

Les bonnes pratiques

Table des matières

I. Contexte	3
II. Les commits	3
III. Exercice : Appliquez la notion	5
IV. L'historique	6
V. Exercice : Appliquez la notion	8
VI. Les branches	9
VII. Exercice : Appliquez la notion	10
VIII. Choisir un workflow	10
IX. Exercice : Appliquez la notion	13
X. Auto-évaluation	13
A. Exercice final	13
B. Exercice : Défi	15
Solutions des exercices	17

I. Contexte

Durée : 45 min

Environnement de travail : GitBash / Terminal

Pré-requis : Connaître les bases de Git et la gestion des branches

Contexte

Lorsque l'on utilise Git pour un projet, il est important que chacun des acteurs l'utilise selon la même méthodologie. En effet, Git peut être utilisé de plusieurs manières possibles : il s'agit d'un outil permissif.

Ainsi, il est important d'adopter de bonnes pratiques et de les faire connaître. Dans le cas contraire, on risque de se retrouver rapidement avec un historique illisible et des problèmes au niveau des fusions.

Contrairement à certains langages informatiques ou logiciels, Git ne propose pas de bonnes pratiques quant à l'utilisation de ses commandes. Elles sont donc le fruit de l'expérience, et leurs sources peuvent être multiples. Cependant, nous allons voir qu'un certain nombre de consensus émergent dans plusieurs domaines, tels que sur les commits, les branches ou l'historique.

II. Les commits

Objectif

- Aborder les bonnes pratiques sur la gestion des commits

Attention

Pour ne plus avoir à rename la branche principale de votre projet de "master" à "main" :

Sur votre git en local vous pouvez exécuter la commande suivante : `git config --global init.defaultBranch main`.

Normalement sur GitHub si vous créez un nouveau dépôt il sera automatiquement avec main en tant que branche principale.

Mise en situation

Les éléments principaux de Git, ceux que nous manipulerons au quotidien, sont les commits. Il est donc important de suivre des bonnes pratiques en terme de nommage et de contenu. Nous allons en voir plusieurs dans cette partie.

Fondamental

Commiter souvent, perfectionner plus tard, publier une fois

Git ne prend connaissance des modifications apportées aux fichiers que lorsqu'ils sont commités. Il est donc fortement conseillé de faire un commit après chaque phase de développement d'une fonctionnalité. Il ne faut pas nécessairement chercher à faire le commit parfait dès le premier jet !

Complément Qu'appelle-ton une phase ?

Les phases correspondent aux différentes étapes du développement d'une fonctionnalité.

Par exemple, si nous développons une nouvelle page de notre site web, les différentes phases seront :

- Écriture du code HTML de la page,
- Écriture du code CSS de la page,
- Vérification et correction du code HTML et CSS.

Exemple Pouvoir revenir en arrière

L'utilité principale des commits réguliers est de pouvoir revenir en arrière sur un travail dans un état semi-stable.

Imaginons que, pour une raison qui nous échappe, notre code ne veuille plus fonctionner et que nous n'arrivions pas à revenir manuellement à un état applicatif stable. Nous pourrions avoir envie de revenir au **dernier point de sauvegarde**.

Si le travail a été sauvegardé régulièrement dans un état stable, il est possible de revenir en arrière simplement sur les dernières actions effectuées. Si nous n'en avons pas fait, nous serons obligés de tout refaire.

L'inconvénient de cette pratique, c'est que nous risquons de nous retrouver avec un historique de commits conséquent pour une seule fonctionnalité. Cependant, si nous le souhaitons, nous pourrions toujours réécrire notre historique pour qu'il reflète ce que nous désirons. Une fois notre historique de commits propre, nous pourrions pousser notre branche.

Fondamental Écrire des messages de commit utiles

Les messages de commits sont très importants. Ils vont permettre de décrire ce que nous avons voulu faire dans le cadre d'une modification et pour quelle raison, sans que nous ayons à lire le code.

Il faut aussi prévoir d'adopter des conventions de nommage (par exemple, les écrire en anglais, pas plus de 72 caractères, message à l'infinif, référence aux tickets liés si un outil de suivi est utilisé...).

Pour que les messages soient les plus pertinents possibles, il faudra probablement réécrire notre historique avant de publier celui-ci.

Exemple Les messages à bannir

En partant de là, il faut donc absolument bannir les messages tels que **Add new feature, Fix problem, Hotfix**, etc. Préférez plutôt des messages explicites :

- *Add user registration page*
- *Fix database error when deleting a product. Fix #1*
- *Disable user subscription when user deletes his account. Close #2*

Fondamental Ne pas commiter des fichiers générés

Les fichiers qui peuvent être régénérés à volonté, tels que le cache, les logs ou les dépendances du projet, doivent être ignorés grâce à un fichier `.gitignore`.

Les fichiers à commiter sont les fichiers qui nous ont demandé un effort de développement ou d'adaptation, par exemple :

- un fichier créé de zéro,
- un fichier généré puis modifié pour convenir à notre besoin.

Syntaxe **À retenir**

Parmi les bonnes pratiques concernant les commits, trois grandes sont à retenir :

- Commiter souvent, perfectionner plus tard, publier une fois
- Écrire des messages de commit utiles
- Ne pas commiter des fichiers générés

Complément

<https://gist.github.com/pandeiro/1552496>

<https://deepsources.io/blog/git-best-practices/>

<https://www.git-tower.com/learn/git/ebook/en/command-line/appendix/best-practices>

https://seesparkbox.com/foundry/semantic_commit_messages

III. Exercice : Appliquez la notion

Clonez le dépôt suivant dans un nouveau répertoire intitulé `bonnes-pratiques-git` : <https://github.com/lpe-acelys/bonnes-pratiques-git>.

Vous prendrez soin de travailler sur la branche `bp-commits` : `git checkout bp-commits`.

Pour les besoins de ce nouveau projet, vous êtes affecté au ticket suivant : Jira #15 - Mise en place d'un formulaire d'édition de profil pour un utilisateur.

Jira¹ est un outil de gestion de projets un peu plus avancé que certains outils (tels que Trello), régulièrement utilisé en entreprise.

Voici les étapes clés devant être mises en place tout au long du développement :

- Mise en place du formulaire de base HTML
- Traitement/Validation du formulaire en PHP
- Mise à jour de la base de données

Il est à noter que, lors du traitement de ce formulaire, une photo de profil est téléchargée par l'utilisateur. Celles-ci seront stockées dans le répertoire `uploads`.

Les conventions de nommage des commits sur ce projet sont les suivantes :

- Pour un commit, on indique systématiquement s'il s'agit d'une fonctionnalité (`ft`) ou d'un bug (`bf`)
- On indique également systématiquement le numéro du ticket en question (`Jira#12`, par exemple)
- Les commits sont rédigés en français

Voici un exemple : `ft - Jira#0 : Initialisation du dépôt`.

Question 1

[solution n°1 p.19]

En partant de ces informations et à partir des bonnes pratiques évoquées, réalisez les commits nécessaires au traitement des différentes phases du ticket qui vous est affecté, en considérant que, par simplification, **une étape du ticket = une entrée dans le fichier `actions.md`**.

Les photos de profil sont des fichiers `.jpeg` qui ne devront pas être présents sur le dépôt.

Question 2

[solution n°2 p.19]

Les photos de profil sont des fichiers `.jpeg` qui ne devront pas être présents dans le répertoire `uploads` et devront être exclus. Modifiez le fichier **`.gitignore`** en conséquence.

¹ <https://www.atlassian.com/fr/software/jira>

IV. L'historique

Objectif

- Aborder les bonnes pratiques sur l'historique

Mise en situation

L'historique d'une branche, c'est un peu ses mémoires. Pour les écrire, nous avons besoin de revenir plusieurs fois dessus, ajouter des informations, etc. Pour nos mémoires, nous remettrons tout au propre à la fin pour que nos paragraphes suivent une suite chronologique.

Pour l'historique de nos branches, il existe deux approches : représenter l'exacte chronologie des développements, ou refléter l'historique attendu à l'origine.

Sausage making

Derrière l'expression *sausage making*, nous retrouvons toutes les opérations de réalisation d'un développement. Souvent, cela va se caractériser par une succession de commits, pas forcément représentatifs de l'objectif à atteindre. Or, si nous sommes amenés à faire des opérations de *cherry-pick* (fait de prendre un commit d'une branche et de l'apposer sur une autre) ou de *rebase*, cela ne sera pas très pratique.

Dans le cas de cherry-picks, il faudra identifier tous les commits à récupérer. Tandis que, si nous n'avons qu'un commit, nous n'aurons qu'à nous soucier de celui-ci.

Dans le cas des rebase, si nous rencontrons des conflits, nous pourrions avoir à résoudre plusieurs fois les mêmes conflits.

Pour toutes ces raisons, il est conseillé de garder le principe : **un commit = une fonctionnalité**. Les commandes `git rebase -i`, `git add -p` et `git reset -p` pourront contribuer à conserver un historique « propre ».

Exemple Réécrire l'historique

Nous pourrions, par exemple, nous retrouver avec cinq commits pour réaliser une seule fonctionnalité, dont certains pourraient contenir des anomalies :

```
1 123fa39 (HEAD -> bugfix) Refactor CSS
2 e4f01e3 More fixes
3 f6a4123 Fix HTML bug
4 a1f0525 Style user page with CSS
5 220bd43 Add user page HTML
6 c4753e3 (main) Add README.md
```

L'idéal serait d'avoir :

```
1 123fa39 (HEAD -> bugfix) Add user page
2 c4753e3 (main) Add README.md
```

Pour la même raison, si notre historique ressemblait à cela :

```
1 123fa39 (HEAD -> bugfix) Fix bug in user page and fix another bug in product page
2 c4753e3 (main) Add README.md
```

Nous pourrions préférer l'historique suivant :

```
1 123fa39 (HEAD -> bugfix) Fix error when no users are displayed in user page
2 5f6e78a Fix product name in product page
3 c4753e3 (main) Add README.md
```

Attention Ne pas changer un historique publié

Dans Git, la réécriture d'historique est très puissante, mais « un grand pouvoir implique de grandes responsabilités ». Elle vous permettra, par exemple, de « mentir » sur l'historique de vos branches en fusionnant des commits entre eux ou en les divisant.

Mais la réécriture d'historique peut poser d'énormes problèmes lorsqu'il s'agit de collaborer avec d'autres développeurs, ou de fusionner ou de rebase des branches.

Exemple

Par exemple, si nous créons une branche `bugfix` depuis `main` et que nous avons l'historique suivant :

```
1 A---B---C---D ← main
2   \
3     E---F ← bugfix
```

La branche `bugfix` contient les commits `A---B---E---F`. Maintenant, réécrivons l'historique de la branche `main` en modifiant le commit `B`. L'historique sera le suivant :

```
1 A---B'---C---D ← main
2
3 A---B---E---F ← bugfix
```

Si nous essayons de fusionner la branche `bugfix` dans `main` après la réécriture de l'historique de `main`, nous aurons de nombreux conflits.

En effet, Git n'est pas capable de détecter que les commits `B` et `B'` sont en grande partie identiques. Pour lui, les commits sont totalement différents. Or, ils possèdent les mêmes modifications : il faudra donc résoudre tous les conflits.

Si nous résolvons les conflits correctement et que nous fusionnons la branche `bugfix` dans `main`, nous aurons l'historique suivant :

```
1 A---B'---C---D---B---E---F ← main
```

Notre commit `B` sera donc un doublon de notre `B'`. Si nous avons rebasé notre branche `bugfix` à partir de `main`, le problème aurait été le même, produisant l'historique suivant :

```
1 A---B'---C---D ← main
2   \
3     B---E---F ← bugfix
```

Une fois qu'une branche est publiée, on ne modifie plus son historique. Cela pourrait faire diverger des branches entre elles, ou bien par rapport à leur *upstream*.

Complément

Dans la pratique, si nous sommes seuls à travailler sur ladite branche et qu'elle n'est la base d'aucune autre branche, il ne devrait pas y avoir de problèmes pour réécrire son historique.

Syntaxe À retenir

- Au cours de nos développements, nous aurons à faire un grand nombre de commits. Il est conseillé qu'un commit corresponde à une fonctionnalité.
- Si nous ne respectons pas cette règle, l'historique peut devenir difficile à lire et les commits complexes à manipuler.
- La réécriture d'historique peut venir à notre secours : cependant, il faut garder à l'esprit qu'elle peut faire diverger des branches entre elles ou par rapport à leur *upstream*.
- Il faut donc éviter de réécrire l'historique des branches publiées.

Complément

<https://gist.github.com/pandeiro/1552496>
<https://deepsources.io/blog/git-best-practices/>
<https://raygun.com/blog/git-workflow/>

V. Exercice : Appliquez la notion

Si cela n'a pas été fait lors de l'exercice précédent, clonez le dépôt suivant dans un nouveau répertoire intitulé `bonnes-pratiques-git` : <https://github.com/lpe-acelys/bonnes-pratiques-git>.

Si vous avez rencontré des difficultés lors de la réalisation de l'exercice précédent, vous pouvez exécuter la commande `git reset --hard origin/branche_en_cours` afin de remettre le dépôt dans son état initial.

Pour cet exercice, vous prendrez soin de travailler sur la branche `bp-historique` : `git checkout bp-historique`.

Pour les besoins de l'exercice, on considérera que cette branche n'a pas été publiée et qu'il s'agit de votre branche de travail locale.

Vous avez précédemment travaillé sur le ticket suivant : Jira #15 - Mise en place d'un formulaire d'édition de profil pour un utilisateur.

Plusieurs commits ont été réalisés au cours des différentes phases de travail. À ce stade, vous considérez que votre travail est terminé.

```
1 $ git log
2 commit 3237294c69bc7f414edbf948fd4a3b75cd894e95 (HEAD -> bp-commits)
3 Author: User
4 Date: Wed May 13 10:16:34 2020 +0200
5 ft - Jira#15: Mise à jour de la base de données
6
7 commit 7eb9ac5e58176c6d7a1785640ea23334ad111a2e
8 Author: User
9 Date: Wed May 13 10:16:03 2020 +0200
10
11 ft - Jira#15: Traitement du formulaire en PHP
12
13 commit b7fc2237e9fb068b6a3887531ea3460362dc62c5
14 Author: User
15 Date: Wed May 13 10:15:12 2020 +0200
16
17 ft - Jira#15: Mise en place du formulaire HTML
18
19 commit 288b182347739311ebe525733aa7cbcf67b44ce3 (origin/main, main)
20 Author: User
21 Date: Wed May 13 10:09:34 2020 +0200
22
23 ft - Jira#0: Initialisation du dépôt
```

Pour rappel, les conventions de nommage des commits sur ce projet sont les suivantes :

- Pour un commit, on indique systématiquement s'il s'agit d'une fonctionnalité (ft) ou d'un bug (bf)
- On indique également systématiquement le numéro du ticket en question (Jira#12 par exemple)
- Les commits sont rédigés en français

Voici un exemple : `ft - Jira#0 : Initialisation du dépôt`.

Question

[solution n°3 p.19]

Avant de publier votre branche, factorisez l'ensemble des commits de travail effectués en un seul commit, que vous renommerez selon les conventions du projet.

Indice :

Voici la commande Git que vous pouvez utiliser : `git rebase -i HEAD~3`.

Indice :

On va devoir conserver (et modifier) le premier commit, tandis que les deux suivants seront squashed.

```
1 e b7fc223 ft - Jira#15: Mise en place du formulaire HTML
2 s 7eb9ac5 ft - Jira#15: Traitement du formulaire en PHP
3 s 3237294 ft - Jira#15: Mise à jour de la base de données
4
```

VI. Les branches

Objectif

- Aborder les bonnes pratiques sur la gestion des branches

Mise en situation

Git propose un système de branches très efficace qui nous permet de développer nos fonctionnalités et de corriger nos anomalies dans des espaces dédiés. Nous allons voir plusieurs bonnes pratiques quant à leur utilisation.

Méthode Ne pas hésiter à utiliser les branches

Si les branches sont si faciles à gérer avec Git, ce n'est pas par hasard. Une branche est un espace dédié idéal pour tous nos développements, qu'il s'agisse d'une nouvelle fonctionnalité, d'une correction d'anomalie ou simplement d'une idée à expérimenter.

Il est important de les utiliser et de ne pas hésiter à le faire. Ainsi, nous pourrions développer, créer un commit, changer de branche et revenir plus tard sur notre branche pour continuer notre travail.

Complément Adopter une nomenclature

Il est aussi conseillé d'adopter une nomenclature pour le nommage de nos branches. En effet, il peut facilement devenir difficile de retrouver la branche d'un développement parmi toutes les branches que nous avons déjà créées.

Il faut donc éviter de les appeler `test` ou `bugfix`, mieux vaut être plus précis. Par exemple, si nous utilisons un système de suivi de ticket, nous pourrions utiliser l'identifiant de celui en rapport avec notre développement dans le nom de la branche.

A minima, il faut essayer de décrire ce que l'on essaie de faire afin de retrouver la branche facilement.

Exemple Exemple de nomenclature

Notre nomenclature pourrait être `<ticket>-<descriptif du ticket>`, ce qui nous donnerait des noms de branche tels que `13-user-page` ou `24-fix-product-creation`.

Complément Supprimer les branches inutiles

Parcourir la liste des branches n'est jamais une sinécure lorsqu'il existe des dizaines, voire des centaines de branches.

Il est donc conseillé de garder sa liste de branches en les supprimant au fur et à mesure qu'elles sont fusionnées ou devenues inutiles, qu'il s'agisse de branches locales ou de branches distantes.

Les commandes `git branch -d` et `git push origin --delete` sont nos alliées dans cette tâche.

Syntaxe À retenir

- Le système de branches fait la force principale de Git et les avantages de leur utilisation sont nombreux, il faut donc les utiliser autant que possible.
- Il est conseillé d'adopter une nomenclature pour ses branches et de supprimer les inutiles, afin de pouvoir retrouver une branche plus facilement.

Complément

<https://www.git-tower.com/learn/git/ebook/en/command-line/appendix/best-practices>

<https://raygun.com/blog/git-workflow/>

VII. Exercice : Appliquez la notion

Pour l'un des projets sur lesquels vous travaillez, la convention de nommage de branches Git est la suivante :

- S'il s'agit d'une fonctionnalité, la branche est préfixée par `ft/`, pour un correctif `bf/`, pour de la documentation `doc/`.
- Une tâche est systématiquement rattachée à un ticket Jira, ainsi on ajoute toujours le numéro du ticket concerné à la suite.
- On nomme ensuite explicitement la branche en `snake_case` et en français.

Voici un exemple pour le ticket #1, où il était demandé de mettre en place la structure du projet : `ft/1_initialisation_structure`.

Question 1

[solution n°4 p.20]

Vous êtes chargé de traiter les tickets suivants :

- #5 - Fonctionnalité : mise en place d'un formulaire de contact
- #6 - Correctif : Connexion LDAP impossible
- #7 - Documentation : Créer un fichier README

Comment devriez-vous nommer les branches afin de procéder au traitement de ces tickets ?

Question 2

[solution n°5 p.20]

Grâce à quelle commande auriez-vous pu créer ces branches ?

VIII. Choisir un workflow

Objectifs

- Comprendre l'utilité d'un workflow Git
- Décrire le fonctionnement du Git Flow

Mise en situation

Très rapidement, lorsque nous sommes amenés à travailler collaborativement avec des fusions de branches et des rebases, nous nous rendons compte qu'il est nécessaire d'adopter une méthode de travail avec Git, aussi appelé **workflow**.

Il en existe plusieurs, et nous verrons plus en détails le fonctionnement du workflow **GitHub Flow**.

Définition Qu'est-ce qu'un workflow Git ?

Un workflow Git met en place de bonnes pratiques en termes de nommage des commits, des branches, de leur fusion, et de quand les pousser.

À vrai dire, nous avons déjà utilisé un workflow, puisque nous avons déjà eu à interagir avec un dépôt distant :

1. Création d'une branche
2. Développement et ajout de commits
3. Validation de nos développements
4. Envoi de nos modifications au dépôt distant

Ce workflow nous est propre, mais lorsque nous intégrerons une équipe de développeurs, nous aurons probablement à nous conformer à un autre.

Exemple Fonctionnement du GitHub Flow

Nous allons voir plus en détails le fonctionnement d'un workflow Git grâce au **GitHub Flow**.

Il a l'avantage d'être très simple, de s'intégrer parfaitement avec les outils de GitHub, et de s'adapter particulièrement bien dans le cadre du déploiement continu, comme dans le cas du développement d'applications web.

Méthode Création de branches

Toutes les branches doivent partir de la branche `main`.

La branche `main` doit toujours être déployable, donc contenir une version stable du projet.

Les noms des branches doivent décrire ce qu'elles contiennent.

Méthode Création de commits

Les commits sont faits sur les branches. Ils doivent avoir un message clair, reflétant la modification effectuée dans le commit.

Méthode Ouverture d'une pull request

Une **pull request** est un mécanisme permettant de proposer, via l'interface de GitHub, une différence entre une branche et une autre. Elle permet de visualiser le travail effectué et d'obtenir de l'aide ou une simple revue de code en vue d'une fusion. Nous pouvons retrouver le terme de **merge request** avec d'autres forges logicielles (GitLab, par exemple).

Le système de **@mention** peut être utilisé pour interpeller une équipe ou un autre utilisateur.

Méthode Discussion et relecture de code

Dans une pull request, il est possible de commenter le code produit ou de poser des questions. Par exemple, nous pourrions indiquer que la pull request ne respecte pas les bonnes pratiques établies sur le projet en termes de normes de codage, de tests.

Si la pull request nous appartient et que nous avons des retours concernant, par exemple, une anomalie présente dans notre code, nous pouvons la corriger et pousser de nouveau les modifications de notre branche. La pull request sera mise à jour automatiquement.

Méthode Déploiement

Une fois notre pull request validée, il est possible de déployer notre branche automatiquement depuis GitHub et de tester nos modifications en production pour valider qu'elles fonctionnent correctement.

Si les modifications ne fonctionnent pas, nous pourrions revenir en arrière en redéployant la branche `main`.

Méthode Merge

Une fois la livraison effectuée et validée, nous pouvons merger notre branche dans la branche `main`. L'historique de la pull request est gardé, permettant de comprendre, plus tard, pourquoi tel ou tel choix a été fait pour ce développement.

Nous pouvons faire référence à une *issue* directement dans le texte de la pull request afin d'effectuer sur celle-ci des opérations automatiques. Par exemple, `Closes #13` fermera l'issue n°13.

Complément D'autres workflows

Ils existent d'autres workflows, mais GitFlow et GitLab Flow sont les plus connus :

- Le `GitFlow` est plus adapté dans le cadre du développement de logiciels pouvant apparaître dans plusieurs versions, comme c'est le cas pour les applications de bureau. Il permet aussi de gérer un système de release qui peut embarquer le contenu de plusieurs branches, et donc plusieurs fonctionnalités.
- Le `GitLab Flow` est une évolution du `GitFlow` pour gérer le passage d'un environnement de déploiement à un autre, par exemple si nous avons un serveur de production à destination des utilisateurs finaux, et un autre de préproduction dont le but est de valider une version de notre application avant de la livrer au grand public.

Syntaxe À retenir

- Un workflow Git permet de décrire une méthode de travail à appliquer par tous les acteurs d'un projet. De cette façon, il est possible de limiter les erreurs en termes de fusion de branches et de tester plus efficacement les applications avant et après leur livraison.
- Plusieurs workflows existent, parmi lesquels on retrouve le `GitHub Flow`, le `Git Flow` et `GitLab Flow`, chacun possédant ses avantages.

ComplémentChoisir un workflow¹GitHub Flow²GitFlow³GitLab Flow⁴**IX. Exercice : Appliquez la notion****Pour réaliser cet exercice, vous devrez disposer d'un compte GitHub.****Question**

[solution n°6 p.20]

- Sur votre compte GitHub, initialisez un nouveau dépôt intitulé "premiere-pr"
- Créez un nouveau dépôt sur votre environnement local et liez-y votre dépôt GitHub de la façon suivante : `git remote add origin url_depot_github.git`
- Créez un nouveau fichier `README.md` et intégrez-y le contenu suivant "Initialisation du fichier"
- Commitez vos modifications et poussez cette branche sur votre dépôt distant : `git push -u origin main`
- Basculez sur une nouvelle branche intitulée `feature/update_readme`
- Modifiez le fichier `README.md` en y ajoutant une nouvelle ligne "Modification du fichier"
- Commitez vos modifications et poussez cette branche sur votre dépôt distant : `git push origin feature/update_readme`
- Dans l'interface de GitHub, créez votre première pull request en comparant `feature/update_readme` à votre branche `main`

X. Auto-évaluation**A. Exercice final****Exercice 1**

[solution n°7 p.21]

Exercice

Lorsque l'on travaille sur une nouvelle fonctionnalité, il est conseillé d'effectuer des commits...

- ☐ Régulièrement, quel que soit l'état de l'application
- ☐ Régulièrement, lorsque l'on se trouve dans un état stable
- ☐ Uniquement si la fonctionnalité est terminée

Exercice

1 <https://gist.github.com/pandeyro/1552496#choose-a-workflow>

2 <https://guides.github.com/introduction/flow/>

3 <https://nvie.com/posts/a-successful-git-branching-model/>

4 https://docs.gitlab.com/ee/topics/gitlab_flow.html

Parmi ces messages de commits, lesquels faudrait-il bannir ?

- ☐ Désactivation des envois de SMS lors d'une nouvelle commande
- ☐ Correctif urgent
- ☐ Correction d'un problème sur l'application
- ☐ Jira #0 - Documentation : Ajout d'informations sur la création de comptes

Exercice

Les fichiers générés automatiquement sont des fichiers...

- ☐ Qu'il faut s'efforcer de commiter
- ☐ Qu'il faut éviter de commiter

Exercice

Dans l'idéal, une fois le développement d'une fonctionnalité terminé, il vaut mieux conserver...

- ☐ Un seul commit
- ☐ Plusieurs commits

Exercice

Il est préférable de modifier l'historique d'une branche...

- ☐ Qui n'a pas encore été publiée
- ☐ Qui a déjà été publiée

Exercice

Grâce à quelle option de la commande `git commit` est-il possible de modifier le message du dernier commit d'une branche ?

- ☐ --edit
- ☐ --modifiy
- ☐ --update
- ☐ --amend

Exercice

Lorsque l'on démarre le développement d'une nouvelle fonctionnalité, il est conseillé...

- ☐ De réaliser ce développement sur une nouvelle branche
- ☐ De réaliser ce développement sur la branche principale du projet

Exercice

Au cours de notre développement, mieux vaut...

- ☐ Conserver toutes ses branches, même si elles ont déjà été fusionnées
- ☐ Supprimer ses branches régulièrement une fois fusionnées

Exercice

Il existe un mécanisme permettant de proposer, via l'interface de GitHub par exemple, une différence entre une branche et une autre en vue d'un merge. Ce mécanisme permet également à nos collègues d'intervenir afin de vérifier que tout est en ordre, par exemple.

On appelle ce mécanisme :

- ☐ Un workflow
- ☐ Le déploiement
- ☐ Une pull request

Exercice

Quelle est la dernière étape du GitHub Flow ?

- ☐ Le merge sur la branche de référence
- ☐ L'envoi du développement au dépôt distant
- ☐ La pull request
- ☐ La création de la branche

B. Exercice : Défi

Aujourd'hui, vous allez vivre la journée typique d'un développeur. Vous travaillez pour le compte d'une école de commerce et vous êtes en charge de l'intranet utilisé par les étudiants.

Clonez ce dépôt : <https://github.com/lpe-acelys/intranet-ecole>.

La branche de référence du projet est la branche `main`.

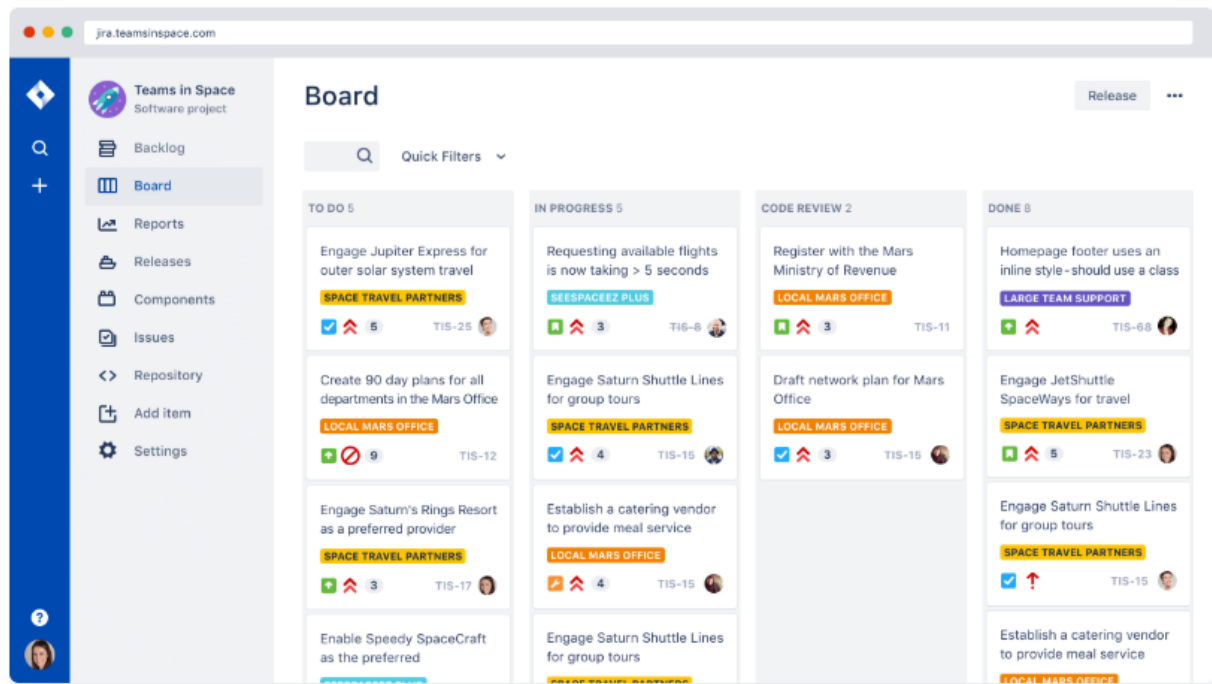
Les conventions de nommage des commits sur ce projet sont les suivantes :

- Pour un commit, on indique systématiquement s'il s'agit d'une fonctionnalité (`ft`) ou d'un bug (`bf`)
- On indique également systématiquement le numéro du ticket en question (`Jira#12` par exemple)
- Les commits sont rédigés en français

Voici un exemple : `ft - Jira#0 : Initialisation du dépôt`.

Pour rappel, Jira¹ est un outil de gestion de projets un peu plus avancé que certains outils tels que Trello, régulièrement utilisé en entreprise. Voici un exemple de ce à quoi peut ressembler le tableau de travail d'une équipe, avec les tâches à réaliser.

¹ <https://www.atlassian.com/fr/software/jira>



Il est 9 h...

Votre journée commence par la prise en charge d'un nouveau ticket Jira.

Il s'agit du ticket #285 - Fonctionnalité : Mise en place d'un agenda pour les étudiants.

Il y aura plusieurs étapes pour réaliser ce ticket :

1. Réalisation de l'interface
2. CRUD de gestion des événements personnels (*Create, Read, Update, Delete*)
3. Synchronisation avec l'agenda de l'école

Il s'agit d'une tâche ambitieuse, mieux vaudra découper les développements en plusieurs étapes.

Question 1

[solution n°8 p.23]

Démarrez cette nouvelle fonctionnalité en partant d'une nouvelle branche, dont le nommage respectera les conventions du projet.

La matinée passe rapidement, mais vous avez réalisé les deux premières étapes :

- Modifiez le fichier `actions.md` pour ajouter une ligne par étape.
- N'oubliez pas d'effectuer les commits en conséquence.

Il est 11 h

Nathalie, du service scolarité, vous appelle paniquée. Les étudiants devaient consulter les résultats des examens, mais les notes n'apparaissent pas correctement.

Vous devez traiter ce problème en urgence et commencez donc par créer un nouveau ticket Jira : # 286 - Correctif : Affichage des notes d'examens.

Question 2

[solution n°9 p.24]

- En partant de la branche `main`, créez une nouvelle branche afin de corriger ce problème.

Il s'agissait d'un problème de connexion à la base de données dû à une erreur de syntaxe.

- Mettez à jour le fichier `actions.md` et commitez les modifications.
- Vous avez jugé que cela ne nécessitait pas une pull request et vous mergez cette branche sur `main`.

Il est 16 h

Félicitations, vous avez sauvé la journée de Nathalie et les étudiants sont ravis ! Votre correctif a pu être déployé en production.

Vous avez repris votre travail depuis quelques heures et avez été d'une efficacité redoutable. Vous considérez votre système d'agenda comme terminé et vous vous préparez à ouvrir votre pull request.

Question 3

[solution n°10 p.25]

- Retournez sur la branche `ft/285-agenda-etudiants`.
- Intégrez la dernière modification manquante "Synchronisation avec l'agenda de l'école" en mettant à jour le fichier `actions.md` et committez ces modifications.
- Votre travail étant terminé, fusionnez ensuite vos commits afin qu'ils ne forment qu'un. N'oubliez pas d'effectuer un renommage en conséquence.

Il est 16 h 30

Votre collègue réputé pour son œil aguerri vous fait remarquer suite à votre pull request que quelques conventions de nommage PHP (PSR) ne sont pas respectées. Vous devez effectuer un dernier correctif et vous pourrez ensuite merger votre branche.

Question 4

[solution n°11 p.26]

- Effectuez un dernier commit sur votre branche `ft/285-agenda-etudiants` afin de corriger le souci PSR évoqué, mettez à jour le fichier `actions.md`.
- Fusionnez ce commit avec le précédent.
- Rebasez votre branche par rapport à `main` en n'oubliant pas de résoudre les conflits.
- Fusionnez votre branche avec `main`.
- Supprimez les branches créées pour les tickets #285 et #286.

Solutions des exercices

p.5 Solution n°1

À la suite de cet exercice, voici à quoi doit ressembler votre historique.

Vous avez dû effectuer 3 commits, correspondant aux différentes phases du ticket :

```

1 # Modification du fichier actions.md
2 git add actions.md
3 $ git commit -m "ft - Jira#15: Mise en place du formulaire HTML"
4
5 # Modification du fichier actions.md
6 git add actions.md
7 $ git commit -m "ft - Jira#15: Traitement du formulaire en PHP"
8
9 # Modification du fichier actions.md
10 git add actions.md
11 $ git commit -m "ft - Jira#15: Mise à jour de la base de données"
12
13 $ git log
1 commit 3237294c69bc7f414edbf948fd4a3b75cd894e95 (HEAD -> bp-commits)
2 Author: User
3 Date:   Wed May 13 10:16:34 2020 +0200
4
5     ft - Jira#15: Mise à jour de la base de données
6
7 commit 7eb9ac5e58176c6d7a1785640ea23334ad111a2e
8 Author: User
9 Date:   Wed May 13 10:16:03 2020 +0200
10
11     ft - Jira#15: Traitement du formulaire en PHP
12
13 commit b7fc2237e9fb068b6a3887531ea3460362dc62c5
14 Author: User
15 Date:   Wed May 13 10:15:12 2020 +0200
16
17     ft - Jira#15: Mise en place du formulaire HTML
18
19 commit 288b182347739311ebe525733aa7cbcf67b44ce3 (origin/main, main)
20 Author: User
21 Date:   Wed May 13 10:09:34 2020 +0200
22
23     ft - Jira#0: Initialisation du dépôt

```

p.5 Solution n°2

Vous avez dû ajouter les fichiers `.jpeg` du répertoire `uploads` à vos fichiers ignorés, comme l'étaient déjà les fichiers `pdf`.

```

1 uploads/*.pdf
2 uploads/*.jpeg

```

p.9 Solution n°3

```

1 $ git rebase -i HEAD~3
1 e b7fc223 ft - Jira#15: Mise en place du formulaire HTML
2 s 7eb9ac5 ft - Jira#15: Traitement du formulaire en PHP
3 s 3237294 ft - Jira#15: Mise à jour de la base de données

1 $ git commit --amend -m "ft - Jira#15: Mise en place formulaire d'édition de profil
  utilisateur"
2 $ git log

1 commit c999bd80c99bbb7b7179c01b46b597cc960f459d (HEAD)
2 Author: User
3 Date:   Wed May 13 10:15:12 2020 +0200
4
5     ft - Jira#15: Mise en place formulaire d'édition de profil utilisateur
6
7 commit 288b182347739311ebe525733aa7cbcf67b44ce3 (origin/main, origin/bp-commits, main, bp-
  commits)
8 Author: User
9 Date:   Wed May 13 10:09:34 2020 +0200
10
11     ft - Jira#0: Initialisation du dépôt
12

```

p. 10 Solution n°4

Voici comment nommer ces branches :

- ft/5_formulaire_contact
- bf/6_connexion_ldap
- doc/7_creation_readme

p. 10 Solution n°5

`git checkout -b nom_de_la_branche`

p. 13 Solution n°6

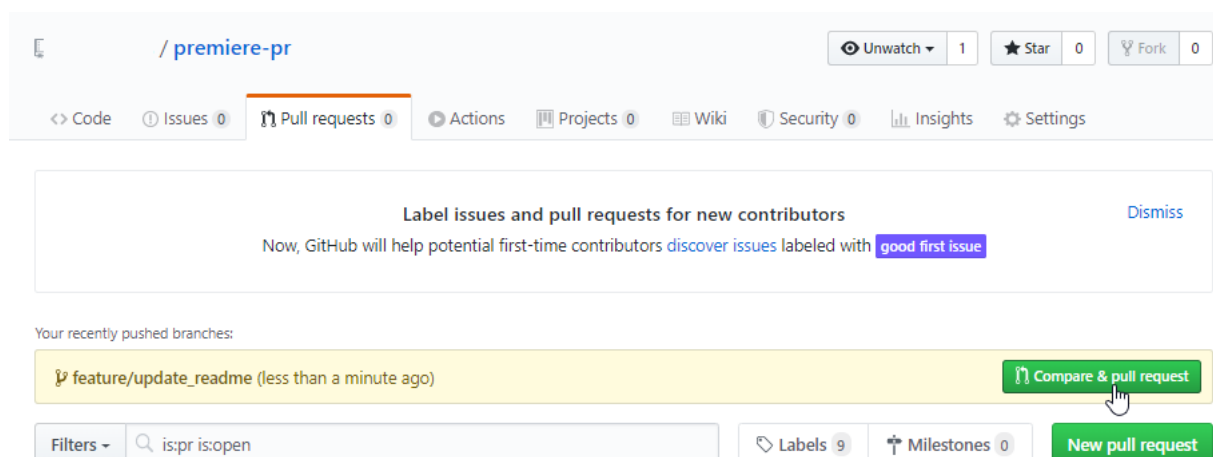
Voici les commandes que vous avez dû effectuer :

```

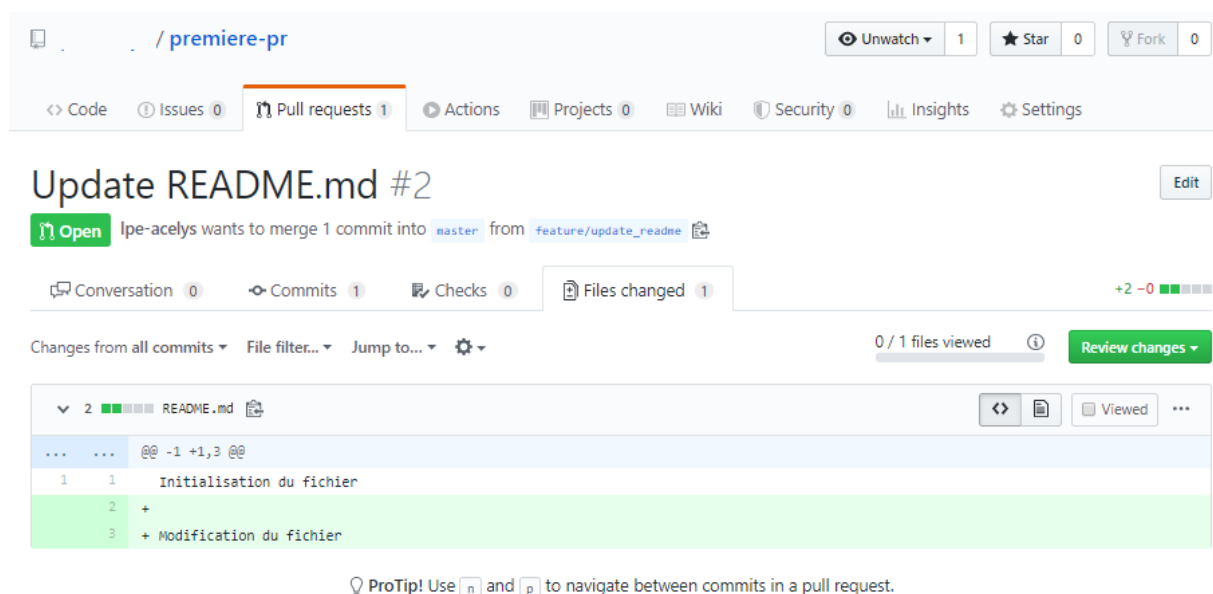
1 mkdir premiere-pr
2 cd premiere-pr
3 git init
4 git remote add origin ...
5 touch README.md
6 echo "Initialisation du fichier" > README.md
7 git commit -am "Initialisation du fichier README"
8 git push -u origin main
9 git checkout -b feature/update_readme
10 echo "Modification du fichier" > README.md
11 git commit -am "Modification du fichier README"
12 git push origin feature/update_readme

```

Une fois votre dépôt mis à jour, vous devriez avoir constaté ceci :




Et une fois votre PR créée :



Exercice p. 13 Solution n°7

Exercice

Lorsque l'on travaille sur une nouvelle fonctionnalité, il est conseillé d'effectuer des commits...

- ☐ Régulièrement, quel que soit l'état de l'application
- ☒ Régulièrement, lorsque l'on se trouve dans un état stable
- ☐ Uniquement si la fonctionnalité est terminée
-  L'utilité principale des commits réguliers est de pouvoir revenir en arrière sur un travail dans un état semi-stable.


Exercice

Parmi ces messages de commits, lesquels faudrait-il bannir ?

- ☐ Désactivation des envois de SMS lors d'une nouvelle commande
Si on lit ce message, on sait ce qui a été fait sans avoir besoin de regarder le détail du commit.
- ☒ Correctif urgent
Correctif de quoi ? Dans quel but ?
- ☒ Correction d'un problème sur l'application
Correctif de quoi ? Dans quel but ?
- ☐ Jira #0 - Documentation : Ajout d'informations sur la création de comptes
Si on lit ce message, on sait ce qui a été fait sans avoir besoin de regarder le détail du commit.


Exercice

Les fichiers générés automatiquement sont des fichiers...

- ☐ Qu'il faut s'efforcer de commiter
- ☒ Qu'il faut éviter de commiter
-  Les fichiers qui peuvent être régénérés à volonté, doivent idéalement être ignorés.
Les fichiers à commiter sont les fichiers qui nous ont demandé un effort de développement ou d'adaptation.


Exercice

Dans l'idéal, une fois le développement d'une fonctionnalité terminé, il vaut mieux conserver...

- ☒ Un seul commit
- ☐ Plusieurs commits
-  Au cours de nos développements, nous aurons à faire un grand nombre de commits. Il est conseillé qu'un commit corresponde à une fonctionnalité.
Si nous ne respectons pas cette règle, l'historique peut devenir difficile à lire, et les commits complexes à manipuler.

Exercice

Il est préférable de modifier l'historique d'une branche...

- ☒ Qui n'a pas encore été publiée
- ☐ Qui a déjà été publiée
-  La réécriture d'historique peut poser d'énormes problèmes lorsqu'il s'agit de collaborer avec d'autres développeurs ou de fusionner ou rebase des branches. Il vaut donc mieux éviter de modifier l'historique d'une branche publiée.


Exercice

Grâce à quelle option de la commande `git commit` est-il possible de modifier le message du dernier commit d'une branche ?

- ☐ --edit
- ☐ --modify
- ☐ --update
- ☒ --amend


Exercice

Lorsque l'on démarre le développement d'une nouvelle fonctionnalité, il est conseillé...

- ☒ De réaliser ce développement sur une nouvelle branche
- ☐ De réaliser ce développement sur la branche principale du projet
-  Une branche est un espace dédié idéal pour tous nos développements, qu'il s'agisse d'une nouvelle fonctionnalité, d'une correction d'anomalie ou simplement d'une idée à expérimenter. Mieux vaut réaliser ce développement sur une branche à part afin de ne pas altérer le statut de notre application sur la branche de référence, tant qu'il n'est pas terminé.

Exercice

Au cours de notre développement, mieux vaut...

- ☐ Conserver toutes ses branches, même si elles ont déjà été fusionnées
- ☒ Supprimer ses branches régulièrement une fois fusionnées
-  Il est conseillé de garder sa liste de branches en les supprimant au fur et à mesure qu'elles sont fusionnées ou devenues inutiles, qu'il s'agisse de branches locales ou de branches distantes, afin de s'y retrouver plus simplement lorsque l'on parcourt nos branches.

Exercice

Il existe un mécanisme permettant de proposer, via l'interface de GitHub par exemple, une différence entre une branche et une autre en vue d'un merge. Ce mécanisme permet également à nos collègues d'intervenir afin de vérifier que tout est en ordre, par exemple.

On appelle ce mécanisme :

- ☐ Un workflow
Le workflow représente l'ensemble de la méthode de travail adoptée.
- ☐ Le déploiement
- ☒ Une pull request

Exercice

Quelle est la dernière étape du GitHub Flow ?

- ☒ Le merge sur la branche de référence
- ☐ L'envoi du développement au dépôt distant
- ☐ La pull request
- ☐ La création de la branche

En étant situé sur la branche `main`, vous avez dû créer une branche intitulée `ft/285-agenda-etudiants` ou équivalent, grâce à `git checkout -b ft/285-agenda-etudiants` par exemple.

Voici les commits que vous devez constater à ce stade :

```

1 # Modification du fichier actions.md
2 git add actions.md
3 $ git commit -m "ft - Jira#285: Réalisation interface"
4
5 # Modification et ajout du fichier actions.md
6 $ git commit -m "ft - Jira#285: Mise en place CRUD événements"
7
8 $ git log
9 commit c0ef2d2237c0a7a7196dd00aea70d525f0b0e92e (HEAD -> correction-q1-ft/285-agenda-
   etudiants)
10 Author: User
11 Date:   Wed May 13 13:26:07 2020 +0200
12
13     ft - Jira#285: Mise en place CRUD événements
14
15 commit fd07280a0fdf7631ac8ae579648baaf940e356cc
16 Author: User
17 Date:   Wed May 13 13:25:30 2020 +0200
18
19     ft - Jira#285: Réalisation interface
20
21 commit 68bec06d4b2e9138c2ecbbd1f542820ed3164245 (origin/main, main)
22 Author: User
23 Date:   Wed May 13 13:22:18 2020 +0200
24
25     ft - Jira#1 : Ajout d'un exemple d'action
26
27 commit a1b7b492786a022bdf92ff38d1e86d55def032c6
28 Author: User
29 Date:   Wed May 13 13:18:48 2020 +0200
30
31     ft - Jira#0 : Initialisation du dépôt
32

```

Voici le contenu du fichier `actions.md`:

```

1 Exemple d'action
2
3 Réalisation de l'interface
4
5 CRUD de gestion des événements personnels
6

```

Si vous avez rencontré un problème au cours de cette étape, procédez de la façon suivante afin de pouvoir continuer l'exercice :

- Placez-vous sur la branche que vous avez créée, ou créez-la au besoin
- Effectuez la commande suivante : `git reset --hard origin/correction-q1-ft/285-agenda-etudiants`

Vous avez dû retourner sur la branche `main` et créer une branche à partir de là intitulée, par exemple, `286-affichage-notes-examens` : `git checkout main && git checkout -b bf/286-affichage-notes-examens`.

Le fichier contenu du fichier `actions.md` à ce stade est le suivant :

```
1 Exemple d'action
2
3 Correctif de l'affichage des notes d'examens
4
1 # Modification du fichier actions.md
2 git add actions.md
3 $ git commit -m "bf - Jira#286 - Affichage des notes, correctif connexion BDD"
4
```

Un `git log` sur la branche `bf/286-affichage-notes-examens` doit indiquer cela :

```
1 $ git log
2 commit 75af52815912a8881729a00f3c15ed0d1c70d33d
3 Author: User
4 Date:   Wed May 13 13:41:33 2020 +0200
5
6     bf - Jira#286 - Affichage des notes, correctif connexion BDD
7
8 commit 68bec06d4b2e9138c2ecbbd1f542820ed3164245 (origin/main)
9 Author: User
10 Date:   Wed May 13 13:22:18 2020 +0200
11
12     ft - Jira#1 : Ajout d'un exemple d'action
13
14 commit a1b7b492786a022bdf92ff38d1e86d55def032c6
15 Author: User
16 Date:   Wed May 13 13:18:48 2020 +0200
17
18     ft - Jira#0 : Initialisation du dépôt
19
```

Ensuite, les commandes effectuées ont dû être les suivantes : `git checkout main` suivie de `git merge bf/286-affichage-notes-examens`.

Si vous avez rencontré des problèmes au cours de cette question :

- Placez-vous sur la branche `main` et effectuez la commande `git reset --hard origin/correction-q2-main`
- Placez-vous sur la branche `bf/286-affichage-notes-examens` (ou équivalent) et effectuez la commande `git reset --hard origin/correction-q2-bf/286-affichage-notes-examens`

p. 17 Solution n°10

Félicitations, votre travail est prêt à être validé par vos collègues !

Voici les commandes que vous avez dû effectuer :

1. `git checkout ft/285-agenda-etudiants`
2. `git add actions.md`
3. `git commit -m "ft - Jira#15: Synchronisation agenda ecole"`
4. `git rebase -i HEAD~3`
5. `git commit --amend -m "ft - Jira#285: Mise en place agenda étudiants"`

6. `git rebase --continue`

```
1 e fd07280 ft - Jira#285: Réalisation interface # on va renommer ce commit
2 f c0ef2d2 ft - Jira#285: Mise en place CRUD événements # on va fusionner ces 2 commits avec
  le premier mais on ne souhaite pas conserver les messages
3 f f7c3371 ft - Jira#285: Synchronisation agenda ecole
4
```

Voici votre nouvel historique, la branche est prête à être soumise à une PR :

```
1 $ git log
2 commit 97cb0c8dc77772a8829a291de1feaedcc83b7913 (HEAD)
3 Author: User
4 Date:   Wed May 13 13:25:30 2020 +0200
5
6     ft - Jira#285: Mise en place agenda étudiants
7
8 commit 68bec06d4b2e9138c2ecbbd1f542820ed3164245 (origin/main)
9 Author: User
10 Date:   Wed May 13 13:22:18 2020 +0200
11
12     ft - Jira#1 : Ajout d'un exemple d'action
13
14 commit alb7b492786a022bdf92ff38d1e86d55def032c6
15 Author: User
16 Date:   Wed May 13 13:18:48 2020 +0200
17
18     ft - Jira#0 : Initialisation du dépôt
19
```

Si vous avez rencontré un problème au cours de cette étape, procédez de la façon suivante afin de pouvoir continuer l'exercice :

- Placez-vous sur la branche `ft/285-agenda-etudiants`
- Effectuez la commande suivante : `git reset --hard origin/correction-q3-ft/285-agenda-etudiants`

p. 17 Solution n°11

- On modifie le fichier d'actions, on l'ajoute puis on commit : `git add actions.md` `git commit -m "ft - Jira#285 - Correction PSR"`.
- `git rebase -i HEAD~2`.

```
1 pick 7a056bb ft - Jira#285: Mise en place agenda étudiants
2 f edabd39 ft - Jira#285 - Correction PSR
```

- `git rebase main` : on constate des conflits dans le fichier `actions.md` qu'il convient de résoudre.

Voici l'historique dont on dispose une fois le rebasage terminé :

```
1 $ git log
2 commit 3d22704a2e90ee9b443c6c99ebe1b2affd7f4b6c
3 Author: User
4 Date:   Wed May 13 13:25:30 2020 +0200
5
6     ft - Jira#285: Mise en place agenda étudiants
7
8 commit 75af52815912a8881729a00f3c15ed0d1c70d33d
9 Author: User
10 Date:   Wed May 13 13:41:33 2020 +0200
11
```

```
12    bf - Jira#286 - Affichage des notes, correctif connexion BDD
13
14 commit 68bec06d4b2e9138c2ecbbd1f542820ed3164245 (origin/main)
15 Author: User
16 Date:   Wed May 13 13:22:18 2020 +0200
17
18    ft - Jira#1 : Ajout d'un exemple d'action
19
20 commit a1b7b492786a022bdf92ff38d1e86d55def032c6
21 Author: User
22 Date:   Wed May 13 13:18:48 2020 +0200
23
24    ft - Jira#0 : Initialisation du dépôt
25
```

- On peut alors fusionner notre branche : `git checkout main` suivie de `git merge ft/285-agenda-etudiants`.
- Et supprimer les branches dont on n'a plus besoin : `git branch -d ft/285-agenda-etudiants` et `git branch -d bf/286-affichage-notes-examens`.