Cours Git - Theorie

Introduction

Définition

Un gestionnaire de versions distribuées est un outil essentiel dans la programmation moderne, facilitant la collaboration sur les projets logiciels en conservant un historique de toutes les modifications apportées au code source. Les systèmes comme Git, Mercurial et Bazaar sont des exemples de ces outils.

Comprendre les systèmes de contrôle de version

Un système de contrôle de version (VCS, Version Control System) est un outil qui aide les développeurs à suivre et à gérer les modifications apportées à un code source au fil du temps. Il conserve un historique de toutes les modifications, permettant ainsi aux développeurs de revenir à une version précédente si nécessaire. Il existe deux types principaux de VCS : centralisés et distribués.

Systèmes de contrôle de version centralisés (CVCS)

Dans un CVCS, comme SVN ou CVS, il y a un seul "référentiel" central de tout le code et de son historique. Les développeurs obtiennent une copie de travail du code, apportent des modifications, puis "commettent" ces modifications dans le référentiel central.

Systèmes de contrôle de version distribués (DVCS)

Dans un DVCS, comme Git, chaque développeur possède une copie complète du référentiel, y compris l'historique des modifications. Cela offre de nombreux avantages, comme la possibilité de travailler hors ligne, une réduction de la dépendance à un serveur central, et une flexibilité accrue pour les différentes méthodologies de développement.

Git : Un exemple de gestionnaire de versions distribuées

Git est le DVCS le plus largement utilisé.

Il a été créé par Linus Torvalds pour le développement du noyau Linux.

Concepts clés de Git

- **Référentiel (repository)** : Un référentiel Git est un ensemble de fichiers et de dossiers que vous souhaitez suivre, dossiers dans lesquels vous effectuez des modifications, et que vous souhaitez enregistrer au fil du temps.
- **Commit** : Un commit est un instantané d'un ensemble de modifications apportées aux fichiers dans votre référentiel. Chaque commit est doté d'un identifiant unique (un "hash") que vous pouvez utiliser pour référencer ce commit spécifique.
- **Branches** : Les branches sont essentiellement des pointeurs vers un commit. Elles sont utilisées pour créer des flux de travail isolés sur le même référentiel, permettant aux développeurs de travailler sur différentes fonctionnalités simultanément sans interférer entre eux.
- **Merge** : L'opération de fusion permet de combiner les modifications de différentes branches ensemble.

Workflow basique avec Git

1. **Clone** : Cette opération vous permet d'obtenir une copie locale d'un référentiel existant.
2. **Pull** : Cette opération vous permet de récupérer les dernières modifications du référentiel distant.
3. **Branch** : Créez une nouvelle branche sur laquelle effectuer vos modifications.
4. **Checkout** : Cette commande vous permet de passer d'une branche à une autre.
5. **Add/Commit** : Ajoutez des fichiers à votre commit et enregistrez vos modifications dans le référentiel.
6. **Push** : Cette commande vous permet d'envoyer vos commits vers un référentiel distant. Cela permet de partager votre travail avec d'autres et de le sauvegarder dans un emplacement externe.

7. **Merge** : Si vous avez fini de travailler sur une branche et que vous souhaitez intégrer vos modifications dans la branche principale (souvent appelée 'master' ou 'main'), vous utiliserez la commande merge.
8. **Pull Request** : Dans certains systèmes comme GitHub, avant de fusionner votre branche avec la branche principale, vous pouvez créer une "Pull Request". C'est une façon de proposer vos modifications à l'équipe, de demander des commentaires et d'effectuer des révisions si nécessaire avant de fusionner.

Avantages d’un gestionnaire de versions distribuées

- Il protège contre le fait qu'une seule machine devienne le SPOF (Point Unique De Défaillance)
- Les contributeurs du projet peuvent continuer à travailler sur leur projet sans avoir à être en ligne
- Il permet de participer à un projet sans nécessiter d'autorisations par un chef de projet (les droits de validation/soumission peuvent donc être donnés après avoir démontré son travail et pas avant)
- La plupart des opérations sont plus rapides car effectuées localement.
- Vous permet de créer des brouillons sans publier les modifications ni déranger les autres contributeurs.
- Permet de conserver un référentiel de référence contenant les versions livrées d'un projet.

Les objets Git

Introduction

Nous allons d'abord explorer son fonctionnement interne. Comme vous le savez, Git est un outil de gestion de versions qui garde en mémoire toutes les modifications apportées aux fichiers, avec des informations telles que le message, la date et l'heure. Pour stocker de manière efficace ces fichiers et informations, Git utilise sa propre structure de fichiers basée sur les objets "blob", "tree" et "commit".

Blob

Un objet "blob" est le plus simple des objets dans Git. Il représente un jeu de données. Dans la plupart des cas, cet ensemble de données est un fichier.

Lorsque vous ajoutez un fichier à votre dépôt Git et que vous validez cette modification, Git crée un objet blob qui contient les données de votre fichier. Chaque blob est identifié par une empreinte SHA-1, qui est générée à partir du contenu du blob.

Exemple

Supposons que vous ayez un fichier appelé "hello.txt" avec le contenu "Hello, World!".

Lorsque vous ajoutez ce fichier à votre dépôt Git et que vous effectuez un commit, Git crée un blob pour ce fichier. Le blob contiendra les données "Hello, World!", et aura un hash SHA-1 unique basé sur ces données.

Tree

Un tree dans Git est un objet qui représente un répertoire dans votre système de fichiers. Il fait le lien entre le nom des fichiers et les blobs. Il peut également lier à d'autres trees pour représenter une structure de répertoire.

Chaque objet tree contient une ou plusieurs entrées, chacune d'elles étant une référence à un blob ou à un autre tree (pour les sous-répertoires), un nom de fichier, et des permissions de fichier.

Exemple

Supposons que vous ayez un répertoire avec deux fichiers : "hello.txt" et "world.txt".

Lorsque vous ajoutez ces fichiers à votre dépôt Git et que vous effectuez un commit, Git crée un tree qui représente ce répertoire. Ce tree contiendra deux entrées : une pour "hello.txt" et une pour "world.txt". Chaque entrée pointera vers le blob correspondant à ce fichier et aura le nom du fichier.

Commit

Un commit est l'objet le plus complexe dans Git. Il représente un point dans l'histoire de votre projet.

Un commit pointe vers un tree qui représente l'état de votre dépôt à ce point précis. Il contient également des métadonnées, comme le nom de l'auteur, l'adresse e-mail, la date du commit, et un message de commit.

Un commit pointe également vers zéro ou plusieurs commits parents. Un commit initial n'a aucun parent, un commit normal en a un, et un commit qui est le résultat d'une fusion de deux branches peut avoir deux parents.

Exemple

Supposons que vous avez ajouté plusieurs fichiers à votre dépôt Git et que vous êtes prêt à effectuer un commit.

Lorsque vous effectuez un commit, Git crée un objet commit qui contient vos informations (nom, e-mail), la date et heure actuelle, votre message de commit, et un pointeur vers la tree qui représente l'état actuel de votre dépôt. Si c'est votre premier commit, il n'aura pas de parents. Si ce n'est pas votre premier commit, il pointera vers le commit que vous avez effectué juste avant

Recapitulatif

Blob

- Un "blob" (ou "binary large object") est un fichier qui est stocké dans git, il permet de stocker un contenu tel qu'une image ou un document.

Analogie

Imaginez un blob comme un coffre-fort qui contient un précieux document (votre fichier). Une fois que le document est à l'intérieur, le coffre-fort est scellé et il ne peut plus être modifié. Le coffre-fort est ensuite marqué avec un numéro de série unique (l'empreinte SHA-1), basé sur le contenu du document à l'intérieur. Vous pouvez avoir de nombreux coffres-forts (blobs) avec différents documents, chacun avec son propre numéro de série unique.

Tree

- Un "tree" est comme une arborescence de fichier, il contient des références aux "blobs" et aux "trees" enfants, c'est un peu comme un dossier qui contient des fichiers et d'autres dossiers.

Analogie

Maintenant, imaginez que vous avez une pièce (un répertoire) où vous stockez tous ces coffres-forts. Dans cette pièce, chaque coffre-fort est associé à une étiquette (le nom du fichier). C'est ce qu'est un tree. C'est comme une pièce qui contient plusieurs coffres-forts (blobs), chacun associé à une étiquette (nom de fichier). Et cette pièce peut aussi contenir d'autres petites pièces (sous-répertoires), qui ont aussi leurs propres coffres-forts et leurs propres étiquettes.

Commit

- Un "commit" est comme un instantané, il contient une référence à un "tree" particulier, ainsi que des informations comme la date, l'auteur, le message de commit. Cela permet de retrouver un état précis d'un projet à un moment donné.

Commit

Un commit est donc simplement un objet stockant :

- Une référence vers un tree
- Un message de commit

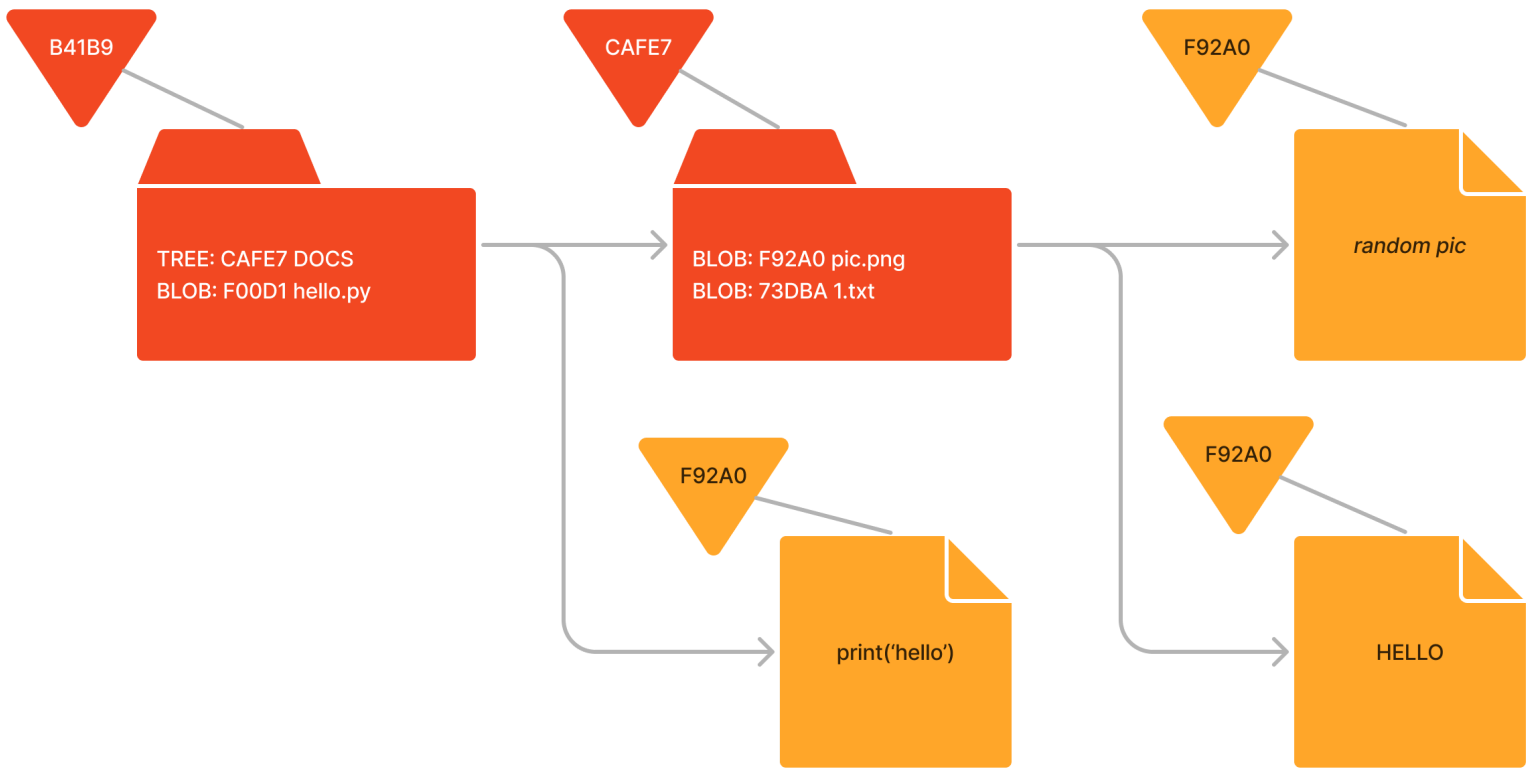
- Le nom de l'auteur du commit
- La date et l’heure du commit
- Une référence vers son commit parent (le commit qui le précède)

Analogie

Enfin, imaginez un agent de sécurité qui fait des rondes dans le bâtiment où se trouve cette pièce. Chaque fois que l'agent fait sa ronde, il note l'état actuel de la pièce, y compris l'emplacement de chaque coffre-fort et de chaque étiquette, ainsi que son propre nom, la date et l'heure, et un bref rapport sur ce qui a changé depuis sa dernière ronde. C'est ce qu'est un commit. C'est comme un enregistrement de l'état de la pièce (l'état de votre projet) à un certain moment dans le temps, avec des métadonnées supplémentaires comme l'agent de sécurité (l'auteur du commit), la date et l'heure, et un rapport sur ce qui a changé (le message de commit).

Visualisation

Schema

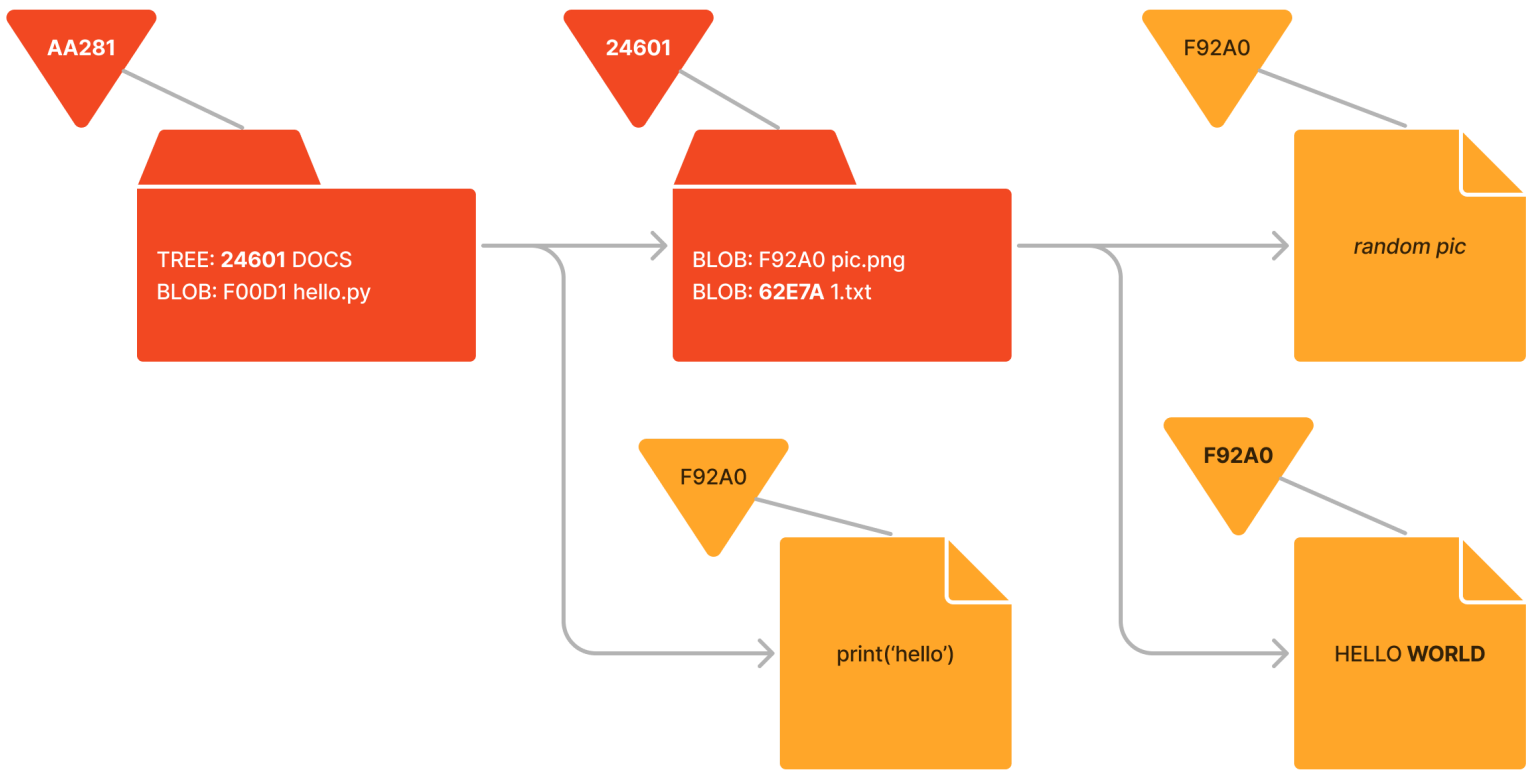


Explications

Dans la slide précédente, nous pouvons voir que chaque tree contient une référence vers un blob. Cependant, un tree peut également contenir une référence vers un autre tree. Cela permet de créer des dossiers contenant d'autres dossiers et fichiers.

Chaque objet git est stocké dans un fichier dont le nom est le hash SHA-1 de l'objet. Donc, les noms des fichiers sont stockes dans le tree, et les noms des dossiers sont stockés dans le tree parent.

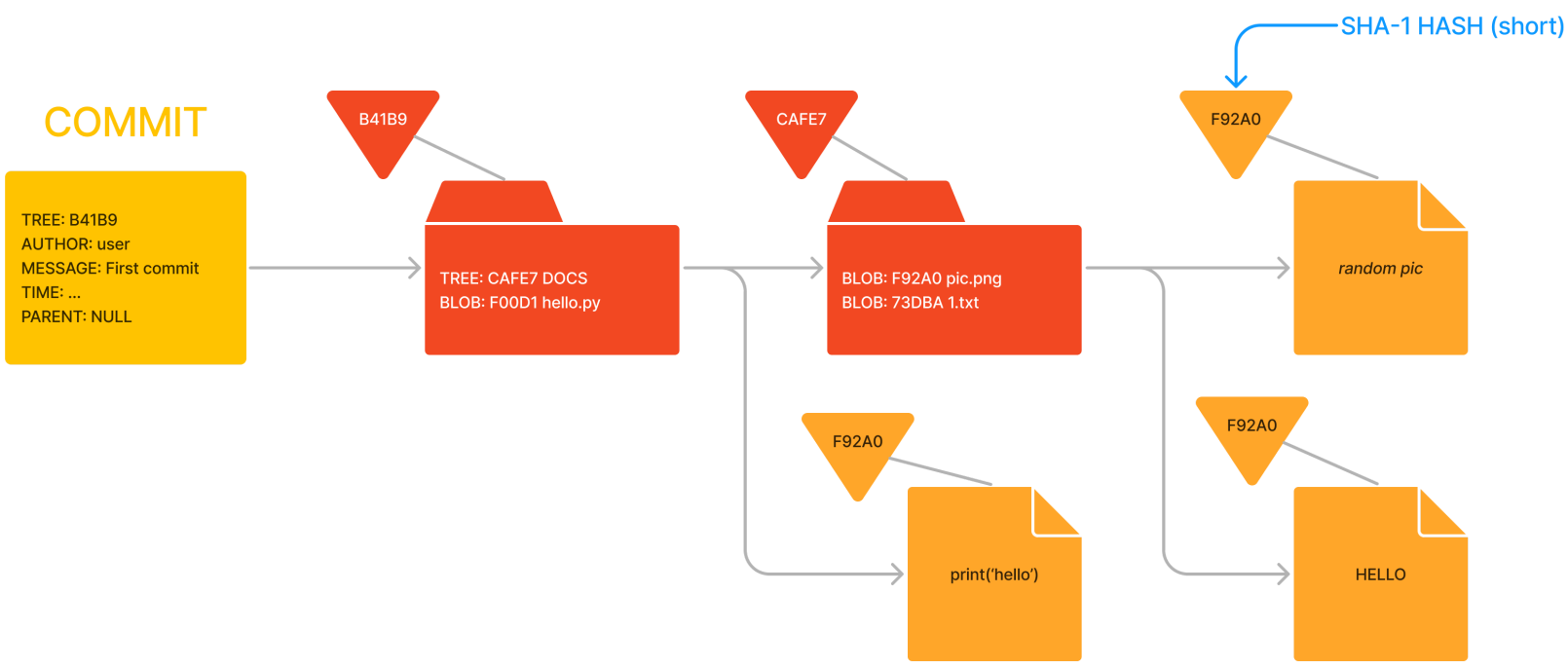
Schema



Explications

Si on modifie le contenu d'un fichier, Git va créer un nouveau blob avec le nouveau contenu, et modifier le tree pour pointer vers ce nouveau blob. puisque le contenu du tree a changé, Git va créer un nouveau tree avec les modifications. Et ce, pour tous ses parents.

Schema



Explications

Un commit pointe donc vers un tree, qui pointe vers des blobs et d'autres trees.

C'est ainsi qu'il garde un snapshot de l'état du projet à un moment donné.

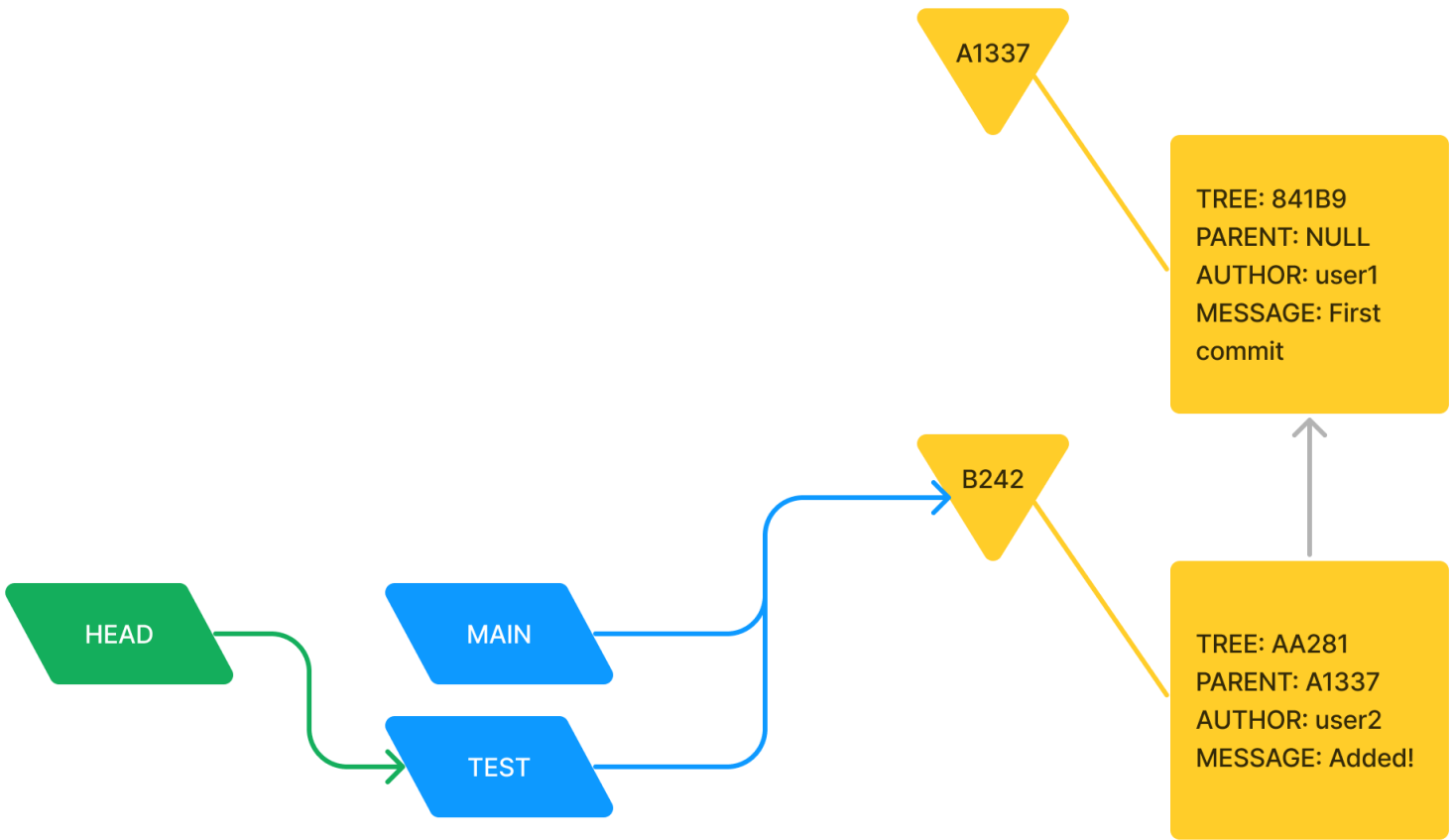
HEAD

Attardons maintenant notre attention sur le pointeur HEAD.

HEAD est un pointeur vers la branche courante.

Par défaut, HEAD pointe vers la branche main. La branche main est la branche principale du projet.

Schema



Explications

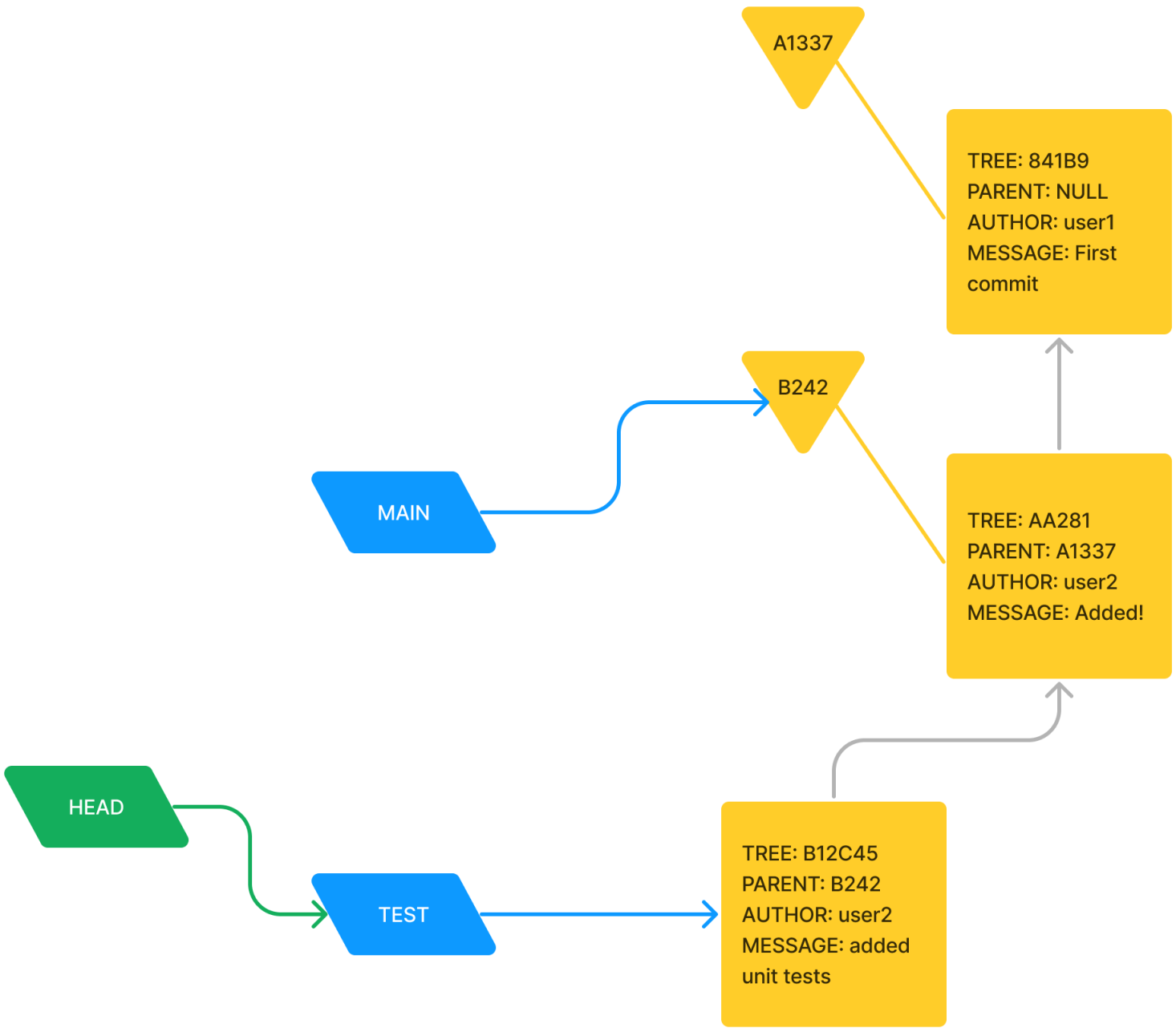
Sur notre schéma, nous avons deux branches: MAIN et TEST.

HEAD pointe vers la branche TEST.

MAIN et TEST pointent vers le meme commit.

Si nous faisons un commit sur la branche TEST, HEAD va pointer vers TEST, et TEST va pointer vers ce commit.

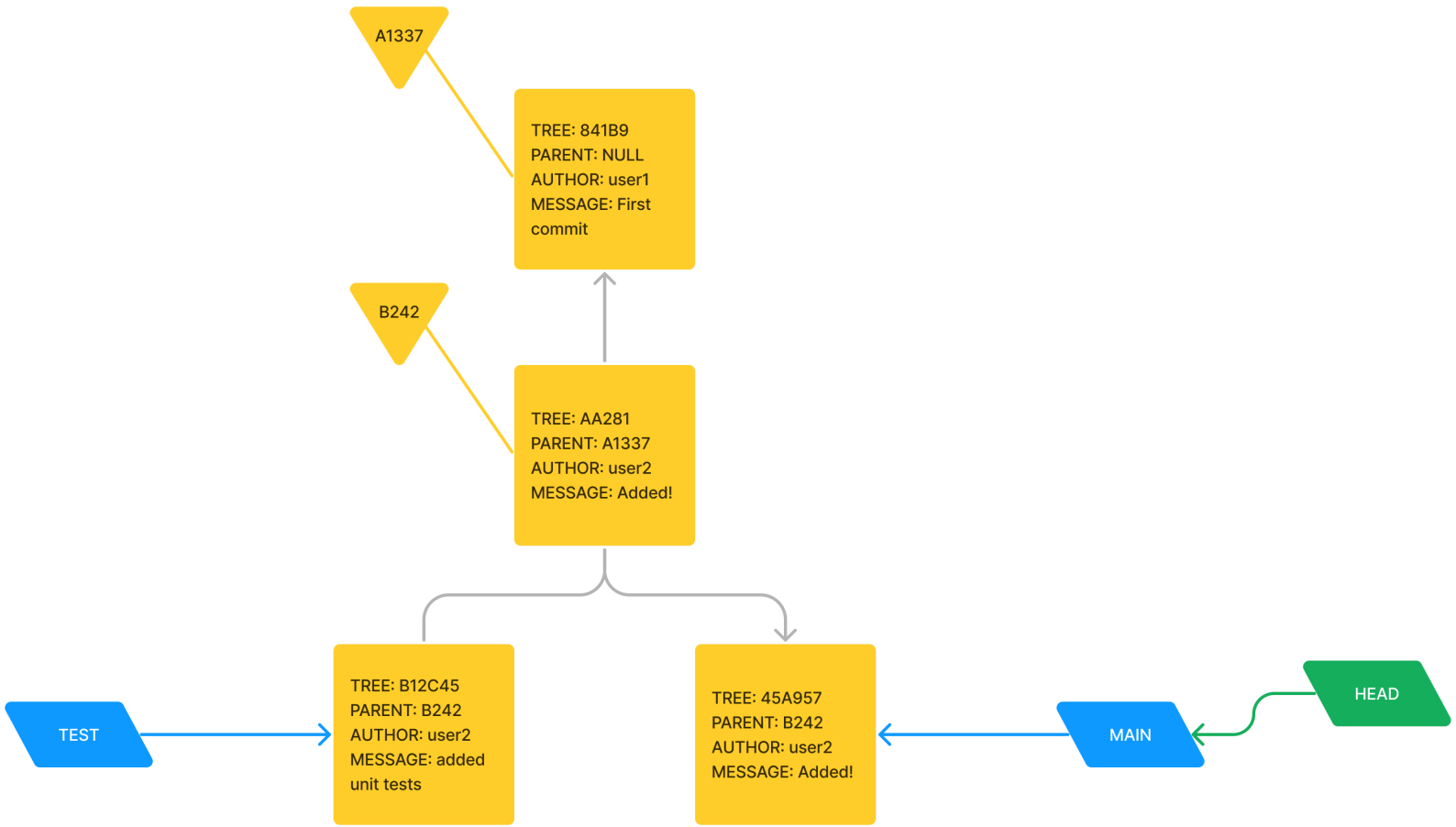
Schema



Explications

Si nous nous deplaçons sur la branche MAIN, et faisons un commit, HEAD va pointer vers MAIN, et MAIN va pointer vers ce commit.

Schema



Patterns et workflows

Le git flow

Qu'est-ce que le Git flow ?

Le Git flow est un type de workflow qui se base sur l'utilisation de branches spécifiques pour gérer les différentes étapes de développement d'un projet. Il a été développé par Vincent Driessen pour gérer les flux de travail d'équipes de développement. Il est souvent utilisé pour les projets de grande taille ou les projets avec plusieurs équipes de développement simultanément.

Comment fonctionne le Git flow ?

Avec le Git flow, il y a une branche principale `main` qui accueille les versions stables de votre projet et une branche `develop` qui accueille les modifications en cours de développement. Il y a également des branches pour les fonctionnalités, les corrections de bugs et les versions de production. Les modifications sont intégrées dans ces branches spécifiques et ensuite fusionnées dans la branche principale après validation.

Il est important de noter que le Git flow nécessite une bonne organisation et une bonne communication entre les membres de l'équipe pour fonctionner efficacement.

Avantages

- Organisation : Le Git flow permet une bonne organisation des modifications et une bonne visibilité sur l'état de développement d'un projet.
- Séparation des modifications : Il permet une séparation des modifications entre les fonctionnalités, les corrections de bugs et les versions de production.
- Possibilité de gérer plusieurs équipes : Il est adapté pour les projets de grande taille ou les projets avec plusieurs équipes de développement simultanément.

Inconvénients

- Complexité : Il nécessite une bonne compréhension des branches et des merge pour être utilisé efficacement, il est donc plus complexe à mettre en place que d'autres types de workflow.
- Communication et coordination : Il nécessite une bonne communication et coordination entre les membres de l'équipe pour fonctionner efficacement.

Quand utiliser le Git flow ?

- Lorsque vous travaillez sur un projet de grande taille ou avec plusieurs équipes de développement simultanément.
- Lorsque vous voulez une bonne organisation des modifications et une bonne visibilité sur l'état de développement d'un projet.
- Lorsque vous voulez une séparation des modifications entre les fonctionnalités, les corrections de bugs et les versions de production.

Quand éviter le Git flow ?

- Lorsque vous travaillez sur un projet de petite ou moyenne taille, il est préférable d'utiliser un autre type de workflow comme un workflow basé sur le tronc pour éviter la complexité supplémentaire liée à la gestion des branches.
- Lorsque vous voulez éviter la complexité supplémentaire liée à la gestion des branches et des merge.
- Lorsque la communication et la coordination entre les membres de l'équipe sont difficiles.

Ce qu'il y a a retenir :

- Le Git flow est un type de workflow qui se base sur l'utilisation de branches spécifiques pour gérer les différentes étapes de développement d'un projet.
- Il permet une bonne organisation des modifications et une bonne visibilité sur l'état de développement d'un projet.
- Il permet une séparation des modifications entre les fonctionnalités, les corrections de bugs et les versions de production.
- Il est adapté pour les projets de grande taille ou les projets avec plusieurs équipes de développement simultanément.
- Il nécessite une bonne compréhension des branches et des merge pour être utilisé efficacement, il est donc plus complexe à mettre en place que d'autres types de workflow.
- Il nécessite une bonne communication et coordination entre les membres de l'équipe pour fonctionner efficacement.
- Il n'est pas adapté pour les projets de petite ou moyenne taille, ou lorsque la communication et la coordination entre les membres de l'équipe sont difficiles.

Trunk Based Workflow

Qu'est-ce qu'un workflow basé sur le tronc (trunk-based workflow) ?

Un workflow basé sur le tronc (ou trunk-based workflow) est un type de workflow qui se base sur l'utilisation d'une seule branche principale (ou tronc) pour accueillir toutes les modifications. Il est souvent utilisé pour les projets de petite ou moyenne taille, ou pour les projets qui nécessitent une intégration rapide des modifications.

Qu'est-ce qu'un workflow basé sur le tronc (trunk-based workflow) ?

Avec un workflow basé sur le tronc, tous les développeurs travaillent sur la même branche principale, et les modifications sont intégrées directement dans cette branche, sans passer par des branches de fonctionnalités ou des branches de bugfix. Cela permet une intégration rapide des modifications, mais il est important de noter qu'il n'y a pas de séparation des modifications et qu'il peut y avoir des risques de conflits.

Il est possible de mettre en place des stratégies de validation des modifications avant intégration pour minimiser les risques de conflits.

Avantages

- Intégration rapide des modifications : Les modifications sont intégrées directement dans la branche principale, sans passer par des branches de fonctionnalités ou des branches de bugfix.
- Simplicité : Ce type de workflow est plus simple à mettre en place et à comprendre que d'autres types de workflow.
- S'intègre bien avec les outils de CI/CD : Ce type de workflow s'intègre bien avec les outils de CI/CD, car il n'y a pas de branches de fonctionnalités ou de branches de bugfix.

Inconvénients

- Risques de conflits : Il n'y a pas de séparation des modifications, il y a donc des risques de conflits.
- Pas de séparation des modifications : Il n'y a pas de séparation des modifications, ce qui peut rendre difficile de comprendre les modifications apportées.

Quand utiliser un workflow basé sur le tronc ?

- Lorsque vous travaillez sur un projet de petite ou moyenne taille.
- Lorsque vous voulez une intégration rapide des modifications.

Quand éviter un workflow basé sur le tronc ?

- Lorsque vous travaillez sur un projet de grande taille ou avec plusieurs équipes de développement simultanément, il est préférable d'utiliser un autre type de workflow comme un Gitflow ou un GitHub flow pour minimiser les risques de conflits.
- Lorsque vous voulez une séparation des modifications pour faciliter la compréhension des modifications apportées.

Résumé des éléments à retenir :

- Le workflow basé sur le tronc est un type de workflow qui se base sur l'utilisation d'une seule branche principale pour accueillir toutes les modifications.
- Il permet une intégration rapide des modifications, mais il y a des risques de conflits.
- Il est important de mettre en place des stratégies de validation des modifications pour minimiser les risques de conflits.
- Il est adapté pour les projets de petite ou moyenne taille ou pour les projets qui nécessitent une intégration rapide des modifications
- Il est plus simple à mettre en place et à comprendre que d'autres types de workflow.
- Il n'y a pas de séparation des modifications, ce qui peut rendre difficile de comprendre les modifications apportées.

Fork based workflow

Qu'est-ce qu'un fork-based workflow ?

Un fork-based workflow est un type de workflow utilisé lorsque vous travaillez sur un projet open-source ou lorsque vous voulez utiliser le code d'un autre projet dans le vôtre, sans pour autant le modifier directement. Il se base sur l'utilisation de "forks" (ou copies) de dépôts distants.

Qu'est-ce qu'un fork-based workflow ?

Avec un fork-based workflow, vous allez créer une copie (ou fork) du dépôt distant sur lequel vous voulez travailler. Vous pourrez alors y apporter des modifications et envoyer des "pull requests" pour intégrer vos modifications dans le dépôt original. Cela permet aux mainteneurs du dépôt d'accepter ou de refuser les modifications proposées.

Il est important de noter que ce type de workflow est plus adapté aux projets open-source ou aux projets qui nécessitent une validation avant intégration des modifications.

Avantages

- Séparation des modifications : Vous pouvez effectuer vos modifications sur votre fork sans affecter le dépôt original.
- Contrôle des modifications : Les mainteneurs du dépôt peuvent valider les modifications avant de les intégrer dans le dépôt original.

Inconvénients

- Complexité supplémentaire : Ce type de workflow est plus complexe que les autres, il nécessite une bonne compréhension des notions de fork et de pull request.
- Temps supplémentaire : Les modifications doivent être approuvées avant d'être intégrées, il faut donc prévoir un temps supplémentaire pour cette validation.

Quand utiliser un fork-based workflow ?

- Lorsque vous travaillez sur un projet open-source ou lorsque vous voulez utiliser le code d'un autre projet dans le vôtre sans le modifier directement.
- Lorsque vous voulez assurer un niveau de qualité élevé en validant les modifications avant de les intégrer dans le dépôt original.

Quand éviter un fork-based workflow ?

- Lorsque vous travaillez sur un projet personnel ou dans une équipe fermée, il est préférable d'utiliser un autre type de workflow comme un Gitflow ou un GitHub flow.
- Lorsque vous voulez éviter la complexité supplémentaire liée à la gestion des forks et des pull requests.

Ce qu'il y a à retenir :

- Le fork-based workflow est un type de workflow utilisé lorsque vous travaillez sur un projet open-source ou lorsque vous voulez utiliser le code d'un autre projet dans le vôtre, sans pour autant le modifier directement.
- Il se base sur l'utilisation de "forks" (ou copies) de dépôts distants et permet de séparer les modifications et de les valider avant de les intégrer dans le dépôt original.
- Ce type de workflow est plus adapté aux projets open-source ou aux projets qui nécessitent une validation avant intégration des modifications.
- Il nécessite une bonne compréhension des notions de fork et de pull request, et peut être plus complexe et prendre plus de temps que d'autres types de workflow.