Git

m2iformation.fr

Sommaire

1. Présentation de Git
2. Les fondamentaux
3. Gestion locale des fichiers
4. Gestion des branches

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

1. Présentation de Git

Définition de gestion de version

“ Un gestionnaire de version est un système qui enregistre **l’évolution d’un fichier** ou d’un **ensemble de fichiers** au cours du **temps** de manière à ce qu’on puisse rappeler une **version antérieure** d’un fichier à tout moment ”

Les systèmes de gestion de version locaux

- Une méthode courant consiste à recopier les fichiers dans un autre répertoire avec un nom différent
- Les développeurs ont ensuite inventé dss VCS locaux qui utilisaient une base de données simple pour conserver les modifications d'un fichier

Les systèmes de gestion de version centralisés

- Pour gérer la collaboration sur des projets, des CVCS ont été inventés
- **avantages**: gestion des droits, visibilité sur l'avancement du projet
- **défauts**: risque de panne du serveur central

Les systèmes de gestion de version distribués

- Les Distributed Version Control Systems (DVCS) dupliquent entièrement le dépôt
- En cas de problème avec le serveur, une restoration peut être effectuée
- Ces outils permettent de mettre en place différentes chaînes de traitements

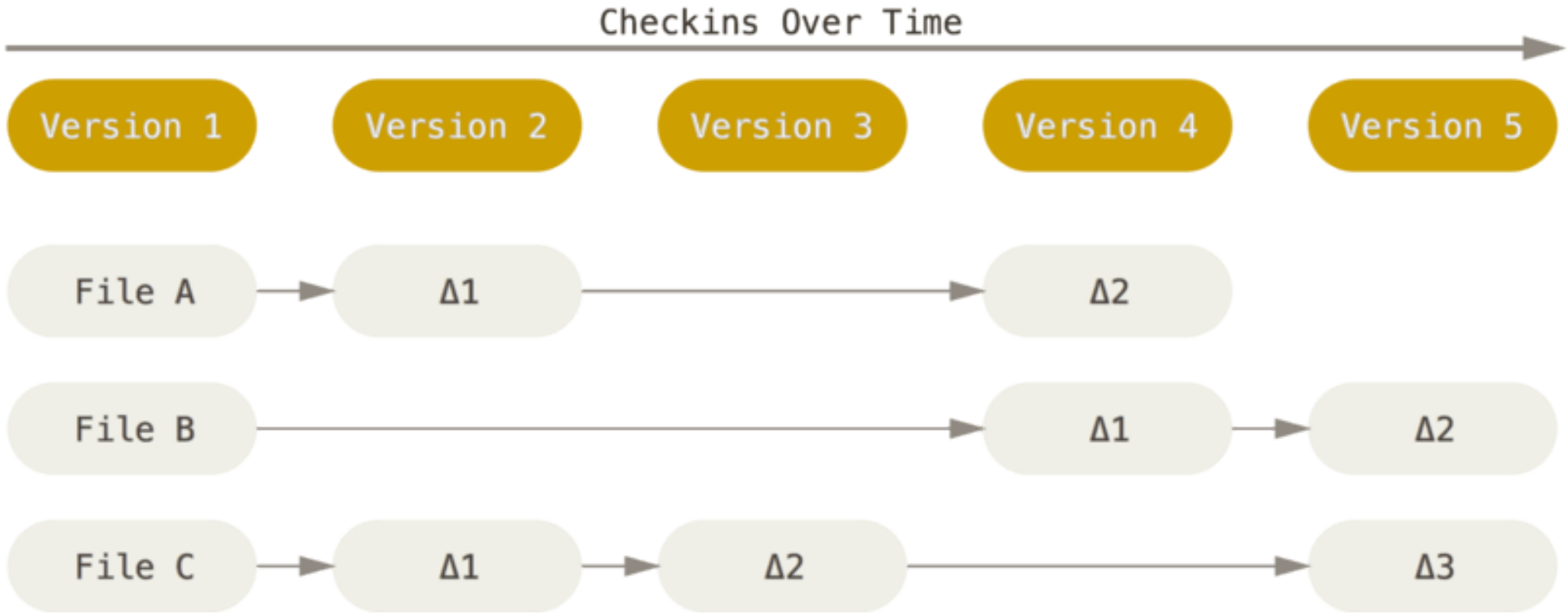
Histoire de Git

- Git a été créé par **Linus Torvald** (inventeur de Linux) en 2005
- Il est né d'un désaccord entre la communauté de Linux avec l'outil BitKeeper (DVCS propriétaire) devenu payant
- Git est basé sur les objectifs suivants:
  - **Vitesse** et **conception simple**
  - Support pour le **développement non linéaire**
  - Système **distribué**
  - Gestion de **projets d'envergures**

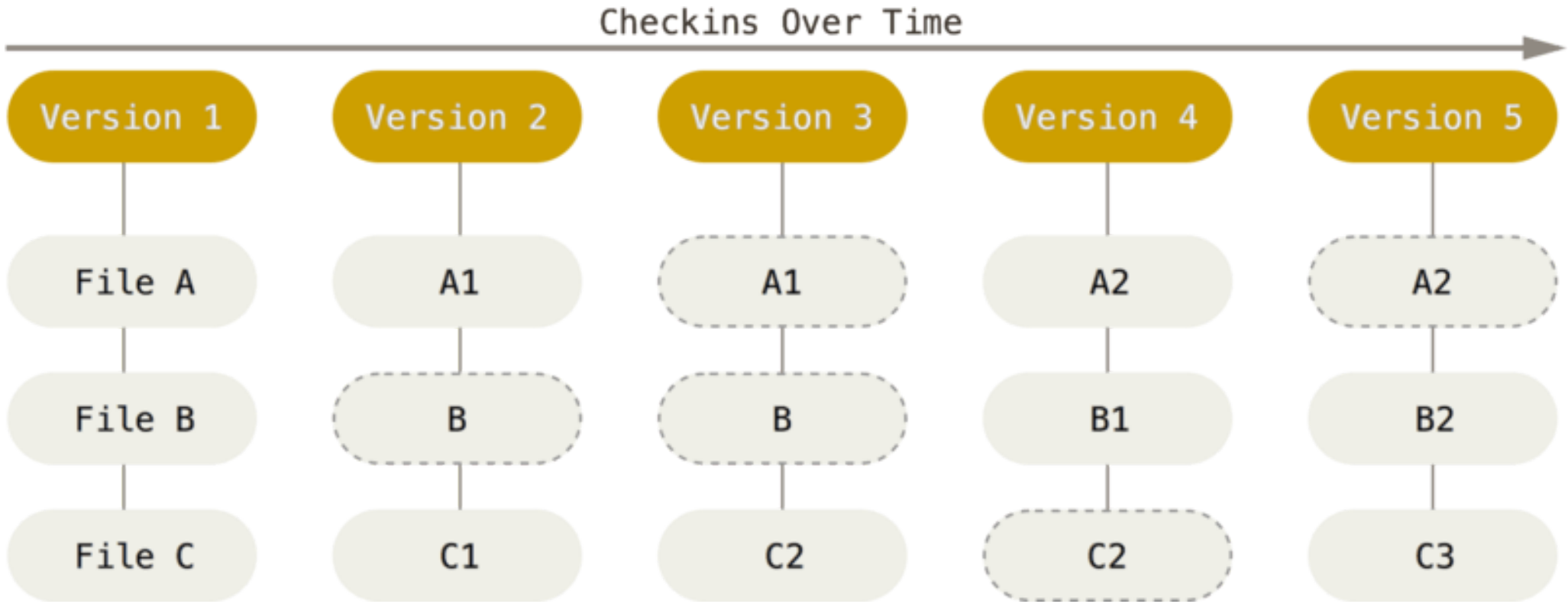
Fonctionnement de Git

- Le système de versionnage de git se base sur le principe de **snapshots** et non pas de différences entre les fichiers
- Chaque fois que l'état d'un projet est validé ou enregistré, git enregistre une référence à ce snapshot
- Si les fichiers n'ont pas été modifiés, git n'enregistrera pas à nouveau le fichier mais fera référence à sa dernière version modifiée

Système basé sur la différence des fichiers



Système basé sur les snapshots (git)



Installation

- Pour installer git il suffit de se rendre à l'URL suivant: https://git-scm.com/downloads

Paramètrage de git

- **git config** est un outil qui permet de voir et modifier les variables de configuration qui contrôlent tous les aspects de l'apparence et du comportement de Git
- Configuration de l'identité (utilisée dans les validations) :

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

- Configuration de l'éditeur (vscode) :

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

Vérification des paramètres

- Définir le nom de la branche par défaut (de base 'master')
  - `git config --global init.defaultBranch main`
- Pour afficher la configuration et son origine
  - `git config --list --show-origin`
- Pour afficher la valeur d'un paramètre
  - `git config [paramètre]`

Obtenir de l'aide

- Pour obtenir la documentation liée à une commande, il suffit d'utiliser les différentes commandes suivantes

```
git help <commande>
git <commande> --help
man git-<commande>
```

- Par exemple pour obtenir l'aide de la commande config: `git help config`
- Pour une documentation concise: `git <command> -h`

2. Les fondamentaux

Glossaire

- **branch** : version du dépôt pour travailler sur une fonctionnalité particulière sans impacter le code courant
- **commit** : enveloppe qui contient une petite portion de codes modifiés d’un ou plusieurs fichiers
- **dépôt** : environnement virtuel d'un projet qui permet d'enregistrer les versions du code et d'y accéder au besoin
- **merge** : action qui applique les changements d’une branche sur une autre

Glossaire

- **HEAD** : pointeur vers le dernier commit de la branche en cours push
- **fetch** : récupère les commits du dépôt distant en local sans les appliquer
- **tag** : pointeur vers un commit particulier (release)
- **pull** : télécharge les commits manquant du dépôt distant sur le dépôt local et les applique
- **Index/staging area** : Zone mémoire qui contient les fichiers mis de côté pour préparer le commit. On dit que les fichiers sont indexés

Initialiser un dépôt

- Pour créer un nouveau dépôt git saisir: `git init`
- Cette commande créé un répertoire **.git** dans le répertoire courant
- Le répertoire **.git** contient tous les fichiers nécessaires au fonctionnement d'un dépôt git

```
├──hooks
├──info
├──objects
└──refs
   ├──heads
   └──tags
```

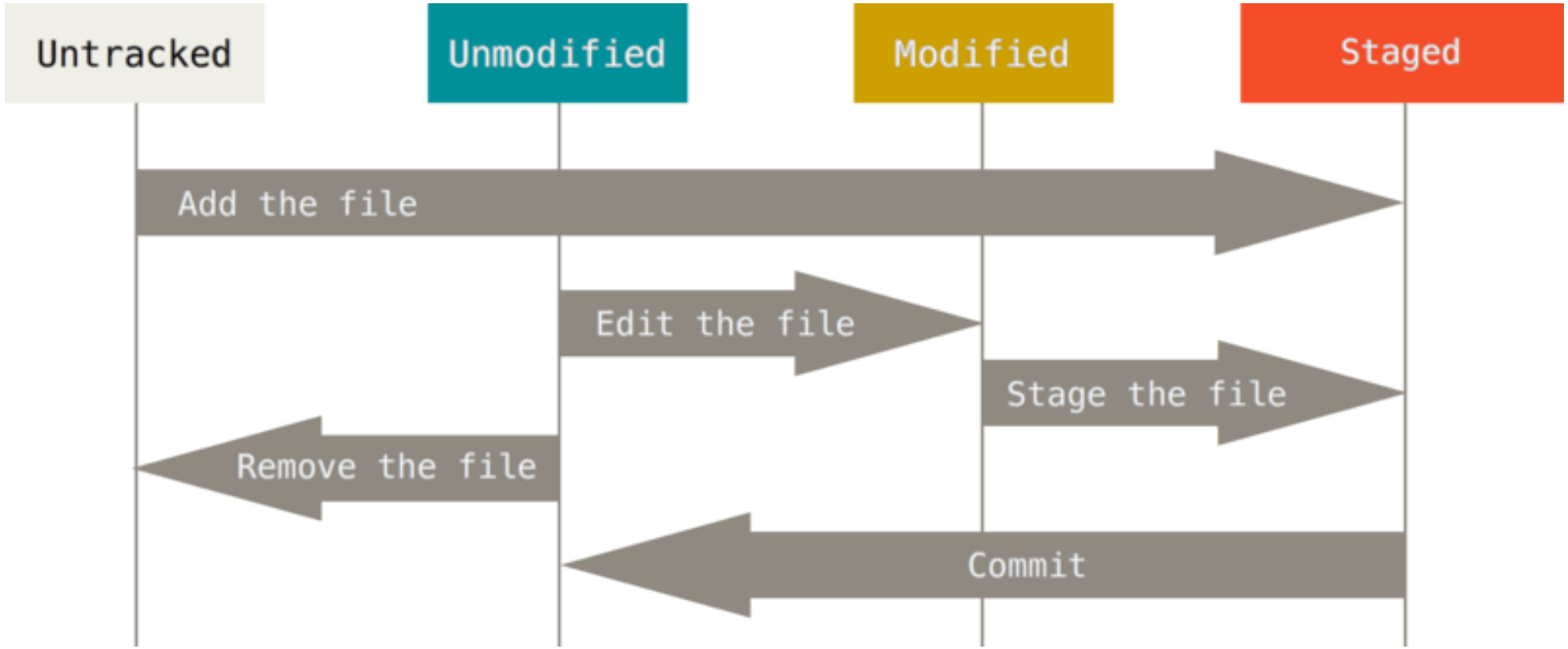
Cloner un dépôt existant

- Pour obtenir les sources d'un dépôt existant `git clone [url]`
- Git reçoit une copie des données sur le serveur distant avec toutes les versions et l'historique du projet
- `git clone https://github.com/utopios/pokemon-app`
- Cette commande créé un répertoire pokemon-app, initialise un répertoire .git avec toutes les données du dépôt

Enregistrer des modifications dans le dépôt

- Chaque fichier dans le répertoire de travail peut avoir deux états
  1. **non suivi** : non connu par git
  2. **suivi** : appartient à un snapshot, il peut être **inchangé**, **modifié** ou **indexé**

Le cycle de vie des fichiers



Ajouter un fichier au suivi de version

- `git add [fichier]` : permet d'ajouter un fichier au suivi de version
- `git add .` : ajoute le répertoire actuel au suivi de version
- `git add --all` : fait la même chose que git add .
- `git add -A` : raccourci de la commande précédente

Vérifier l'état des fichiers

- L'outil principal pour déterminer quels fichiers sont dans quel état est la commande : `git status`

```
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file.txt
```

Ignorer des fichiers

- Certains fichiers ne nécessitent pas d'être versionnés, par exemple : les fichiers de compilations, les logs, les fichiers temporaires, les clés d'api ...
- Pour éviter de tracer ces fichiers dans git, il faut créer un fichier `.gitignore` à la racine du projet
- Les lignes commençant avec le caractère `#` sont des commentaires
- Si la ligne termine par `/` cela indique un dossier

Exemple .gitignore java

```
# fichiers de class compilés
*.class

# Fichiers logs
*.log

# Package
*.jar
*.war

# répertoire secret
secret/
```

Comparer des changements

- `git diff` est une commande Git qui lance une fonction de différenciation sur les sources de données Git
- Ces sources de données peuvent être des commits, des branches, des fichiers, etc.
- `git diff` montre les lignes exactes qui ont été ajoutées, modifiées ou effacées — le patch en somme
- `git diff --staged` : permet de comparer les fichiers indexés et le dernier snapshot

Valider des modifications

- Une fois le staging area (zone d'index) dans l'état désiré, il faut valider les modifications
- Pour valider les modifications: `git commit`
- Pour documenter les modifications d'un commit, on ajoute **toujours** un message descriptif au commit avec le paramètre `git commit -m "<message>"`
- Cette opération créé un nouveau **snapshot**

Tp commits

- Créer un nouveau *repository git*
- Ajouter un fichier et le commiter
- Modifier le fichier et le commiter

Ajouter et valider des modifications

- Il existe un raccourci qui permet d'ajouter les fichiers en suivi de version directement via l'opérateur de commit
- `git -a` : cette commande ordonne à tous les fichiers de se placer dans la zone d'index et de réaliser la validation ce qui permet de s'absoudre de taper `git add`
- cas d'usage : `git -am "maj fichiers"`
- ⚠️ si des fichiers ont été créés dans le répertoire, il faudra veiller à les ajouter au suivi de version sans quoi l'opération précédente ne fonctionnera pas pour ces fichiers

Visualiser l'historiques des validations

- Pour visualiser l'historique des commits : `git log`
- Cette commande invoque les commits du plus récents aux plus anciens
- Le paramètre `git log -p -<nombre>` permet d'afficher les différences entre les validations, on peut préciser le nombre de commit que l'on souhaite afficher
- `git log --stat` permet d'afficher les statiques des commits

Annuler une action

`git reset [mode] [commit]` : reset la branche courante

- commit :
  - id du commit sur lequel on veut positionner le sommet de la branche
  - si vide, on laisse la branche où elle est (utile pour resetter l'index ou la working copy)
- mode :
  - `--soft` : ne touche ni à l'index, ni à la working copy (alias "je travaillais sur la mauvaise branche")
  - `--hard` : reset l'index et la working copy (alias "je mets tout à la poubelle")
  - `--mixed` : reset l'index mais pas la working copy, (alias "finalement je ne vais pas commiter tout ça") c'est le mode par défaut.
- Le mode par défaut (mixed) n'entraîne pas de perte de données, on retire juste les changements de l'index.

Annuler une action

- Pour revenir sur une working copy propre (c'est-à-dire supprimer tous les changements non commités) :
  - `git reset --hard`

Les dépôts distants

- Pour visualise les serveurs distants enregistrés : `git remote`
- S'il s'git d'un dépôt cloné vous devriez au moins vor l'origine `origin`
- Pour voir l'url du dépôt distant : `git remote -v`
- Ajouter des dépôts distants : `git remote add [nomcourt] [url]`
- Supprimer un dépôt distants : `git remote rm [nomcourt]`
- Récupérer toutes les données que vous ne possédez pas encore : `git fetch [remote-name]`
- Pousser son travail sur un dépôt distant :

Étiquetage

À l'instar de la plupart des VCS, Git donne la possibilité d'étiqueter un certain état dans l'historique comme important.

- Lister vos étiquettes (Tag) : `git tag`
- Crée un tag : `git tag -a nom_du_tag -m "message"`
- Supprimer un tag : `git tag -d <nom-d-etiquette>`

Les alias Git

Git ne complète pas votre commande si vous ne la tapez que partiellement. Si vous ne voulez pas avoir à taper l'intégralité du texte de chaque commande, vous pouvez facilement définir un alias pour chaque commande en utilisant git config.

```
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.st status
```

Les branches avec Git

Presque tous les VCS proposent une certaine forme de gestion de branches. Créer une branche signifie diverger de la ligne principale de développement et continuer à travailler sans impacter cette ligne. Elles permettent le développement de différentes versions en parallèle :

- Version en cours de développement
- Version en production (correction de bugs)
- Version en recette
- ...

Les branches avec Git

- branch == pointeur sur le dernier commit (sommet) de la branche
  - les branches sont des références
- master == branche principale (trunk)
- HEAD == pointeur sur la position actuelle de la working copy
- On parle de “merge” lorsque tout ou partie des modifications d'une branche sont rapatriées dans une autre
- On parle de “feature branch” pour une branche dédiée au développement d'une fonctionnalité (ex : gestion des contrats...)

Les branches avec Git

Création

- `git branch` liste des branches (locales)
- `git branch <mabranche>` création de la branch
- `git checkout <mabranche>` se positionner sur la branch
- `git checkout -b <mabranche>` création et se positionner sur la branch
- `git branch --move mauvais-nom-de-branche nom-de-branche-corrigé` renommez la branche localement

Les branches avec Git

Suppression

- `git branch -d <mabranche>` suppression de la branch mabranche (erreur si pas merge)
- `git branch -D <mabranche>` suppression de la branch mabranche (forcé)

Tp branches / Checkout / reset / Tags

- Créer un nouveau *repository* Git
- Ajouter un fichier et le commiter
- Ajouter un deuxième fichier et le commiter
- Vérifier l'historique (on doit avoir 2 commits)
- Faire des modifications sur le deuxième fichier et le commiter
- Annuler les modifications du dernier commit
- Vérifier l'historique (on doit avoir 2 commits)
- Créer une branche à partir du 1er commit
- Faire un commit sur la branche
- Vérifier l'historique (on doit avoir 2 commits)

Tp branches / Checkout / reset / Tags

- Lister les branches (on doit avoir 1 branche)
- Tagger la version
- Revenir au sommet de la branche *master*
- Lister les tags (on doit avoir un tag)
- supprimer la branche
- Lister les branches (on doit avoir une seule branche : master)

Merge

La fusion (merge) en Git est le processus d'intégration des modifications de deux branches distinctes en une seule, combinant ainsi les modifications apportées à chaque branche en un seul ensemble cohérent.

- Permet de fusionner 2 branches / Réconcilier 2 historiques
- Rapatrier les modifications d'une branche dans une autre
- ⚠️ par défaut le merge concerne tous les commits depuis le dernier merge / création de la branche

Merge

Pour faire un merge (fusion), d'une branch A vers une branche B :

- Assurez-vous d'être sur la branche de destination (branche B) en utilisant la commande : `git checkout B`
- Ensuite effectuez le merge en utilisant la commande : `git merge A`

Cela intégrera les modifications de la branche A dans la branche B.

Merge

Git peut automatiquement effectuer le merge si les changements dans les deux branches ne se chevauchent pas. Cependant, s'il y a des conflits (lorsque le même fichier a été modifié dans les deux branches), Git vous demandera de résoudre ces conflits manuellement. Ouvrez les fichiers en conflit, modifiez-les pour résoudre les différences.
Pour voir quelles branches ont déjà été fusionnées dans votre branche courante, lancez `git branch --merged`

Rebase

Le rebase Git réécrit l'historique des commits en déplaçant une branche vers un autre point de départ.

- Modifie / réécrit l'historique
- Modifie / actualise le point de départ de la branche
- Remet nos commits au dessus de la branche contre laquelle on rebase

Rebase

Pour effectuer un rebase de la branche A sur la branche B
Assurez-vous d'abord d'être sur la branche A en utilisant la commande :

- `git checkout A`

Exécutez la commande de rebase pour réécrire l'historique de la branche A sur la base de la branche B

- `git rebase B`

Une fois que le rebase est terminé, la branche A aura l'historique réécrit sur la base de la branche B.

Rebase

Git peut rencontrer des conflits pendant le rebase. Si cela se produit, résolvez les conflits un par un en éditant les fichiers concernés.
Après chaque résolution de conflit, utilisez :

- `git add nom_du_fichier`

Une fois que tous les conflits sont résolus et que vous avez ajouté les fichiers modifiés, continuez le rebase en utilisant :

- `git rebase --continue`

Si à tout moment vous souhaitez annuler le rebase et revenir à l'état précédent, utilisez la commande :

- `git rebase --abort`

Rebaser ou Fusionner

- Rebase : pour la mise à jour des branches avant merge linéaire (commits indépendants) ex : corrections d'anomalies → on ne veut pas de commit de merge.
- Merge sans rebase : pour la réintégration des feature branches (on veut garder l'historique des commits indépendants sans polluer l'historique de la branche principale)

Tp rebase

- Créer un nouveau repository Git
- Ajouter un fichier et le commiter (C1), le modifier et le commiter (C2)
- Créer une branche B1 à partir de C1
- Faire une modification du fichier et commiter C3
- Merger B1 dans master de manière à avoir un historique linéaire

Tp Merge

- Créer un nouveau repository git
- Ajouter un fichier et le commiter (C1)
- Créer une feature branch B1 à partir de C1
- Faire une modification du fichier et commiter (C2)
- Merger B1 dans master de manière à avoir un commit de merge dans master

Tp Conflit

- Créer un nouveau repository Git
- Ajouter un fichier et le commiter (C1)
- Modifier la première ligne du fichier et commiter (C2)
- Créer une feature branch B1 à partir de C1
- Faire une modification de la première ligne du fichier et commiter (C3)
- Merger B1 dans master en résolvant les conflits

Pousser les branches

Lorsque vous souhaitez partager une branche avec le reste du monde, vous devez la pousser sur un serveur distant sur lequel vous avez accès en écriture. Vos branches locales ne sont pas automatiquement synchronisées sur les serveurs distants.

- `git push -u repository_distant ma_branche_de_developpement` pousser une branche locale vers un dépôt distant et en même temps configurer le suivi de cette branche (option `-u`)
- `git push origin --all` : pousser toutes les branches locales en une seule fois
- `git pull` : tirer une branche (fetch et merge)

Suppression de branches distantes

Supposons que vous en avez terminé avec une branche distante.

- `git push nom_remote --delete nombranch` : effacer votre branch nombranch su serveur

Cela ne fait que supprimer le pointeur sur le serveur. Le serveur Git garde généralement les données pour un temps jusqu'à ce qu'un processus de nettoyage (garbage collection) passe. De cette manière, si une suppression accidentelle a eu lieu, les données sont souvent très facilement récupérables.

Tp Git Distant

- Créer un nouveau repository Git (R1)
- Ajouter un fichier et le commiter (C1)
- Cloner le repository (protocole file) (R2)
- Lister toutes les branches locales et distantes (on doit avoir une branche locale, une branche remote et une remote head)
- Sur R1 modifier le fichier et commiter (C2)
- Sur R2 récupérer le commit C2 (vérifier avec git log)
- Sur R2 créer une nouvelle branche (B1), faire une modification du fichier, commiter (C3)
- Publier B1 sur sur R1 (vérifier avec git branch -a sur R1)
- Créer une branche B2 sur R1

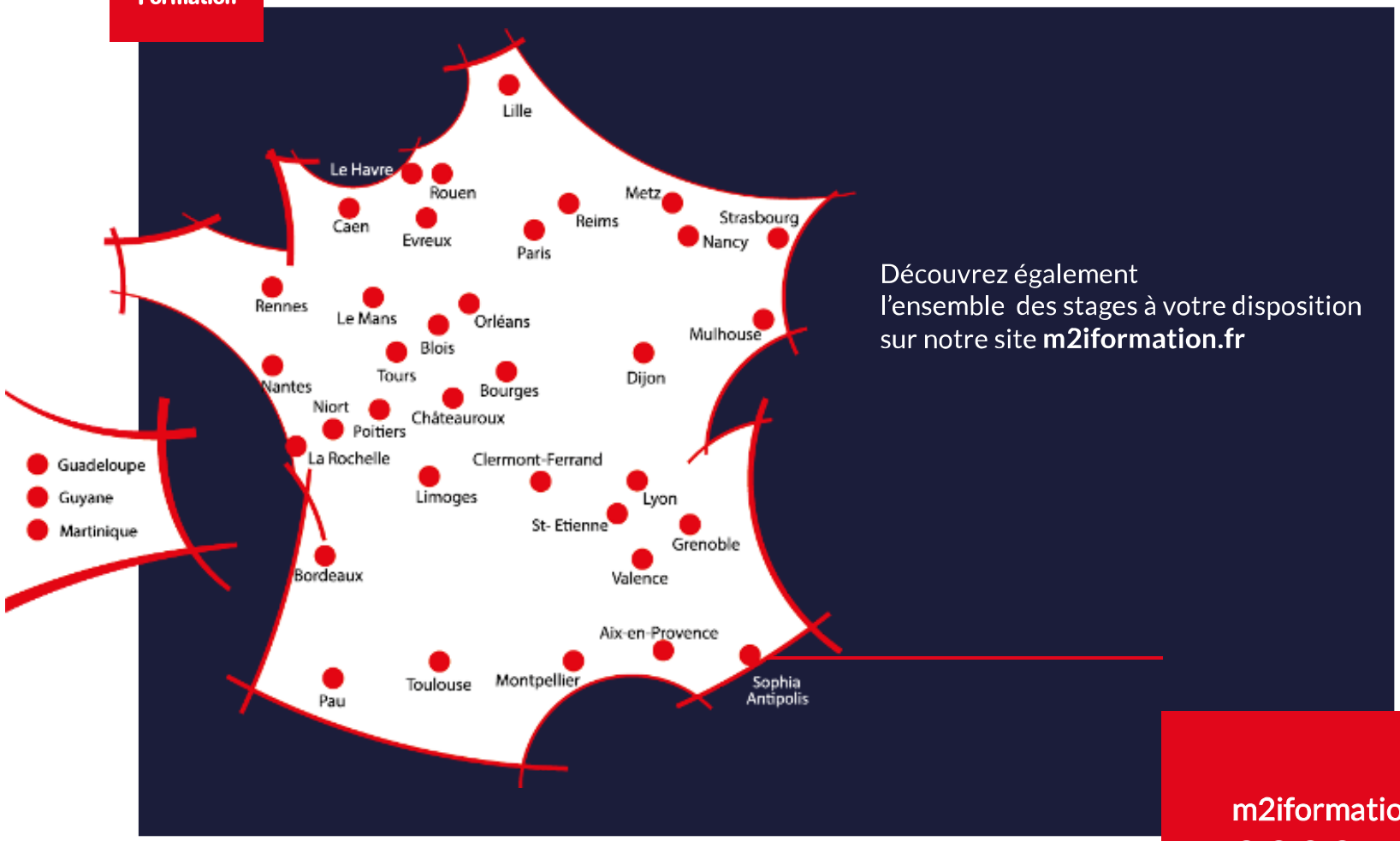
Tp Git Distant

- Récupérer B2 sur R2 (vérifier avec git branch -a sur R2)
- Tagger B2 sur R2 (T1)
- Publier T1 sur R1
- Vérifier que le Tag T1 est sur R1 (git tag -l)
- Sur R1 B1 modifier la première ligne du fichier et commiter (C4)
- Sur R2 B1 modifier la première ligne du fichier et commiter (C5)
- Publier C5 sur R1 B1 (conflit)
- Résoudre le conflit
- Vérifier la présence d'un commit de merge sur R1 B1

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Merci pour votre attention

Des questions ?



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr