Les bonnes pratiques

Table des matières

I. Contexte

Durée : 1 heure.

Environnement de travail : PC ou Mac avec 8 Go de RAM minimum.

Prérequis : notions de Git et GitHub.

Contexte

Git et GitHub sont deux outils très populaires de gestion de versions utilisés par les développeurs du monde entier pour collaborer sur des projets de logiciels. Bien que Git soit un outil de gestion de versions décentralisé permettant à plusieurs développeurs de travailler simultanément sur un même code, GitHub est une plateforme de collaboration basée sur Git qui fournit des fonctionnalités supplémentaires telles que l'hébergement de code, le suivi des problèmes, la gestion de projets et la collaboration sociale. Il est important de noter que GitLab, AWS CodeCommit et Bitbucket sont d'autres exemples de plateformes de collaboration de développement basées sur Git offrant des fonctionnalités similaires à celles de GitHub.

Dans ce cours, nous avons mis en avant quelques bonnes pratiques et erreurs à éviter lors de l'utilisation de Git et GitHub. Il est important de noter que ces conseils ne sont pas exhaustifs et qu'ils peuvent varier en fonction du contexte du projet. Il est donc conseillé d'adapter ces pratiques en fonction des besoins et des spécificités de chaque projet.

Cependant, en suivant ces bonnes pratiques, les développeurs peuvent faciliter la collaboration, améliorer la qualité et la transparence du code, tout en réduisant les erreurs et en assurant la conformité aux normes de qualité et de performance du projet. L'utilisation efficace de Git et GitHub peut donc aider les développeurs à travailler de manière plus productive et à livrer des logiciels de qualité supérieure dans des délais plus courts.

II. Bonnes pratiques Git

A. Bonnes pratiques Git

Utilisation de la branche principale

Utiliser une branche principale stable est une bonne pratique dans le développement de logiciels, car elle permet de maintenir une version stable et fonctionnelle du code tout au long du processus de développement. La branche principale stable est généralement nommée «*master*» ou «*main*» et ne doit contenir que du code testé et validé. Cela permet de maintenir une version fonctionnelle du code, ce qui est essentiel pour assurer la qualité du logiciel et éviter les erreurs.

En utilisant une branche principale stable, tous les membres de l'équipe travaillent à partir d'une version commune du code. Cela facilite la collaboration et permet de mieux coordonner le travail entre les différents développeurs. Si plusieurs développeurs travaillent sur la même branche, cela peut causer des conflits et rendre la fusion du code plus difficile. En utilisant une branche principale stable, les développeurs travaillent sur des branches isolées et ne fusionnent le code que lorsqu'il est testé et validé.

La branche principale stable est souvent utilisée pour le déploiement en production. En ayant une version stable et testée du code sur la branche principale, vous pouvez déployer le code plus facilement et réduire les risques d'erreurs ou de bugs.

Choisir des noms explicites

Il est important de nommer les branches et les commits de manière explicite, car cela permet de mieux comprendre l'historique des changements sur un projet et de faciliter la collaboration entre les membres de l'équipe.

Méthode

Il y a plusieurs raisons pour lesquelles il est important de nommer les branches et les commits de manière explicite. Tout d'abord, il faut comprendre que la nomination des branches et des commits de manière explicite est un élément clé pour comprendre l'historique des changements sur un projet. En effet, des noms clairs et précis permettent de savoir rapidement ce qui a été modifié, pourquoi et par qui. Cela facilite également la collaboration entre les membres de l'équipe, car les autres membres peuvent mieux comprendre les modifications effectuées et les raisons pour lesquelles elles ont été apportées.

En plus de simplifier la collaboration, la nomination des branches et des commits de manière claire et précise peut également éviter les confusions et les erreurs lors de la fusion du code. Les noms explicites permettent de savoir exactement quels sont la branche ou le commit que vous devez fusionner avec une autre branche, ce qui peut vous faire gagner du temps et éviter les erreurs potentielles.

Par ailleurs, la nomination explicite des branches et des commits permet de simplifier la recherche. Des noms distincts et évidents facilitent la recherche dans l'historique des changements pour trouver des informations spécifiques. Vous pouvez ainsi mieux rechercher et trouver les informations dont vous avez besoin sans perdre de temps à chercher dans des noms de branches ou de commits confus ou peu précis.

Exemple

```
*   0d799dc (HEAD -> main) Merge branch 'new-payment-gateway' into main
|\
| * d0e621b (new-payment-gateway) Add PayPal integration
| | * 5e53d19 (account-management) Add password reset feature
| | * 0e8e584 Add 2-factor authentication
| | * e76b71f Add profile editing feature
| |/
|/|
* | 2fd3472 Add shopping cart functionality
| * a123e45 Add credit card payment feature
| * 3c3d928 Add checkout page
|/
* f14e910 Initial commit

```

Dans cet exemple simple créé avec «***git graph***», on comprend immédiatement, grâce à une bonne nomination, ce que les commits apportent et la fonctionnalité de la branche.

Pour pouvoir utiliser la commande «***git graph***», il faut au préalable configurer le terminal avec cette commande :

```
$ git config --global alias.graph "log --all --graph --decorate --oneline"
```

L'utilisation de l'anglais pour nommer les branches et les commits dans les projets Git est une pratique courante en France et dans le monde de la programmation en général. Cette pratique permet d'assurer la compréhension des noms pour tous les membres de l'équipe, quelle que soit leur langue maternelle, et de faciliter l'utilisation de plateformes internationales telles que GitHub ou GitLab.

À noter que les bonnes pratiques en matière de gestion des branches Git peuvent varier selon les projets et les équipes, mais voici une approche courante qui utilise les branches suivantes : master, dev, release, hotfix et feature.

Master ou Main : la branche «*master*» ou «*main*» représente la branche principale du projet. Elle devrait contenir la version la plus stable et la plus à jour du code.

Dev : la branche «*dev*» est la branche de développement principale. Les développeurs travaillent sur cette branche pour ajouter de nouvelles fonctionnalités et effectuer des modifications importantes du code.

Release : la branche «*release*» est créée à partir de la branche «*dev*» pour préparer une version du projet pour une publication ou une mise en production. Les corrections de bugs mineurs sont effectuées sur cette branche et elle est soumise à des tests approfondis avant d'être fusionnée avec la branche principale.

Hotfix : la branche « *hotfix* » est créée à partir de la branche « *master* » pour corriger rapidement des bugs critiques qui ne peuvent pas attendre la prochaine version du projet. Les corrections apportées à cette branche sont testées et fusionnées avec la branche principale dès que possible.

Feature : la branche « *feature* » est créée à partir de la branche « *dev* » pour ajouter de nouvelles fonctionnalités ou des modifications importantes du code. Cette branche est soumise à des tests et des relectures avant d'être fusionnée avec la branche principale.

Schéma représentant la nomination des différentes branches Git

Source : Ultrazohm[1]

Ces branches sont généralement utilisées en conjonction avec des processus de développement agiles et des méthodologies de gestion de projet telles que Scrum. Cependant, leur utilisation peut varier selon les projets et les équipes. Il est important de bien comprendre les avantages et les inconvénients de chaque approche de gestion de branches pour trouver celle qui convient le mieux à votre projet.

Bonnes pratiques avec les commits

Il existe plusieurs méthodes qui donnent lieu à des bonnes pratiques pour faire des commits. Les commits atomiques, par exemple, sont des commits qui n'implémentent qu'une seule fonctionnalité ou qu'une seule correction de bug. En faisant des commits atomiques, on facilite la revue de code et la compréhension des changements apportés au projet. Cette façon de faire permet également d'annuler des modifications spécifiques plus facilement en cas de besoin.

Une autre pratique recommandée lors de l'utilisation des commits, c'est d'écrire des messages de commit clairs et concis. Pourquoi les messages de commit doivent-ils être clairs et concis ? Afin de donner l'occasion à tous les membres de l'équipe de comprendre rapidement les changements apportés par le commit. Le message de commit doit décrire le changement de manière précise et fournir suffisamment d'informations pour permettre à n'importe quel membre de l'équipe qui interviendrait de comprendre pourquoi le changement a été apporté.

Il est important d'éviter les commits inutiles. Cela peut être, par exemple, des commits qui ne modifient que des espaces blancs ou qui ne font que renommer des fichiers. Il faut être conscient qu'ils peuvent rendre l'historique de Git confus et difficile à lire. Il est donc approprié de regrouper ce genre de modifications avec un commit de modification, lorsque cela est possible, puis de l'ajouter à la description de celui-ci.

Un autre écueil dans lequel il ne faut pas tomber c'est de faire des commits volumineux. Si des commits sont volumineux et incluent plusieurs modifications non liées les unes aux autres, ils seront difficiles à comprendre et à gérer, et ils peuvent introduire des erreurs difficiles à identifier. Ainsi pour éviter les commits volumineux, il est recommandé de faire des commits atomiques.

1 https://docs.ultrazohm.com/general/project_structure/contribution_workflow.html

Méthode

Voici la liste des commandes à utiliser pour suivre le processus recommandé avant de faire un commit :

1. « ***git status*** » : cette commande permet de voir les fichiers qui ont été modifiés, supprimés ou ajoutés depuis le dernier commit, ainsi que les fichiers qui ont été ajoutés ou modifiés mais qui ne sont pas encore suivis par Git. Pour afficher les changements non suivis, utilisez la commande suivante :

```
$ git status
```

2. « ***git add*** » : après avoir vérifié les modifications avec « ***git status*** », on peut utiliser la commande « ***git add*** » pour ajouter les fichiers nécessaires à notre commit. Pour ajouter un fichier spécifique, utilisez la commande suivante en remplaçant « *nom-du-fichier* » par le nom du fichier que vous souhaitez ajouter :

```
$ git add nom-du-fichier
```

Pour ajouter tous les fichiers modifiés, si vous êtes certain de vouloir tout intégrer au commit, utilisez la commande suivante :

```
$ git add
```

3. « ***git diff*** » : après avoir ajouté les fichiers avec « ***git add*** », on peut utiliser la commande « ***git diff*** » pour vérifier que les modifications ajoutées correspondent bien à nos intentions. Cette commande permet de voir les modifications qui seront incluses dans le commit. Pour afficher les différences entre l'index et le dernier commit, utilisez la commande suivante :

```
$ git diff --cached
```

4. « ***git commit*** » : enfin, après avoir vérifié les modifications avec « *git diff* », on peut utiliser la commande « ***git commit*** » pour faire le commit en incluant un message de commit clair et concis. Pour créer un commit avec un message de commit en ligne de commande, utilisez la commande suivante en remplaçant « *message de commit* » par le message que vous souhaitez inclure :

```
$ git commit -m 'message de commit »
```

Tester son code

Effectuer des tests avant la fusion est une bonne pratique recommandée lors de l'utilisation de Git. Cette étape consiste à s'assurer que les modifications apportées au code sont fonctionnelles et n'ont pas d'effets indésirables sur le projet.

Il est recommandé d'effectuer des tests sur un environnement de développement ou de *staging* avant de fusionner les modifications dans la branche principale du projet. Les tests peuvent inclure des tests unitaires, des tests d'intégration et des tests fonctionnels, en fonction de la complexité du projet.

L'utilisation de tests avant la fusion permet de s'assurer que les modifications sont conformes aux spécifications du projet et qu'elles n'affectent pas négativement d'autres parties du code. Cela peut aider à prévenir les erreurs et les problèmes de qualité du code avant qu'ils ne soient déployés dans l'environnement de production.

B. Exercice : Quiz

Question 1

Quelle est la branche principale d'un projet Git ?

- ○ Dev
- ○ Release
- ○ Master/Main

Question 2

À quoi sert la branche « *dev* » ?

- ○ À créer des fonctionnalités
- ○ À maintenir une version stable du code
- ○ À préparer une version du projet pour une publication

Question 3

Qu'est-ce qu'un commit atomique ?

- ○ Un commit qui inclut plusieurs modifications non liées les unes aux autres
- ○ Un commit qui implémente une seule fonctionnalité ou une seule correction de bug
- ○ Un commit qui ajoute des fichiers non nécessaires au projet

Question 4

À quoi servent les tests avant la fusion dans Git ?

- ○ À garantir que le code est fonctionnel et ne génère pas de bug avec d'autres parties du code
- ○ À supprimer les fichiers inutiles du projet
- ○ À créer une nouvelle branche Git

Question 5

Que doit contenir la branche principale d'un projet Git ?

- ○ Du code non testé
- ○ Du code testé et validé
- ○ Du code en cours de développement

III. Bonnes pratiques Github

A. Bonnes pratiques Github

Les issues

Une issue est un ticket ou une demande de modification sur un projet GitHub. Les issues sont généralement utilisées pour faciliter la collaboration entre les membres de l'équipe travaillant sur un projet. Elles permettent de signaler les problèmes, les idées ou les améliorations à apporter de manière claire et organisée. Elles peuvent aussi contenir des descriptions détaillées de la tâche à effectuer, des étapes à suivre, des captures d'écran, des liens vers des références, etc.

Autre fonctionnalité des issues : elles peuvent être assignées à des membres de l'équipe pour leur attribution. Les membres assignés reçoivent des notifications lorsqu'une issue est mise à jour ou fermée. Les commentaires peuvent être ajoutés aux issues pour discuter des solutions ou pour fournir des mises à jour sur l'avancement des travaux.

Autre avantage, elles peuvent être triées et filtrées en fonction de leur type, de leur priorité ou de leur statut.

Comment définir les issues ?

Avant même de créer une issue, il est important d'identifier les tâches nécessaires pour atteindre les objectifs du projet. Par exemple, vous pouvez lister les fonctionnalités à implémenter, les bogues à corriger ou les améliorations à apporter. Ensuite, pour chaque tâche identifiée, créez une nouvelle issue sur GitHub en cliquant sur l'onglet « ***Issues*** » de votre projet, puis sur le bouton « ***New issue*** ». Pensez à ajouter une description détaillée de la tâche, y compris les étapes à suivre pour la réaliser.

Afin de mieux organiser les issues, vous pouvez ajouter des labels à chaque issue. Par exemple, vous pouvez utiliser des labels comme « *bug* », « *feature* », « *documentation* », etc. Ces labels vous permettent de trier et de filtrer les issues en fonction de leur type. De plus, vous pouvez assigner des personnes responsables de la réalisation de chaque tâche en utilisant le champ « ***Assignees*** ». Les personnes assignées recevront des notifications lorsqu'une issue est mise à jour ou fermée.

Les labels vous aident à définir les priorités pour chaque issue, tandis que les milestones vous permettent de regrouper les issues en fonction de leur date d'échéance ou de leur importance.

En outre, vous avez la possibilité de suivre les progrès de chaque issue en utilisant les commentaires. Les commentaires sont là pour vous accorder la possibilité de discuter des problèmes et des solutions avec les autres membres de l'équipe, de poser des questions et de fournir des mises à jour.

Il est également recommandé de créer des issues qui correspondent aux branches pour mieux organiser et suivre les travaux en cours sur un projet de développement. En créant des issues qui correspondent aux branches, vous pouvez mieux visualiser les tâches à effectuer pour chaque branche et suivre leur progression. Par exemple, si vous créez une branche pour travailler sur une nouvelle fonctionnalité, vous pouvez également créer une issue qui correspond à cette branche. Dans cette issue, vous pouvez décrire la fonctionnalité à développer, les tests à effectuer, les dépendances à installer, etc. Les développeurs peuvent alors travailler sur la branche en suivant les instructions de l'issue et fermer l'issue une fois que la fonctionnalité est développée et testée. Cependant, il est important de ne pas créer trop d'issues pour éviter de perdre en clarté et en lisibilité. Il est préférable de créer des issues générales pour les grandes tâches et de détailler les tâches plus petites dans les commentaires de l'issue. De cette façon, vous pouvez mieux suivre les travaux en cours tout en évitant de surcharger l'interface de gestion de projet avec trop d'issues.

Méthode **Créer une issue sur Github**

1. Connectez-vous à votre compte GitHub et accédez au projet sur lequel vous souhaitez créer une issue.
2. Dans la barre de navigation supérieure, cliquez sur l'onglet « ***Issues*** ».
3. Cliquez sur le bouton vert « ***New issue*** » pour créer une nouvelle issue.
4. Rédigez un titre pour votre issue. Ce titre doit être court et précis pour décrire le problème ou la fonctionnalité à ajouter.
5. Rédigez une description détaillée de l'issue. Décrivez le problème que vous rencontrez, les étapes pour le reproduire ou la fonctionnalité que vous souhaitez ajouter. Vous pouvez également ajouter des images, des liens ou du code si nécessaire.
6. Ajoutez des labels pour organiser votre issue. Les labels permettent de catégoriser les issues en fonction de leur type ou de leur priorité. Vous pouvez ajouter des labels prédéfinis ou en créer de nouveaux en cliquant sur « ***Edit labels*** ».
7. Assignez l'issue à un ou plusieurs membres de l'équipe en utilisant le champ « ***Assignees*** ». Les personnes assignées recevront des notifications pour les mises à jour de l'issue.
8. Ajoutez des références si nécessaire. Vous pouvez ajouter des références à des problèmes ou des pull-requests liés en utilisant le champ « ***References*** ». (Les pull-requests sont expliquées dans le paragraphe suivant, « *Les pull-requests github ou merge request sur gitlab* ».)
9. Configurez les options supplémentaires si nécessaire. Vous pouvez modifier les paramètres tels que l'assignation automatique, la demande de review, etc.
10. Cliquez sur « ***Submit new issue*** » pour créer l'issue.

Une fois l'issue créée, vous pouvez suivre son évolution en consultant les commentaires et les mises à jour ajoutées par les membres de l'équipe. Vous pouvez également utiliser les options de tri et de recherche pour trouver des issues similaires ou suivre les tendances des problèmes.

Les pull-requests github ou merge request sur gitlab

Une pull-request (ou «*demande d'extraction*» en français) est une fonctionnalité de github (merge request sur gitlab) qui permet aux membres de l'équipe de proposer des modifications à un code existant et de les soumettre pour examen et fusion dans la branche principale. En gros, cela permet à un contributeur de faire une demande de fusion des modifications qu'il a apportées sur une branche vers une autre branche, généralement la branche principale. Cette demande de fusion peut être examinée, discutée, et éventuellement validée ou rejetée par d'autres membres de l'équipe avant que les modifications soient effectivement fusionnées dans la branche principale.

L'utilisation de pull-requests dans le processus de développement logiciel présente de nombreux avantages pour les équipes de développement. L'un des principaux avantages est la collaboration accrue entre les membres de l'équipe. Les pull-requests permettent aux contributeurs de proposer et de discuter des modifications avec d'autres membres avant la fusion. Cette interaction encourage les commentaires et les suggestions qui peuvent améliorer le code, favorisant ainsi la qualité et la fiabilité du produit final.

En plus d'améliorer la qualité du code, les pull-requests offrent également une transparence totale sur les modifications apportées au code. Les modifications sont documentées et accessibles à tous les participants, ce qui permet de maintenir un suivi clair et une compréhension partagée des modifications apportées. Cette transparence peut aider à résoudre les problèmes plus rapidement et à garantir que tous les membres de l'équipe sont informés des changements importants.

De plus, les pull-requests peuvent être automatisés à l'aide d'outils tels que des tests de validation et des vérifications de code. Cette automatisation garantit que les modifications apportées sont conformes aux normes de qualité et de performance du projet. Elle permet également de détecter et de corriger les erreurs et les problèmes potentiels avant qu'ils ne soient déployés dans l'environnement de production, réduisant ainsi les risques d'interruption du service ou de perturbation des utilisateurs.

Méthode

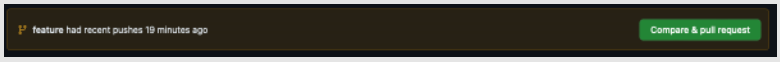
1. Tout d'abord, assurez-vous d'avoir une branche de fonctionnalité distincte sur laquelle vous avez effectué des modifications que vous souhaitez fusionner dans la branche principale. Si vous n'avez pas déjà une branche de fonctionnalité, créez-en une en utilisant la commande :

```
$ git switch -c nom_de_votre_branche
```

2. Faites vos modifications sur la branche en commitant comme vous en avez l'habitude et lorsque vous estimez que votre travail et fini et que la nouvelle fonctionnalité et opérationnelle pusher vers le dépôt distant Github :

```
$ git push
```

3. Accédez à la page de votre repository sur GitHub et cliquez sur le bouton «***Compare & pull requests***» en haut de la page.



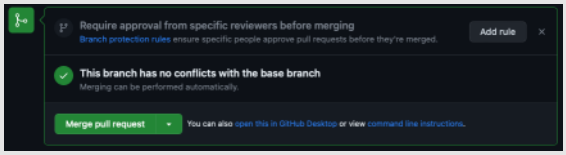
Bouton « Compare & pull request »

4. GitHub vous envoie automatiquement les modifications de votre dernier commit et la branche que vous souhaitez fusionner.

Vous pouvez simplement ajouter une description détaillée de votre pull-request dans le bloc de commentaire fourni par GitHub. Expliquez ce que vous avez modifié, et pourquoi, et ajoutez des captures d'écran ou des liens vers des problèmes connexes si nécessaire. Vous pouvez aussi associer votre pull-request à une issue avec #numéro_issue

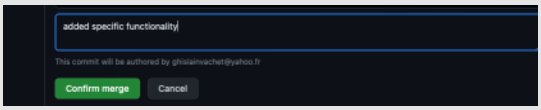
Vous pouvez également spécifier des relecteurs pour votre pull-request en utilisant la section « ***Reviewers*** » située à droite de la page. Cela permettra à ces membres de l'équipe de recevoir une notification pour examiner votre pull-request et fournir des commentaires ou des suggestions.

5. Cliquez sur « ***Create pull request*** » et GitHub affichera automatiquement les différences entre les deux branches, vous permettant de vérifier les modifications que vous souhaitez fusionner. Assurez-vous que tout est correct et que les modifications sont bien celles que vous souhaitez fusionner. S'il n'y a pas de conflit, vous devriez avoir ce message :



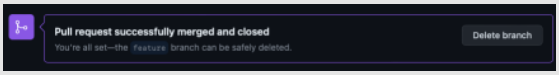
Merge pull request

6. Cliquez sur « ***Merge pull request*** » et ajoutez une description détaillée de votre pull-request, en expliquant ce que vous avez modifié et pourquoi.



Confirm merge

7. Cliquez sur « ***Confirm merge*** » et vous devriez avoir ce message :



Pull request successfully

B. Exercice : Quiz

Question 1

Qu'est-ce qu'une issue sur GitHub ?

- ○ Une demande de modification sur un projet GitHub
- ○ Un outil de mesure de la qualité du code
- ○ Un module d'authentification pour l'accès aux dépôts Git

Question 2

Quel est l'avantage de créer des issues qui correspondent aux branches ?

- ○ Permet de mieux visualiser les tâches à effectuer.
- ○ Regrouper les issues en fonction de leur date d'échéance ou de leur importance
- ○ Éviter de surcharger l'interface de gestion de projet avec trop d'issues

Question 3

Que peut-on ajouter aux issues pour faciliter leur organisation ?

- Des descriptions détaillées des tâches à effectuer
- Des références à des problèmes ou des pull-requests liés
- Des labels pour trier et filtrer les issues en fonction de leur type ou de leur priorité

Question 4

Qu'est-ce qu'une pull-request sur GitHub ?

- Une demande d'extraction des modifications proposées sur une branche vers une autre branche
- Un ticket ou une demande de modification sur un projet GitHub
- Un outil de mesure de la qualité du code

Question 5

Pourquoi les pull-requests encouragent-ils la collaboration entre les membres de l'équipe ?

- Parce qu'il donne la possibilité d'échanger entre les membres de l'équipe sur le code, les idées et autres
- Parce qu'ils permettent de regrouper les issues en fonction de leur date d'échéance ou de leur importance
- Parce qu'ils permettent de détecter et de corriger les erreurs et les problèmes potentiels avant qu'ils ne soient déployés dans l'environnement de production

IV. Essentiel

L'utilisation de bonnes pratiques sur Git et GitHub est essentielle pour optimiser la collaboration entre les membres d'une équipe travaillant sur un projet de développement. La gestion de version avec Git permet de travailler en toute sécurité, de conserver l'historique des modifications, de revenir en arrière si nécessaire, et de gérer efficacement les conflits.

GitHub, quant à lui, offre de nombreuses fonctionnalités pour améliorer la collaboration et la transparence, telles que les issues, les pull-requests et les commentaires. Les issues permettent de signaler les problèmes, les idées ou les améliorations à apporter de manière claire et organisée, et les pull-requests facilitent la proposition et la validation des modifications de code.

Il est important de suivre les bonnes pratiques telles que la création de branches pour les fonctionnalités et les corrections de bogues, l'utilisation de messages de commit clairs et précis, la vérification des conflits avant de fusionner les branches, l'ajout de commentaires pour faciliter la collaboration et la relecture de code pour améliorer la qualité.

En adoptant ces pratiques, les membres de l'équipe peuvent travailler plus efficacement, être plus productifs et qualitatifs sur la transparence du code, tout en réduisant les erreurs et en garantissant la conformité aux normes de qualité et de performance du projet. En somme, l'utilisation des bonnes pratiques sur Git et GitHub peut considérablement améliorer la productivité et la qualité du travail en équipe dans le domaine du développement logiciel.

V. Auto-évaluation

A. Exercice

Vous travaillez sur un projet open-source en collaboration avec d'autres développeurs. Vous avez remarqué que de nombreuses issues sont ouvertes, mais pas encore résolues. Vous voulez vous assurer que toutes les issues sont bien assignées à des membres de l'équipe pour être résolues rapidement.

Question 1

Décrivez les étapes pour assigner à chaque issue un membre de l'équipe qui sera responsable de la résolution de l'issue. Ajoutez des commentaires pour ses issues en décrivant les étapes à suivre pour la résoudre. Puis créez votre première branche qui correspond à votre première issue, faites trois commits dessus et faites un git merge.

Vous travaillez sur un projet de développement de logiciel et vous devez gérer plusieurs branches selon les bonnes pratiques Git. Vous voulez simuler un projet avec plusieurs branches pour comprendre comment elles sont gérées et comment les commits sont effectués régulièrement pour maintenir l'intégrité du projet.

Question 2

Simulez un projet de développement de logiciel avec les branches master, dev, release, hotfix et feature. Utilisez Git pour effectuer des commits réguliers et maintenir l'intégrité du projet. Utilisez ensuite la commande « ***git graph*** » pour visualiser la structure de votre projet avec les différentes branches et les commits associés.

B. Test

Exercice 1 : Quiz

Question 1

Comment s'appelle la branche utilisée pour préparer une version du projet pour une publication ?

- ○ Dev
- ○ Release
- ○ Hotfix

Question 2

Comment nommer les branches et les commits dans un projet Git ?

- ○ Avec des noms aléatoires
- ○ Avec des noms génériques comme « *fonctionnalité 1* », « *fonctionnalité 2* », etc.
- ○ Avec des noms explicites décrivant les modifications apportées

Question 3

Comment créer une issue sur GitHub ?

- ○ En utilisant la commande « ***git add -A*** »
- ○ En cliquant sur l'onglet « ***Issues*** » de votre projet, puis sur le bouton « ***New issue*** » et en ajoutant une description détaillée de la tâche à effectuer
- ○ En cliquant sur le bouton « ***Pull request*** » de votre projet, puis en ajoutant une description détaillée des modifications apportées

Question 4

Qu'est-ce qu'un label sur GitHub ?

- ○ Un commentaire ajouté sur une pull-request
- ○ Une étiquette attribuée à une issue
- ○ Une demande de modification de code proposée par un membre de l'équipe

Question 5

À quoi sert le champ «***Assignees***» lors de la création d'une issue sur GitHub ?

- ○ Il permet de définir la priorité de l'issue
- ○ Il permet de trier et de filtrer les issues en fonction de leur statut
- ○ Il permet d'assigner des personnes responsables de la réalisation de la tâche