GIT
Les fondamentaux

Table des matières

Qu'est-ce que GIT ?

Git est un système de contrôle de version décentralisé (DVCS) largement utilisé pour le suivi des modifications apportées à un ensemble de fichiers, généralement dans le cadre du développement de logiciels. Il a été créé par Linus Torvalds en 2005 et est devenu un élément essentiel du développement de logiciels collaboratifs.

Git permet aux développeurs de travailler ensemble sur des projets, de suivre les modifications apportées à un code source au fil du temps, de gérer différentes branches de développement, de fusionner des modifications de manière fluide et de gérer les conflits de fusion. Il assure également la sauvegarde de l'historique complet des modifications, ce qui facilite le suivi des bugs et la réversion des changements indésirables.

Parmi les principales caractéristiques de Git, on trouve la possibilité de travailler hors ligne, la gestion efficace des branches, la distribution décentralisée (ce qui signifie que chaque utilisateur dispose d'une copie complète de l'historique du projet), la rapidité et la flexibilité.

Des plates-formes telles que GitHub, GitLab et Bitbucket fournissent des services d'hébergement Git basés sur le web, ce qui facilite la collaboration et le partage de code entre développeurs.

Vidéo d'introduction

1. Résumé

Dans cette vidéo, vous allez apprendre qu'est-ce que Git, comment il fonctionne et comment vous en servir facilement dans vos différents projets informatiques.

La première question qu'on est en droit de se poser est : qu'est-ce que Git?
C'est un outil permettant de contrôler la ou les versions d'un projet informatique.

Prenons l'exemple de Hugo, un jeune étudiant qui doit réaliser un site web pour son projet. Sans Git, il rencontre des problèmes de collaboration avec son camarade Michael, comme la perte de travail ou l'attente de mises à jour.

Avec Git, Hugo crée un dépôt sur GitHub et utilise des commandes comme `git clone`, `git add`, `git commit` et `git push` pour gérer son code.

Lorsque Michael rejoint le projet, ils utilisent Git pour collaborer efficacement.

Une "branche" est une version du projet.
La branche principale est généralement appelée "master".

Pour éviter les conflits, Hugo et Michael créent des branches séparées pour différentes fonctionnalités. Ils utilisent `git merge` pour fusionner leurs travaux.

La clé de la collaboration réussie avec Git est la communication.

Les équipes doivent discuter de la répartition des tâches et éviter de travailler sur les mêmes fonctionnalités en même temps.

Git est un outil puissant pour la collaboration et le contrôle de version.

Bien utilisé, il peut simplifier le processus de développement et éviter de nombreux problèmes.

2. Points importants :

1. Git permet de suivre et de contrôler les versions d'un projet.
2. Les branches permettent de travailler sur différentes fonctionnalités en parallèle.
3. Il est essentiel de communiquer avec les membres de l'équipe lors de l'utilisation de Git.
4. La branche "master" est la version principale du projet.

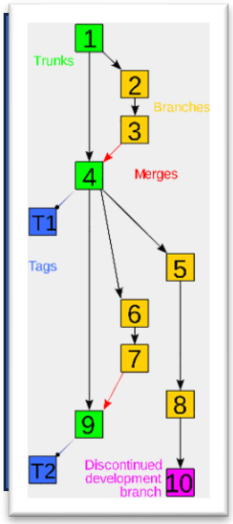
3. Concepts clés :

1. **Git** : Un outil de contrôle de version.
2. **GitHub** : Une plateforme d'hébergement pour les projets Git.
3. **Branche** : Une ligne de vie du projet qui enregistre les différentes versions.
4. **Commit** : Un ensemble de modifications empaquetées.
5. **Push** : Envoyer des modifications à un dépôt distant.
6. **Pull** : Récupérer des modifications d'un dépôt distant.
7. **Merge** : Fusionner deux branches.

1. Introduction

1. Historique de Git :

Émergence des systèmes de contrôle de version :



Au fur et à mesure que les projets logiciels devenaient plus complexes et impliquaient un plus grand nombre de développeurs, est née la nécessité d'une méthode structurée pour suivre et gérer les changements apportés au code source. Les systèmes de contrôle de version ont été développés en réponse à cette exigence. Ces systèmes permettent non seulement de suivre les modifications apportées à chaque fichier, mais aussi de revenir à n'importe quelle version précédente, offrant ainsi une grande sécurité et flexibilité aux développeurs.

Création de Git :

L'histoire de Git commence avec un personnage bien connu de la communauté technologique : Linus Torvalds. Après avoir créé le noyau Linux, Torvalds s'est retrouvé confronté à des défis uniques en matière de gestion des contributions de milliers de développeurs du monde entier. En 2005, en réponse à des besoins spécifiques et à des désaccords avec les systèmes de contrôle de version existants, il a conçu Git. Plus qu'un simple outil, Git a été pensé pour répondre aux exigences de vitesse, d'efficacité et de fiabilité du projet Linux.

Popularité de Git :

Il n'a pas fallu longtemps après sa création pour que Git gagne en popularité. Sa conception distribuée, permettant à chaque développeur de disposer d'une copie complète de l'historique du code, l'a distingué des autres systèmes. De plus, sa performance, sa flexibilité et sa robustesse en ont fait le système de contrôle de version de choix pour de nombreux développeurs et entreprises à travers le monde. Aujourd'hui, Git est largement reconnu et adopté, servant de fondement à de nombreuses plateformes populaires comme GitHub, GitLab, et Bitbucket.

2. Pourquoi utiliser un système de contrôle de version ?

La gestion de projets, en particulier ceux qui s'étendent sur de longues périodes avec plusieurs participants, peut être une tâche complexe. Les systèmes de contrôle de version, tels que Git, offrent des solutions essentielles à ces défis.

Traçabilité

Au cœur de tout projet, la question de la transparence et de la responsabilité est primordiale. Grâce à Git, chaque modification, chaque ligne de code ajoutée ou supprimée est soigneusement enregistrée. Cela donne aux équipes la possibilité non seulement de voir ce qui a été modifié, mais aussi par qui et pour quelle raison. La traçabilité renforce la confiance au sein des équipes et garantit que chaque changement peut être examiné et compris.

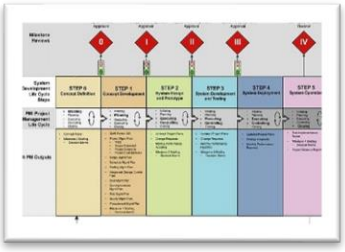
Collaboration

Dans le monde du développement logiciel, la collaboration est la clé. Git facilite le travail collectif en permettant à plusieurs individus d'apporter des contributions simultanées à un projet. Au lieu de risquer d'écraser le travail des autres, les systèmes de contrôle de version comme Git utilisent des branches pour que chaque développeur puisse travailler en parallèle. Lorsque leur travail est terminé, ces branches peuvent être fusionnées, assurant que toutes les contributions sont incluses et intactes.

Sauvegarde et Restauration

Tout développeur sait qu'une erreur peut survenir à tout moment. Qu'il s'agisse d'une simple faute de frappe ou d'un bug complexe, le besoin de revenir à une version précédente du projet est souvent nécessaire. Avec Git, chaque état du projet est enregistré. Ainsi, si une erreur est détectée, il est possible de revenir à un état antérieur, éliminant le risque de perdre un travail précieux.

Gestion des versions



Au fur et à mesure que les projets évoluent, il est essentiel de pouvoir identifier des étapes spécifiques. Que ce soit pour le lancement d'une nouvelle fonctionnalité ou la correction d'un bug, la capacité de marquer des versions spécifiques garantit que les équipes peuvent référencer, déployer ou revenir à des étapes déterminées avec confiance et clarté.

3. Principales différences entre Git et d'autres systèmes de contrôle de version.

Distribué vs. Centralisé :

Contrairement à d'autres systèmes comme SVN, chaque utilisateur de Git a une copie complète de l'historique du projet, ce qui offre une plus grande flexibilité et résilience.

Au cœur de l'évolution du contrôle de version se trouve la distinction entre les systèmes distribués et centralisés. Là où des systèmes comme SVN (Subversion) adoptent une approche centralisée, nécessitant un serveur principal pour stocker l'ensemble de l'historique, Git opère sur une base distribuée. Cela signifie que chaque utilisateur de Git détient une copie complète de l'historique du projet directement sur sa machine.

Cette structure distribuée non seulement accroît la résilience (si un dépôt est perdu, n'importe quelle copie peut le restaurer), mais offre également une flexibilité incomparable. Les utilisateurs peuvent travailler hors ligne, faire des commits locaux, et synchroniser leurs changements avec le dépôt central une fois en ligne.

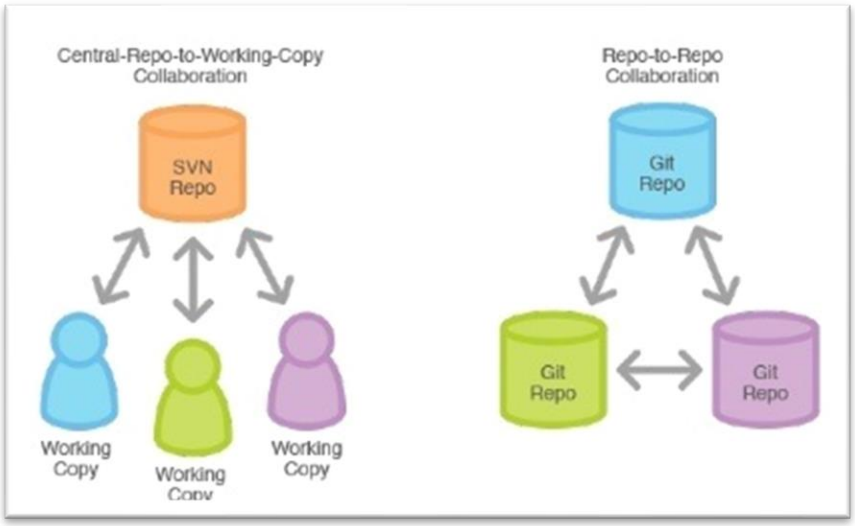


Figure 1 : https://www.quora.com/What-are-the-pros-and-cons-of-using-branches-in-Git-vs-SVN

Intégrité des données :

Git utilise une cryptographie pour garantir l'intégrité et la consistance du code source.
Grâce à la cryptographie, chaque commit ou ensemble de modifications est associé à une empreinte digitale unique, garantissant ainsi l'intégrité et la consistance du code source à chaque étape.

Performance :

La performance est également un point fort de Git. Conçu pour traiter efficacement des opérations telles que la fusion, le branchement et le clonage, il surpasse souvent d'autres systèmes de contrôle de version en termes de rapidité et d'efficacité, surtout dans des projets de grande envergure.

Flexibilité :

Enfin, l'une des raisons pour lesquelles Git est devenu le choix prédominant pour tant de développeurs est sa flexibilité. Au lieu de contraindre les utilisateurs à suivre un modèle unique, Git embrasse et facilite diverses

approches, qu'il s'agisse de développement linéaire, de branches spécifiques à des fonctionnalités, ou d'autres méthodologies complexes. Cette capacité à s'adapter à différents flux de travail le rend non seulement puissant, mais aussi incroyablement polyvalent.

4. Conclusion

Au fil des années, de nombreux outils ont été conçus pour aider les développeurs à gérer leurs codes. Parmi eux, Git s'est démarqué, offrant une solution robuste aux défis persistants de la collaboration et du suivi des modifications.
Git est un outil puissant qui a révolutionné la manière dont les développeurs collaborent et gèrent leurs projets.

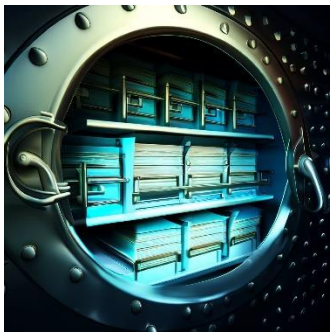
Sa conception permet une traçabilité détaillée des changements, offrant ainsi une vue transparente de l'évolution du projet. Au-delà du travail en équipe, même pour ceux qui travaillent en solo, Git est une outil essentielle, guidant les développeurs à travers leurs propres changements et réflexions. **Que vous travailliez en solo ou en équipe, comprendre Git est essentiel pour garantir une collaboration fluide et une gestion efficace du code.**

Maintenant que nous avons une compréhension de base de ce qu'est Git et de son importance, plongeons-nous dans ses concepts fondamentaux pour mieux appréhender son fonctionnement.

2. Concepts fondamentaux de Git

1. Définitions : dépôt (repository), commit, branche, fusion (merge), fork, clone.

Définitions essentielles :



Dépôt (Repository) : Le dépôt est le cœur d'un projet sous Git. Il agit comme une base de données contenant tous les fichiers de votre projet ainsi que l'historique complet de toutes les modifications effectuées depuis sa création. Chaque projet géré avec Git est stocké dans son propre dépôt, garantissant ainsi une traçabilité et une gestion optimale des versions du projet.

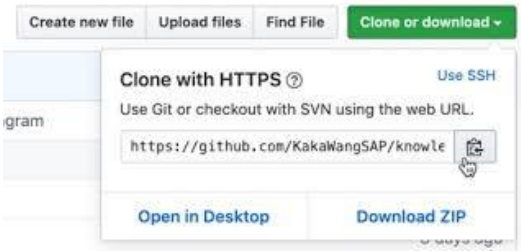
Commit : Lorsque vous voulez enregistrer l'état actuel de votre projet, vous créez un "commit". Pensez-y comme à une capture instantanée, ou une photographie, de votre projet à ce moment précis. Chaque commit est unique et est identifié par un code SHA. Lors de la création d'un commit, il est essentiel d'y associer un message descriptif pour détailler les changements ou les ajouts effectués.

Branche : Visualisez l'évolution de votre projet comme une ligne temporelle. Dans cette perspective, une branche est une version dérivée de votre projet principal, permettant de développer, tester ou corriger certaines parties sans perturber la version principale, souvent dénommée **master** ou **main**.

Fusion (Merge) : Lorsque le travail sur une branche est terminé et validé, il est souvent nécessaire de le réintégrer à la version principale ou à une autre branche. Cette action est appelée "fusion". Par exemple, une fonctionnalité développée sur une branche dédiée sera fusionnée avec la branche principale une fois achevée.

Fork : Sur des plateformes collaboratives comme GitHub, "forker" signifie créer une copie personnelle d'un dépôt qui appartient à une autre personne ou organisation. Cette action permet d'apporter ses propres modifications à un projet sans impacter le dépôt original.

Clone : Cloner est l'action de télécharger localement un dépôt existant. Cela vous permet d'avoir une copie du projet sur votre machine, de travailler dessus, puis éventuellement de synchroniser vos modifications avec le dépôt d'origine.



2. Cycle de vie d'un fichier dans Git.

L'un des avantages de Git est sa capacité à suivre avec précision l'évolution des fichiers au fil du temps. Cette traçabilité est rendue possible grâce à une série d'états que chaque fichier peut traverser. Ces états aident les développeurs à comprendre exactement où en est chaque fichier dans le processus de développement.

Les trois états principaux d'un fichier :

1. **Modifié (Modified) :**

- À ce stade, vous avez apporté des modifications à un fichier qui est déjà sous le contrôle de version de Git. Cependant, ces modifications n'ont pas encore été "présentées" pour être incluses dans le prochain commit.
- Par exemple, si vous corrigez un bug ou ajoutez une nouvelle fonction à un script, ce fichier est considéré comme modifié.

2. **Indexé (Staged) :**

- Une fois que vous êtes satisfait des modifications apportées à votre fichier, vous pouvez le "mettre en scène" – stage-. Cela signifie que vous avez marqué le fichier modifié pour qu'il soit inclus dans le prochain commit.

- L'étape de mise en scène vous donne une grande flexibilité. Vous pouvez choisir d'indexer seulement certains fichiers et laisser d'autres de côté pour un prochain commit.

3. **Validé (Committed) :**

- À ce stade, vous avez confirmé toutes les modifications que vous aviez mises en scène. Ces modifications sont enregistrées dans la base de données de Git, et le fichier revient à un état non modifié jusqu'à ce que vous apportiez de nouvelles modifications.
- Le commit représente une capture instantanée de votre projet, permettant ainsi de revenir à cet état précis si nécessaire.

Processus général de gestion des fichiers :

1. **Modification des fichiers**

Tout commence lorsque vous apportez des modifications à un ou plusieurs fichiers dans votre répertoire de travail. À ce moment-là, Git reconnaît que ces fichiers ont changé, mais ne les a pas encore enregistrés.

2. **Indexation des modifications**

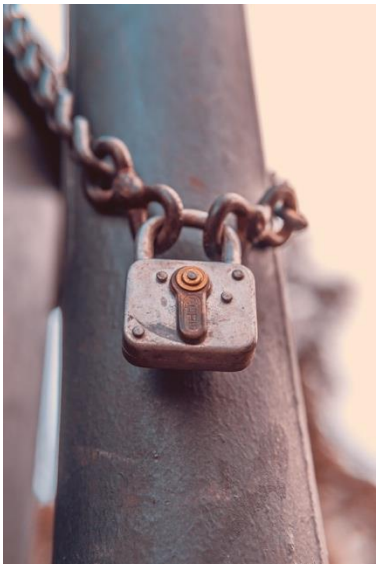
En utilisant la commande `git add`, vous ajoutez vos modifications à une zone intermédiaire appelée "zone d'index" ou "staging area". Cette étape vous permet de préparer et d'organiser vos modifications avant de les valider définitivement.

3. **Validation des modifications**

Une fois que vous avez soigneusement organisé vos modifications et que vous êtes prêt à les enregistrer, vous utilisez la commande `git commit`. Cela crée un nouvel instantané de votre projet avec un message descriptif que vous fournissez, capturant ainsi toutes les modifications que vous avez mises en scène.

En comprenant le cycle de vie d'un fichier dans Git, vous gagnez en précision et en efficacité dans la gestion de vos projets. Cela permet non seulement de suivre les modifications, mais aussi d'organiser et de contrôler avec soin les évolutions du code.

3. La structure d'un commit.



La structure d'un commit :

1. Un identifiant unique (SHA)

Définition : SHA signifie "Secure Hash Algorithm". Dans le contexte de Git, il s'agit d'une chaîne de caractères générée automatiquement pour chaque commit, assurant son unicité.

Importance : Cet identifiant permet à Git de retracer et de différencier chaque commit. Si deux commits ont le même SHA (ce qui est extrêmement rare), cela signifierait qu'ils ont exactement le même contenu, le même parent, la même date, le même auteur, etc.

Exemple : Un SHA ressemble à `e83c5163316f89bfbde7d9ab23ca2e25604af290`.

2. Le ou les auteurs

Définition : Il s'agit des personnes qui ont contribué à ce commit spécifique. Dans la plupart des cas, il n'y a qu'un seul auteur, mais Git permet d'attribuer un commit à plusieurs auteurs si nécessaire.

Importance : Cela permet de savoir qui a fait quoi, facilitant la collaboration et le suivi des contributions.

Exemple : Un auteur est généralement représenté par son nom complet et son adresse e-mail, comme "Jean Dupont jean.dupont@email.com".

3. La date

Définition : La date et l'heure exactes auxquelles le commit a été créé.

Importance : Cette information aide à retracer l'évolution chronologique d'un projet et à comprendre quand certaines modifications ont été apportées.

Exemple : `Tue Mar 15 17:45:28 2023 +0200.`

4. Un message associé :

Définition : Une description courte et significative des modifications apportées lors de ce commit.

Importance : Le message guide les autres contributeurs (et vous-même dans le futur) pour comprendre rapidement les raisons et le contexte des modifications.

Conseils :

- Commencez le message par une ligne de résumé courte (moins de 50 caractères).
- Fournissez une description plus détaillée en dessous si nécessaire.
- Utilisez le temps présent ("ajoute une fonctionnalité" plutôt que "a ajouté une fonctionnalité").

5. Un lien vers le ou les commits parents

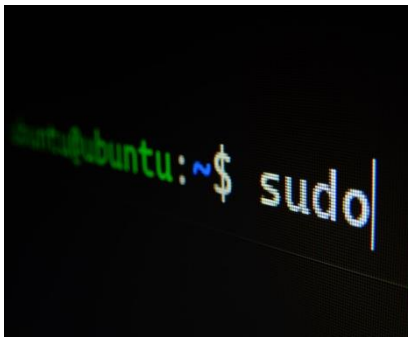
Définition : Chaque commit, à l'exception du tout premier commit d'un dépôt, a un ou plusieurs commits "parents". Cela représente l'historique direct précédant le commit actuel.

Importance : Ces liens permettent de créer la chaîne de l'historique du projet, offrant la possibilité de naviguer à travers les différentes étapes de son évolution.

Exemple : En visualisant l'historique avec des outils comme `git log`, vous pouvez voir les relations entre les commits et suivre les liens vers leurs parents.
Avec cette structure, chaque commit est un instantané complet de l'état d'un projet à un moment donné, tout en conservant le contexte et l'historique nécessaires pour comprendre et naviguer dans l'évolution du projet.

3. Premiers pas avec Git

a. Installation de Git



Pour les utilisateurs Windows : Téléchargez et installez le logiciel à partir du site officiel. L'installation ajoutera également une interface en ligne de commande appelée "Git ".

Pour les utilisateurs macOS : sur les dernières versions, GIT est déjà installé.

Pour les utilisateurs Linux : Utilisez votre gestionnaire de paquets, par exemple sur Debian/Ubuntu, utilisez `sudo apt-get install git.`

Une fois installé, vous pouvez vérifier la version de Git avec la commande : `git --version.`

b. Configuration initiale

Avant de commencer à utiliser Git, il est essentiel de se présenter à Git via deux configurations principales :

- **Nom** : Identifiez-vous avec votre nom : `git config --global user.name "Votre Nom"`
- **Email** : Identifiez-vous avec votre email : `git config --global user.email "votre.email@example.com"`

Il est également utile de configurer un éditeur par défaut pour écrire des messages de commit ou résoudre des conflits. Par exemple, pour utiliser nano :

```
git config --global core.editor nano
```

c. Création d'un nouveau dépôt

Un dépôt (ou "repository") est un espace de stockage pour un projet. Il contient tous les fichiers du projet ainsi que l'historique des modifications.

Pour initialiser un nouveau dépôt :

- Naviguez vers le répertoire de votre projet : `cd chemin/vers/votre/projet`
- Exécutez la commande : `git init MonPremierDepot`
- Une fois cette commande exécutée, un nouveau sous-répertoire .git est créé. C'est ici que Git stocke toutes les métadonnées et l'historique du projet.

d. Ajout de fichiers et premier commit

Un "commit" est une capture instantanée de l'état actuel des fichiers du projet.

Ajout de fichiers à la zone de préparation :
Pour ajouter des fichiers en vue d'un commit, utilisez : `git add nom_du_fichier` ou pour ajouter tous les fichiers : `git add .`

Statut des fichiers : Avant de faire un commit, vous pouvez voir quels fichiers ont été modifiés ou ajoutés avec la commande : `git status`.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git add demofile

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   demofile
```

Faire un commit : Une fois que vous avez ajouté tous les fichiers souhaités, vous pouvez faire un commit : `git commit -m "Message décrivant les modifications effectuées"`
Astuce : Un bon message de commit est concis (moins de 50 caractères pour le titre, suivi d'une explication plus détaillée si nécessaire) et décrit clairement ce que le commit fait.

```
$ git commit -m "modified"
[master 7e14679] modified
 1 file changed, 1 insertion(+)
```

Un bon message de commit décrit clairement les changements. Il permet à toute l'équipe de comprendre l'historique du projet. Pensez concis, pensez clair !

4. Travail en solo avec Git

Figure 2 : ajout d'un message qui ne sert à rien

a. Les commandes essentielles : `git status`, `git add`, `git commit`, `git log`.

git status : Affiche l'état actuel de votre espace de travail. Il montre quels fichiers ont été modifiés, quels fichiers sont en attente d'être commités et quels fichiers ne sont pas encore suivis par Git.
Exemple : Imaginons que vous ayez modifié un fichier appelé "index.html". Après avoir fait cette modification, tapez `git status` dans le terminal. Vous verrez quelque chose comme : `modified: index.html`.

git add : Ajoute des modifications dans votre espace de travail à la zone de préparation (aussi appelée "staging area"). C'est une étape intermédiaire avant de valider ces modifications dans un commit.
Exemple : Pour ajouter notre fichier "index.html" à la zone de préparation, tapez : `git add index.html`.

git commit : Enregistre vos modifications dans l'historique du dépôt. Chaque commit est un instantané de votre code à un moment donné.
Exemple : Après avoir ajouté votre fichier à la zone de préparation, tapez : `git commit -m "Description de mes modifications".` La partie `-m` permet d'ajouter un message à votre commit.

git log : Affiche l'historique des commits. Cela vous donne une vue d'ensemble des modifications apportées au fil du temps.
Exemple : Tapez simplement `git log` pour voir tous les commits précédents, avec leur auteur, date et message.

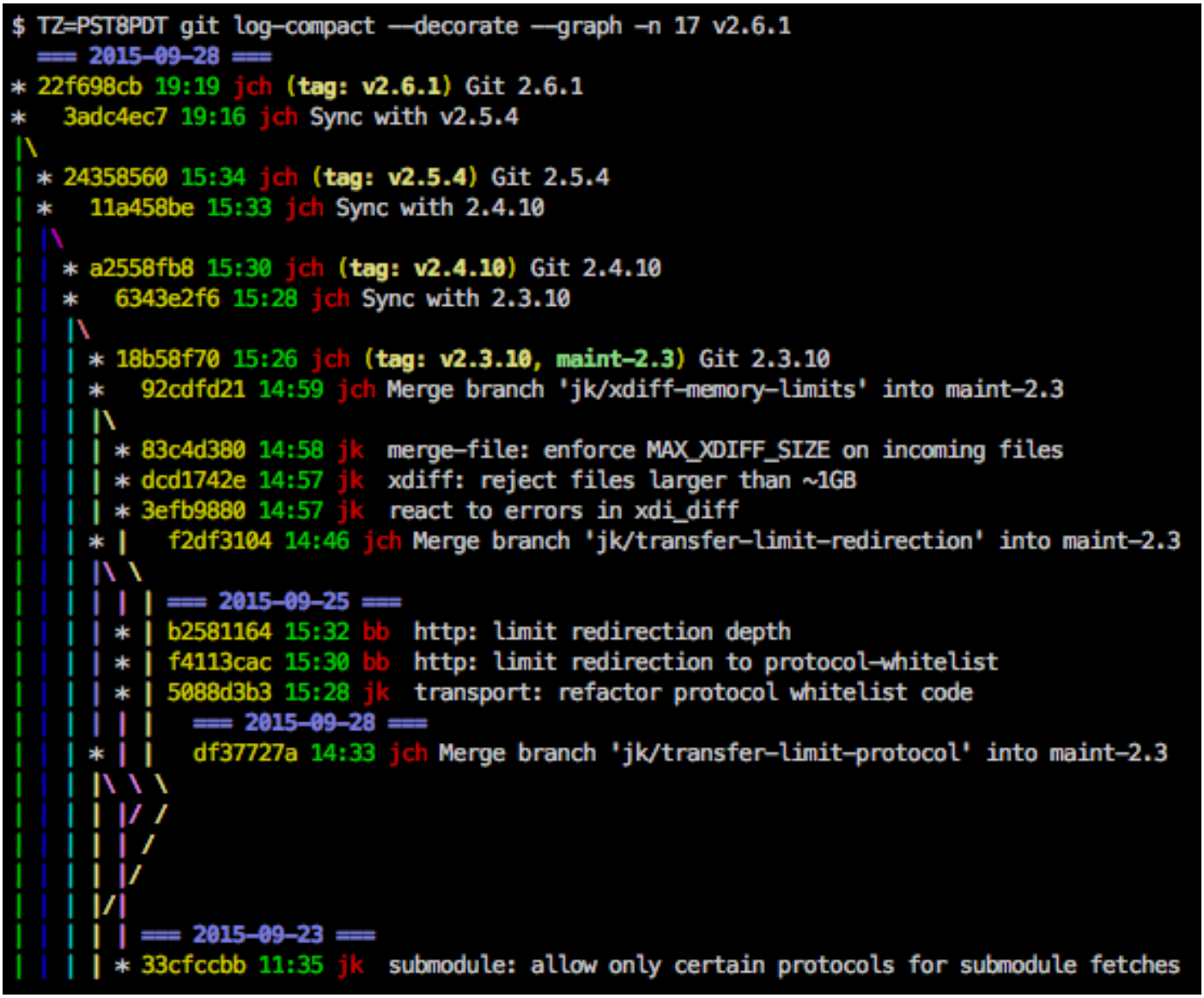


Figure 3 https://mackyle.github.io/git-log-compact/

b) Explorer l'historique

Naviguer parmi les commits :

Chaque commit a un identifiant unique appelé "hash". Avec cet identifiant, vous pouvez revenir à n'importe quel commit.

Exemple : Si vous souhaitez voir le code à un commit spécifique, utilisez la commande `git checkout [hash_du_commit].`

Annuler des changements :

Il y a deux principales commandes pour cela :

git checkout : Permet de revenir à un état précédent d'un fichier.

Exemple : Si vous voulez annuler les modifications apportées à "index.html", tapez : `git checkout -- index.html.`

git reset : Revertit les modifications jusqu'à un certain commit. Il y a trois principales options avec `git reset: soft, mixed et hard.`

Exemple : Pour annuler le dernier commit mais garder les modifications dans votre espace de travail, utilisez : `git reset --mixed HEAD~1.`

c) Utiliser des branches

Les branches sont essentiellement des pointeurs vers des commits. Elles sont utiles pour développer des fonctionnalités isolément les unes des autres.

git branch : Affiche toutes les branches de votre dépôt. La branche actuelle sera indiquée par une étoile à côté.

Exemple : Tapez simplement `git branch.`

git checkout : Permet de se positionner sur une branche

Exemple : Pour aller sur la branche *main* tapez simplement `git checkout main`

git checkout -b : Crée une nouvelle branche et y bascule.

Exemple : Pour créer une nouvelle branche appelée "nouvelle-fonctionnalite", tapez : `git checkout -b nouvelle-fonctionnalite`

Cette commande revient à faire un `git branch.` puis un `git checkout` sur la branche voulue

git merge : Fusionne une branche dans une autre.

Exemple : Si vous êtes sur la branche "master" et que vous souhaitez y fusionner "nouvelle-fonctionnalite", tapez : `git merge nouvelle-fonctionnalite`.

Conclusion :

Travailler en solo avec Git est essentiellement un jeu d'équilibriste entre la création de commits clairs, l'exploration de l'historique et la gestion de branches pour organiser votre travail. Avec la maîtrise de ces commandes, vous serez bien équipé pour gérer votre code de manière efficace et structurée

5. Travailler en équipe avec Git

a. Introduction à GitHub, GitLab, et Bitbucket

Quand on parle de Git, il est presque impossible de ne pas mentionner les plateformes qui permettent aux équipes de collaborer en utilisant Git. Voici les trois les plus populaires :

	**GitHub** : C'est probablement la plateforme la plus connue. GitHub offre une interface conviviale, des intégrations d'outils tiers, et héberge de nombreux projets open-source. Depuis 2018, il appartient à Microsoft.

	**GitLab** : Contrairement à GitHub, GitLab offre à la fois des solutions hébergées et auto-hébergées. Il possède également une suite d'outils CI/CD intégrés, ce qui le rend populaire pour le déploiement continu
	**Bitbucket** Propriété d'Atlassian, cette plateforme est souvent utilisée par des équipes qui utilisent également Jira, Trello ou d'autres outils Atlassian

b. Cloner un dépôt

Le clonage vous permet de copier un dépôt existant sur votre machine locale. C'est la première étape pour commencer à collaborer sur un projet.

Pour cloner un dépôt :

```
git clone [URL_DU_DEPOT]
```

Cela crée un dossier avec le nom du dépôt dans votre répertoire actuel, contenant tout le code et l'historique du dépôt.

c. Travailler avec des remotes

Un "remote" est une version de votre dépôt qui est hébergée sur Internet ou sur un réseau.

Ajouter un remote : Si vous avez initialement cloné un dépôt, le remote d'origine (souvent appelé "origin") est automatiquement ajouté. Sinon, vous pouvez l'ajouter avec :

```
git remote add [NOM_DU_REMOTE] [URL_DU_REMOTE]
```

Pousser des changements (push) : Une fois que vous avez fait des commits en local, vous pouvez les "pousser" vers un remote pour que d'autres puissent les voir et y travailler.

```
git push [NOM_DU_REMOTE] [BRANCHE]
```

Tirer des changements (pull) : Pour obtenir les derniers changements d'une branche depuis un remote :

```
git pull [NOM_DU_REMOTE] [BRANCHE]
```

d. Gérer les conflits

Quand ?

Un conflit survient lorsque deux personnes modifient la même partie d'un fichier et que Git ne sait pas quelle modification garder.

Lors d'un push ou d'un merge, il peut avoir un ou des conflits.

Par exemple sur un git merge dev :

```
PremierProjet % git merge dev
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
PremierProjet %
```

GIT vous indique qu'il y a un conflit, il est nécessaire de le corriger manuellement.
Il indique également qu'à la fin il faut faire un commit.

Voici comment résoudre les conflits :

1. **Identifier les conflits** : Exécutez `git status`. Les fichiers avec des conflits seront répertoriés.
2. **Ouvrir les fichiers et résoudre les conflits** : Git marque les conflits dans le fichier avec des `<<<<<<<`, `=======` et `>>>>>>>`. Supprimez ces marques et décidez quelles modifications conserver.
3. **Valider les changements résolus** : Une fois les conflits résolus, ajoutez les fichiers avec `git add` et faites un nouveau commit.

Il est toujours préférable de communiquer avec l'équipe pour résoudre les conflits. Connaître l'intention derrière chaque changement facilite grandement la résolution des problèmes.

6. Bonnes pratiques et astuces

a. Commit souvent, mais avec des messages clairs

Principe : L'un des avantages de Git est qu'il permet d'enregistrer des "instantanés" de votre travail à différents moments. Ces instantanés sont appelés "commits".

Bonnes pratiques :

- Faites des commits régulièrement pour sauvegarder votre progression.
- Chaque commit devrait représenter une unité logique de travail.
- Évitez les commits trop volumineux qui englobent de multiples modifications sans relation.

Astuces pour de bons messages de commit :

- Commencez par une ligne résumant en 50 caractères ou moins.

- Si nécessaire, fournissez une description plus détaillée après une ligne vide.
- Décrivez ce que vous avez fait et pourquoi.

b. Utiliser des branches pour les nouvelles fonctionnalités

Principe : Les branches sont comme des chemins parallèles où vous pouvez développer des fonctionnalités ou corriger des bugs sans affecter la branche principale (souvent appelée "master" ou "main").

Bonnes pratiques :

- Créez une nouvelle branche pour chaque nouvelle fonctionnalité ou correction de bug.
- Cela permet de développer en isolation, rendant la fusion plus facile plus tard.

Commande utile :

```
git checkout -b [nom_de_la_branche]
```

c. La règle d'or : ne jamais réécrire l'histoire publique

Principe : Git permet de réécrire l'historique des commits, mais cela peut être dangereux dans un environnement collaboratif.

Bonnes pratiques :

- Ne réécrivez jamais l'historique de branches que d'autres personnes utilisent.
- Cela peut causer des désynchronisations et des confusions pour les autres collaborateurs.

Rappel : Les commandes comme `git rebase` ou `git push --force` peuvent modifier l'historique. Utilisez-les avec prudence!

d. Utiliser git stash pour sauvegarder des changements temporaires

Principe : Parfois, vous avez des modifications en cours dont vous n'êtes pas encore prêt à commiter, mais vous devez changer de tâche ou de branche. `git stash` est là pour ça!

Comment cela fonctionne :

- `git stash` met vos modifications en attente, vous laissant avec un répertoire propre.
- Une fois que vous êtes prêt à récupérer ces modifications, utilisez `git stash apply`.

Bonnes pratiques :

- Utilisez `git stash list` pour voir tous vos stashes.
- N'oubliez pas de "dé-stasher" lorsque vous revenez à votre tâche initiale.