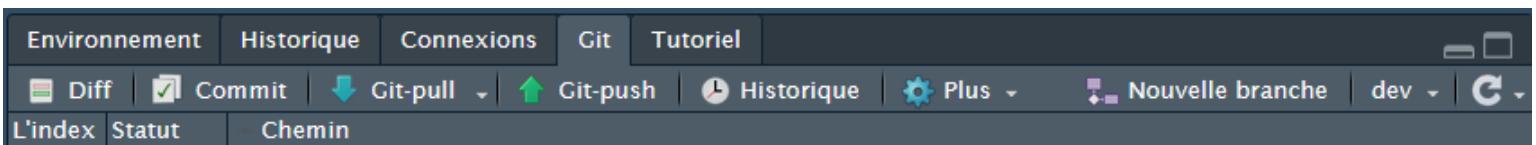
- Un fichier **.git**  est présent dans l'explorateur de fichier.



Note

Pensez à cocher la case Montrer les fichiers cachés

- À l'ouverture du projet R, un onglet Git s'affiche dans RStudio :



=> Une utilisation de Git en cliquant sur les boutons de RStudio

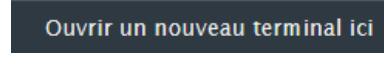
2.7 Git avec le terminal

- En local, vous pouvez directement utiliser les commandes du terminal sans RStudio.

Pour cela, il vous suffit de vous placer dans le bon répertoire (par exemple avec la commande `cd` pour [change directory](#)).

- Sur un server Posit comme Cerise, le terminal est désormais intégré dans les sessions RStudio à côté de la console classique de R 

Vous pouvez alors soit utiliser la même méthode qu'en local avec la commande `cd` soit de préférence :

- Vous placez dans le répertoire voulu dans l'explorateur de fichiers
- Cliquez sur le bouton [Plus](#) en haut à droite
- Cliquez sur le bouton  [Ouvrir un nouveau terminal ici](#)

[**=> Une utilisation de Git en tapant des commandes qui commencent par git...**](#)



2.8 Git avec le package R gitssp

Un package à installer et chargé comme d'habitude.

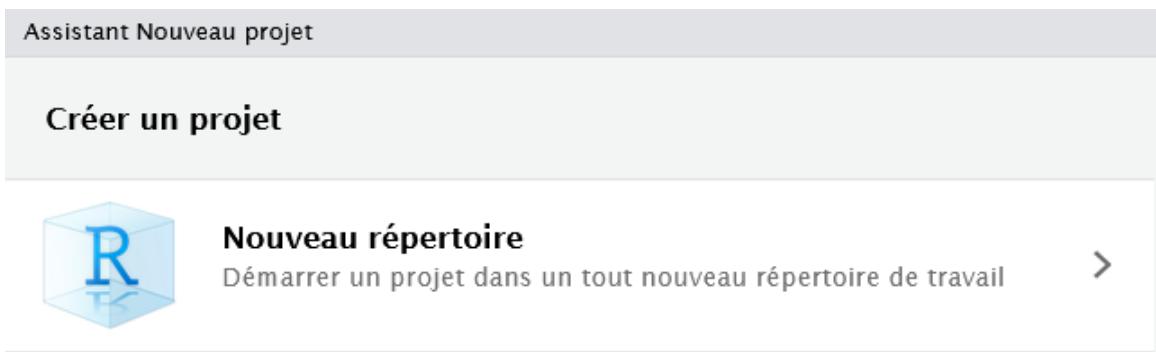
Pour l'instant, les sources ne sont présentes que sur la forge interne du MASAF :

```
1 remotes::install_git("https://forge.agriculture.rie.gouv.fr/gitlab/ssp/bmis/gitssp",
2                         dependencies = T,
3                         git = "external")
```

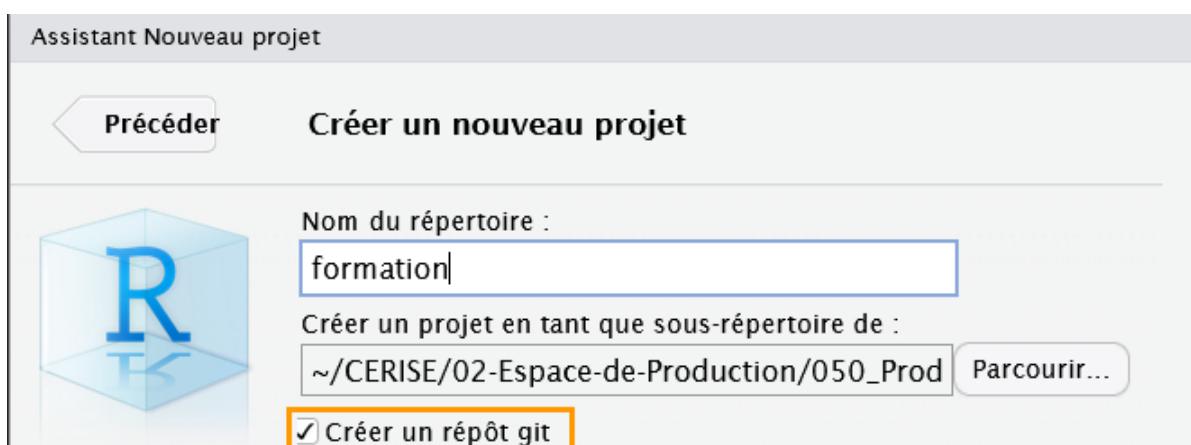


2.9 Initialiser un dossier local en dépôt Git avec RStudio

Créer un nouveau projet puis choisir un nouveau répertoire :



Après avoir choisi le nom du nouveau projet, cocher la case “Créer un répôt git” :



2.10 Initialiser un dossier local en dépôt Git avec le terminal

La commande `git init` permet d'initialiser un dossier avec Git

Cette commande crée un nouveau sous-répertoire nommé `.git` qui contient tous les fichiers nécessaires au dépôt — un squelette de dépôt Git.

Pour l'instant, aucun fichier n'est encore versionné !

Important

Il s'agit d'être vigilant sur l'emplacement du terminal au moment du lancement de la commande `git init` au risque d'initialiser le mauvais répertoire !

Astuce

Lors de l'initialisation du dépôt Git, on peut directement choisir le nom de la branche avec la commande `git init --initial-branch=<nom-de-branche>`.
=> voir [partie 4](#) de ce support



2.11 Initialiser un dossier local en dépôt Git avec gitssp

Le package R `{gitssp}` permet d'initialiser un dossier local avec Git tout en le synchronisant avec une forge.

Pour en savoir plus, voir le [mode d'emploi](#).

Cela sera abordé dans la [partie 4](#) de cette formation.



Note

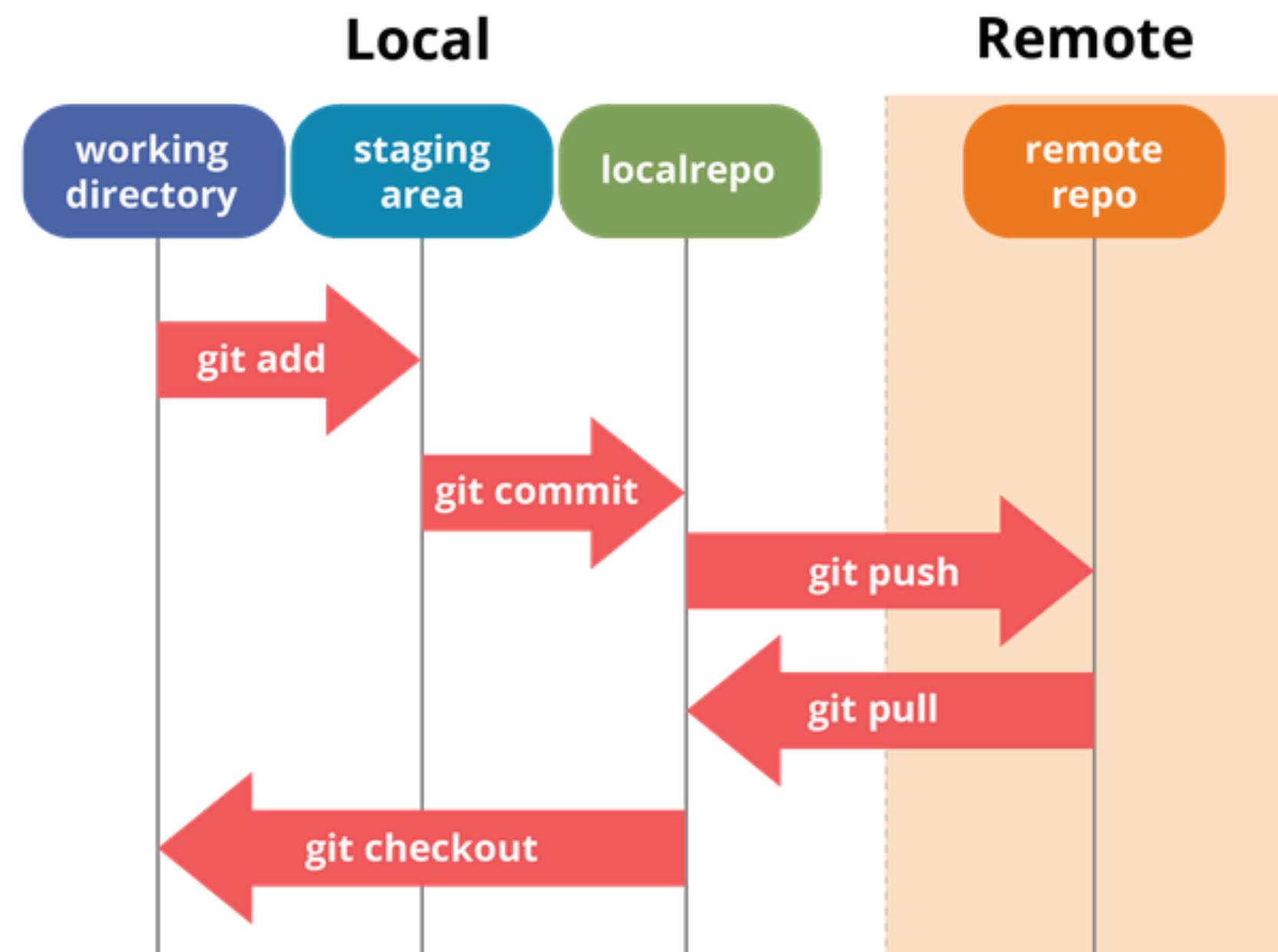
D'autres packages comme `{usethis}` ou `{gert}` permettent de manipuler Git avec des fonctions R. Vous trouverez leurs liens dans [la partie bibliographique de ce support](#)



3 Les zones de Git



3.1 Fonctionnement



Source : itnext.io



3.2 Les zones de Git :

Pour évoquer le fonctionnement de Git et son articulation avec une forge, cela nécessite d'apprendre à utiliser de nouveaux termes (voir lexique diapo suivante).

- **working directory** : répertoire de travail de l'agent
- **staging area** ou **index** : zone tampon dans laquelle l'agent regroupe les changements en vue du prochain commit.
- **local repository** : dépôt local présent dans votre espace de travail (.git)
- **remote repository** : dépôt distant sur une forge



*Liberté
Égalité
Fraternité*

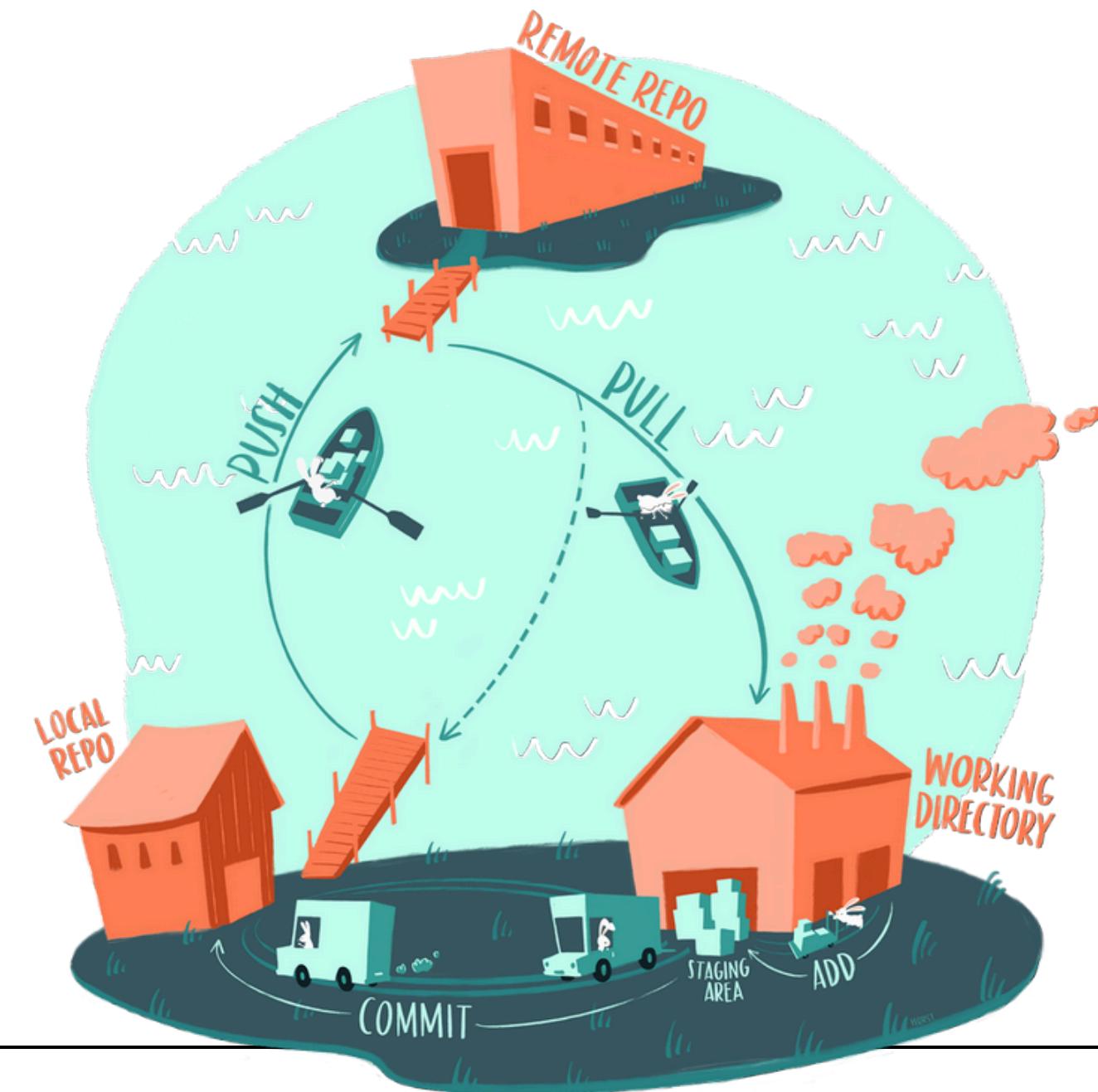
3.3 Les commandes principales de Git

- `git add` : ajout des changements à l'index
- `git commit` : enregistre les changements placés dans l'index
- `git push` : envoie les changements vers le dépôt distant
- `git pull` : télécharge les changements depuis le dépôt distant
- `git checkout` : permet de se déplacer dans l'arbre (retour vers le passé, changement de branche)



3.4 Une 2ème illustration pour fixer les idées :)

Source : [Allison Horst](#)



3.5 Le working directory

Il s'agit de notre **copie de travail** c'est-à-dire le répertoire et sous-répertoires dans lesquels se trouvent les fichiers suivis par Git.

C'est là que l'agent effectue les changements sur les fichiers.

Il correspond à votre **espace de travail**.

Avis aux utilisateurs novices

Faire très attention au début de l'utilisation de Git : en fonction de la commande Git que vous allez utiliser (par exemple un `git checkout` pour changer de branche ou revenir dans le passé), **Git va changer le contenu de cet espace sans vous avertir**.

3.6 La staging area ou l'index

Zone tampon dans laquelle l'agent regroupe les changements en vue du prochain commit.

L'agent peut **ajouter** ou **retirer** les changements à l'index.

Il peut faire cela en une seule ou plusieurs opérations (voir diapos suivantes).

 **Attention**

Ajouter des changements à plusieurs fichiers en même temps nécessite d'être sûr de ce que l'on fait et d'avoir correctement configuré le fichier `.gitignore` (voir diapositive suivante).



3.7 Ajouter des changements à l'index

Pour ajouter des changements :

R Avec RStudio

- Un seul fichier : en cochant un fichier dans l'onglet Git

Staged	Status	Path
<input checked="" type="checkbox"/>	M	03_basegit.qmd

- Plusieurs fichiers : en cochant plusieurs fichiers dans l'onglet Git

Staged	Status	Path
<input checked="" type="checkbox"/>	M	03_basegit.qmd
<input checked="" type="checkbox"/>	A	img/git_add_un_fichier.png

En ligne de commande :

- Un seul fichier : `git add <file-name>`
- Tous les fichiers : `git add .`



3.8 Retirer des changements à l'index

Pour retirer des changements :

Avec RStudio

En décochant les fichiers dans l'onglet Git

En ligne de commande :

- Un seul fichier : `git reset <file-name>`
- Tous les fichiers : `git reset`



3.9 Les états des fichiers avec RStudio

L'onglet de RStudio permet d'avoir des repères visuels sur les changements apportés à chacun de vos fichiers :



- **Deleted** : le fichier a été supprimé du working directory
- **Modified** : le contenu du fichier a été modifié
- **Untracked** : le fichier a été ajouté au working directory et Git ne l'a jamais vu auparavant

3.10 Les états des fichiers avec le terminal

La commande `git status` affiche l'état du working directory et de l'index.

Voici un exemple d'équivalence entre les icônes RStudio et le résultat renvoyé par la commande `git status`:

```
M 03_basegit.qmd
? ? img/etats_fichiers.png
? ? img/onglet_terminal.png
? ? img/ouvrir_nouveau_terminal_ici.png
```

```
c:\users\damien.dotta\DEMESIS\Formations_Git\formation_git_2024>git status
on branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

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:   03_basegit.qmd

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    img/etats_fichiers.png
    img/onglet_terminal.png
    img/ouvrir_nouveau_terminal_ici.png

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



3.11 Les états des fichiers APRÈS l'ajout à l'index

Après l'ajout de fichier à l'index, 2 nouvelles icônes apparaissent dans RStudio (même fonctionnement avec le terminal) :

- Added
- Renamed

- **Added** : Git comprend que vous voulez ajouter le fichier au dépôt
- **Renamed** : Git comprend que le fichier a été renommé

Changement

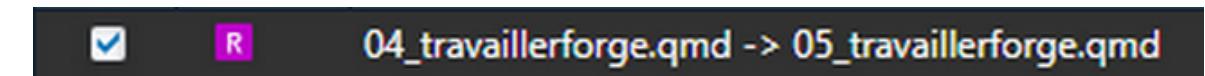
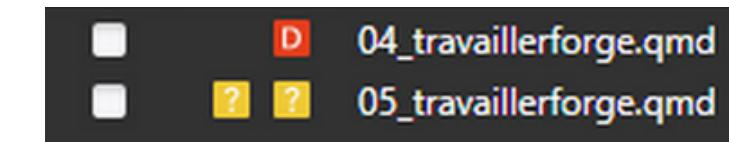
AVANT l'ajout à l'index

APRÈS l'ajout à l'index

Ajout de fichier

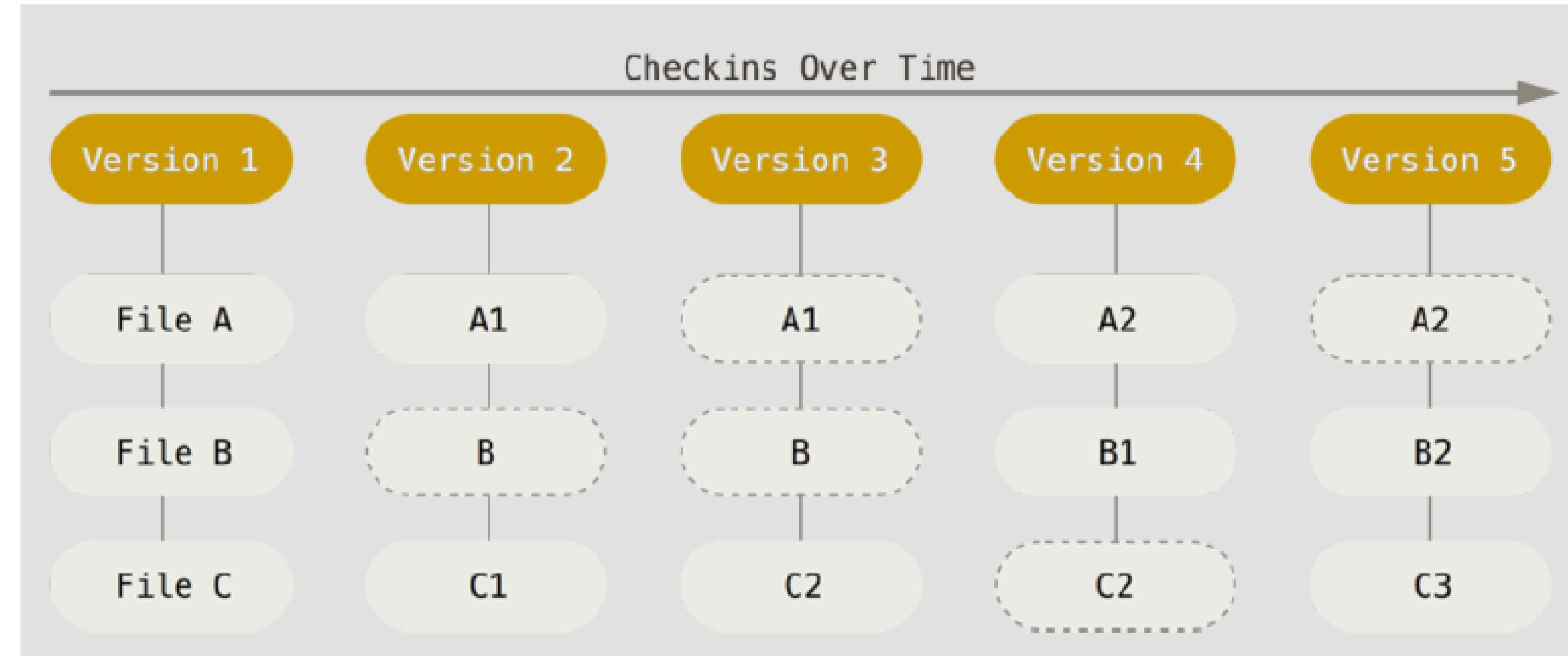


Renommage de fichier





3.12 Comment Git stocke les différentes versions d'un fichier ?

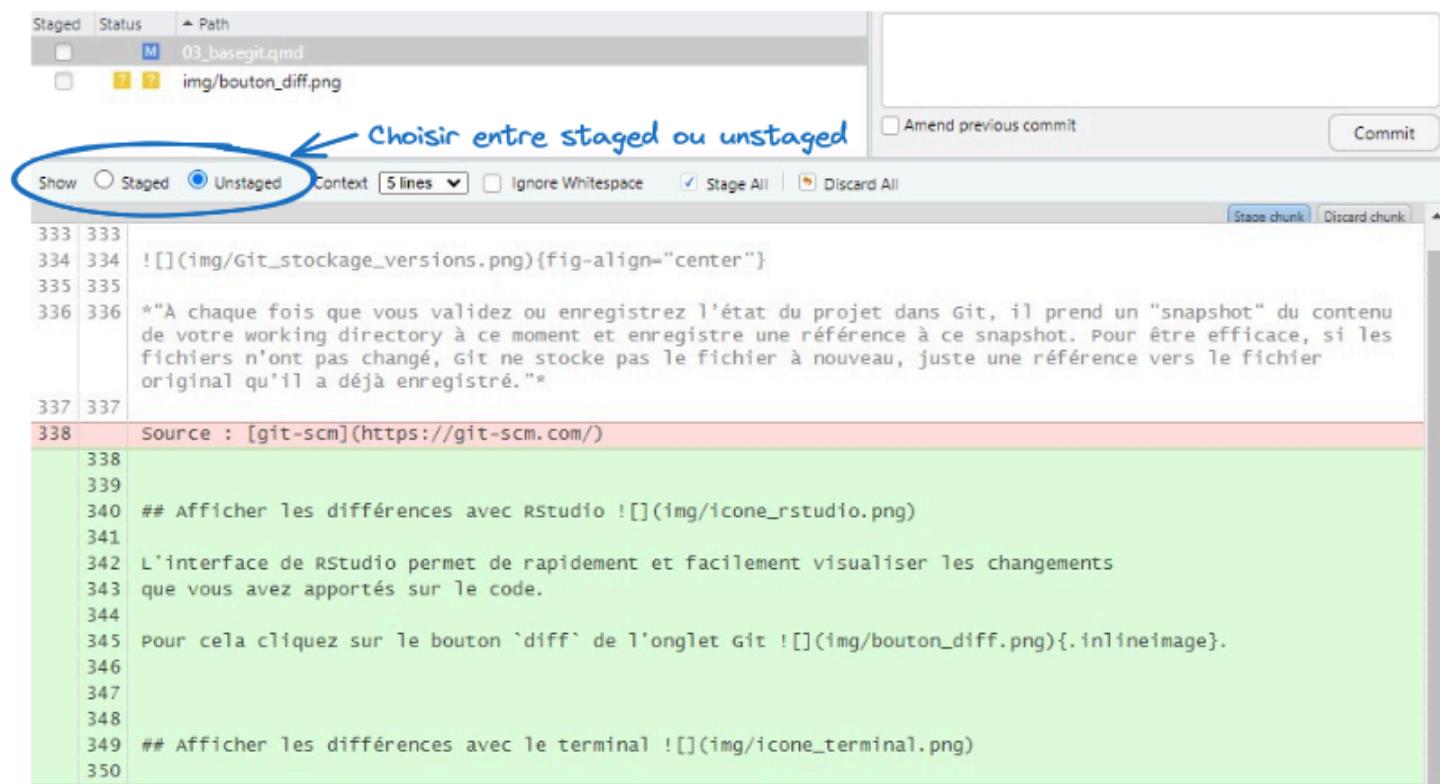


À chaque fois que vous validez ou enregistrez l'état du projet dans Git, il prend un "snapshot" du contenu de votre working directory à ce moment et enregistre une référence à ce snapshot. Pour être efficace, si les fichiers n'ont pas changé, Git ne stocke pas le fichier à nouveau, juste une référence vers le fichier original qu'il a déjà enregistré.

3.13 Afficher les différences avec RStudio

L'interface de RStudio permet de rapidement et facilement visualiser les changements que vous avez apportés sur le code.

Pour cela cliquez sur le bouton **diff** de l'onglet Git .



Choisir entre staged ou unstaged

```

Staged Status Path
M 03_basegit.qmd
? img/bouton_diff.png

Show Staged Unstaged Context 5 lines Amend previous commit Commit
Stage chunk Discard chunk

333 333 {fig-align="center"}
334 334 
335 335 
336 336 
337 337 
338 338 
339 339 
340 340 ## Afficher les différences avec RStudio 
341 341 
342 342 L'interface de RStudio permet de rapidement et facilement visualiser les changements
343 343 que vous avez apportés sur le code.
344 344 
345 345 Pour cela cliquez sur le bouton 'diff' de l'onglet Git {.inlineimage}.
346 346 
347 347 
348 348 
349 349 ## Afficher les différences avec le terminal 
350 350

```

- Le code supprimé s'affiche en rouge
- Le code ajouté s'affiche en vert



3.14 Afficher les différences avec le terminal ➔

La commande `git diff` affiche les différences.

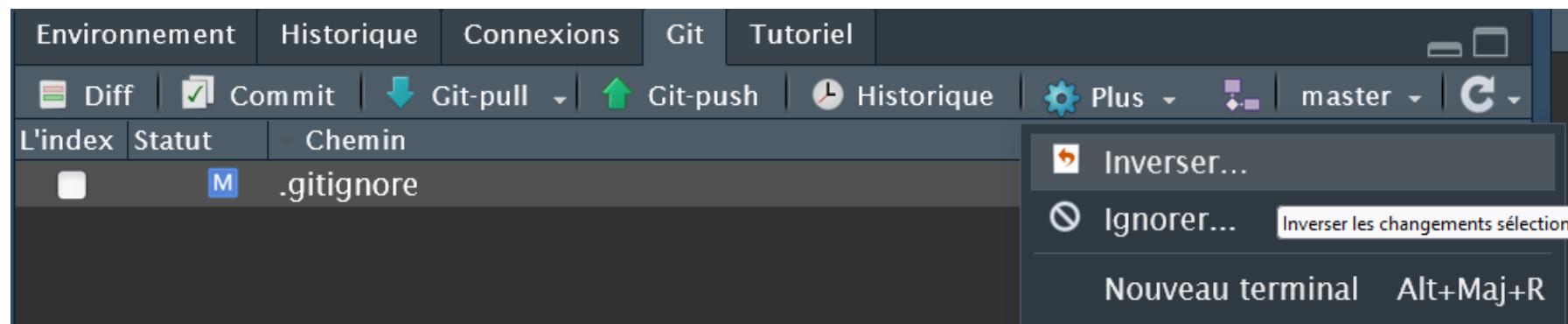
```
c:\Users\damien.dotta\DEMESIS\Formations_Git\formation_git_2024>git diff
diff --git a/03_basegit.qmd b/03_basegit.qmd
index 2977212..a76bb11 100644
--- a/03_basegit.qmd
+++ b/03_basegit.qmd
@@ -335,7 +335,19 @@ Après l'ajout de fichier à l'index, 2 nouvelles icônes apparaissent dans RStudio
**"À chaque fois que vous validez ou enregistrez l'état du projet dans Git, il prend un "snapshot" du contenu de votre working directory à ce moment et enregistre une référence à ce snapshot. Pour être efficace, si les fichiers n'ont pas changé, Git ne stocke pas le fichier à nouveau, juste une référence vers le fichier original qu'il a déjà enregistré."*
-Source : [git-scm](https://git-scm.com/)
+
+## Afficher les différences avec RStudio 
+
+L'interface de RStudio permet de rapidement et facilement visualiser les changements
+que vous avez apportés sur le code. ■
+
+Pour cela cliquez sur le bouton `diff` de l'onglet Git {.inlineimage}. ■
+
+
+## Afficher les différences avec le terminal 
+
```

Pour afficher les différences sur les fichiers déjà ajoutés à l'index, la commande est `git diff --staged`.

Le terminal permet d'aller plus loin dans l'observation des différences, par exemple en comparant 2 commits...

3.15 Inverser les changements avec RStudio

Dans RStudio, sélectionner le fichier sur lequel vous souhaitez inverser les changement puis cliquez sur le bouton **Plus** de l'onglet Git puis choisir le bouton **Inverser** :



Un message de confirmation vous demande si vous êtes sûr de votre action :





3.16 Inverser les changements avec le terminal ➔

La commande git restore permet de restaurer des fichiers

Action

Inverser les changements pas encore ajoutés à l'index

Commande

`git restore <file-name>`

Inverser les changements déjà ajoutés à l'index

`git restore --staged <file-name>`

3.17 Le commit

Action de figer l'état du dépôt sous forme de [snapshot](#)

Un bon commit :

- Change une seule chose
- Peut être décrit avec un message clair et explicite

Erreur courante du débutant sur Git :

- Il code pendant des heures/jours/semaines
- Ajoute toutes les modifications à l'index avec [git add .](#)
- Puis il commite...



3.18 Caractéristiques du commit

- Un commit possède **un identifiant unique** (SHA)

Il informe les utilisateurs :

- Qui ? Quoi ? Où ? Quand ? => Avec ses **métadonnées**
- Quoi ? Où ? Comment ? => Avec l'affichage des **différences**
- Pourquoi ? => Avec **le message du commit**



3.19 Les messages de commit

D'après [Conventional Commits](#), l'écriture des messages de commits devrait suivre la structure minimale suivante :

<type>: <description>

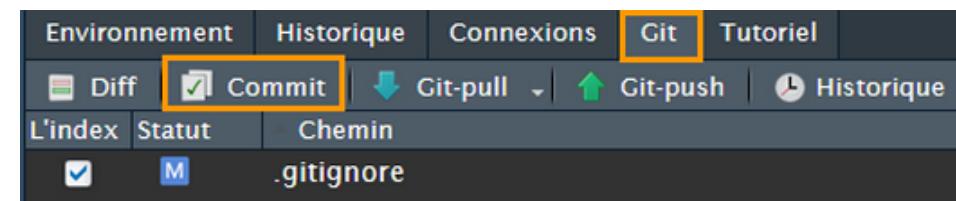
Le tableau suivant reprend les types les plus couramment utilisés :

Type	Detail
wip	Développement d'une fonctionnalité
feat	Ajout d'une fonctionnalité
fix	Correction d'un bug/erreur
doc	Documentation
deprecated	Fonctionnalité dépréciée
chore	Tâches de routine/automatisées

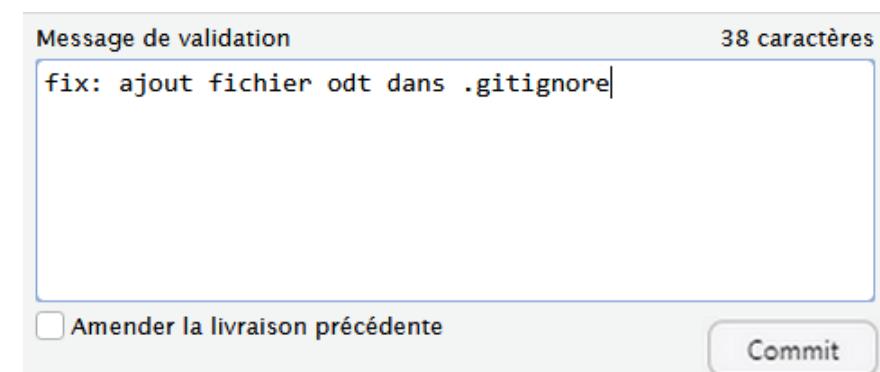
3.20 Comment committer avec RStudio ?

Après avoir ajouté les fichiers modifiés à l'index, il est temps de “committer” les changements c'est-à-dire de prendre une photo / un snapshot de ces fichiers modifiés.

Dans RStudio, il faut cliquer sur le bouton “Commit” de l'onglet “Git”



Puis, renseigner dans la nouvelle fenêtre qui s'affiche le message du commit et cliquer sur le bouton “Commit” :





3.21 Comment committer avec le terminal ? ➔

La commande `git commit -m "message du commit"` permet à la fois de faire le commit tout en renseignant le message associé

Voici un exemple :

`git commit -m "fix: ajout fichier odt dans .gitignore"`



La commande `git commit` propose de nombreuses options qui sont disponibles [ici](#).

3.22 Exercice 1

Exercice 1

- Créer un projet R “formation-git” dans votre espace personnel Cerise en l’initialisant avec Git
- Vérifier que votre projet est bien versionné avec Git
- Ajouter les nouveaux fichiers à l’index et faire un 1er commit
- Créer des scripts R à l’intérieur de votre projet
- Enregistrer vos scripts
- Ajouter ces scripts à l’index et faire votre 2ème commit

BONUS : faire l’exercice avec RStudio et pour ceux qui sont en avance avec le terminal :)



3.23 Modifier le dernier commit

Il y a au moins deux raisons de revenir sur un commit :

- Modifier le message de commit (même pour une faute de frappe)
- Ajouter ou supprimer des fichiers au commit.

R

Cocher la case `Amend previous commit` en dessous du message de commit précédent.



>

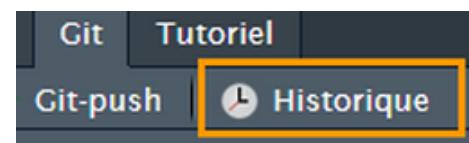
La commande à utiliser est :
`git commit --amend -m 'Mon nouveau message de commit'`

Attention ! Cette commande ne doit être effectuée que si vous n'avez pas encore poussé le commit sur dépôt distant sinon les autres utilisateurs du commit seront affectés puisque le SHA-1 est changé



3.24 Consulter l'historique avec RStudio (1/2) R

En cliquant sur le bouton **Historique** de RStudio :



Ce qui permet d'accéder à la fenêtre suivante :

RStudio: Review Changes

Subject	Author	Date (UTC)	SHA
HEAD -> refs/heads/main origin/main doc: ajout de ce qu'on versionne et ceux qu'on ne versionne pas Damien Dotta <damien.dotta@live.fr>	Damien Dotta <damien.dotta@live.fr>	2024-03-29	efb2b38f
doc: ajout zones de Git	Damien Dotta <damien.dotta@live.fr>	2024-03-29	24be7385
doc: ajout source pour photo de Linus Torvalds	Damien Dotta <damien.dotta@live.fr>	2024-03-29	60ff4056
doc: ajout diapo sur la génèse de Git	Damien Dotta <damien.dotta@live.fr>	2024-03-29	a35854c3
doc: ajout illustrations git workflow + lexique	Damien Dotta <damien.dotta@live.fr>	2024-03-29	d29397cc
doc: ajout workflow git et gitlab	Damien Dotta <damien.dotta@live.fr>	2024-03-29	74f160a2

Commits 1-32 of 32

```

SHA      efb2b38f3f4c7719201119c5bf1dea575e0621ab
Author   Damien Dotta <damien.dotta@live.fr>
Date (UTC) 2024-03-29 16:13
Subject  doc: ajout de ce qu'on versionne et ceux qu'on ne versionne pas
Parent   24be738505d6d45c9939663456019e22566575b2
03_basegit.qmd
@@ -1,5 +1,59 @@
1 1 # Les bases de Git {.backgroundTitre}
2 2
3 ## Qu'est-ce qu'on versionne ?
4
5 on versionne les fichiers de type texte donc par exemple les programmes R, SAS,
6 La documentation au format texte, Markdown, quarto..., les fichiers de configuration
7 de type yaml par exemple...
8 Eventuellement, on peut aussi versionner des images si on en a besoin dans une application
9 ou une documentation.
10
11 ## Qu'est-ce qu'on NE versionne PAS ? (1/2)
12

```



3.25 Consulter l'historique avec RStudio (2/2)

RStudio permet facilement de :

- Balayer l'historique du projet en sélectionnant les différents commits.
- Accéder aux informations correspondants à chaque commit.
- Cliquer sur les fichiers concernés par chaque commit et afficher leur état au moment choisi.

3.26 Consulter l'historique avec le terminal

» Avec la commande `git log` : Exemple :

```
C:\Users\damien.dotta\DEMESIS\Formations_Git\formation_git_2024>git log
commit efb2b38f3f4c7719201119c5bf1dea575e0621ab (HEAD -> main, origin/main)
Author: Damien Dotta <damien.dotta@live.fr>
Date:   Fri Mar 29 17:13:00 2024 +0100
```

doc: ajout de ce qu'on versionne et ceux qu'on ne versionne pas

Ce qu'on y trouve :

- l'identifiant du commit
- l'auteur du commit (nom + email)
- la date et l'heure du commit
- le message du commit



Note

La commande `git log --oneline` affiche les informations concernant l'historique de façon plus compacte mais moins riche.



*Liberté
Égalité
Fraternité*

3.27 Fréquence des commits

Quelle fréquence pour les commit ?

- Aussi souvent que possible
- Regrouper les modifications en lots qui “font sens”, avec un message pertinent qui résume bien les modifications apportées
- **Il ne faut committer sur la base du temps passé mais plutôt sur la base des fonctionnalités.**



3.28 Exercice 2 (1/2)

Exercice 2 (1/2)

Reprendre [l'exercice 1](#)

- Apporter des changements dans un de vos scripts R
- Enregistrer votre script modifié
- Que se passe t'il dans l'onglet “Git” de RStudio ? Avec la commande `git status` ?
- Ajouter vos changements à l’index et faire un nouveau commit

- Supprimer un de vos scripts
- Que se passe t'il dans l'onglet “Git” de RStudio ? Avec la commande `git status` ?
- Ajouter vos changements à l’index et faire un nouveau commit

VOIR LA SUITE DE L'EXERCICE SUR LA PROCHAINE DIAPOSITIVE...

3.29 Exercice 2 (2/2)

Exercice 2 (suite)

- Ajouter un fichier de type tableau dans votre projet
- Que se passe t'il dans l'onglet “Git” de RStudio ? Avec la commande `git status` ?
- Modifier le fichier `.gitignore` pour faire en sorte que Git ignore ce fichier tableau.
- Ajouter vos changements sur `.gitignore` à l’index et faire un nouveau commit
- Ajouter un nouveau script `Analyse.R` qui contient uniquement la ligne `library(dplyr)`
- Ajouter vos changements à l’index et faire un nouveau commit
- Consulter l’historique de votre projet avec RStudio et le terminal



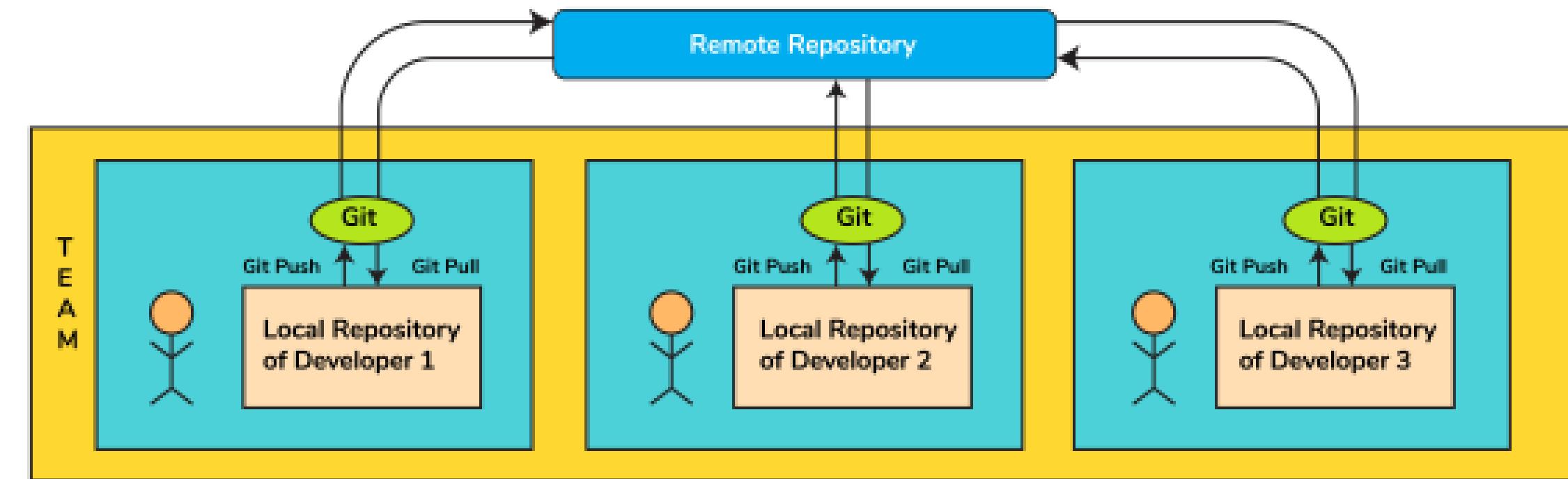
4 Travailler avec une forge

4.1 Git - un modèle distribué

- **Dépôt local** : dépôt de l'agent sur son espace de travail (sous Cerise ou en local par ex.)
- **Dépôt distant (remote)** : dépôt correspondant situé sur la forge.

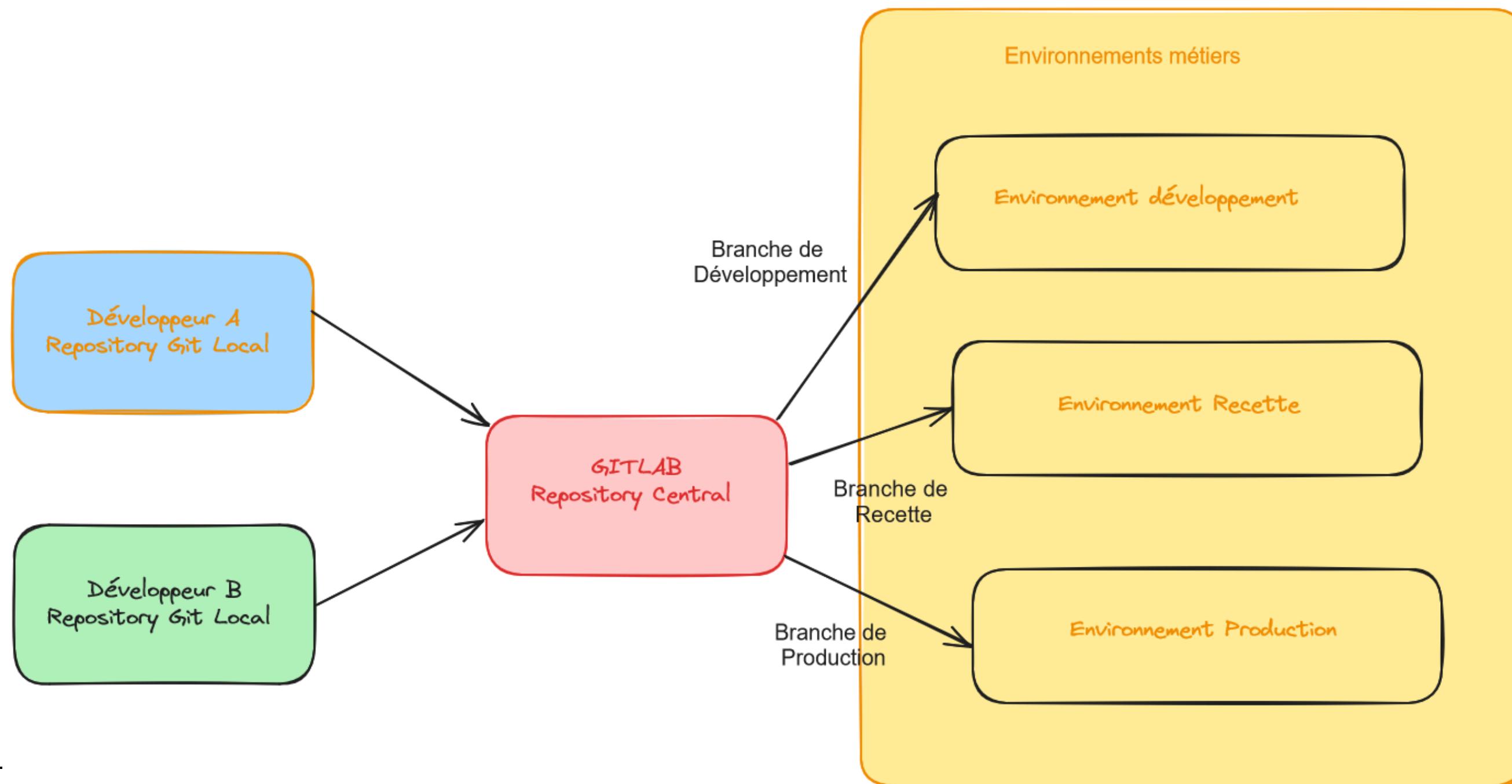
Convention

Par défaut, le dépôt distant porte l'alias `origin`





4.2 Architecture générale



4.3 C'est quoi une forge ?

Définition Wikipédia :

En informatique, une forge est un système de gestion et de maintenance collaborative de texte.

C'est un espace où on peut archiver du code informatique (R mais pas que bien sûr) qui offre des fonctionnalités supplémentaires à Git comme :

- la gestion des tickets (issues) ;
- les merge requests ou pull requests ;
- un affichage plus agréable du dépôt avec une interface web ;
- la possibilité d'ajouter un wiki ;
- l'intégration et le déploiement continus...

4.4 Exemples de forges

Les plus connues : [Gitlab](#), [Github](#), [Bitbucket](#)...

Une forge peut être **interne** ou **externe** à un ministère.

- URL de la forge interne du MASAF : <https://forge.agriculture.rie.gouv.fr/gitlab/>
- URL de la forge interne du MTECT : <https://gitlab-forge.din.developpement-durable.gouv.fr/>
- URL du SSM Agriculture (SSP) sur Github : <https://github.com/SSM-Agriculture>
- URL du MTECT sur Github : <https://github.com/MTES-MCT>

 À retenir !

- Les forges externes ont vocation à héberger les projets open-source. Il s'agit donc d'être très vigilant sur le code déposé sur ces forges

4.5 Dialoguer avec Gitlab

Pour travailler avec Gitlab, il faut mettre en place un mode d'authentification.

2 modes principaux existent :

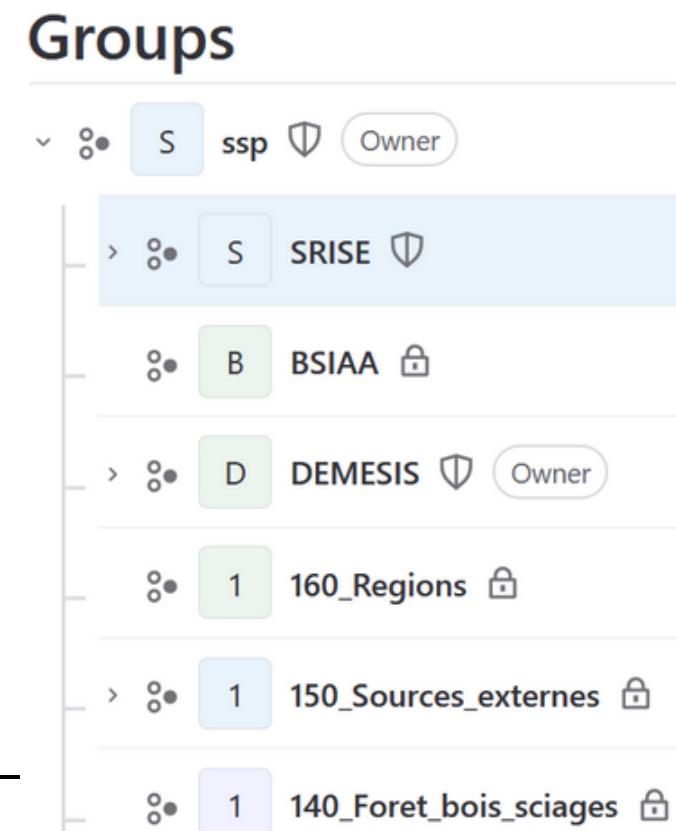
- SSH
- HTTPS

Au MASAF, [une procédure de configuration existe et a été validée en mars 2024](#) pour établir la connexion entre Cerise et la forge gitlab interne. Celle-ci est basée sur le mode [HTTPS](#) et utilise le credential helper de Git qui évite d'avoir à re-saisir son mot de passe (ou token) à chaque action.

4.6 Les groupes dans les forges Gitlab

- Une forge Gitlab est structurée par **groupe**.
- Chaque **groupe** est constitué de **projets** (les fameux dépôts distants).
- Chaque **groupe** est associé à des utilisateurs dont chacun dispose **d'un niveau de permissions** sur les projets du groupe et sur le groupe lui-même.

Aperçu des groupes au
MASAF/SSP :



The screenshot shows a 'Groups' section in a GitLab interface. At the top, there is a header with a search bar and a user icon. Below the header, a list of groups is displayed in a tree structure:

- ssp (Owner)
 - SRISE (Owner)
 - BSIAA (Locked)
 - DEMESIS (Owner)
 - 160_Regions (Locked)
 - 150_Sources_externes (Locked)
 - 140_Foret_bois_sciages (Locked)

4.7 Permissions et rôles dans Gitlab

- Les **permissions** se traduisent par plusieurs **rôles** qui sont détaillés dans la [documentation de Gitlab](#).

Ce qu'il faut retenir :

Rôle	Permissions
Owner	Les propriétaires ont un contrôle total sur le projet ou le groupe.
Maintainer	Les mainteneurs ont généralement des droits similaires aux propriétaires, mais ils n'ont pas accès à l'intégralité des paramètres du projet/groupe.
Developer	Les développeurs ont des droits d'écriture sur le projet ou le groupe.
Reporter	Les rapporteurs ont des droits de lecture sur le projet ou le groupe.
Guest	Les invités ont des droits d'accès limités et sont souvent utilisés pour donner un accès en lecture seule à des personnes extérieures au projet.

4.8 Bonnes pratiques avec les forges

Voici ci-dessous quelques bonnes pratiques (non exhaustives) à ajouter à un projet R lorsque celui-ci est amené à être partagé sur une forge (Gitlab ou Github ou autre).

Dans l'idéal votre projet doit contenir :

- Un fichier [README.md](#) : un document rédigé en [markdown](#) qui inclue notamment des informations sur :
 - Ce que le projet fait
 - Son emplacement sous Cerise
 - Ses pré-requis et son utilisation/installation (par exemple s'il s'agit d'une application Shiny, les instructions de lancement ou s'il s'agit d'un package R, les instructions d'installation...)
 - Ses principales fonctionnalités
 - L'identité de l'équipe qui maintient le dépôt
- Un fichier [.gitignore](#) (voir [cette diapo](#))
- Un fichier [CHANGELOG.md](#) ou [NEWS.md](#) : lui aussi rédigé avec markdown, il permet de suivre les principales modifications apportées au projet par les développeurs. Il est très utile en association avec les [tags](#).

4.9 Le système de ticketing (issues)

Une issue est une tâche à effectuer : il peut s'agir tout autant de correctifs à apporter au projet que de fonctionnalités à ajouter.

Ouvrir une issue depuis Gitlab permet de :

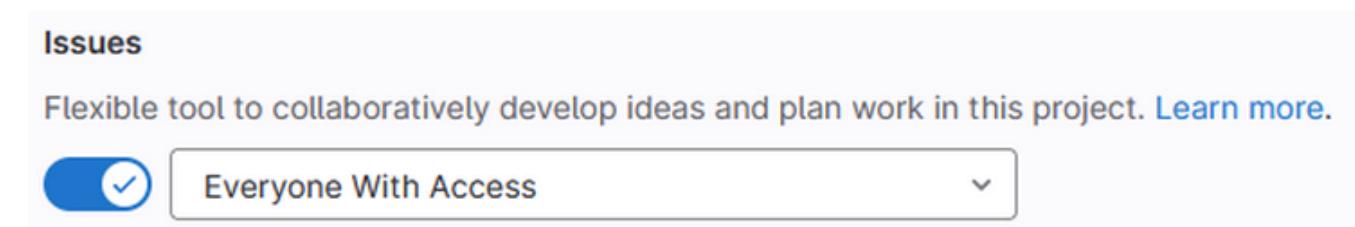
- **Discuter un point** : il est possible d'échanger à plusieurs dans une issue pour définir comment la traiter.
- **Assigner une tâche à une personne** : lorsqu'on crée une issue, on peut assigner la tâche à une ou plusieurs personnes. Elle s'affiche alors sur le tableau de bord.
- **Définir une échéance, classer une issue avec un label** : le tableau de bord permet alors d'avoir une vision complète des issues en cours.

4.10 Activer les issues sur la forge interne du MASAF 🦚

Par défaut, les issues ne sont pas activées à la création d'un dépôt sur la forge Gitlab du MASAF.

Pour les activer, il faut :

- Cliquer sur **Settings** dans le bandeau de gauche puis **General**
- Cliquer sur le bouton **Expand** à côté de **Visibility**, **project features**, **permissions**
- Activer les issues :



- Enregistrer les changements :

Save changes

4.11 Créer une issue 🦅

The screenshot shows a sidebar with the following menu items: 'test' (selected), 'Project information', 'Issues' (selected), 'List' (highlighted), 'Boards', 'Service Desk', and 'Milestones'. A large blue button labeled 'New issue' is centered at the bottom of the sidebar.

Puis renseigner les informations concernant l'issue.

The screenshot shows the 'New Issue' creation form. Handwritten annotations in orange point to specific fields:

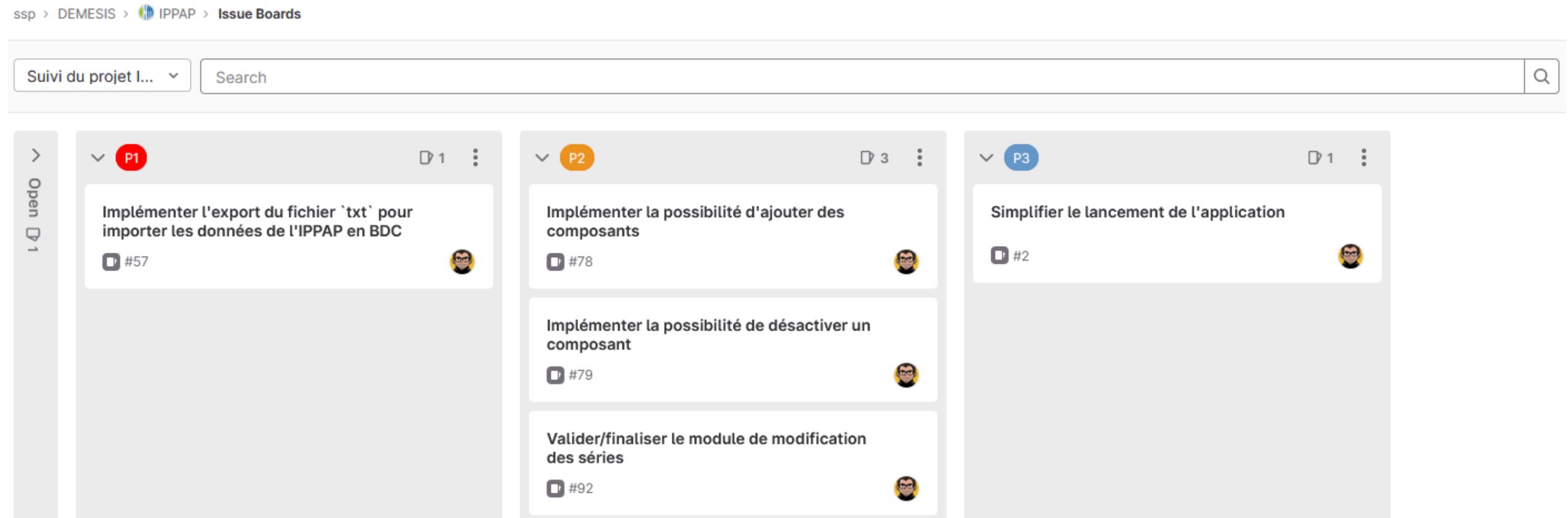
- An arrow points to the 'Title (required)' field containing '[Bug] - Le filtre sur les départements ne fonctionne plus' with the text: "Donner un titre à l'issue".
- An arrow points to the 'Description' rich text editor with the text: "Préciser des détails si nécessaire".
- An arrow points to the 'Assignee' dropdown set to 'Damien DOTTA' with the text: "Assigner la tâche à un membre".
- An arrow points to the 'Due date' field set to '2024-04-11' with the text: "Fixer une date limite".
- An arrow points to the 'Milestone' dropdown set to 'Réunion de suivi du 11 avril' with the text: "Associer un jalon à l'issue".
- An arrow points to the 'Labels' dropdown set to 'PI' with the text: "Affecter l'issue à un label".

At the bottom left are 'Create issue' and 'Cancel' buttons.

4.12 Afficher les issues sous forme de tableau (board) 🦸

Si vous utilisez **des labels** pour classer vos issues, Gitlab propose un affichage sous forme de tableau de bord qui permet de faire **un suivi de projet sous forme de tâches (Kanban)**.

Exemple :



The screenshot shows a GitLab Kanban board interface. At the top, there's a navigation bar with 'ssp > DEMESIS > IPPAP > Issue Boards'. Below it is a search bar with 'Suivi du projet I...' and a 'Search' button. The board itself has three columns labeled P1, P2, and P3. Each column contains a list of tasks:

- P1:** Implémenter l'export du fichier 'txt' pour importer les données de l'IPPA en BDC (status: 1, created by #57)
- P2:** Implémenter la possibilité d'ajouter des composants (status: 3, created by #78)
Implémenter la possibilité de désactiver un composant (status: 1, created by #79)
- P3:** Simplifier le lancement de l'application (status: 1, created by #2)

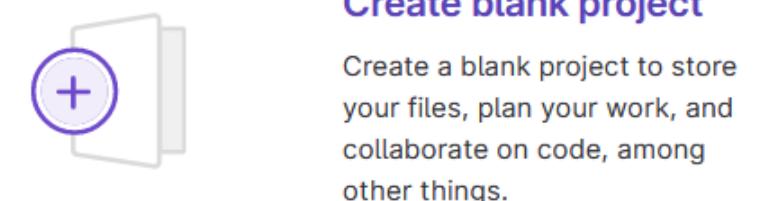
4.13 Lier un dépôt local à un dépôt distant (1/3)

1ère étape commune à toutes les méthodes : création d'un dépôt vide sur la forge Gitlab

- Cliquez sur le bouton “New project” situé en haut à droite :



- Choisir “blank project” :



4.14 Lier un dépôt local à un dépôt distant (2/3)

- Renseigner le nom du projet, éventuellement une courte description, l'espace d'appartenance du projet dans la forge, son niveau de visibilité et penser à décocher l'option “Initialize repository with a README”. Puis cliquer sur “Create Project”.

The screenshot shows the 'Create blank project' interface on a web application. At the top, there's a purple icon with a plus sign and the text 'Create blank project'. Below it, a sub-instruction reads: 'Create a blank project to store your files, plan your work, and collaborate on code, among other things.' The main form fields are as follows:

- Project name:** A text input field containing the value 'test'. A note below it specifies: 'Must start with a lowercase or uppercase letter, digit, emoji, or underscore. Can also contain dots, pluses, dashes, or spaces.'
- Project URL:** A dropdown menu showing the URL 'https://forge.agriculture.rie.gouv.fr/gitlab/' followed by the path 'ssp/bmis'.
- Project slug:** A text input field containing the value 'test'.
- Visibility Level:** A dropdown menu with two options: 'Private' (unchecked) and 'Internal' (checked). A note explains: 'Project access must be granted explicitly to each user. If this project is part of a group, access is granted to members of the group.'
- Project Configuration:** A section containing two checkboxes:
 - Initialize repository with a README**: A note below it says: 'Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.'
 - Enable Static Application Security Testing (SAST)**: A note below it says: 'Analyze your source code for known security vulnerabilities. [Learn more](#).

At the bottom of the form are two buttons: 'Create project' (in blue) and 'Cancel'.

4.15 Lier un dépôt local à un dépôt distant (3/3)

La méthode à utiliser est différente selon les cas :

1. **Vous voulez créer un nouveau répertoire dans votre working directory**
2. **Vous avez déjà un répertoire dans votre working directory**

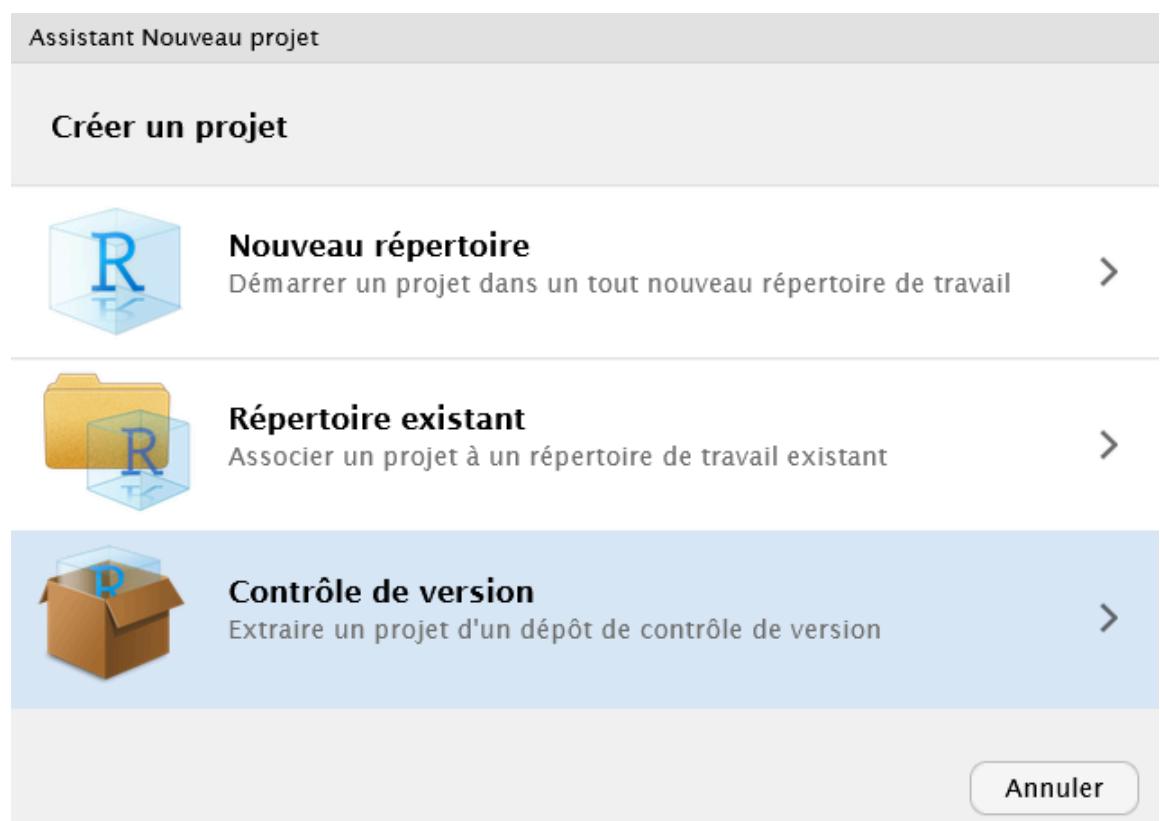
À retenir

Pour vérifier qu'un dépôt local est lié à un dépôt distant, la commande du terminal `git remote -v` est très utile car elle renvoie l'URL du dépôt distant s'il existe.

4.16 Lier un dépôt local à un dépôt distant avec RStudio (1/5) R

CAS 1 : création d'un nouveau répertoire

Créer un nouveau projet, puis choisir “Contrôle de version” :



4.17 Lier un dépôt local à un dépôt distant avec RStudio (2/5)

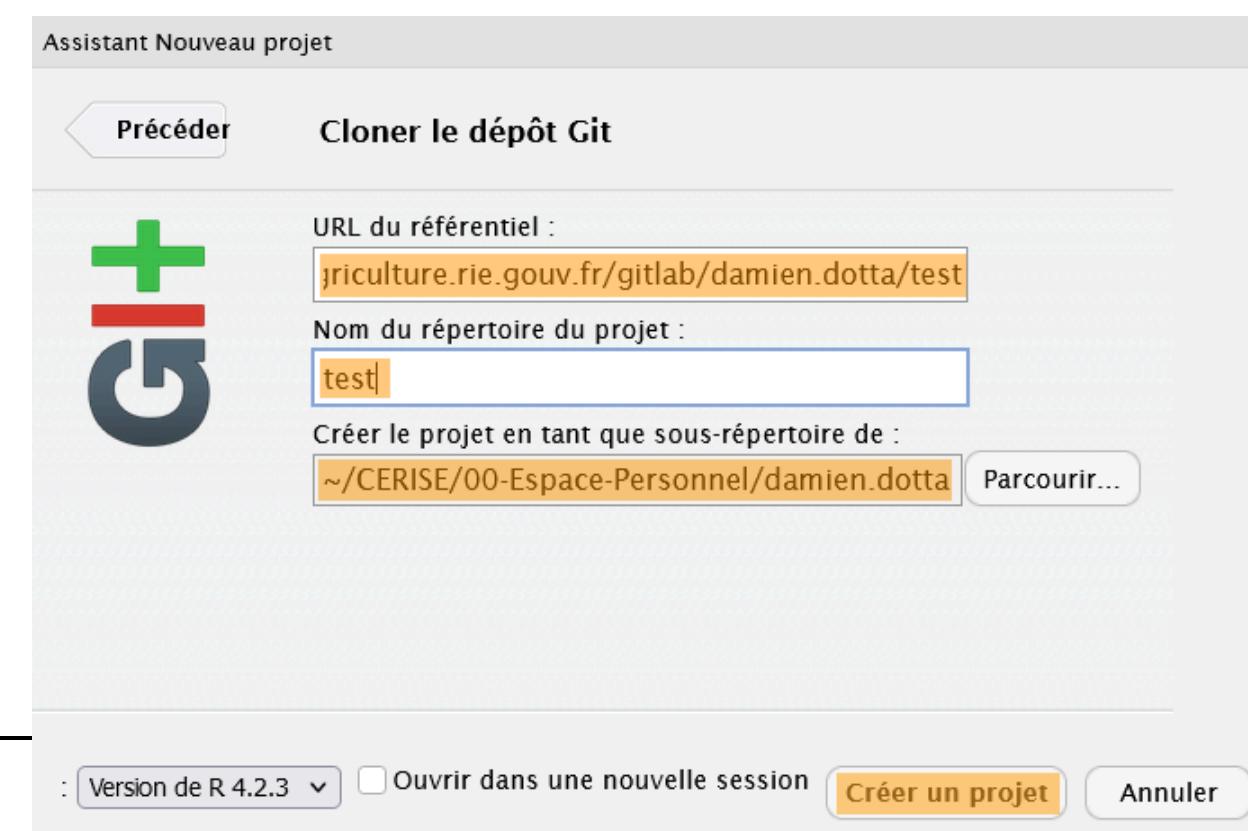
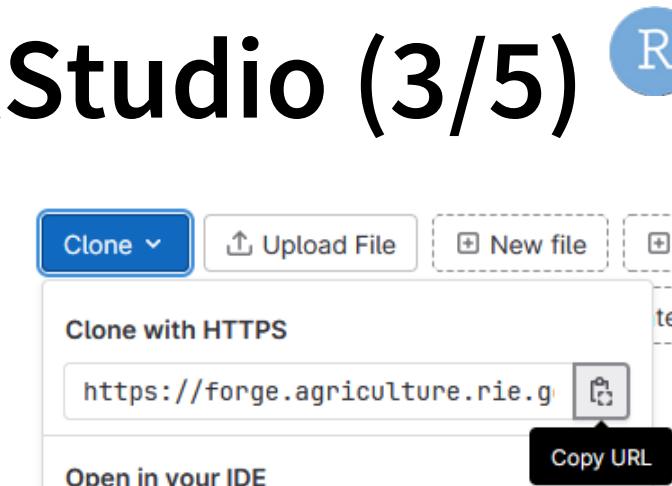
Puis choisir “Git” :



4.18 Lier un dépôt local à un dépôt distant avec RStudio (3/5)

Puis renseigner :

- L'URL du dépôt distant (voir ci-contre comment la récupérer) ->
- Le nom du répertoire du projet
- Le chemin du dépôt local dans votre espace de travail



4.19 Lier un dépôt local à un dépôt distant avec RStudio (4/5)

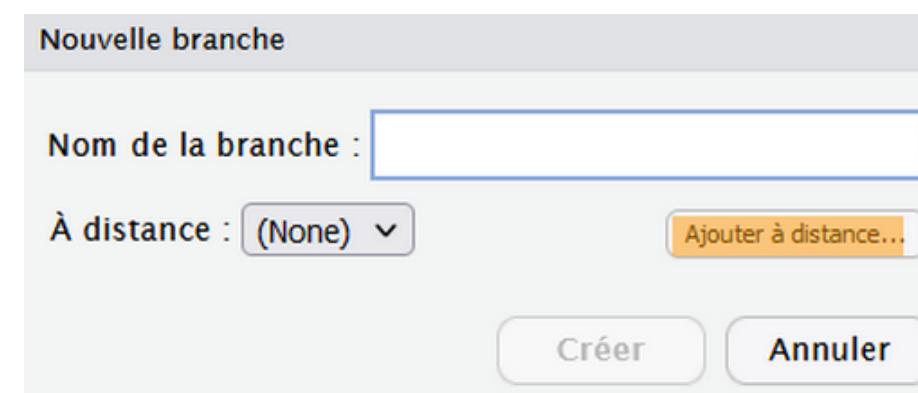
CAS 2 : un répertoire existe déjà

Dans ce cas, l'interface de RStudio est mal pensée et **il est beaucoup plus intuitif d'utiliser le terminal** (voir [ici](#))
 La fonctionnalité d'ajout d'un dépôt distant est accessible en cliquant sur le bouton “Nouvelle branche” :(



Entrer l'alias origin et l'URL du dépôt distant :

Puis cliquer sur le bouton “Ajouter à distance” :



Puis cliquer sur “Ajouter” puis “Annuler” sur l'écran précédent (on ne veut pas créer de nouvelle branche ici)



4.20 Lier un dépôt local à un dépôt distant avec RStudio (5/5) R

Avec la procédure présentée dans le cas 2, les boutons “Git-pull” et “Git-push” apparaissent en grisé et ne sont pas disponibles pour l’utilisateur :



Pour dégriser ces boutons, **le terminal est indispensable** pour taper la commande :

`git push -u <alias> <nom-branche>`

Puis en cliquant sur le bouton de rafraîchissement du l’onglet “Git” C .



Note

Pour information, le `-u` de la commande ci-dessus est nécessaire car dans le cas 2, aucune branche n’existe dans le dépôt local. Cette instruction permet simplement de créer une branche locale qui suit la branche distante existante sur le dépôt distant.

```
To https://forge.agriculture.rie.gouv.fr/gitlab/damien.dotta/formation.git
* [new branch]      main -> main
```

4.21 Lier un dépôt local à un dépôt distant avec terminal (1/2) ➔...

CAS 1 : création d'un nouveau répertoire

- Lancer le terminal dans le répertoire où on souhaite créer le nouveau dossier correspondant au dépôt distant (rappel sur [comment faire](#))
- Taper la commande `git clone <URL-depot-distant>`

Par exemple :

```
git clone https://forge.agriculture.rie.gouv.fr/gitlab/damien.dotta/test.git
```

- Ouvrir le nouveau répertoire et commencer à y travailler comme d'habitude en y créant un projet, des scripts...

4.22 Lier un dépôt local à un dépôt distant avec terminal (2/2) ➔

CAS 2 : un répertoire existe déjà

- Lancer le terminal dans le répertoire que l'on souhaite lié au dépôt distant (rappel sur [comment faire](#))
- Taper la commande `git remote add origin <URL-depot-distant>`

Par exemple :

```
git remote add origin https://forge.agriculture.rie.gouv.fr/gitlab/damien.dotta/test.git
```

4.23 Lier un dépôt local à un dépôt distant avec gitssp (1/2)

Le package [gitssp](#) peut également être utilisé pour effectuer la liaison.

Plusieurs branches distantes (développement, recette et production) peuvent être créées en une seule instruction R.

Procédure :

- Créer un projet .Rproj dans votre working directory (sans choisir “Contrôle de version”)
- Charger le package avec [library\(gitssp\)](#)
- Lancer la fonction R [ajouter_git\(\)](#)

4.24 Lier un dépôt local à un dépôt distant avec gitssp (2/2)

La fonction `ajouter_git()` comprend 2 arguments :

- `origin` : URL du dépôt distant (obligatoire)
- `dossier` : chemin vers le projet .Rproj que vous avez créé au préalable

Par exemple :

```
1 ajouter_git(  
2   origin = "https://forge.agriculture.rie.gouv.fr/gitlab/damien.dotta/test.git",  
3   dossier = "~/test/"  
4 )
```

4.25 Pousser les changements sur la forge avec RStudio

Après avoir ajouté vos changements à l'index et fait vos commits, vous pouvez “pousser” ces modifications sur la forge avec RStudio.

Dans l'exemple ci-dessous, après avoir committer une fois, RStudio me prévient dans l'onglet “Git” que je suis sur la branche **développement** d'un commit :



Pour pousser ce commit sur le dépôt distant, il suffit simplement de cliquer sur le bouton  Git-push

4.26 Pousser les changements sur la forge avec le terminal ➔

Voilà comment pousser vos modifications sur la forge avec le terminal :

- Taper la commande `git status` pour récupérer l'information de la situation de notre dépôt local par rapport au dépôt distant.

```
[damien.dotta@stats-prod-rstudio-3.zsg.cdpagri.fr yoyo]$ git status
On branch developpement
Your branch is ahead of 'origin/developpement' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

- Taper la commande `git push` pour pousser ce commit sur le dépôt distant.

```
[damien.dotta@stats-prod-rstudio-3.zsg.cdpagri.fr yoyo]$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 342 bytes | 114.00 KiB/s, done.
```

4.27 Utilisation des instructions `git push` et `git pull`

Dans les slides précédentes, une utilisation succincte des instructions `git push` et `git pull` a été faite.

Pour être sûr de maîtriser vos interactions entre le dépôt local et le dépôt distant, il est recommandé d'utiliser la syntaxe plus complète sur le modèle suivant :

- `git push <alias> <nom-branche>`
- `git pull <alias> <nom-branche>`

Exemples :

- `git push origin developpement`
- `git pull origin production`

4.28 Exercice 3

Exercice 3

Reprendre [l'exercice 2](#)

- Aller sous dans la forge Gitlab dans le groupe « ssp/BAC-A-SABLE »
- Créer un dépôt vide dans ce groupe au format “prenom-nom-formation-git” dans la forge Gitlab
- Lier votre projet R à ce dépôt distant
- Poussez vos commits déjà effectués sur le dépôt distant

- Apporter des changements dans un de vos scripts R (sauf [Analyse.R](#))
- Ajouter vos changements à l’index, faire un nouveau commit et pousser sur le dépôt distant

4.29 Récupérer les changements en local avec RStudio

Pour récupérer sur le dépôt local des changements qui ont été poussés sur le dépôt distant avec RStudio.

Exemples de cas d'usage :

- Si un membre de l'équipe projet a poussé des modifications avant vous sur le dépôt distant.
- Si vous voulez récupérer les dernières modifications apportées à la branche distante suite à une `merge request` (voir plus loin)

Pour récupérer les commits depuis le dépôt distant, il suffit simplement de cliquer sur le bouton 

Dans le cas d'un travail en équipe, il s'agit d'une opération à faire très fréquemment.

Git vous empêchera de pousser vos modifications sur le dépôt distant si vous n'avez pas au préalable récupérer toutes les dernières modifications qui s'y trouvent.

Message qui s'affiche dans ce cas :

```
>>> git push origin HEAD:refs/heads/developpement
To https://forge.agriculture.rie.gouv.fr/gitlab/damien.dotta/test.git
! [rejected]      HEAD -> developpement (fetch first)
error: failed to push some refs to 'https://forge.agriculture.rie.gouv.fr/gitlab/damien.dotta/test.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.
|
```

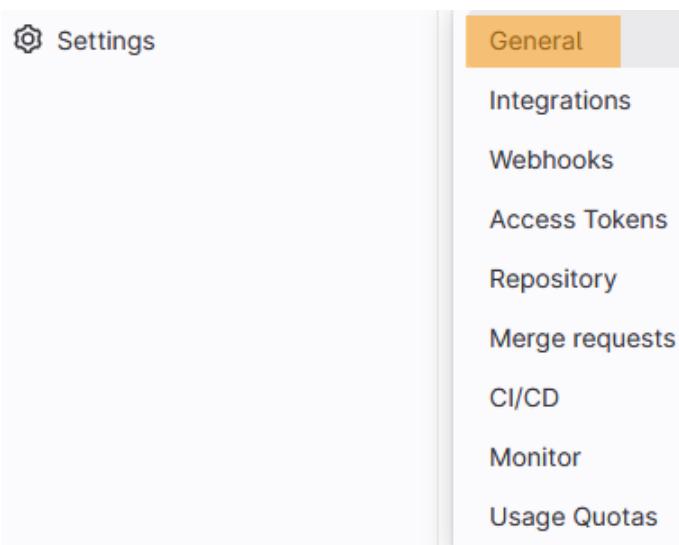


4.30 Récupérer les changements en local avec le terminal ➔

- Taper la commande `git pull` pour récupérer les derniers commits depuis dépôt distant.

```
[damien.dotta@stats-prod-rstudio-3.zsg.cdpagri.fr yoyo]$ git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 343 bytes | 5.00 KiB/s, done.
```

4.31 Supprimer un dépôt dans Gitlab



Descendre en bas de la page puis :

Delete project

Deleting the project will delete its repository and all related resources, including issues and merge requests.

Deleted projects cannot be restored!

Delete project

4.32 Créer des tags avec le terminal ➔

Les tags sont des étiquettes qui pointent vers des points spécifiques dans l'historique de Git.

Ils sont très utiles pour étiqueter les versions de vos projets (v0.1, v0.2...).

Pour créer un tag v0.1 sur le dernier commit :

```
git tag -a v0.1 -m "Message de tag"
```

Pour créer un tag v0.1 a posteriori (avec l'ID du commit) :

```
git tag -a v0.1 0af8bfd9 -m "Message de tag"
```

Pour partager les tags sur le dépôt distant :

```
git push origin --tags
```

Pour lister les tags :

```
git tag
```



4.33 Créer des tags avec gitlab

The screenshot shows the GitLab interface for creating a new tag. On the left, a sidebar menu is open with the following items:

- T test
- Project information
- Repository
- Files
- Commits
- Branches
- Tags** (highlighted)
- Contributor statistics
- Graph
- Compare revisions

The main content area is titled "New Tag". It contains the following fields:

- Tag name:** v0.1
- Create from:** 0af8bfd988cf6d61ed4f63b4067cf7b232380c09
- Message:** Version de l'application pour recette de fin mars 2024

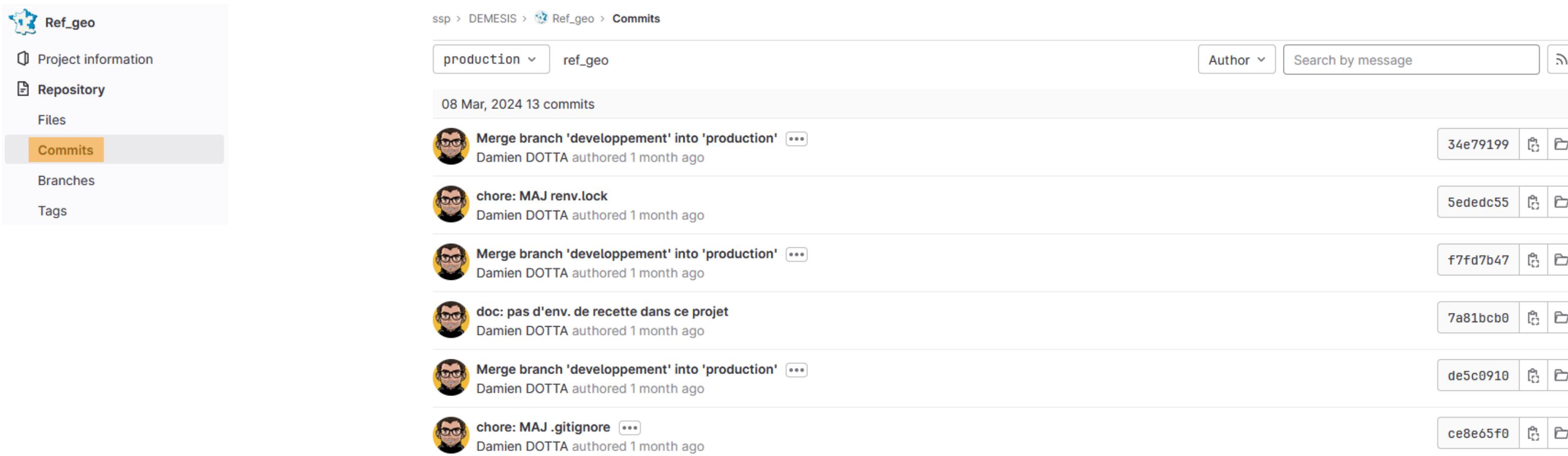
At the bottom, there are "Create tag" and "Cancel" buttons.

Il reste à faire un `git pull` pour récupérer le nouveau tag dans votre working directory.

4.34 Revenir en arrière avec Gitlab (1/3)

L'interface de Gitlab permet de facilement se déplacer dans l'historique d'un projet versionné.

Par exemple :

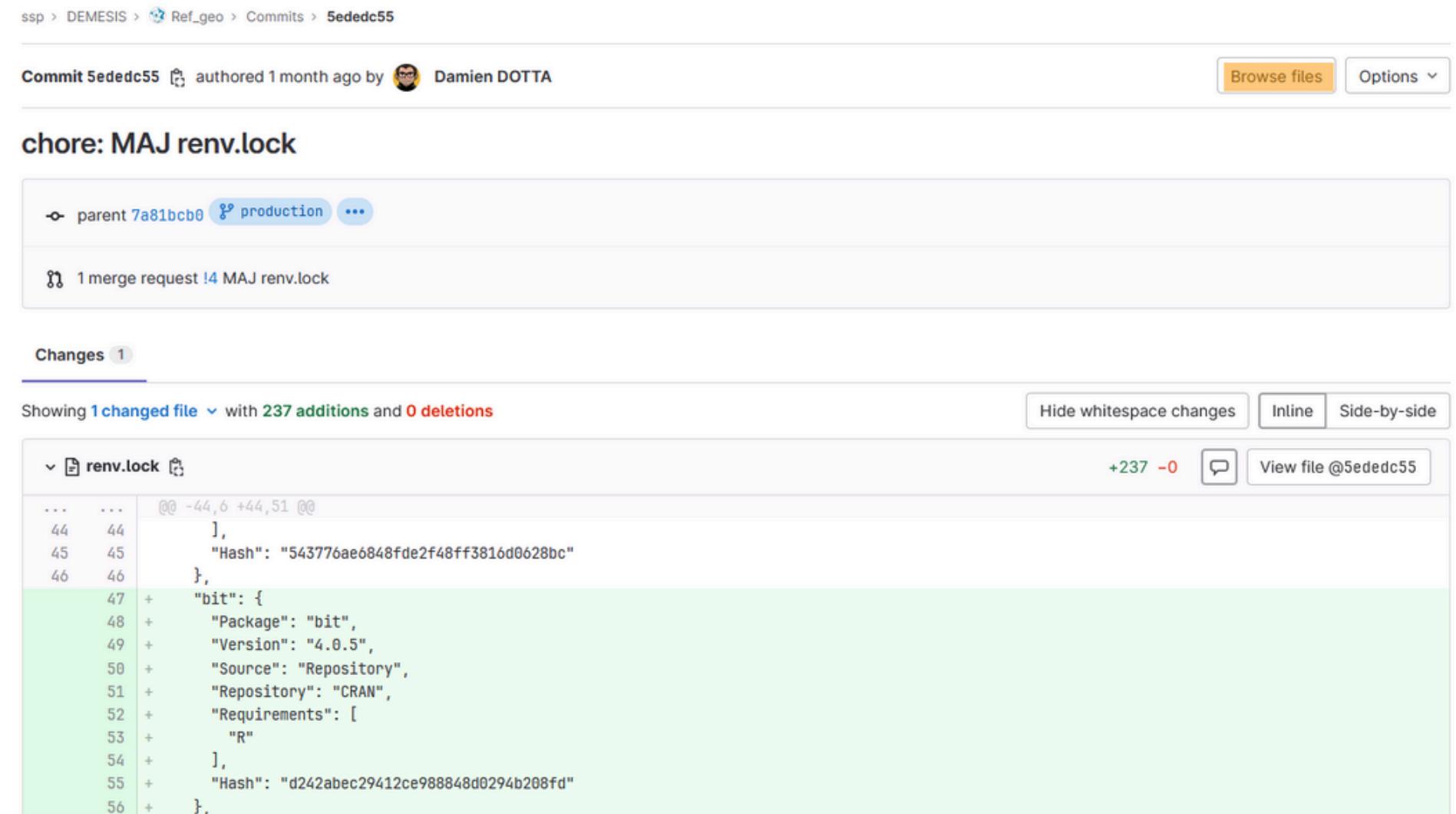


The screenshot shows the Gitlab interface for the 'Ref_geo' project. On the left, a sidebar lists 'Project information', 'Repository' (selected), 'Files', 'Commits' (selected), 'Branches', and 'Tags'. The main area displays a commit history for the 'production' branch. The commits are as follows:

Commit Message	Author	Date	SHA
Merge branch 'développement' into 'production'	Damien DOTTA	1 month ago	34e79199
chore: MAJ renv.lock	Damien DOTTA	1 month ago	5edede55
Merge branch 'développement' into 'production'	Damien DOTTA	1 month ago	f7fd7b47
doc: pas d'env. de recette dans ce projet	Damien DOTTA	1 month ago	7a81bcb0
Merge branch 'développement' into 'production'	Damien DOTTA	1 month ago	de5c0910
chore: MAJ .gitignore	Damien DOTTA	1 month ago	ce8e65f0

4.35 Revenir en arrière avec Gitlab (2/3)

Cliquer sur le commit d'intérêt et dans la fenêtre qui en détaille le contenu, cliquer sur “Browse file” :



The screenshot shows a Gitlab commit detail page for a commit named `Sededc55`. The commit message is `chore: MAJ renv.lock`. It has one parent commit `7a81bcb0` and one merge request `!4` titled `MAJ renv.lock`. Below the commit details, there is a section titled "Changes" with a count of 1. It shows a diff for the file `renv.lock`. The diff highlights new code added to the file, specifically a new object with properties like "bit", "Package", "Version", "Source", "Repository", "Requirements", and "R". The code is as follows:

```
... ... @@ -44,6 +44,51 @@  
 44 44    ],  
 45 45    "Hash": "543776ae6848fde2f48ff3816d0628bc"  
 46 46  },  
 47 + "bit": {  
 48 +   "Package": "bit",  
 49 +   "Version": "4.0.5",  
 50 +   "Source": "Repository",  
 51 +   "Repository": "CRAN",  
 52 +   "Requirements": [  
 53 +     "R"  
 54 +   ],  
 55 +   "Hash": "d242abec29412ce988848d0294b208fd"  
 56 + },
```

4.36 Revenir en arrière avec Gitlab (3/3)

On accède ainsi à l'état du projet tel qu'il était au moment du commit d'intérêt.

The screenshot shows a Gitlab repository interface. At the top, there's a breadcrumb navigation: ssp > DEMESIS > Ref_geo > Repository. Below this is a commit card for a commit by Damien DOTTA 1 month ago, with the message "chore: MAJ renv.lock". To the right of the commit message is the commit hash "5edede55" and a copy icon. Below the commit card is a navigation bar with dropdowns for "5edede553e011aaabd..." and "ref_geo / +", and buttons for "History", "Find file", "Web IDE", a download icon, and "Clone". The main area displays a table of files with their last commit details:

Name	Last commit	Last update
App	feat: envoi fichiers sur gitlab	1 month ago
renv	feat: MAJ fichiers du renv	1 month ago
.Rprofile	feat: MAJ fichiers du renv	1 month ago
.gitignore	chore: MAJ .gitignore	1 month ago
README.md	doc: pas d'env. de recette dans ce projet	1 month ago
Ref_geo.Rproj	Ajout de la version : 'developpement'	1 month ago
app.R	feat: envoi fichiers sur gitlab	1 month ago
renv.lock	chore: MAJ renv.lock	1 month ago

Remarque : des manipulations quasi-identiques permettent de faire la même chose avec les tags.



4.37 Revenir en arrière avec RStudio

Avec RStudio, en cliquant sur le bouton “Historique” :

On accÈde aux diffÈrentes versions des fichiers qui ont t modifi au fil du temps

Remarque : avec RStudio il n'est pas possible de revenir en arrière sur l'ensemble du projet

Sujet Auteur Date (UTC) SHA

HEAD -> refs/heads/master origin/master origin/HEAD feat: ajout fonction de suppression Damien DOTTA <damien.dotta@agricultur 2024-04-08 e0777bb9
feat: renommer git_commandes_system par gitssp Damien DOTTA <damien.dotta@agricultur 2024-04-08 eddd087d
Merge branch 'master' of https://forge.agriculture.rie.gouv.fr/gitlab/ssp/bmis/gitssp Damien DOTTA <damien.dotta@agricultur 2024-04-08 56339565
feat: ajout fonctions pour lister les branches Damien DOTTA <damien.dotta@agricultur 2024-04-08 f7c0abb6
feat: ajout fonctions pour lister les branches Damien DOTTA <damien.dotta@agricultur 2024-04-08 ed17b931
feat: ajout fonction pour supprimer les branches locales Damien DOTTA <damien.dotta@agricultur 2024-04-08 e91ac4aa
fix: ajout argument par défaut pour push_branch gitlab Damien DOTTA <damien.dotta@agricultur 2024-04-08 823512e0

SHA e91ac4aa142abbc5f6b8614748becbda57d13bb2
Author Damien DOTTA <damien.dotta@agriculture.gouv.fr
Date (UTC) 2024-04-08 12:43
Subject feat: ajout fonction pour supprimer les branches locales
Parent 823512e993cb745037eb0d5d80dd80c622c37050

[data-raw/git_commandes_system.R](#) 2 fichiers modifiés dans ce commit
[data/git_commandes_system.rda](#)

cliquer ici pour voir l'état du fichier [git_commandes_system.R](#)

Voir le fichier @ e91ac4aa

```
@@ -12,14 +12,15 @@ git_commandes_system <- list(  
12 12 add_origin = function(origin) system(paste0("git remote add origin ", origin)),  
13 13 get_origin = function() system("git config --get remote.origin.url", intern = TRUE),  
14 14 get_current_branch = function() system("git symbolic-ref --short HEAD", intern = TRUE),  
15 delete_branch_local = function(newBranch) system(paste0("git branch -d ", newBranch)),  
15 16 add_commit = function(message = "Initial commit") system(paste0("git commit -m ", message))
```

```
# Code pour pouvoir creer `git_commandes_system.rda`
git_commandes_system <- list(
  get_user = function() system("git config --global --get user.name", intern = TRUE),
  get_email = function() system("git config --global --get user.email", intern = TRUE),
  add_user_name = function() system(paste0('git config --global user.name "', 
                                         chaine_en_titre(gsub('\\.', ' ', Sys.getenv('USER'))), '')),
  add_user_email = function() system(paste0('git config --global user.name "', 
                                         Sys.getenv('USER'), "@agriculture.gouv.fr", '')),
  set_default_branch = function() system("git config --global init.defaultBranch main", intern = TRUE),
  init_repo = function(branch) system("git init --initial-branch=main"),
  add_origin = function(origin) system(paste0("git remote add origin ", origin)),
  get_origin = function() system("git config --get remote.origin.url", intern = TRUE),
  get_current_branch = function() system("git symbolic-ref --short HEAD", intern = TRUE),
  delete_branch_local = function(newBranch) system(paste0("git branch -d ", newBranch)),
  add_commit = function(message = "Initial commit") system(paste0(
    'git add . && git commit -m "', message, "'")),
  fetch_origin = function() system("git fetch"),
  create_branch_local = function(newBranch) system(paste0('git checkout -b ', newBranch)),
  switch_branch_local = function(branch) system(paste0('git checkout ', branch)),
  push_branch_gitlab = function(newBranch, wait = FALSE) system(paste0('git push -u origin ', newBranch), wait = wait),
  simple_push = function(branch, wait = FALSE) system(paste0('git push origin ', branch), wait = wait),
  simple_pull = function(branch, wait = FALSE) system(paste0('git pull origin ', branch), wait = wait),
  save_git_password = function(password, origin = "https://forge.agriculture.rie.gouv.fr/gitlab/bmis/gitssp.git") {
    system("git config --global credential.helper 'store'")
    protocol <- str_match(origin, "(https?)://([^.]+)")[2]
    host <- str_match(origin, "(https?)://([^.]+)")[3]
    cat(paste0(protocol, '://', Sys.getenv('USER'), ':', password, '@', host), file = "~/.git-credentials")
  },
  clone_all_project = function(origin, repertoire = "") system(paste0('git clone ', origin, ' ', repertoire)),
  clone_branch_project = function(origin, branch, repertoire){
    system(paste0('git clone ', origin, ' --branch ', branch, ' --single-branch ', repertoire))
  }
)
usethis::use_data(git_commandes_system, overwrite = TRUE)
```

4.38 Revenir en arrière avec le terminal

Pour revenir à la version du tag v0.1 :

```
git checkout v1.0
```

Pour revenir à la version du commit 0af8bfd9 :

```
git checkout 0af8bfd9
```

Pour remonter de 4 commits dans le passé :

```
git checkout nom-de-branche~4
```

Pour revenir au présent :

```
git checkout nom-de-branche
```

 Note

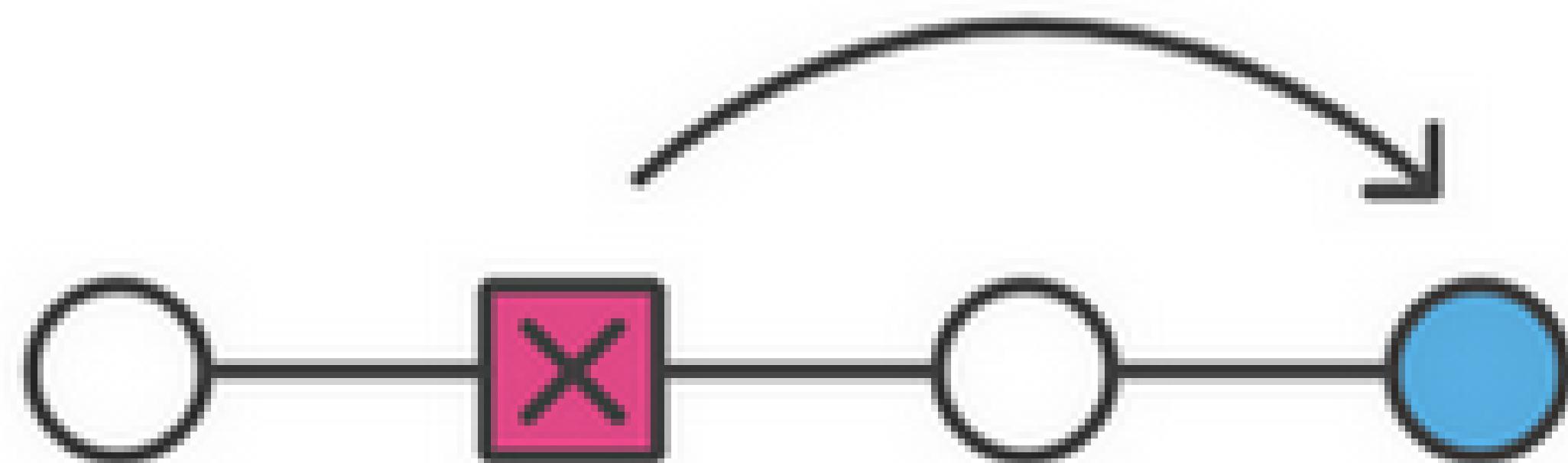
Les instructions Git ci-dessus peuvent aussi se décliner sur un fichier spécifique.

Par exemple avec la structure suivante pour un commit et un fichier particulier : `git checkout commitID file-name`

4.39 Inverser un commit

La commande `git revert SHA-commit` permet d'inverser/de défaire ce qui avait été fait au moment du commit d'intérêt.

Attention, le `git revert` crée un nouveau commit (d'inversion c'est-à-dire que les lignes ajoutées seront supprimées et les fichiers supprimés seront recréés...).



4.40 Si vous ne deviez retenir que quelques commandes ! *À retenir !*

Action

R

>-

Vous ajoutez vos fichiers à l'index

Cochez les fichiers

`git add`

Vous committez vos changements

Appuyez sur le bouton “Commit” et vous renseignez votre message de commit

`git commit -m "Votre message"`

Vous envoyez vos commits en local sur le serveur distant

Appuyez sur le bouton “Push”

`git push origin <nom-de-branche>`

Supplément si vous travaillez à plusieurs :

Récupérer les éventuels commits de vos collègues

Appuyez sur le bouton “Pull”

`git pull origin <nom-de-branche>`

4.41 Autres instructions Git

Il y a encore pleins de concepts et d'instruction Git très utiles que vous découvrirez (ou pas) :

- Le **fork** : travailler avec une copie d'un dépôt existant
- **git cherry-pick** : sélectionner et appliquer un seul commit d'une branche à une autre
- **git stash** : mettre de côté les modifications d'un répertoire de travail
- **git blame** : montrer la révision et l'auteur qui ont modifié en dernier chaque ligne d'un fichier
- **git rebase** : réappliquer une série de commits d'une branche sur une autre branche
- **git bisect** : trouver par recherche binaire la modification qui a introduit une erreur

...

4.42 Exercice 4

Exercice 4

Reprendre [l'exercice 3](#)

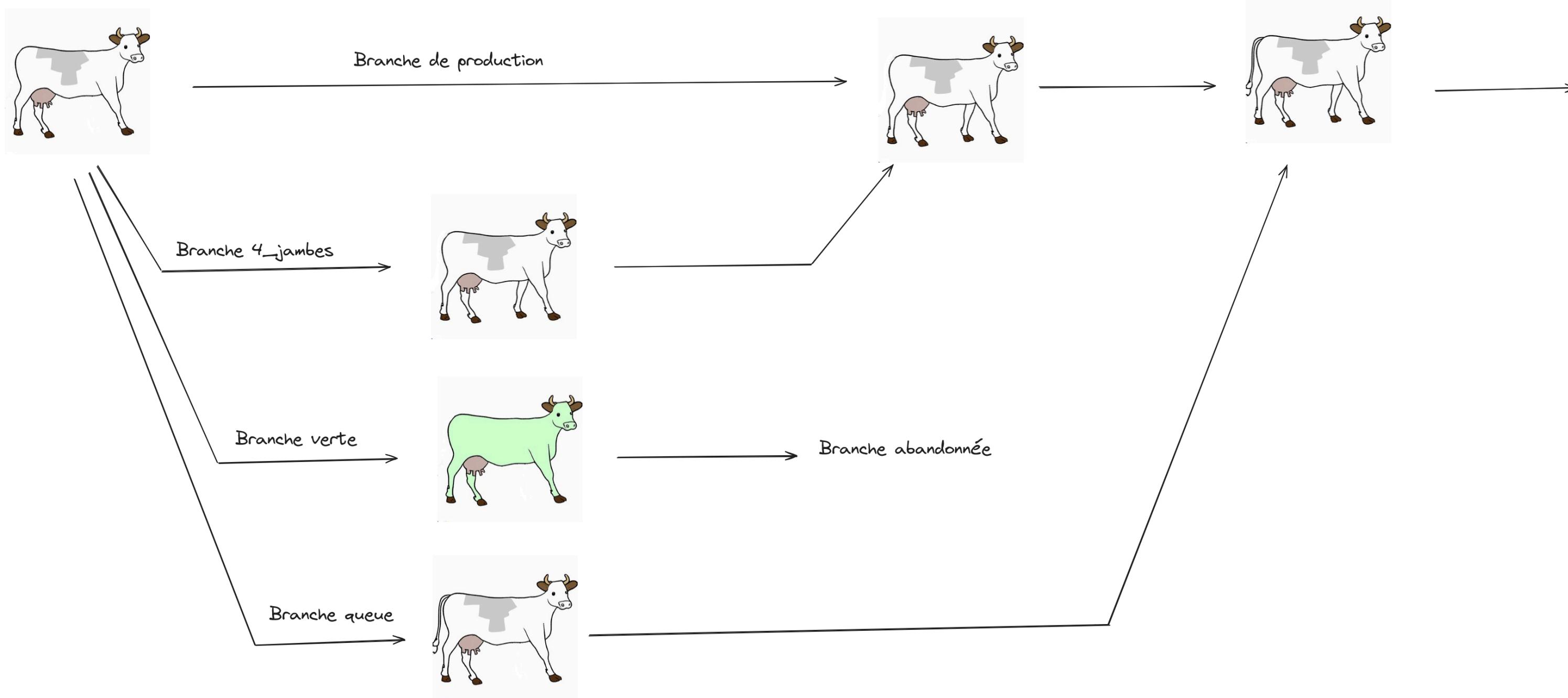
- Parcourir l'historique de votre dépôt distant avec Gitlab (les solutions avec RStudio et le terminal ont été abordées lors de l'exercice 2).
- Faire apparaître les différences entre deux versions consécutives du projet
- Afficher une version passée du projet (par exemple son état au 2ème commit)



5 Les branches



5.1 Définition des branches



5.2 Quand créer une branche ?

On crée une branche pour :

- Continuer le développement de manière isolée sans altérer le reste du dépôt
 - Pour **corriger un bug**
 - Pour **implémenter une nouvelle fonctionnalité**
 - Pour **refactorer le code...**
- Isoler les différentes étapes du développement (une branche = un environnement)

5.3 Noms des branches

Concernant le noms des branches, une bonne pratique est d'utiliser la structure suivante : <catégorie>-<nom>

Catégorie	Signification
hotfix	Pour résoudre rapidement des problèmes critiques
bugfix	Pour résoudre des erreurs/bugs
feature	Pour ajouter/supprimer/modifier des fonctionnalités
test	Pour tester une idée expérimentale (hors issue)
issue-X	Pour faire référence à l'issue n°X

5.4 Fermer une issue à partir d'une branche

Avec les forges Gitlab et Github, la fermeture des issues peut être **automatiquement gérée via les messages de commit.**

Pour cela, vous pouvez utiliser des **mots-clés** comme (au choix) :

- Close
- Fix
- Resolve
- Implement

En précisant le **#numero-issue**.

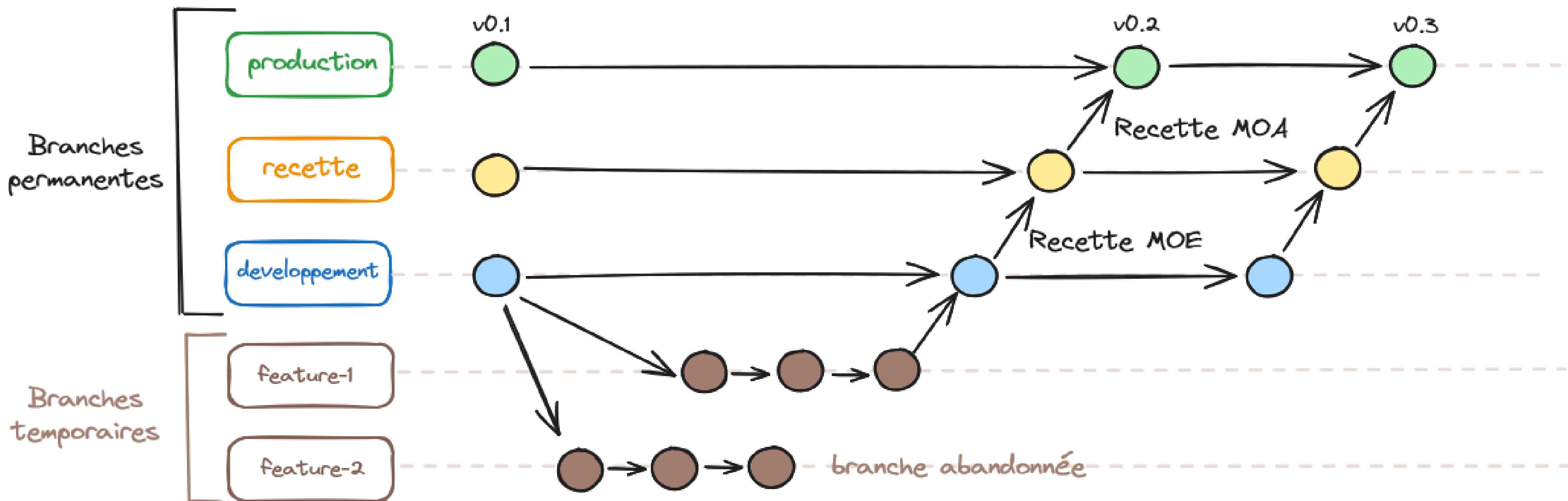
Exemple de message de commit :

```
feat: Ajout documentation utilisateurs
```

```
Close #31, en lien avec #29
```

5.5 Organisation au DEMESIS

Modèle de branches au DEMESIS



5.6 Que deviennent les branches ?

Les branches permanentes persistent tout au long de la vie du projet.

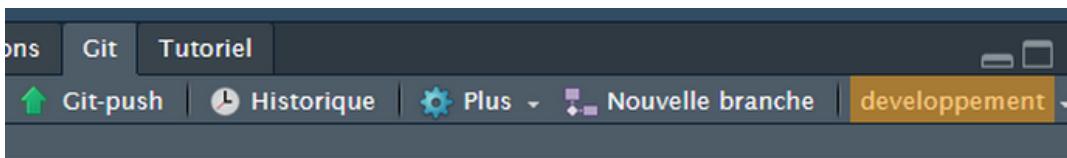
Les branches temporaires :

- Apparaissent et disparaissent au fil de la vie du projet
- Ont une durée de vie limitée...
- ... et doivent donc être détruites (manuellement ou suite à une fusion)

5.7 Savoir sur quelle branche on se situe



RStudio indique la branche sur laquelle on se situe en haut à droite de l'onglet “Git” :



`git status` est la commande la plus simple

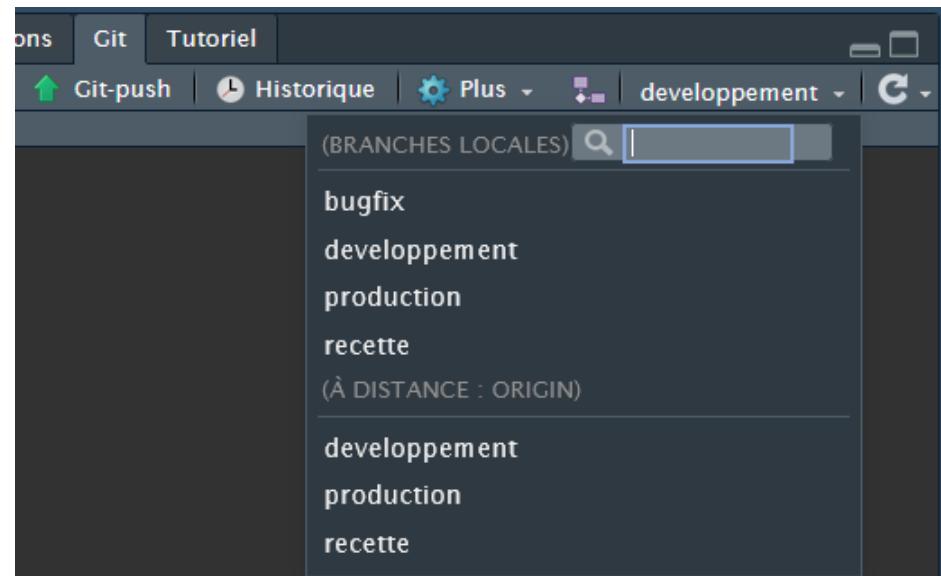
```
[damien.dotta@stats-prod-rstudio-3.zsg.cdpagri.fr test]$ git status
On branch developpement
Your branch is up to date with 'origin/developpement'.

nothing to commit, working tree clean
```



5.8 Lister les branches

Cliquer sur la petite flèche située à côté du nom de la branche courante :



- Pour afficher uniquement les branches locales

`git branch`

- Pour afficher les branches locales et distantes

`git branch -a`



- Pour afficher uniquement les branches locales

`gitssp$list_branch_local()`

- Pour afficher les branches locales et distantes

`gitssp$list_branch_all()`



5.9 Liste des branches et position sur les branches

Lorsqu'on affiche la liste des branches, la branche sur laquelle on se situe est identifiée avec un astérisque * devant.

```
[damien.dotta@stats-prod-rstudio-3.zsg.cdpagri.fr test]$ git branch -a
bugfix
* developpement
production
recette
remotes/origin/developpement
remotes/origin/production
remotes/origin/recette
```

5.10 Créer une nouvelle branche avec RStudio

Pour créer une nouvelle branche avec RStudio, il faut :

- Cliquer sur le bouton “Nouvelle branche” dans l’onglet Git



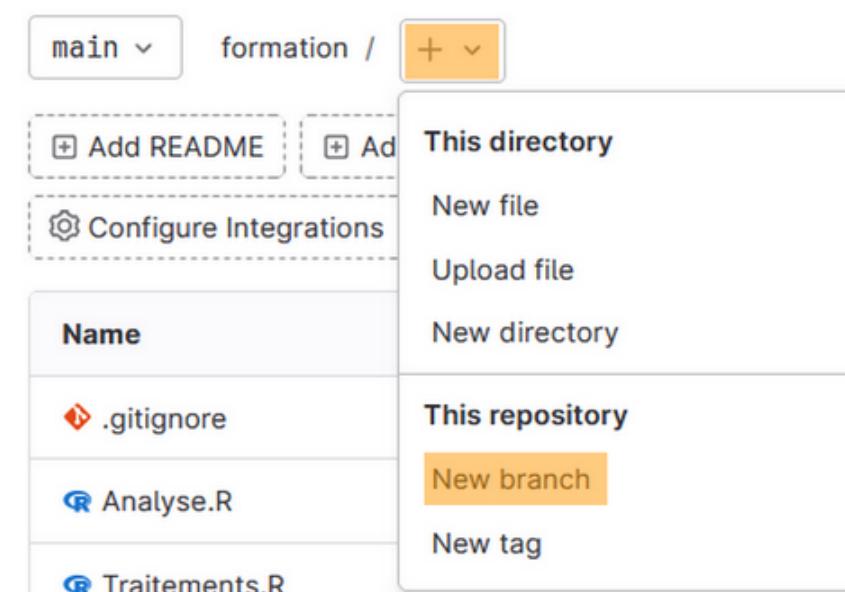
- Renseigner le nom de la nouvelle branche



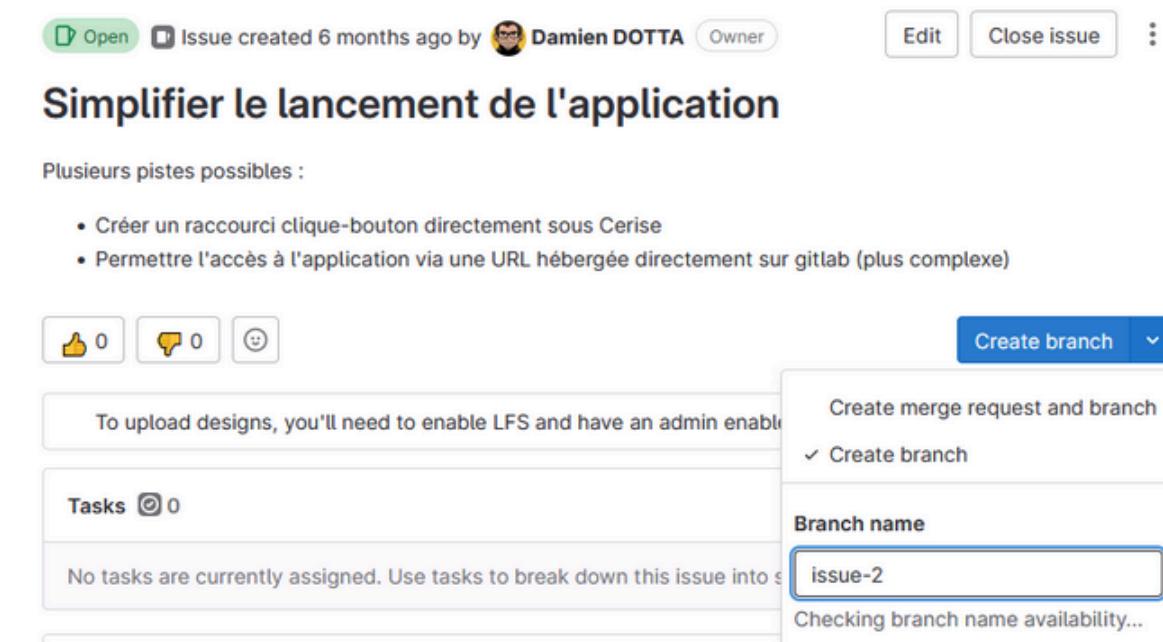
Cette procédure va créer la nouvelle branche à la fois sur le dépôt local mais aussi sur le dépôt distant.

5.11 Créer une nouvelle branche à partir de Gitlab

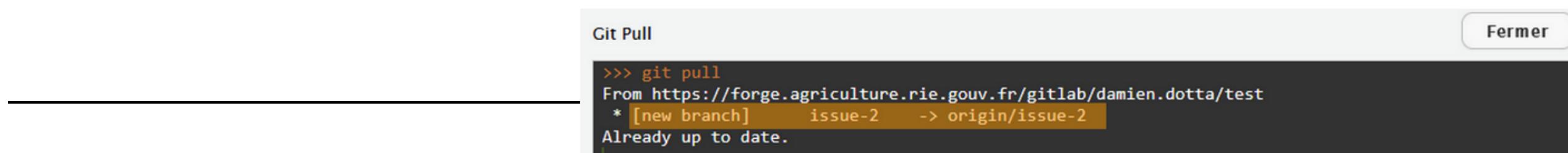
Depuis l'interface Gitlab, on peut créer une branche depuis la page d'accueil du dépôt :



Depuis l'interface Gitlab et à partir d'une issue, on peut aussi créer une branche :



Pour récupérer cette branche distante dans votre dépôt local, on fera un `git pull` dans le terminal ou RStudio.



```
>>> git pull
From https://forge.agriculture.rie.gouv.fr/gitlab/damien.dotta/test
 * [new branch]      issue-2    -> origin/issue-2
Already up to date.
```

5.12 Créer une nouvelle branche avec le terminal ➔

Pour créer une branche dans le dépôt local avec le terminal, les commandes `git branch` et `git checkout` sont utilisées.

Créer une nouvelle branche “bugfix” dans le dépôt local :

`git branch bugfix`

Créer une nouvelle branche “bugfix” dans le dépôt local et se placer dessus :

`git checkout -b bugfix`

Synchroniser la nouvelle branche locale “bugfix” avec la branche distante :

`git push -u origin bugfix`

5.13 Créer une nouvelle branche avec gitssp



Créer une nouvelle branche

```
gitssp$create_branch_local("nom-de-branche")
```

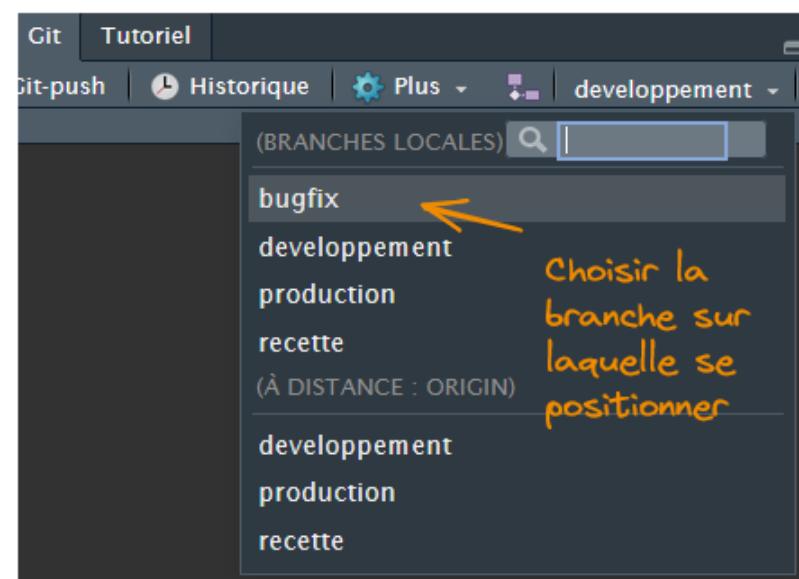
Pousser la branche locale sur le dépôt distant

```
gitssp$push_branch_gitlab("nom-de-branche")
```



5.14 Changer de branche

R



>



Avec la commande dans le terminal : Avec l'instruction R :

`git checkout nom-de-branche`

`gitssp$switch_branch_local("nom-de-branche")`



5.15 Supprimer des branches locales



Avec l'IHM RStudio, on ne peut pas supprimer de branches locales.



Avec la commande dans le terminal : Avec l'instruction R :
`git branch -d nom-de-branche` `gitssp$delete_branch_local("nom-de-branche")`



Important

Git empêche la suppression de branche si vous êtes positionnés dessus. Dans ce cas, il faut au préalable changer de branche avec `git checkout nom-de-branche`



5.16 Supprimer des branches distantes



Avec l'IHM RStudio, on ne peut pas supprimer de branches distantes.



```
git push origin --delete nom-de-branche OU
git fetch --prune
```



```
gitssp$delete_branch_gitlab("nom-de-branche")
```

GitLab permet de protéger les branches spécifiques en empêchant les utilisateurs non autorisés de les modifier.
Pour choisir les branches protégées, se rendre dans le bandeau de gauche [>settings](#) puis [Repository](#) et dérouler le menu [Branche rules](#).
Puis sélectionner la ou les branches à protéger (voir [cette page](#) pour en savoir plus sur ce qu'il est possible de faire avec les branches protégées.)

[production](#) [default](#) [protected](#)

- Allowed to push and merge: 1 role
- Allowed to merge: 1 role

5.17 Fusionner deux branches

Une fois les changements effectués sur votre branche de travail, on effectue **une fusion de branche** pour rapatrier ces modifications sur une branche permanente.

Dans le cadre d'un travail collectif, les fusions permettent à plusieurs membres d'une équipe de combiner leur travail.



Avec **RStudio** on ne peut pas fusionner 2 branches



Avec **le terminal** on peut fusionner 2 branches.

Par exemple :

1. On se place sur la branche de développement : `git checkout développement`
2. On la fusionne avec la branche de travail : `git merge branche-de-travail`
3. On pousse sur le dépôt distant : `git push origin développement`
4. On supprime la branche de travail locale : `git branch -d branche-de-travail`

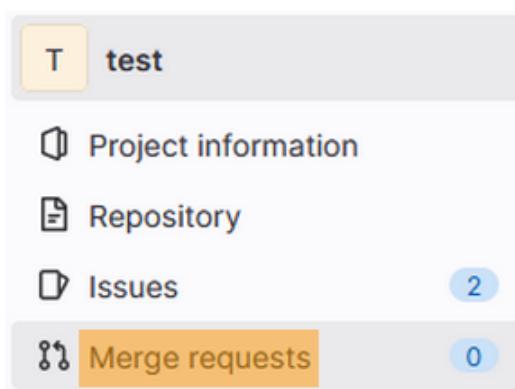


5.18 Fusionner deux branches avec Gitlab (1/3)

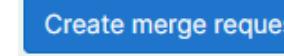
Il est conseillé de passer par Gitlab pour effectuer les fusions de branches.

Sous Gitlab, elle se nomment **merge request** (et **pull request** sous Github).

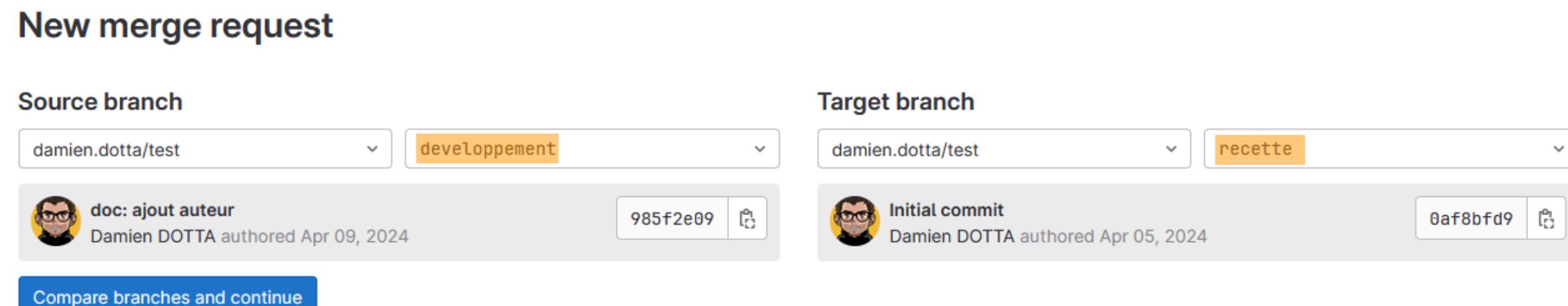
Cliquer sur “Merge requests” dans le menu de gauche :



Puis selon l’endroit par lequel vous passez, cliquer sur l’un des boutons suivants :

 ou 

Choisir la branche source et la branche cible :



New merge request

Source branch: damien.dotta/test | developpement

Target branch: damien.dotta/test | recette

doc: ajout auteur
Damien DOTTA authored Apr 09, 2024

Initial commit
Damien DOTTA authored Apr 05, 2024

Compare branches and continue

5.19 Fusionner deux branches avec Gitlab (2/3)

New merge request

From `developpement` into `recette` [Change branches](#)

Title (required)

`Envoi des développements effectués en mars 2024 pour recette côté métier`

Mark as draft

Drafts cannot be merged until marked ready.

Description

[Write](#) [Preview](#)



Fonctionnalités supplémentaires :
- Widget régional ;
- Export zip ;
- Nouvelle fenêtre d'aide...

Supports Markdown. For quick actions, type `/`.

5.20 Fusionner deux branches avec Gitlab (3/3) 🚀

Ouvreur de la MR ou responsable du domaine du projet

Assignee → Damien DOTTA

Reviewer → Hervé LEMAINTEC → Relecteur

Milestone → Jalon associé à la MR → Livraison de l'application de f...

Labels → Label associé à la MR → PAT 2024

Merge options → Laisser cocher pour les branches temporaires
DÉCOCHER pour les branches permanentes

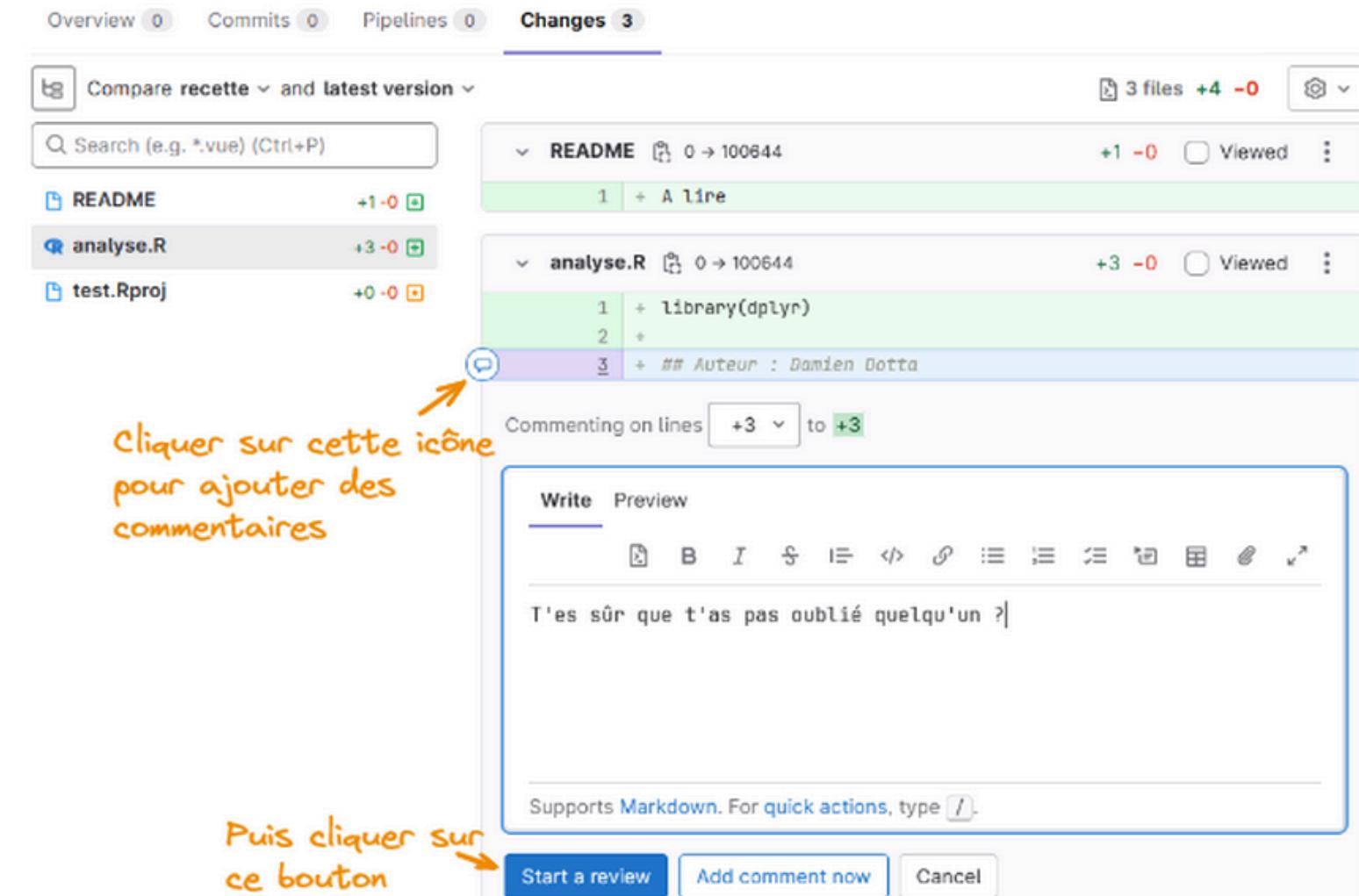
Delete source branch when merge request is accepted.

Squash commits when merge request is accepted. ?

Create merge request **Cancel**

5.21 Les relectures ou revues de code avec Gitlab (1/2)

Les fusions de branches sont aussi l'occasion d'effectuer des **revues de code** entre les membres de l'équipe projet.



Overview 0 Commits 0 Pipelines 0 Changes 3

Compare recette v and latest version v

Search (e.g. *.vue) (Ctrl+P)

README +1 -0

analyse.R +3 -0

test.Rproj +0 -0

3 files +4 -0

Viewed

README 0 → 100644 +1 -0 Viewed

A lire

analyse.R 0 → 100644 +3 -0 Viewed

library(dplyr)

Auteur : Damien Dotta

Commenting on lines +3 to +3

Write Preview

T'es sûr que t'as pas oublié quelqu'un ?

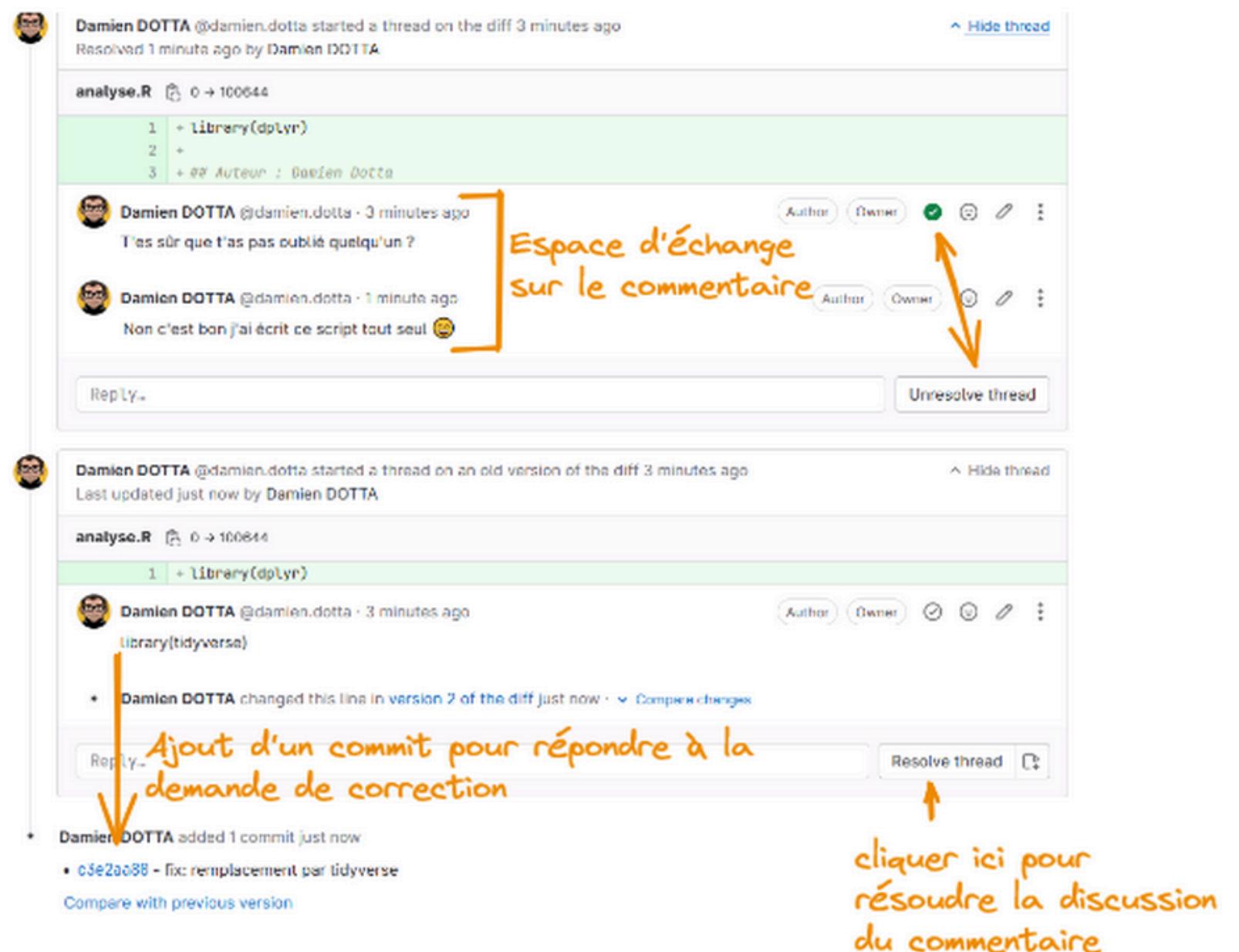
Supports Markdown. For quick actions, type /.

Start a review Add comment now Cancel

Cliquez sur cette icône pour ajouter des commentaires

Puis cliquez sur ce bouton

5.22 Les relectures ou revues de code avec Gitlab (2/2)



The screenshot shows two separate code review threads on GitLab:

Top Thread: Damien DOTTA started a thread on the diff 3 minutes ago. The diff shows changes to `analyse.R` from version 0 to 100644. The changes are:

```
1 + library(dplyr)
2 +
3 + @@ Auteur : Damien Dotta
```

Comments:

- Damien DOTTA: T'es sûr que t'as pas oublié quelqu'un?
- Damien DOTTA: Non c'est bon j'ai écrit ce script tout seul 😊

Bottom Thread: Damien DOTTA started a thread on an old version of the diff 3 minutes ago. The diff shows changes to `analyse.R` from version 0 to 100644. The changes are:

```
1 + library(dplyr)
```

Comments:

- Damien DOTTA: library(tidyverse)
- Damien DOTTA: Damien DOTTA changed this line in version 2 of the diff just now · Complex changes

A commit was added:

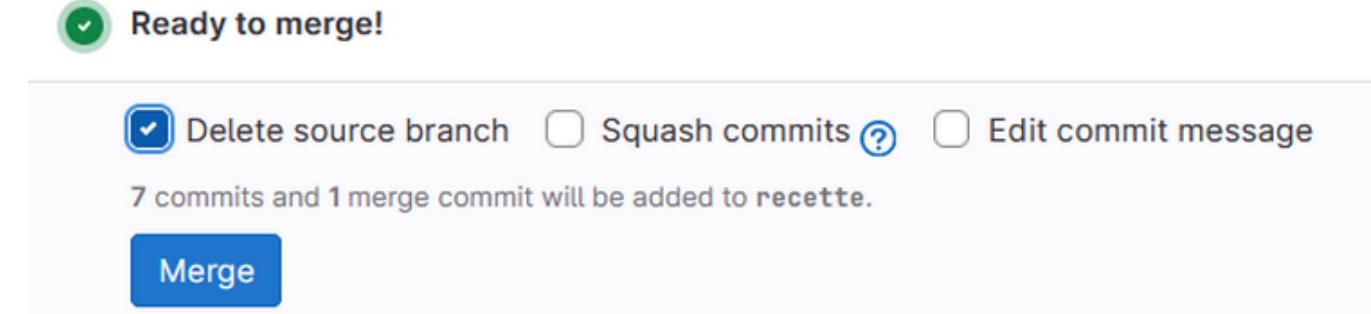
- Damien DOTTA added 1 commit just now
- c3e2aa88 - fix remplacement par tidyverse

Annotations in orange:

- An annotation points to the comment area in the top thread with the text: "Espace d'échange sur le commentaire".
- An annotation points to the "Reply" button in the bottom thread with the text: "Ajout d'un commit pour répondre à la demande de correction".
- An annotation points to the "Resolve thread" button in the bottom thread with the text: "cliquer ici pour résoudre la discussion du commentaire".

5.23 Effectuer le merge avec Gitlab

Une fois la revue de code effectuée (pas obligatoire), appuyer sur le bouton “Merge” pour valider la fusion :



Important

Après avoir effectué la fusion sur gitlab, ne pas oublier de faire un `git pull` dans votre working directory afin de rapatrier les changements issus du merge !

5.24 Stratégies de fusion

Sans rentrer dans le détail, il existe 2 principales stratégies de merge dans Git :

- le [fast-forward-merge](#) ou [merge rapide](#) : utilisée principalement pour la **correction de petites fonctionnalités**
Il a lieu lorsque le chemin entre la pointe de la branche-de-travail et la branche permanente **est linéaire**.
- le [3-way-merge](#) ou [merge à 3 sources](#) : utilisée pour l'intégration de **fonctionnalités de plus long terme**
Il a lieu lorsque la branche permanente a été modifiée pendant les développements effectués sur la branche-de-travail.

Pour en savoir plus, consulter [cette page](#) très complète sur le sujet et en français.

5.25 Définition des conflits

Les conflits surviennent généralement lorsque deux personnes ont modifié les mêmes lignes dans un fichier, ou si par exemple un développeur a supprimé un fichier alors qu'un autre développeur le modifiait.

Dans ces cas, Git ne peut pas déterminer automatiquement la version correcte

Comportement de Git lors d'un conflit :

- **Les conflits n'affectent que le développeur qui effectue le merge**
- Les autres membres de l'équipe ne sont pas conscients du conflit.
- Git marquera le fichier comme étant en conflit et bloquera le processus de merge.

=> Il incombe alors aux développeurs de résoudre le conflit.

Source: [Atlassian](#)



5.26 Gérer les conflits (1/2)

Que ce soit avec le terminal ou sur Gitlab, le (seul) avantage des conflits est qu'on est vite avertis :)



```
[damien.dotta@stats-prod-rstudio-3.zsg.cdpagri.fr form]$ git merge bugfix-tidyr
Auto-merging Analyse.R
CONFLICT (content): Merge conflict in Analyse.R
Automatic merge failed; fix conflicts and then commit the result.
```

Merge blocked: merge conflicts must be resolved.

Merge details

- The source branch is [2 commits behind](#) the target branch
- 1 commit and 1 merge commit will be added to [main](#).
- Source branch will be deleted.

Et si jamais cela ne suffisait pas, **Gitlab nous envoie un mail** pour nous dire qu'une fusion n'a pas pu avoir lieu.

5.27 Gérer les conflits (2/2)

Gitlab nous propose 2 façons de résoudre les conflits, soit localement (ce qui correspond à la procédure à suivre avec le terminal) soit directement dans l'interface de Gitlab :

Resolve locally

Resolve conflicts

- **La résolution sur Gitlab :**

Cela signifie que l'arbitrage par l'équipe projet se fait directement sur l'interface Gitlab. Il s'agira ensuite de récupérer ces modifications sur le dépôt local.

- **La résolution locale :**

Cela signifie que l'on rapatrie dans son working directory les fichiers tels qu'ils sont en cours lors de la fusion.

Après arbitrage de l'équipe projet sur les lignes de code à retenir, il s'agit ensuite de pousser la modification sur le dépôt distant.

=> Dans ce cas, on utilisera le terminal

5.28 Délimiter les conflits

Pour délimiter la zone de conflit, Git utilise les marqueurs et annotations suivantes :

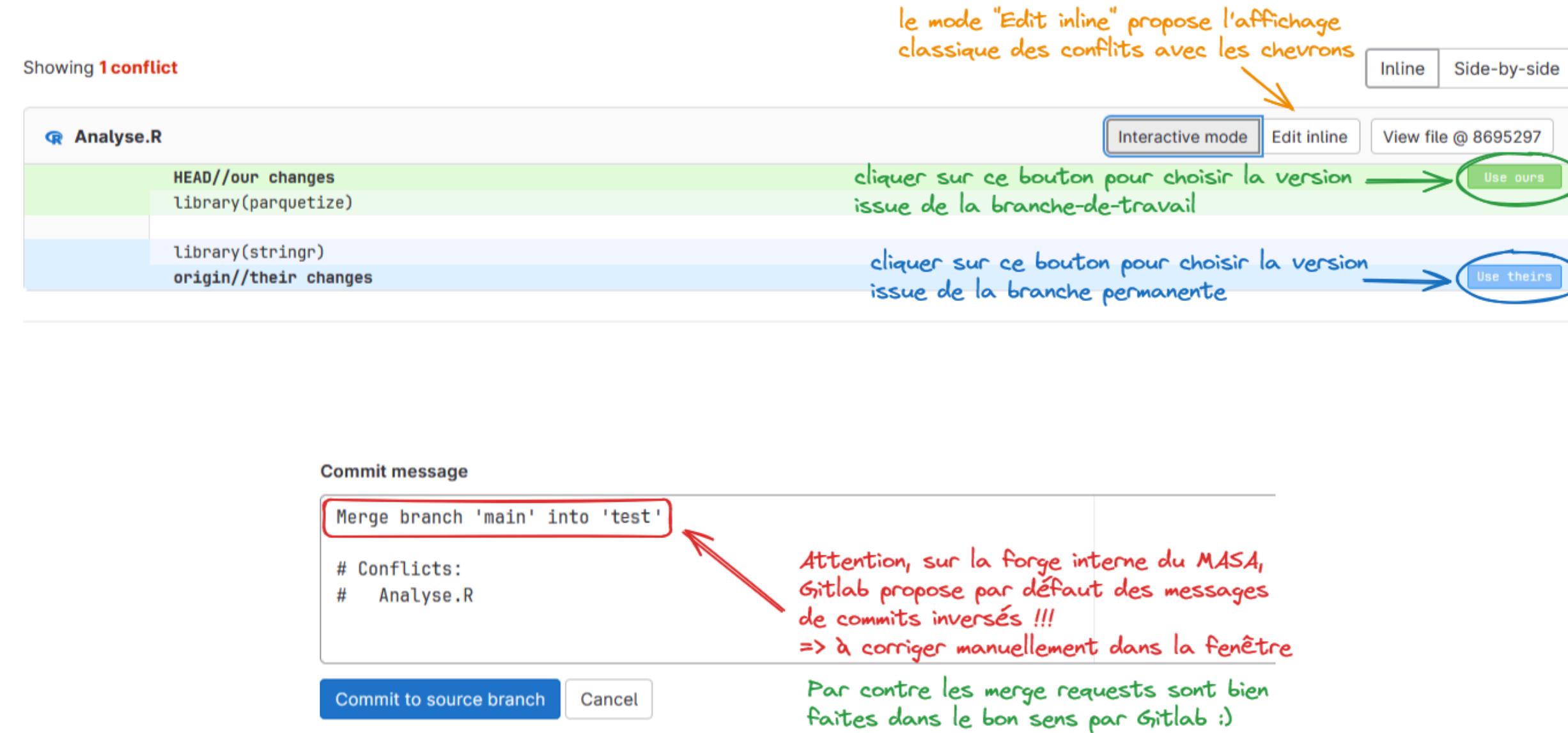
- La ligne qui commence par <<<<< marque le début de la zone conflictuelle.
- Le marqueur ====== représente la frontière entre les deux zones à fusionner.
- La ligne qui commence par >>>>> délimite la fin de la zone conflictuelle.

Exemple d'affichage d'un conflit :

```
1 <<<<< HEAD
2 library(tidyverse)
3 ======
4 library(tidyr)
5 >>>>> bugfix-tidyr
```

5.29 Résoudre les conflits sur Gitlab 🦸

Pour les “petits” conflits, il est conseillé d’utiliser Gitlab (surtout pour les novices en Git).



Showing 1 conflict

Analyse.R

HEAD//our changes
library(parquetize)

library(stringr)
origin//their changes

le mode "Edit inline" propose l'affichage classique des conflits avec les chevrons

Inline Side-by-side

Interactive mode Edit inline View file @ 8695297

cliquer sur ce bouton pour choisir la version issue de la branche-de-travail → Use ours

cliquer sur ce bouton pour choisir la version issue de la branche permanente → Use theirs

Commit message

Merge branch 'main' into 'test'

Conflicts:
Analyse.R

Attention, sur la forge interne du MASA, Gitlab propose par défaut des messages de commits inversés !!!
=> à corriger manuellement dans la fenêtre

Par contre les merge requests sont bien faites dans le bon sens par Gitlab :)

Commit to source branch Cancel

5.30 Résoudre les conflits localement avec le terminal ➔

Pour résoudre **des conflits qui ont lieu sur plusieurs fichiers**, rapatrier les fichiers localement avec le terminal **pour les analyser sur RStudio** peut être une bonne idée.

Cependant les instructions proposées par Gitlab ne fonctionnent pas tout le temps.

Le DEMESIS vous propose de suivre la procédure suivante dans ce cas :

```
1 git checkout branche-de-travail
2 git pull origin main
3 git status
4 # Arbitrer dans le code :
5 # Supprimer les marqueurs de conflit à la main soit :
6 # Pour conserver la version distante, utiliser la commande : git checkout --theirs nom-du-fichier.
7 # Pour conserver la version locale, utiliser la commande : git checkout --ours nom-du-fichier.
8 git status
9 git add .
10 git commit -m "Résolution conflit"
11 git push origin branche-de-travail
```

5.31 Gérer les conflits (5/5)

Après une fusion, on peut **supprimer les branches temporaires** qu'elles soient distantes ou locales.

- Pour **les branches distantes**, elles sont supprimées dès lors que cette case est cochée dans Gitlab (sinon voir [ici](#)) :



Delete source branch when merge request is accepted.

- Pour supprimer **les branches locales** devenues inutiles, voir [ici](#)

5.32 Éviter au maximum les conflits

Quelques conseils :

- S'interroger sur la question de l'intersection des modifications et des fichiers potentiellement impactés lors de la planification des tâches
- Communiquer avec les autres membres de l'équipe pour dire qui travaille sur quoi
- La durée de vie des branches doit être la plus courte possible
- Modulariser son code (par ex. sous forme de fonctions)
- Résoudre les conflits le plus tôt possible et ne pas laisser la situation empirer.
- Faire des “bons” commits fréquemment

5.33 Exercice 5

Exercice 5 - Simulation d'un conflit

Reprendre [l'exercice 4](#)

- Dans votre projet R, créer 2 branches `bugfix-tidyverse` et `bugfix-tidyr`
- Se positionner sur la branche `bugfix-tidyverse` et corriger le script `Analyse.R` en modifiant l'unique ligne avec `library(tidyverse)`.
- Pousser la branche `bugfix-tidyverse` et la fusionner avec la branche permanente sur Gitlab (en supprimant la branche distante `bugfix-tidyverse`).
- Récupérer les résultats du merge dans le dépôt local sur votre branche permanente/principale.
- Supprimer la branche locale `bugfix-tidyverse` qui n'est plus utile
- Se positionner maintenant sur la branche `bugfix-tidyr` et corriger le script `Analyse.R` en modifiant l'unique ligne avec `library(tidyr)`.
- Pousser la branche `bugfix-tidyr` et la fusionner avec la branche permanente sur Gitlab.
- Normalement vous devriez faire face à un joli conflit 😊
- Résoudre le conflit

Source : exercice issue de [cet article de blog](#)

5.34 Exercice 6 collectif (1/2)

Exercice 6 (facultatif)

1. Se mettre par groupes de 2/3 personnes (qui appartient au même groupe sous Gitlab) : – une personne aura la responsabilité d'être mainteneur – une à deux personnes seront développeurs
2. Le mainteneur crée un dépôt sur Gitlab ; il/elle donne des droits aux développeurs du projet.
3. Chaque personne du groupe crée une copie locale (clone) du projet sous Cerise
4. Créer un fichier `votre_nom-votre_prenom.md`, écrire trois phrases de son choix, puis commit et pousse les modifications

5.35 Exercice 6 collectif (2/2)

Exercice 6 (facultatif)

A ce stade, une seule personne (la plus rapide) devrait ne pas avoir rencontré de rejet du push. C'est normal ! Avant d'accepter une modification, Git vérifie en premier lieu la cohérence de la branche avec le dépôt distant. Le premier ayant fait un push a modifié le dépôt commun ; les autres doivent intégrer ces modifications dans leur version locale (pull) avant d'avoir le droit de proposer un changement.

5. Pour ceux dont le push a été refusé, effectuer un pull des modifications distantes
6. Dans RStudio, afficher l'historique du projet et regarder la manière dont ont été intégrées les modifications des collaborateurs
7. Effectuer à nouveau un push de vos modifications locales
8. Les derniers membres du groupe devront refaire les étapes précédentes, potentiellement plusieurs fois, pour pouvoir push les modifications locales

Source : exercice issue de [ce support de formation d'@Inseefrlab](#)



6 Pour en savoir plus





6.1 Ressources complémentaires ou pour chercher de l'aide

- Consultez [le wiki](#) de la formation qui contient :
 - Les corrections des exercices à télécharger et imprimer ;
 - Des ressources complémentaires (cheatsheet, tutoriels de connexion à Gitlab depuis Cerise...)
- Poser vos questions directement sur [l'espace discussions](#) de la formation.

The screenshot shows a GitLab interface. At the top, there's a navigation bar with links for Code, Issues (2), Pull requests, Discussions (selected), Actions, Projects, Wiki, Security, Insights, and Settings. Below the navigation is a search bar with the query "is:open". To the right of the search bar are buttons for Sort by: Latest activity, Label, Filter: Open, and New discussion. On the left, there's a sidebar with Categories: View all discussions, Annonces, Demandes d'ajouts dans la formations / Idées, Questions/réponses, and Sondages. The main area displays two discussions:

- Comment gérer les informations sensibles dans le code à pousser sur la forge ?**
bonnes pratiques
ddotta asked 42 minutes ago in Questions/réponses · Answered
- Je n'arrive plus à créer de dépôt personnel sur la forge interne gitlab**
droits
ddotta asked 51 minutes ago in Questions/réponses · Answered

6.2 Packages R pour utiliser Git

- [usethis](#) - Un package R qui contient plusieurs fonctions très utiles pour les opérations courantes avec Git (par Posit).
- [gert](#) qui est un client Git pour R très efficace (par rOpenSci).



6.3 Bibliographie à consulter

En français :

- Documentation officielle de Git en français
- Les fiches consacrées à Git dans utilitr
- Formation aux bonnes pratiques avec Git et R par l'Insee
- Tutoriel Git par Atlassian

En anglais :

- Happy Git and Github for the useR par Jennifer Bryan (Posit)
- Cheatsheet - Using Git and Github with RStudio



6.4 Remerciements

Le DEMESIS remercie :

- [@laurentC35](#) : l'auteur des fonctions du package [gitssp](#)
 - [@Inseefrlab](#) et en particulier [@linogaliana](#), [@oliviermeslin](#) et [@avouacr](#) pour leur support très inspirant sur les bonnes pratiques avec R
 - Par avance tous les relecteurs/contributeurs à cette nouvelle formation qui en feront un meilleur support pour la communauté française de R et en particulier pour les utilisateurs de RStudio.
- ⋮