

La gestion des branches

Table des matières

I. Contexte	3
II. Définir et créer une branche	3
III. Exercice : Appliquez la notion	5
IV. Utiliser des branches	5
V. Exercice : Appliquez la notion	7
VI. Fusionner des branches et gérer les conflits	7
VII. Exercice : Appliquez la notion	10
VIII. Pousser et récupérer des branches distantes	11
IX. Exercice : Appliquez la notion	13
X. Auto-évaluation	14
A. Exercice final	14
B. Exercice : Défi	15
Solutions des exercices	16

I. Contexte

Durée : 1 h

Environnement de travail : GitBash / Terminal

Pré-requis : Aucun

Contexte

Lorsque nous travaillons sur un projet de développement, nous rencontrons plusieurs problématiques. L'une d'entre elles est le fait de pouvoir travailler sur de nouvelles fonctionnalités en parallèle d'autres fonctionnalités développées par nous-mêmes ou par d'autres développeurs.

Pour nous faciliter cette tâche, les VCS (et Git en particulier) nous proposent un système de branches sur lesquelles nous pourrions travailler sans impacter le travail d'autres personnes, et gérer toute la vie du développement d'une fonctionnalité.

II. Définir et créer une branche

Objectifs

- Découvrir ce qu'est une branche
- Créer nos premières branches

Mise en situation

Lorsque nous débutons le développement d'une fonctionnalité, nous voudrions pouvoir créer un espace dans lequel produire notre code sans risquer de casser le code que nous avons fait jusqu'à maintenant. C'est pour cela que nous avons besoin d'un système de branches.

Jusqu'à présent, avec Git, nous avons travaillé sur une branche sans le savoir. Cette branche par défaut se nomme `master`. Avec GitBash, nous pouvons le voir directement dans notre ligne de commande à partir du moment où nous nous trouvons dans le répertoire d'un projet Git (ici entre parenthèses).

```
1 user@machine ~/Projects/exemple-git (master)
```

Remarque

Il se peut que nous n'ayons pas cette information directement dans notre ligne de commande si nous n'utilisons pas GitBash, mais nous pouvons simplement afficher la liste des branches avec la commande `git branch`.

```
1 * master
```

La branche sur laquelle nous nous trouvons est marquée d'une astérisque.

Le plus souvent, nous pouvons voir le système de branches comme un arbre. Un arbre possède un tronc et des branches. Dans notre cas, le tronc possède le code commun à toutes les branches (ici, le tronc serait la branche `master`), tandis que les branches poussant sur le tronc sont des ramifications qui n'ont aucun rapport entre elles.

Cependant, tout comme il est permis avec un arbre d'avoir des branches poussant sur d'autres branches, il peut être possible avec Git de créer des branches partant d'autres branches.

Méthode Créer une branche

Prenons le cas où nous nous situons sur la branche `master`. Pour créer une branche, il suffit de faire appel à la commande `git branch`, mais en lui spécifiant cette fois le nom de la branche à créer.

```
1 git branch bugfix
```

Ici, nous créons une branche nommé `bugfix`, à partir de la branche `master`. Si nous listons nos branches de nouveau avec la commande `git branch`, nous aurons le résultat suivant :

```
1 bugfix
2 * master
```

Ce que l'on voit ici, c'est que nous nous situons toujours sur notre branche `master`, mais que nous avons bien créé notre branche `bugfix`.

Une fois la branche créée, on peut se placer sur celle-ci avec la commande `git checkout` en spécifiant le nom de la branche.

```
1 git checkout bugfix
```

On peut le vérifier soit directement dans la ligne de commande, soit de nouveau grâce à la commande `git branch`.

```
1 * bugfix
2 master
```

Si nous regardons la liste de nos commits ou le contenu de nos fichiers, aucune différence ne sera notable car, à cet instant précis, les deux branches sont dans le même état.

Créer une branche plus simplement

Nous savons comment créer nos branches avec `git branch` et en changer avec `git checkout`. Il existe cependant une commande nous permettant de faire les deux en une. La commande `git checkout` avec l'option `-b` nous permet de changer de branche et de la créer à la volée. Si elle existe déjà, la commande échouera.

Exemple

```
1 git checkout -b bugfix
2 # équivaut à git branch bugfix && git checkout bugfix
```

Suite à cette commande, la branche `bugfix` a été créée et nous nous trouvons sur elle.

Syntaxe À retenir

- Le système de branches nous permet de créer des espaces de travail distincts qui permettent de développer sans impacter le code produit sur d'autres branches.
- La commande `git branch` permet de lister l'ensemble des branches d'un projet Git, mais aussi de créer de nouvelles branches lorsqu'on lui passe en paramètre le nom de la branche.
- La commande `git checkout` permet de changer de branche en lui spécifiant le nom de la branche sur laquelle nous voulons nous placer.
- La commande `git checkout -b` permet de créer une branche et d'en changer en une seule fois.

Complément

Les branches en bref¹

¹ <https://git-scm.com/book/fr/v2/Les-branches-avec-Git-Les-branches-en-bref>

III. Exercice : Appliquez la notion

Question

[solution n°1 p.17]

Dans cet exercice, vous allez réaliser les tâches suivantes :

1. Depuis la branche `master`, créez une branche `tunnel-achat` sur laquelle vous développerez une fonctionnalité de tunnel d'achat qui comprendra deux étapes : la mise au panier et le paiement.
2. Créez deux branches partant de la branche `tunnel-achat`. Pour cela, une fois la première branche créée, placez-vous dessus et créez deux nouvelles branches : `mise-au-panier` et `paiement`.
3. Créez une branche depuis le tronc principal, la branche `master`, qui vous servira à développer votre espace client. Cette branche s'appellera `espace-client`.

L'arborescence des branches sera la suivante :

```

1 master
2 |
3 |   tunnel-achat
4 |   |   mise-au-panier
5 |   |   paiement
   |   espace-client

```

IV. Utiliser des branches

Objectif

- Manipuler, modifier et supprimer des branches

Mise en situation

Nous savons créer des branches et en changer, mais nous n'avons pas encore vu le cloisonnement des espaces que nous avons créés. Nous verrons ensuite comment modifier et supprimer des branches existantes.

L'indépendance de l'historique

Chaque branche possède son propre historique de commits. Notre branche étant créée, tous les nouveaux commits apportés à celle-ci n'appartiendront qu'à elle. Si nous choisissons de repasser sur une autre branche, nous ne verrons que les commits présents sur la branche en cours.

Exemple

Modifions l'un de nos fichiers, puis créons un nouveau commit sur notre branche `bugfix`.

```
1 git commit -am "Un commit sur la branche bugfix"
```

Si nous affichons la liste de nos commits, nous avons le résultat suivant (avec la commande `git log --oneline`):

```

1 aa18e18 (HEAD -> bugfix) Un commit sur la branche bugfix
2 0c15bb5 (master) Mon premier commit

```

Nous pouvons voir ici deux commits : le premier est hérité de la branche `master`, à partir de laquelle nous avons créé notre branche `bugfix`, tandis que le second est celui que nous venons de créer.

Si nous décidons de retourner sur la branche `master` et d'exécuter de nouveau la liste des commits, nous obtiendrons ceci :

```
1 0c15bb5 (HEAD -> master) Mon premier commit
```

Notre commit `aa18e18` n'apparaît plus, mais il est toujours présent sur la branche `bugfix`. Les modifications apportées à la branche `bugfix` n'ont pas impacté la branche `master`. Cela nous permet de développer en toute sécurité sans impacter les développements présents sur le tronc commun.

Renommer une branche

Pour renommer une branche, il faut utiliser la commande `git branch` avec l'option `-m` (ou `--move`) suivie du nom de la nouvelle branche. Cela nous permet, par exemple, de corriger une faute de frappe dans le nom de la branche.

Exemple

Renommons la branche `bugfix` en `correction-bug`:

```
1 git branch -m correction-bug
```

La branche est instantanément renommée en `correction-bug`.

Supprimer une branche

La commande `git branch` sert aussi à supprimer des branches. Une fois notre développement terminé, ou simplement parce que nous avons créé une branche qui n'a plus d'utilité, nous voudrions la supprimer.

Pour cela, rien de plus simple, la commande `git branch` accepte une option `-d` pour supprimer une branche si celle-ci a déjà été fusionnée.

Il faudra se positionner sur une autre branche que celle que nous souhaitons supprimer, sinon la commande échouera.

Attention

Si l'on souhaite supprimer une branche qui n'a pas été fusionnée, il est possible d'utiliser la commande `git branch -D`, ce qui permet de forcer la suppression.

Attention cependant, car si nous supprimons une branche, **nous ne pourrons plus la récupérer**.

Exemple

La branche `correction-bug` a été fusionnée, on souhaite désormais la supprimer :

```
1 git branch -d correction-bug
```

La branche est instantanément supprimée et n'apparaît plus dans la liste des branches.

Syntaxe À retenir

- La commande `git branch -m` permet de modifier le nom d'une branche, tandis que la commande `git branch -D` permet de la supprimer.

Complément

Les branches en bref¹

¹ <https://git-scm.com/book/fr/v2/Les-branches-avec-Git-Les-branches-en-bref>

V. Exercice : Appliquez la notion

Question

[solution n°2 p.17]

Nous partons de l'arborescence de branches suivante :

```

1 master
2 |
3 |--- tunnel-achat
4 |   |--- mise-au-panier
5 |   |--- paiement
   |--- espace-client
  
```

Pour créer l'arborescence de départ, vous pourrez vous aider de la commande suivante :

```

1 git branch tunnel-achat master && git branch espace-client master && git branch mise-au-panier
   tunnel-achat && git branch paiement tunnel-achat
  
```

Nous voulons corriger l'organisation de nos branches. Utilisez les commandes que vous venez d'apprendre afin d'obtenir l'arborescence de branches suivante :

```

1 master
2 |
3 |--- purchase-tunnel
4 |   |--- cart-management
5 |   |--- shipment-info
   |--- payment-info
  
```

Une fois votre arborescence modifiée, créez un fichier `README.md` et faites un commit nommé `Add readme file` sur la branche `payment-info`, puis faites un `git log --oneline` afin de vérifier que votre commit est présent.

Revenez ensuite sur la branche `master` pour vérifier que le commit n'y est pas et que le fichier est absent.

VI. Fusionner des branches et gérer les conflits

Objectifs

- Apprendre à fusionner des branches
- Comprendre les conflits de fusion

Mise en situation

Lorsque nous avons fait des commits sur une branche et finalisé notre développement, nous aurons besoin de fusionner notre travail sur une branche plus générale, la branche `master` par exemple. Cette fusion (ou *merge* en anglais) est grandement facilitée par Git et nous allons voir dans cette partie comment fusionner une branche avec une autre.

Exemple

Partons du principe que nous avons deux branches : une branche `master` dont l'historique contient un commit, et une `bugfix` dont l'historique contient le commit de la branche `master` et un nouveau commit que nous venons de faire.

Résultat de la commande `git log --oneline` pour `master` :

```

1 0c15bb5 (HEAD -> master) Mon premier commit
  
```

Résultat de la commande `git log --oneline` pour `bugfix` :

```

1 06a8fb8 (HEAD -> bugfix) Un commit sur la branche bugfix
2 0c15bb5 (master) Mon premier commit
  
```

Nous voyons bien que la différence entre les deux branches est l'ajout du commit 06a8fb8 sur la branche `bugfix`.

Complément

Si nous voulons afficher les modifications apportées entre ces branches, nous pouvons utiliser la commande `git diff master..bugfix`:

```
1 diff --git a/README.md b/README.md
2 index 49c2190..842be52 100644
3 --- a/README.md
4 +++ b/README.md
5 @@ -1,3 +1,3 @@
6  # Fichier README.md
7
8 -Il ne contient pas grand chose.
9 +Il ne contient pas grand chose !
10
```

Nous voyons ici qu'une ligne du fichier `README.md` a été modifiée, de *Il ne contient pas grand chose.* à *Il ne contient pas grand chose !*.

Comment fusionner nos branches ?

Afin de reporter les modifications apportées sur la branche `bugfix` sur la branche `master`, nous allons devoir fusionner ces branches. Pour cela, nous allons devoir nous positionner sur la branche sur laquelle nous souhaitons récupérer les modifications, et demander à Git de les récupérer en lui indiquant de quelle branche elles proviennent.

Exemple

Pour fusionner la branche `bugfix` à la branche `master`, nous devons d'abord nous positionner sur la branche `master`, puis indiquer à Git de merger la branche `bugfix`.

```
1 git checkout master
2 git merge bugfix
```

Résultat :

```
1 Updating 0c15bb5..06a8fb8
2 Fast-forward
3  README.md | 2 +-
4  1 file changed, 1 insertion(+), 1 deletion(-)
```

Git nous indique qu'il est passé du commit 0c15bb5 au commit 06a8fb8 et a modifié le fichier `README.md`. Au total, un fichier a été modifié, et une insertion et une suppression ont été effectuées, ce qui correspond au remplacement de la ligne que nous avons vu dans l'exemple précédent.

Le cas des conflits

Lorsque nous fusionnons des branches, il se peut que nous fassions face à des conflits. Ces conflits sont le résultat d'une incapacité de Git à résoudre une fusion par lui-même. Cela arrive fréquemment lorsqu'un même endroit d'un fichier a été modifié par plusieurs commits concurrents. Les conflits sont la façon qu'a Git de nous dire d'effectuer la fusion manuellement.

Exemple

Reprenons notre premier exemple, mais en ajoutant un commit à la branche `master` qui modifie la même ligne du fichier `README.md` que la modification apportée à la branche `bugfix`.

Résultat de la commande `git log --oneline` pour `master`:

```
1 5d25bd2 (HEAD -> master) Une correction directement sur master
2 0c15bb5 Mon premier commit
```

Résultat de la commande `git log --oneline` pour `bugfix`:

```
1 06a8fb8 (HEAD -> bugfix) Un commit sur la branche bugfix
2 0c15bb5 (master) Mon premier commit
```

Cette fois, nous voyons bien que la différence entre les deux branches est l'ajout du commit `06a8fb8` sur la branche `bugfix` et l'ajout du commit `5d25bd2` sur la branche `master`.

Complément

La commande `git diff master..bugfix` affiche :

```
1 diff --git a/README.md b/README.md
2 index c548287..842be52 100644
3 --- a/README.md
4 +++ b/README.md
5 @@ -1,3 +1,3 @@
6  # Fichier README.md
7
8 -Il ne contient pas grand chose. Dommage.
9 +Il ne contient pas grand chose !
10
```

Nous voyons ici qu'une ligne du fichier `README.md` a été modifiée de *Il ne contient pas grand chose. Dommage.* en *Il ne contient pas grand chose !*.

Exemple

Nous pouvons maintenant fusionner la branche `bugfix` dans la branche `master`.

```
1 git checkout master
2 git merge bugfix
```

Résultat :

```
1 Auto-merging README.md
2 CONFLICT (content): Merge conflict in README.md
3 Automatic merge failed; fix conflicts and then commit the result.
```

Git nous indique qu'il a rencontré un conflit dans le fichier `README.md` et qu'il attend que nous résolvions le conflit, puis que nous fassions un commit. Git a en fait modifié le fichier `README.md` pour y faire apparaître les deux modifications. Rendons-nous dans le fichier pour voir ce qu'il se passe :

```
1 # Fichier README.md
2
3 <<<<<< HEAD
4 Il ne contient pas grand chose. Dommage.
5 =====
6 Il ne contient pas grand chose !
7 >>>>>> bugfix
8
```

Les lignes 3 à 7 possèdent le détail du conflit. Ici, Git nous indique que la ligne *Il ne contient pas grand chose. Dommage.* provient de `HEAD`, c'est-à-dire de la branche actuelle `master`, et *Il ne contient pas grand chose !* provient de la branche `bugfix`.

À partir de là, il nous appartient de corriger le conflit nous-mêmes, puisque Git ne sait pas le résoudre seul. Nous pourrions garder l'un ou l'autre des messages, mais modifions le fichier `README.md` en optant pour ce contenu :

```
1 # Fichier README.md
2
3 Il ne contient pas grand chose ! Dommage !
4
```

Pour valider nos résolutions de conflit, il faut ajouter le fichier, puis faire un commit.

```
1 git add README.md
2 git commit
```

La commande `git commit` va ouvrir un fichier pour indiquer le message de commit. Nous pouvons le laisser en l'état, nous reviendrons sur la signification de ce commit plus tard. Sauvegardons et fermons ce fichier, puis exécutons la commande `git log --oneline`.

```
1 5ebb148 (HEAD -> master) Merge branch 'bugfix'
2 5d25bd2 Une correction directement sur master
3 06a8fb8 (bugfix) Un commit sur la branche bugfix
4 0c15bb5 Mon premier commit
```

La branche `master` comporte maintenant, dans l'ordre chronologique (donc de bas en haut) : le commit commun aux deux branches `0c15bb5`, le commit de la branche `bugfix` `06a8fb8`, le commit de la branche `master` `5d25bd2` et un **commit de merge**. Ce commit de merge est très important, car il contient la résolution de notre conflit.

Syntaxe À retenir

- Les fusions (ou *merges* en anglais) nous permettent de récupérer des modifications apportées sur une branche sur un tronc commun. Pour cela, il faut se placer sur la branche de destination et indiquer à Git de fusionner la branche la plus à jour avec la commande `git merge`.
- Les fusions font partie des opérations les plus complexes avec les VCS. Heureusement, Git nous facilite grandement le travail. Cependant, quelques fois, Git ne peut pas résoudre les fusions seul et produit des conflits. Ces conflits doivent être corrigés manuellement pour finaliser la fusion des deux branches, et ainsi produire un commit de merge.

Complément

Branches et fusions : les bases¹

VII. Exercice : Appliquez la notion

Question

[solution n°3 p.17]

Exécutez les étapes suivantes :

1. Sur une branche `master` sans commits, créez un fichier `README.md` avec le texte *Hello world*
2. Ajoutez le fichier et créez un commit nommé `Add README.md`
3. Créez une branche `bugfix` depuis ce commit, placez-vous dessus, puis modifiez le texte du fichier pour ajouter un point à la fin avec la commande `sed -i s/world/world./g README.md`

¹ <https://git-scm.com/book/fr/v2/Les-branches-avec-Git-Branches-et-fusions%C2%A0%3A-les-bases>

4. Créez un commit nommé `Add point at the end of README.md`
5. Placez-vous de nouveau sur la branche `master` puis modifiez le fichier pour ajouter un point d'exclamation à la fin avec la commande `sed -i s/world/world\!/g README.md`
6. Créez un commit nommé `Add exclamation mark at the end of README.md`
7. Fusionnez la branche `bugfix` avec la branche `master` (vous devriez avoir des conflits)
8. Résolvez les conflits en gardant les modifications de la branche `bugfix` et terminez la fusion

VIII. Pousser et récupérer des branches distantes

Objectifs

- Définir l'upstream d'une branche et la pousser
- Récupérer une branche distante
- Supprimer une branche distante

Mise en situation

L'un des principaux intérêts à l'utilisation des branches dans Git est la possibilité de travailler dans un espace cloisonné dans le cadre de travaux collaboratifs. À cette fin, nous aurons probablement besoin d'interagir avec un dépôt distant et donc d'y pousser nos branches, ainsi que de récupérer les branches d'autres personnes.

Définir l'upstream d'une branche et la pousser

Une fois notre branche créée, nous pouvons définir son **upstream**. L'upstream est notre branche distante, c'est-à-dire la représentation de notre branche locale sur le dépôt distant. En effet, puisque nous travaillons en local avec Git, notre branche n'est pas disponible depuis notre dépôt distant. L'upstream est composée du **remote**, c'est-à-dire l'alias du serveur distant tel qu'il a été défini dans la commande `git remote add`, ainsi que du nom de la branche sur le dépôt distant.

Exemple

Si nous tentons d'exécuter la commande `git push` depuis la branche `bugfix` sans spécifier l'upstream, voilà ce que nous devrions avoir :

```
1 fatal: The current branch bugfix has no upstream branch.
2 To push the current branch and set the remote as upstream, use
3
4     git push --set-upstream origin bugfix
```

Ce que Git nous dit, c'est que notre branche n'a pas d'upstream, donc pas de référence à une branche distante. Pour résoudre ce problème, nous devons pousser notre branche en spécifiant explicitement l'upstream (paramètres `origin` et `bugfix`) et en indiquant qu'on veut le configurer automatiquement pour cette branche pour les prochaines fois (option `--set-upstream` ou `-u`).

```
1 git push --set-upstream origin bugfix
```

Résultat :

```
1 Enumerating objects: 5, done.
2 Counting objects: 100% (5/5), done.
3 Delta compression using up to 12 threads
4 Compressing objects: 100% (2/2), done.
5 Writing objects: 100% (3/3), 323 bytes | 323.00 KiB/s, done.
6 Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
```

```

7 remote:
8 remote: Create a pull request for 'bugfix' on GitHub by visiting:
9 remote: https://github.com/git/exemple-git/pull/new/bugfix
10 remote:
11 To github.com:git/exemple-git.git
12 * [new branch] bugfix -> bugfix
13 Branch 'bugfix' set up to track remote branch 'bugfix' from 'origin'.
14

```

Remarque

À partir du moment où l'upstream est défini, nous pouvons nous contenter d'utiliser la commande `git push` sans paramètres.

Récupérer une branche distante

Tout au long du développement de notre projet, nous aurons probablement à récupérer de nouvelles versions de nos branches. La plupart du temps, il pourrait s'agir de la branche `master`, mais nous pourrions avoir besoin de récupérer d'autres branches. Pour cela, il faudra se positionner sur la branche en question, puis utiliser la commande `git pull` dans le cas où la branche existait déjà en local et que le code n'était pas à jour.

Exemple

Partant du principe que la branche `other-bugfix` existe sur le dépôt distant et que nous ne l'avons pas encore en local, nous ne pourrions pas passer directement sur cette branche avec la commande `git checkout other-bugfix`.

```

1 error: pathspec 'other-bugfix' did not match any file(s) known to git

```

Pour récupérer cette branche, il faudra exécuter la commande `git fetch origin` puis `git checkout other-bugfix`.

```

1 $ git fetch origin
2 From github.com:git/exemple-git
3 * [new branch]      other-bugfix -> origin/other-bugfix
4
5 $ git checkout other-bugfix
6 Switched to a new branch 'other-bugfix'
7 Branch 'other-bugfix' set up to track remote branch 'other-bugfix' from 'origin'.

```

Dans le cas où la branche récupérée n'existait pas et qu'on vient de faire un `git fetch`, la branche est à jour. Dans le doute, ou simplement par la suite pour récupérer les nouvelles mises à jour, nous pouvons utiliser la commande `git pull` (puisque l'upstream est défini automatiquement par Git) ou `git pull origin other-bugfix`.

Supprimer une branche distante

Une fois nos modifications fusionnées, il est d'usage de supprimer la branche « morte » afin de ne garder que les branches toujours actives et d'ainsi pouvoir les parcourir plus simplement en cas de besoin. Nous savons comment supprimer une branche locale, mais nous pouvons aussi supprimer une branche distante avec la commande `git push` et l'option `--delete`.

Exemple

Supprimons la branche distante `other-bugfix` qui n'est plus utilisée.

```
1 git push origin --delete other-bugfix
```

Résultat :

```
1 To github.com:git/exemple-git.git
2 - [deleted]          other-bugfix
```

Syntaxe **À retenir**

- La représentation d'une branche locale sur le dépôt distant s'appelle l'upstream. L'upstream est composé de l'alias du dépôt distant et du nom de la branche distante, par exemple `origin bugfix`.
- Les commandes `git push` et `git pull` suivies de l'upstream permettent respectivement de pousser et de récupérer les modifications d'une branche.
- La commande `git fetch origin` permet de récupérer les branches distantes.
- La commande `git push origin --delete` suivie du nom de la branche distante permet de supprimer la branche du dépôt distant.

Complément

Branches de suivi à distance¹

IX. Exercice : Appliquez la notion

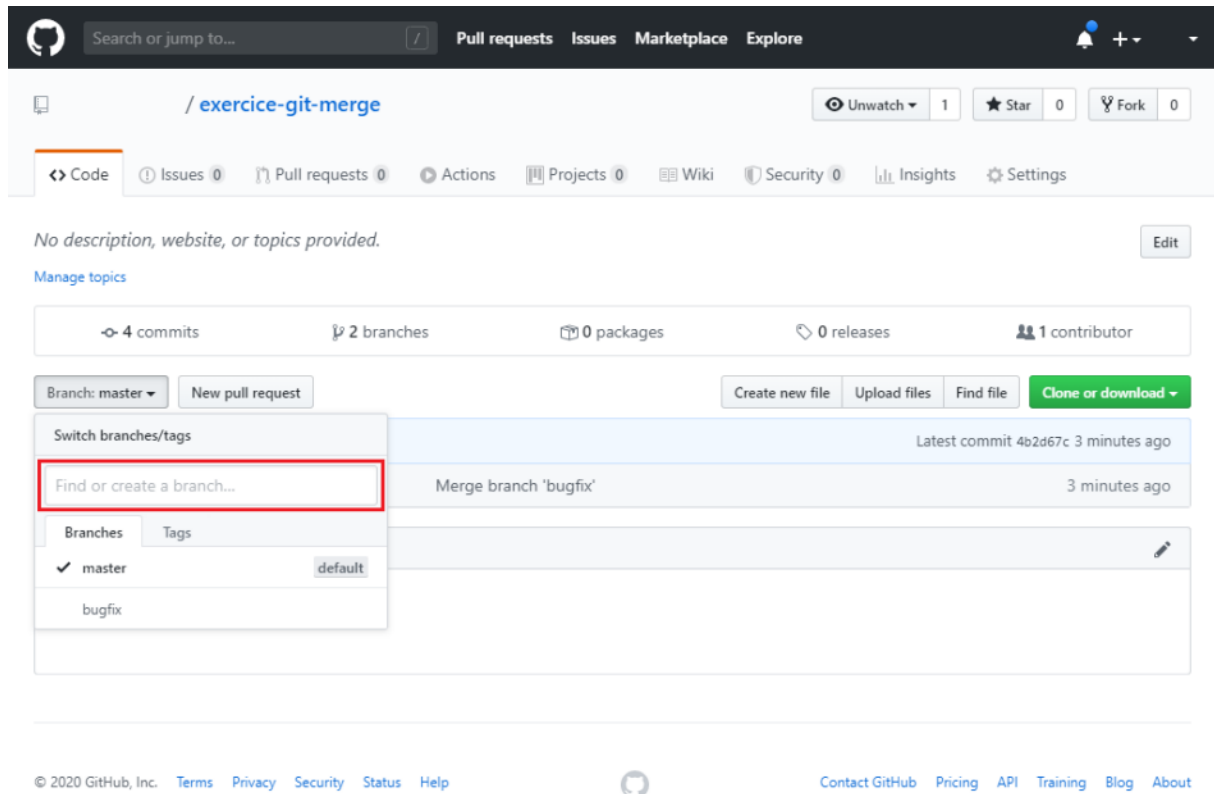
Question

[solution n°4 p.18]

Dans cet exercice :

1. Rendez-vous sur GitHub et créez un dépôt nommé **exercice-git-merge**
2. Ajoutez le remote à votre dépôt existant
3. Poussez-y vos branches `master` (sur laquelle vous créez un fichier simple) et `bugfix` (sur laquelle vous modifierez le fichier en question)
4. Depuis GitHub, créez une branche nommée `other-bugfix` dans la fenêtre des branches (cf. capture d'écran)
5. Depuis votre poste, récupérez la branche distante
6. Supprimez la branche `other-bugfix` du dépôt distant

¹ <https://git-scm.com/book/fr/v2/Les-branches-avec-Git-Branches-de-suivi-%C3%A0-distance>



X. Auto-évaluation

A. Exercice final

Exercice 1

[solution n°5 p.19]

Exercice

À quoi servent les branches ?

- ☐ À pouvoir développer sans casser le code du tronc commun
- ☐ À faciliter le travail collaboratif
- ☐ À récupérer des commits perdus

Exercice

Comment s'appelle la branche par défaut dans Git ?

Exercice

Quelle commande permet d'afficher la liste des branches locales et de créer une branche ?

Exercice

Quelle commande permet de changer de branche ?

Exercice

Comment peut-on créer une branche `bugfix` depuis la branche `master` et s'y placer ?

- ☐ `git checkout bugfix` depuis la branche `master`
- ☐ `git checkout -b bugfix` depuis la branche `master`
- ☐ `git checkout -b bugfix` depuis n'importe quelle branche
- ☐ `git checkout -b bugfix master` depuis n'importe quelle branche

Exercice

Quelle commande permet de fusionner une branche avec une autre ?

Exercice

Que faire lorsque l'on rencontre des conflits ?

- ☐ Laisser Git corriger les conflits et faire un commit
- ☐ Corriger soi-même les conflits et faire un commit
- ☐ Laisser Git corriger les conflits, ajouter les fichiers corrigés et faire un commit
- ☐ Corriger soi-même les conflits, ajouter les fichiers corrigés et faire un commit

Exercice

Qu'est-ce qu'un commit de merge ?

- ☐ Une référence vers le commit que l'on a gardé
- ☐ Un commit contenant la résolution des conflits
- ☐ Un commit indiquant qu'il n'y a pas eu de conflit

Exercice

Comment fait-on pour définir l'upstream d'une branche en même temps qu'on la pousse ?

- ☐ `git push --upstream origin bugfix`
- ☐ `git push --set-upstream origin bugfix`
- ☐ `git push -set-upstream origin bugfix`
- ☐ `git push -u origin bugfix`

Exercice

Quelle commande permet de récupérer les branches distantes ?

B. Exercice : Défi

Dans cet exercice, nous allons appliquer l'ensemble de nos connaissances avec les branches pour travailler sur un fichier HTML comportant une liste de fruits.

Question

[solution n°6 p.20]

Exécutez les étapes suivantes :

1. Sur une branche `master` sans commits, créez un fichier `index.html` (cf. contenu en fin d'exercice)
2. Ajoutez le fichier et créez un commit nommé `Add fruit list`
3. Poussez la branche `master` sur votre dépôt distant en définissant l'upstream après l'avoir créé sur GitHub
4. Créez une branche `fix-typos` depuis ce commit, placez-vous dessus, puis modifiez le texte du fichier pour corriger les fautes de frappe avec la commande `sed -i 's/Pome/Pomme/g; s/Carote/Carotte/g; s/Tommate/Tomate/g; s/Cerisse/Cerise/g' index.html`
5. Créez un commit nommé `Fix typos in fruit list`, puis poussez-le sur GitHub
6. Créez une nouvelle branche `remove-vegetables` depuis `master`, puis modifiez le fichier pour corriger la liste en supprimant les légumes avec la commande `sed -i '/Carote/d; /Salade/d; /Tomate/d' index.html`
7. Créez un commit nommé `Remove vegetables from fruit list`, puis poussez-le sur GitHub
8. Fusionnez la branche `fix-typos` dans la branche `master` (vous ne devriez pas avoir de conflits), poussez `master`
9. Fusionnez la branche `remove-vegetables` dans la branche `master` (vous devriez avoir des conflits)
10. Résolvez les conflits en gardant les corrections des deux branches et terminez la fusion, puis poussez `master` de nouveau
11. Supprimez les branches `fix-typos` et `remove-vegetables` sur le dépôt distant et en local

Fichier **index.html** du point 1 :

```

1 <!DOCTYPE html>
2 <html lang="fr">
3   <head>
4     <meta charset="UTF-8">
5     <meta name="viewport"
6       content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-
7 scale=1.0, minimum-scale=1.0">
8     <meta http-equiv="X-UA-Compatible" content="ie=edge">
9     <title>Document</title>
10  </head>
11  <body>
12    <h1>Liste des fruits :</h1>
13    <ul>
14      <li>Pome</li>
15      <li>Carote</li>
16      <li>Salade</li>
17      <li>Poire</li>
18      <li>Tommate</li>
19      <li>Cerisse</li>
20    </ul>
21  </body>
22 </html>

```

Solutions des exercices

p.5 Solution n°1

Attention

Il faut faire un premier commit avec un premier fichier (README par exemple). Sinon vous ne pourrez pas créer de branches et les gérer.

```
1 # 1
2 git checkout master
3 git branch tunnel-achat
4
5 # 2
6 git checkout tunnel-achat
7 git branch mise-au-panier
8 git branch paiement
9
10 # 3
11 git checkout master
12 git branch espace-client
```

p.7 Solution n°2

```
1 # On supprime la branche espace-client
2 git branch -D espace-client
3
4 # On se place sur la branche tunnel-achat et on la renomme en purchase-tunnel
5 git checkout tunnel-achat && git branch -m purchase-tunnel
6
7 # On renomme les branches mise-au-panier en cart-management et paiement en payment-info
8 git branch -m mise-au-panier cart-management
9 git branch -m paiement payment-info
10
11 # On crée la branche shipment-info à partir de purchase-tunnel
12 git checkout -b shipment-info purchase-tunnel
13
14 # On change de branche, on crée un fichier README.md vide et on ajoute un commit
15 git checkout payment-info
16 touch README.md
17 git add README.md
18 git commit -m "Add readme file"
19
20 # On passe sur la branche master et on constate bien que notre commit et notre fichier
    n'existent pas
21 git checkout master
22 git log --oneline
23 ls README.md
```

p.10 Solution n°3

```
1 # 1
2 echo -e "Hello world\n" > README.md
3
4 # 2
5 git add .
```

```

6 git commit -m"Add README.md"
7
8 # 3
9 git checkout -b bugfix
10 sed -i s/world/world./g README.md
11
12 # 4
13 git commit -am"Add point at the end of README.md"
14
15 # 5
16 git checkout master
17 sed -i s/world/world\!/g README.md
18
19 # 6
20 git commit -am"Add exclamation mark at the end of README.md"
21
22 # 7
23 git merge bugfix
24
25 # 8
26 vi README.md

```

README.md avec conflits :

```

1 <<<<<< HEAD
2 Hello world!
3
4 =====
5 Hello world.
6 >>>>>> bugfix
7

```

Ne gardez que la ligne 4 (bloc bugfix).

README.md avec conflits corrigés :

```

1 Hello world.
2
1 # 8 bis
2 git add .
3 git commit

```

p. 13 Solution n°4

1. Rendez-vous sur la page <https://github.com/new>

```

1 # 2
2 git remote add origin git@github.com:sarramegnag/exercice-git-merge.git
3
4 # 3
5 git push --set-upstream origin master
6 git checkout bugfix
7 git push --set-upstream origin bugfix
8
9 # 5
10 git fetch origin
11 git checkout other-bugfix
12
13 # 6

```

```
14 git push origin --delete other-bugfix
```

Exercice p. 14 Solution n°5**Exercice**

À quoi servent les branches ?

- ☒ À pouvoir développer sans casser le code du tronc commun
- ☒ À faciliter le travail collaboratif
- ☐ À récupérer des commits perdus

Exercice

Comment s'appelle la branche par défaut dans Git ?

main

Exercice

Quelle commande permet d'afficher la liste des branches locales et de créer une branche ?

git branch

Exercice

Quelle commande permet de changer de branche ?

git checkout

Exercice

Comment peut-on créer une branche `bugfix` depuis la branche `master` et s'y placer ?

- ☐ `git checkout bugfix` depuis la branche `master`
- ☒ `git checkout -b bugfix` depuis la branche `master`
- ☐ `git checkout -b bugfix` depuis n'importe quelle branche
- ☒ `git checkout -b bugfix master` depuis n'importe quelle branche

Exercice

Quelle commande permet de fusionner une branche avec une autre ?

git merge

Exercice


Que faire lorsque l'on rencontre des conflits ?

- ☐ Laisser Git corriger les conflits et faire un commit
Si on rencontre des conflits, c'est que Git n'a pas pu les corriger lui-même.
- ☐ Corriger soi-même les conflits et faire un commit
Il ne faut pas oublier d'ajouter les fichiers corrigés.
- ☐ Laisser Git corriger les conflits, ajouter les fichiers corrigés et faire un commit
Si on rencontre des conflits, c'est que Git n'a pas pu les corriger lui-même.

- ☒ Corriger soi-même les conflits, ajouter les fichiers corrigés et faire un commit

Exercice

Qu'est-ce qu'un commit de merge ?

- ☐ Une référence vers le commit que l'on a gardé
- ☒ Un commit contenant la résolution des conflits
- ☐ Un commit indiquant qu'il n'y a pas eu de conflit
-  Un commit de merge n'apparaît que dans le cas de conflits qui ont été résolus après un merge. Ce commit contient la résolution des conflits.

Exercice

Comment fait-on pour définir l'upstream d'une branche en même temps qu'on la pousse ?

- ☐ `git push --upstream origin bugfix`
- ☒ `git push --set-upstream origin bugfix`
- ☐ `git push -set-upstream origin bugfix`
- ☒ `git push -u origin bugfix`

Exercice

Quelle commande permet de récupérer les branches distantes ?

`git fetch`

p. 16 Solution n°6

```

1 # 2
2 git add index.html
3 git commit -m"Add fruit list"
4
5 # 3
6 git remote add origin git@github.com:<username>/defi-git.git
7 git push -u origin master
8
9 # 4
10 git checkout -b fix-typos
11 sed -i -e 's/Pome/Pomme/g; s/Carote/Carotte/g; s/Tommate/Tomate/g; s/Cerisse/Cerise/g'
    index.html
12
13 # 5
14 git add index.html
15 git commit -m"Fix typos in fruit list"
16 git push -u origin fix-typos
17
18 # 6
19 git checkout -b remove-vegetables master
20 sed -i '/Carote/d; /Salade/d; /Tomate/d' index.html
21
22 # 7
23 git add index.html
24 git commit -m"Remove vegetables from fruit list"
25 git push -u origin remove-vegetables

```

```

26
27 # 8
28 git checkout master
29 git merge fix-typos
30 git push
31
32 # 9
33 git merge remove-vegetables
34
35 # 10
36 # Corrigez les conflits (cf fichier plus bas)
37 git add index.html
38 git commit
39 # Validez le commit de merge
40 git push
41
42 # 11
43 git push origin --delete fix-typos && git branch -D fix-typos
44 git push origin --delete remove-vegetables && git branch -D remove-vegetables

```

Vous devriez terminer avec 4 commits et une branche locale ainsi qu'une branche distante :

```

1 $ git log --oneline
2 c914a36 (HEAD -> master, origin/master) Merge branch 'remove-vegetables'
3 43eb496 Remove vegetables from fruit list
4 032943a Fix typos in fruit list
5 38c904e Add fruit list
6
7 $ git branch -a
8 * master
9  remotes/origin/master

```

Fichier **index.html** avec conflits résolus :

```

1 <!DOCTYPE html>
2 <html lang="fr">
3   <head>
4     <meta charset="UTF-8">
5     <meta name="viewport"
6       content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-
7 scale=1.0, minimum-scale=1.0">
8     <meta http-equiv="X-UA-Compatible" content="ie=edge">
9     <title>Document</title>
10  </head>
11  <body>
12    <h1>Liste des fruits :</h1>
13    <ul>
14      <li>Pomme</li>
15      <li>Poire</li>
16      <li>Cerise</li>
17    </ul>
18  </body>
19 </html>

```