VERSIONNING

1. Historique des systèmes de versionnement

a. Avant Git : Les systèmes de contrôle de version centralisés

- **RCS (Revision Control System)** : Un des premiers systèmes (1982), permettant de suivre les modifications sur des fichiers uniques.
- **CVS (Concurrent Versions System)** : Évolution de RCS, introduisant une gestion de projets multi-utilisateurs (1986).
- **Subversion (SVN)** : Une amélioration de CVS avec des fonctionnalités modernes comme le commit atomique (2000).

b. Apparition des systèmes distribués

- Les limitations des systèmes centralisés (p. ex. dépendance à un serveur central) ont conduit au développement de systèmes distribués :
    - **BitKeeper** : L'un des premiers systèmes distribués, utilisé par le projet Linux avant Git.
    - **Git (2005)** : Créé par Linus Torvalds pour gérer le développement rapide et collaboratif du noyau Linux.

c. Pourquoi Git est-il révolutionnaire ?

- Conception distribuée : Chaque utilisateur possède une copie complète du dépôt.
- Vitesse : Git est optimisé pour des projets volumineux.
- Gestion avancée des branches : Intégration et collaboration simplifiées.
- Sécurité : Les données et historiques sont cryptographiquement protégés.

2. Concepts de Git

a. Organisation d'un repository Git

1. **Dépôt (Repository)** :
    - Dossier contenant tous les fichiers d'un projet ainsi que leurs historiques.
    - Composé d'une section de travail (working directory), d'une zone de transit (staging area), et d'un répertoire Git (.git).
2. **Branche** :
    - Une branche est une série de commits isolés. La branche main (ou master) est souvent la branche principale.
    - Utilisées pour travailler sur différentes fonctionnalités ou correctifs sans perturber la branche principale.

b. Modèle de stockage

1. **Snapshots au lieu de différences** :

    - Git ne stocke pas de différences entre les versions des fichiers (comme SVN) mais prend un instantané complet des fichiers à chaque commit.
    - Les fichiers non modifiés pointent vers les mêmes données.
2. **Objets Git** :
    - **Blob** : Contient les données des fichiers.
    - **Tree** : Représente une structure de répertoire.
    - **Commit** : Pointe vers un arbre, identifie un instantané et contient des métadonnées (message, auteur, timestamp).
3. **Hash SHA-1** :
    - Identifie de manière unique chaque objet dans Git.

c. Répertoires clés dans .git

1. **objects/** : Contient tous les objets Git (blobs, trees, commits).
2. **refs/** : Stocke les références aux branches et tags.
3. **HEAD** : Pointeur vers la branche actuellement active.

Analyse des avantages induits par Git

1. Travail local

L'une des forces principales de Git est son caractère **distribué**, permettant un travail local complet sans connexion à un serveur central.

- **Autonomie complète** :
    - Chaque développeur possède une copie complète du dépôt, y compris l'historique des versions.
    - Possibilité de consulter, modifier, committer, et expérimenter localement avant toute interaction avec les autres contributeurs.
- **Rapidité** :
    - Les opérations comme les commits, les différences (git diff), ou les consultations d'historique sont rapides, car elles se font localement, sans requêtes réseau.
- **Travail hors-ligne** :
    - Les développeurs peuvent travailler sans dépendre d'une connexion Internet, ce qui est crucial pour les zones à faible connectivité ou en déplacement.

2. Intégrité

Git garantit l'intégrité des données grâce à sa conception robuste.

- **SHA-1 pour le suivi des objets** :
    - Chaque objet (commit, fichier, etc.) est identifié par un hash SHA-1 unique, garantissant son immuabilité. Toute modification dans l'historique ou les fichiers est détectable immédiatement.
- **Système de sauvegarde intégré** :
    - Les données ne peuvent pas être perdues facilement grâce aux multiples copies (locale et distante).

- **Historique fiable** :
    - Les métadonnées (dates, auteurs, messages) et les relations entre les commits (graphe acyclique dirigé) assurent un suivi clair et précis des modifications.

Panorama des offres autour de Git

Git est utilisé à la fois en local et comme base pour des services collaboratifs.

1. Hébergement de dépôts Git

Ces services permettent de centraliser les dépôts tout en offrant des fonctionnalités collaboratives :

- **GitHub** :
    - Service leader pour les projets open-source et commerciaux.
    - Fonctionnalités : gestion de projets (issues, pull requests), CI/CD intégrée (CI/CD est un terme générique couvrant plusieurs phases DevOps. CI (intégration continue) est la pratique consistant à intégrer des modifications de code dans un dépôt plusieurs fois par jour.
    - CD a deux significations : la **livraison** continue automatise les intégrations de code, tandis que le **déploiement** continu publie automatiquement les versions finales aux utilisateurs finaux.) (GitHub Actions), sécurité (dependabot).
- **GitLab** :
    - Plateforme DevOps complète intégrant Git, CI/CD, et monitoring.
    - Offres open-source et payantes adaptées aux entreprises.
- **Bitbucket** (par Atlassian) :
    - Idéal pour les équipes utilisant l'écosystème Atlassian (Jira, Confluence).
    - Fort focus sur les entreprises et intégrations.

2. Clients Git

Pour simplifier l'utilisation de Git, surtout pour les débutants :

- **Interface graphique (GUI)** :
    - **GitHub Desktop** : Solution simple pour débutants, intégrée avec GitHub.
    - **Sourcetree** : Outil riche pour Windows et macOS, idéal pour visualiser les branches.
    - **GitKraken** : Client multiplateforme avec une interface intuitive et de puissants outils de gestion de branches.
- **Extensions d'IDE** :
    - **Visual Studio Code** : Intégration native avec Git pour suivre, committer, et pousser les modifications.
    - **JetBrains IDEs** (IntelliJ IDEA, PyCharm, etc.) : Intégrations Git avancées pour des workflows fluides.

3. Automatisation et collaboration

- **Outils de CI/CD** :
    - Intégrations avec Git pour déployer automatiquement les changements (ex. : GitHub Actions, GitLab CI, Jenkins).
- **Gestion des branches collaboratives** :
    - Fonctions avancées comme le **Git Flow**, facilitant le travail en équipe sur des branches bien définies.

Installation et mise en place de Git

L'installation de Git dépend de votre système d'exploitation.

Ci-dessous Les étapes détaillées pour installer Git à partir des sources et le configurer.

1. Prérequis

Avant d'installer Git à partir des sources :

- Assurez-vous que votre système dispose des outils nécessaires :
    - **Linux/Unix** : Installez les paquets gcc, make, curl, zlib, et d'autres bibliothèques mentionnées ci-dessous.
    - **Windows** : Utilisez un environnement comme MinGW ou WSL (Windows Subsystem for Linux).
- Vérifiez que votre système a une connexion Internet pour télécharger les sources.

2. Téléchargement des sources

1. Rendez-vous sur la page officielle de Git : https://git-scm.com.
2. Téléchargez la version la plus récente de Git en récupérant l'archive source ou en clonant le dépôt officiel :

```
git clone https://github.com/git/git.git
```

3. Compilation et installation

Pour Linux/Unix :

1. **Installez les dépendances nécessaires** :
   Sur les distributions basées sur Debian (comme Ubuntu), exécutez :

```
sudo apt update
sudo apt install gcc make libssl-dev libcurl4-gnutls-dev libexpat1-dev gettext zlib1g-dev
```

2. **Accédez au répertoire des sources** :

```
cd git
```

3. **Construisez Git à partir des sources** :

```
make prefix=/usr/local all
sudo make prefix=/usr/local install
```

4. **Vérifiez l'installation** :

```
git --version
```

Pour macOS :

1. Installez **Homebrew** si ce n'est pas encore fait :

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

2. Installez Git via Homebrew (recommandé pour macOS) :

```
brew install git
```

Pour Windows :

1. Téléchargez Git pour Windows depuis Git SCM Downloads.
2. Suivez l'assistant d'installation, en vous assurant de :
   - Sélectionner un éditeur par défaut (ex. : Vim ou Nano).
   - Configurer le chemin (PATH) pour inclure Git.
3. Ouvrez **Git Bash** pour commencer à utiliser Git.

4. Configuration initiale

1. **Configurer l'utilisateur** (nom et email) :

```
git config --global user.name "Votre Nom"
git config --global user.email "votre.email@example.com"
```

2. **Vérifiez les paramètres configurés** :

```
git config --list
```

3. **Définir l'éditeur par défaut** (facultatif) :
   - Pour Vim :

```
git config --global core.editor "vim"
```

   - Pour VS Code :

```
git config --global core.editor "code --wait"
```

4. **Configurer les couleurs pour plus de lisibilité** :

```
git config --global color.ui auto
```

5. Vérification finale

Testez Git en initialisant un dépôt local et en réalisant un premier commit :

```
mkdir test-git
cd test-git
git init
touch README.md
git add README.md
git commit -m "Premier commit"
```

Les différentes aides disponibles pour Git

Git offre de nombreuses aides pour apprendre et utiliser efficacement l'outil.

1. Aides intégrées dans Git

Git intègre des commandes de documentation et d'assistance accessibles directement en ligne de commande :

a. Commande d'aide générale

Pour afficher l'aide générale :

```
git help
```

b. Documentation pour une commande spécifique

Pour obtenir des détails sur une commande :

```
git help <commande>
# Exemple :
git help commit
```

- Utilisez aussi `--help` :

  ```
  git commit --help
  ```

c. Options pratiques

- **Raccourcis pour accéder à l'aide** :
  Appuyez sur `q` pour quitter la vue d'aide.
- **Documentation complète** :
  Vous pouvez lire la documentation Git en ligne via https://git-scm.com/docs.

2. Aides en ligne et communautés

a. Documentation officielle

- Le manuel officiel de Git est une ressource complète et gratuite : https://git-scm.com/book.

b. Forums et communautés

- **Stack Overflow** : Recherchez des solutions aux problèmes spécifiques ou posez vos questions.
- **Communautés GitHub et GitLab** : Les forums de ces plateformes offrent souvent des discussions riches et pertinentes.

c. Tutoriels interactifs

- **Git Immersion** : http://gitimmersion.com : Explorez Git à travers des exercices pratiques.
- **Learn Git Branching** : https://learngitbranching.js.org/ : Un simulateur visuel pour comprendre les branches.

d. Formation vidéo

- Les plateformes comme **YouTube**, **Udemy**, ou **LinkedIn Learning** proposent des tutoriels adaptés aux débutants et aux experts.

3. Création d'un repository Git

Un repository est l'espace dans lequel Git stocke vos fichiers et leur historique. Les étapes pour en créer un.

a. Création d'un repository local

1. **Initialisez un dépôt Git** dans un répertoire :

```
mkdir mon-projet
cd mon-projet
git init
```

   Cela crée un dossier .git qui contient toutes les données liées au suivi des versions.

2. **Ajoutez des fichiers au dépôt** :

```
echo "# Mon Projet" > README.md
git add README.md
```

3. **Effectuez un premier commit** :

```
git commit -m "Premier commit"
```

b. Création d'un repository distant

1. Créez un dépôt sur une plateforme comme **GitHub**, **GitLab**, ou **Bitbucket**.

2. Reliez votre dépôt local au dépôt distant :

```
git remote add origin https://github.com/votre-utilisateur/mon-projet.git
```

3. Poussez votre projet local vers le dépôt distant :

```
git push -u origin master
```

4. Aides graphiques

Pour ceux qui préfèrent une interface graphique :

- Utilisez **GitHub Desktop**, **Sourcetree**, ou **GitKraken**.
- Ces outils simplifient les actions comme les commits, pushes, pulls, et la gestion des branches.

1. Initialisation d'un dépôt local

Créez un nouveau projet avec Git :

```
# Créez un répertoire pour le projet
mkdir mon-projet
cd mon-projet

# Initialisez un dépôt Git
git init
```

- Vous verrez un message confirmant l'initialisation :
  Initialized empty Git repository in /path/to/mon-projet/.git
- Un dossier caché .git est créé, contenant les fichiers de configuration et l'historique.

2. Suivi et committement de fichiers

a. Ajouter un fichier et suivre ses modifications

1. Créez un fichier dans le projet :

```
echo "Bienvenue dans mon projet" > README.md
```

2. Ajoutez ce fichier à la zone de transit (staging area) :

```
git add README.md
```

3. Vérifiez l'état du dépôt :

```
git status
```

    - Le fichier devrait apparaître comme "prêt à être committé".

b. Effectuer un commit

1. Enregistrez les changements avec un message descriptif :

```
git commit -m "Ajout du fichier README.md"
```

2. Vérifiez l'historique des commits :

```
git log
```

3. Création et fusion de branches

a. Créez une nouvelle branche

1. Créez une branche appelée feature1 :

```
git branch feature1
```

2. Passez sur cette branche :

```
git checkout feature1
```

3. Vérifiez sur quelle branche vous travaillez :

```
git branch
```

b. Modifiez un fichier et commitez dans la nouvelle branche

1. Modifiez le fichier README.md :

```
echo "Ajout d'une fonctionnalité dans feature1" >> README.md
git add README.md
git commit -m "Ajout d'une fonctionnalité dans feature1"
```

c. Fusionnez la branche dans main

1. Retournez sur la branche principale :

```
git checkout main
```

2. Fusionnez la branche feature1 dans main :

```
git merge feature1
```

3. Vérifiez que les modifications sont présentes dans main :

```
cat README.md
```

4. Exploration de l'historique avec **git log**

1. Affichez l'historique complet :

```
git log
```

   - Vous verrez les commits avec leurs identifiants SHA-1, auteurs, dates et messages.
2. Utilisez des options pour un affichage simplifié :

```
git log --oneline --graph --all
```

   - Affiche un résumé graphique de l'historique, utile pour comprendre les branches et les fusions.

Explorer les commandes pour restaurer des modifications : **git checkout**, **git revert**, **git reset**

Ces trois commandes servent à restaurer, annuler ou revenir à des versions précédentes dans un dépôt Git.

1. **git checkout**

La commande git checkout permet de basculer entre les branches ou de restaurer des fichiers spécifiques. Elle peut être utilisée pour plusieurs actions :

a. Changer de branche

```
git checkout <nom-de-branche>
# Exemple : changer pour la branche 'feature1'
git checkout feature1
```

b. Restaurer un fichier spécifique

Si vous avez modifié un fichier dans votre répertoire de travail, mais que vous souhaitez revenir à la version la plus récente enregistrée dans le commit actuel :

```
git checkout -- <nom-du-fichier>
# Exemple : revenir à la version de 'README.md'
git checkout -- README.md
```

c. Revenir à un commit spécifique

Vous pouvez aussi revenir à un commit précis (en utilisant son identifiant SHA-1) pour tester ou examiner l'état du dépôt à un moment donné :

```
git checkout <sha1-du-commit>
# Exemple : revenir à un commit spécifique
git checkout 123abc4
```

2. git revert

La commande git revert permet de créer un commit inverse d'un commit précédent. Cela annule les modifications d'un commit, mais crée un nouveau commit dans l'historique. Elle est souvent utilisée dans un contexte de collaboration pour annuler un commit sans modifier l'historique.

a. Annuler un commit spécifique

Si vous souhaitez annuler un commit mais garder l'historique intact :

```
git revert <sha1-du-commit>
# Exemple : annuler un commit avec un SHA-1 spécifique
git revert 123abc4
```

- Après cette commande, Git ouvre un éditeur pour vous permettre de modifier le message du commit de réversion.
- Une fois le message confirmé, Git créera un nouveau commit annulant les effets du commit ciblé.

b. Annuler plusieurs commits

Vous pouvez aussi annuler une série de commits en utilisant git revert avec plusieurs identifiants :

```
git revert <commit-ancien>..<commit-nouveau>
```

3. git reset

La commande git reset permet de réinitialiser le HEAD (pointeur sur le dernier commit) à un commit précédent et de modifier la zone de transit (index) ou le répertoire de travail, selon l'option choisie.

a. git reset avec l'option --soft

Le reset "soft" réinitialise l'historique des commits mais garde les modifications dans l'index (zone de transit). Les fichiers restent prêts à être re-committés.

```
git reset --soft <sha1-du-commit>
# Exemple : revenir au commit spécifique mais garder les fichiers dans la zone de transit
git reset --soft 123abc4
```

b. git reset avec l'option --mixed (par défaut)

Le reset "mixed" réinitialise l'historique des commits et l'index, mais conserve les modifications dans votre répertoire de travail. Les fichiers modifiés ne sont plus en attente de commit.

```
git reset --mixed <sha1-du-commit>
# Exemple : revenir à un commit antérieur sans conserver les fichiers en staging
git reset --mixed 123abc4
```

c. `git reset` avec l'option `--hard`

Le reset "hard" réinitialise l'historique, l'index et le répertoire de travail. Cela signifie que toutes les modifications locales seront perdues, il est donc important de l'utiliser avec prudence.

```
git reset --hard <sha1-du-commit>
# Exemple : réinitialiser complètement au commit spécifié
git reset --hard 123abc4
```

Testez les interactions avec un dépôt distant en configurant un dépôt GitHub

Une fois que vous avez travaillé localement, vous pouvez configurer un dépôt distant sur GitHub pour synchroniser vos modifications.

1. Créer un dépôt GitHub

1. **Accédez à GitHub** : https://github.com.
2. **Créez un nouveau dépôt** :
   - Cliquez sur **New** en haut à droite.
   - Donnez un nom à votre dépôt, par exemple `mon-projet`.
   - Vous pouvez choisir de le rendre privé ou public.
   - Ne cochez pas l'option **Initialize this repository with a README** si vous avez déjà un projet local.

2. Lier votre dépôt local au dépôt distant sur GitHub

1. **Initialisez le dépôt local et effectuez un commit initial**, si ce n'est pas déjà fait :

   ```
   git init
   git add .
   git commit -m "Premier commit"
   ```

2. **Ajoutez le dépôt distant GitHub** :

   ```
   git remote add origin https://github.com/votre-utilisateur/mon-projet.git
   ```

3. **Poussez les modifications sur GitHub** :
   - Poussez la branche `main` vers GitHub :

     ```
     git push -u origin main
     ```

- Si vous avez créé une autre branche, poussez-la également :

```
git push -u origin feature1
```

3. Interagir avec le dépôt distant

a. Récupérer les dernières modifications depuis GitHub

Si d'autres collaborateurs ont fait des changements, vous pouvez récupérer et fusionner ces modifications :

```
git pull origin main
```

b. Pousser de nouvelles modifications sur GitHub

Après avoir fait des changements et commis, vous pouvez envoyer les mises à jour vers GitHub :

```
git push origin main
```

Expérimenter avec les commandes de restauration et la gestion de branches

Voici un exercice pratique pour explorer les commandes de restauration (git reset, git revert, git checkout) et apprendre à gérer des branches tout en collaborant avec GitHub.

1. Mise en place de l'exercice

1. **Créez un nouveau dépôt local** :

```
mkdir gestion-historique
cd gestion-historique
git init
```

2. **Ajoutez un fichier et effectuez un premier commit** :

```
echo "Version 1" > fichier.txt
git add fichier.txt
git commit -m "Ajout de fichier.txt avec Version 1"
```

3. **Ajoutez des modifications supplémentaires et commitez-les** :

```
echo "Version 2" >> fichier.txt
git add fichier.txt
git commit -m "Mise à jour : ajout de Version 2"
```

```
echo "Version 3" >> fichier.txt
git add fichier.txt
git commit -m "Mise à jour : ajout de Version 3"
```

4. **Vérifiez l'historique des commits** :

```
git log --oneline
```

Vous devriez voir trois commits listés.

2. Expérimenter les commandes de restauration

a. Revenir à un commit précédent avec git reset

1. Identifiez l'ID du commit de "Version 2" avec :

```
git log --oneline
```

2. Effectuez un reset "soft" vers ce commit :

```
git reset --soft <sha1-de-version-2>
```

Cela conservera les modifications de "Version 3" dans la zone de transit.

3. Si vous voulez tout annuler (reset "hard"), exécutez :

```
git reset --hard <sha1-de-version-2>
```

b. Annuler un commit avec git revert

1. Ajoutez une nouvelle version au fichier et commitez-la :

```
echo "Version 4" >> fichier.txt
git add fichier.txt
git commit -m "Ajout de Version 4"
```

2. Annulez ce commit en créant un nouveau commit inverse :

```
git revert HEAD
```

3. Vérifiez que la modification introduite par "Version 4" est annulée tout en conservant l'historique.

c. Restaurer un fichier avec git checkout

1. Modifiez le fichier fichier.txt :

```
echo "Modification non désirée" >> fichier.txt
```

2. Annulez cette modification locale :

```
git checkout -- fichier.txt
```

Le fichier reviendra à son état du dernier commit.

3. Travailler avec des branches

a. Créez une nouvelle branche

1. Créez une branche nouvelle-feature et basculez dessus :

```
git branch nouvelle-feature
git checkout nouvelle-feature
```

2. Modifiez le fichier et ajoutez une fonctionnalité :

```
echo "Nouvelle fonctionnalité ajoutée" >> fichier.txt
git add fichier.txt
git commit -m "Ajout d'une nouvelle fonctionnalité"
```

b. Fusionnez la branche dans main

1. Basculez sur la branche principale :

```
git checkout main
```

2. Fusionnez nouvelle-feature dans main :

```
git merge nouvelle-feature
```

4. Pousser les modifications vers GitHub

a. Configurez un dépôt GitHub distant

1. Créez un dépôt GitHub appelé gestion-historique.
2. Ajoutez le dépôt distant à votre projet local :

```
git remote add origin https://github.com/<votre-utilisateur>/gestion-historique.git
```

b. Poussez les branches vers GitHub

1. Poussez la branche principale :

```
git push -u origin main
```

2. Poussez la branche nouvelle-feature :

```
git push -u origin nouvelle-feature
```

c. Collaborer sur GitHub

1. Créez une Pull Request sur GitHub pour fusionner nouvelle-feature dans main.
2. Résolvez les conflits s'il y en a et finalisez la fusion.

Prochaines étapes

- Essayez de simuler des conflits de fusion en modifiant le même fichier dans différentes branches.
- Testez `git stash` pour mettre temporairement de côté des modifications non terminées.
- Explorez des commandes avancées comme `git rebase` pour réorganiser les commits.