

# FORMATION **git**



Anthony Di Persio





Qu'est-ce que Git ?

# GIT ?

- Git est un système de gestion de versions créé en 2005.
- C'est un système de gestion de versions décentralisé, c'est-à-dire que chaque développeur possède sur son poste sa ou ses propres versions du projet.
- C'est ce qui le différencie d'un système centralisé où tout repose sur un serveur central




# GIT ?

- Git permet :
  - De retrouver facilement toutes les versions d'un projet.
  - De faire cohabiter différentes versions d'un projet.
  - De fusionner différentes versions d'un projet.
- Git n'est pas un système de sauvegarde, il n'est pas conçu pour cet usage.

# GIT ?

- Git s'articule autour de plusieurs éléments :
  - Le dépôt
  - Les commits
  - Les branches





Un peu de  
vocabulaire...

# dépôt ?

- Le dépôt est le dossier (nommé .git à la racine du projet) qui va contenir toutes les versions du projet.
- Il existe deux types de dépôts
  - Local
  - Distant
- Nous développerons la notion de dépôt distant plus tard dans la formation

# Branches ?

- Les branches sont utilisées pour développer des fonctionnalités isolées des autres.
- La branche master est la branche par défaut quand vous créez un dépôt.
- Utilisez les autres branches pour le développement et fusionnez ensuite à la branche principale quand vous avez fini.
- Les branches permettent de garder une version du code source toujours fonctionnelle, tout en travaillant sur plusieurs fonctionnalités



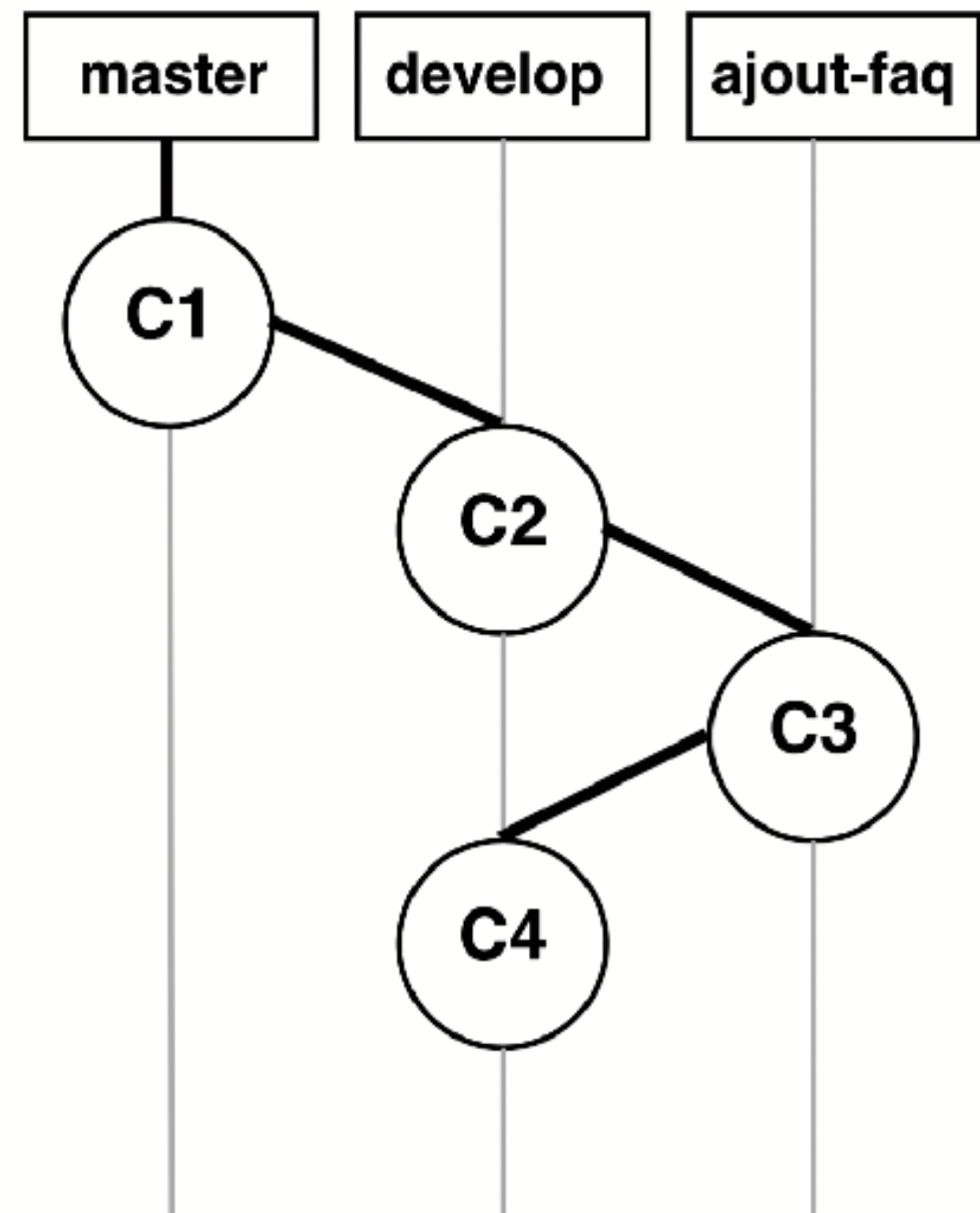
# Commits ?

- Les commits représentent chaque incrément du projet.
- Ils sont identifiés par Git par une empreinte (hash en anglais) de 160 bits (soit 40 caractères hexadécimaux).
- Cela signifie que Git va identifier chacun de ces commits par des identifiants de ce type :

`aef6153a6fa0e4880d9b8d0be4720f78e895265d.`

# Branches et Commits

Les branches et les commits sont les éléments principaux de Git. Les développeurs les manipulent quotidiennement.





# Les trois Zones de stockage...

# Les trois Zones de stockaGE

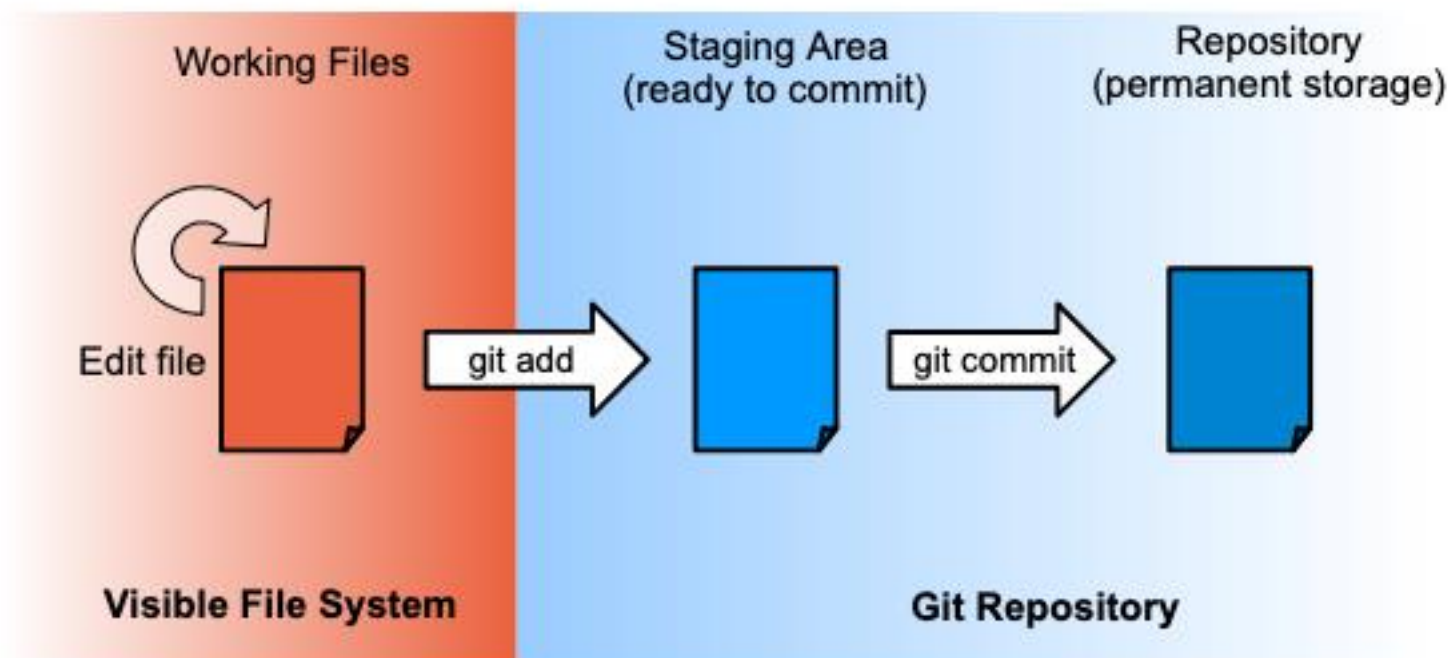
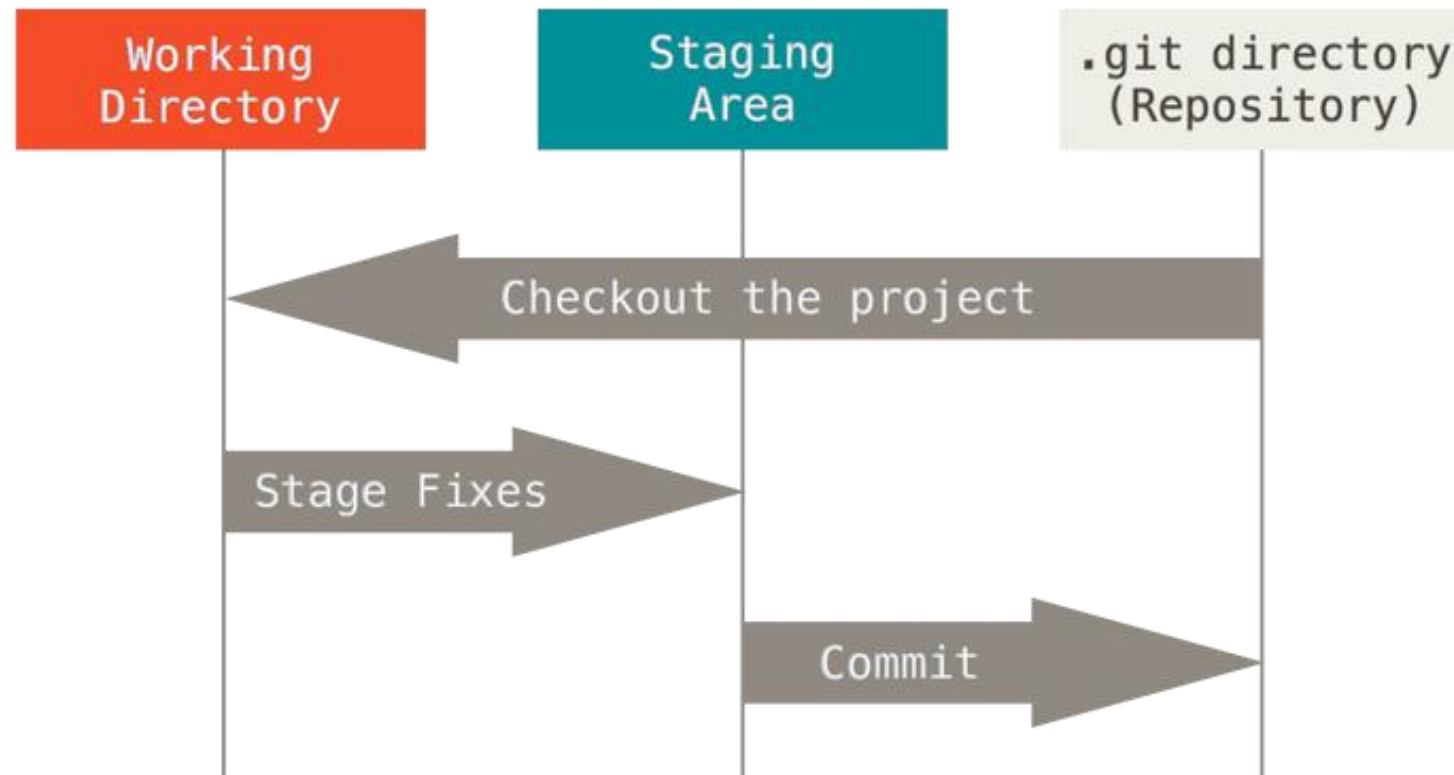
- Git gère trois zones où les fichiers peuvent résider :
  - le répertoire de travail (working directory) local où sont réalisés les changements
  - la « staging area » (aussi appelé index c'est le terme que nous employerons) où sont pré-enregistrés les changements (en attente de commit)
  - le dépôt git où sont enregistrés les changements



# Les trois Zones de stockaGE

- Lorsqu'un développeur souhaite enregistrer son travail dans Git, il effectue un commit qui contiendra les modifications à ajouter au projet
- Avant d'être intégrées dans un commit, les modifications passent dans l'index (staging area)
- L'index est une zone d'attente avant le commit

# Les trois Zones de stockaGE





# Installer GIT

# Installer GIT



<https://git-scm.com/>





# Installer GIT

Pour valider l'installation de git,  
ouvrez le terminal et tapez :

```
git --version
```



# Des Commandes

# Finaliser la Configuration de GIT

- Quand ? Après l'installation de Git
- Pourquoi ?
  - Avant de commencer à faire quoi que ce soit il est important de configurer Git. Par défaut la configuration est plutôt satisfaisante mais on va devoir configurer les informations nous concernant.
  - Ces informations seront visible dans l'historique des modifications et permettront de savoir plus tard qui à fait quoi... Pratique pour le travail en équipe.

# Finaliser la Configuration de GIT

- Comment ?

- De manière générale, en utilisant la commande `git config clé valeur`
- Dans notre cas nous allons utiliser les deux commandes suivantes :

✂ `git config --global user.name "Jean-Michel Apeuprè"`

✂ `git config --global user.email "jm.apeupre@kamoulox.fr"`



# Finaliser la Configuration de GIT

- A savoir
  - Il existe trois niveaux de configuration configurables à l'aide des arguments :

 `--system`

 `--global`

 `--local`

# Créer un dépôt

- Quand ? Au début du projet
- Pourquoi ? Dans le but de versionner le projet
- Comment ?
  - En utilisant la commande `git init` dans le dossier du projet
  - Cette commande va créer un dossier `.git` à la racine du projet

# Créer un dépôt

- Pour utiliser la commande `git init` je dois me situer dans le dossier de mon projet mais comment faire ?
  - Commande `cd` pour changer de dossier
  - Commande `mkdir` pour créer le dossier s'il n'est pas encore présent
  - Commande `ls` pour lister les fichiers du dossier courant

# Créer un dépôt

- A savoir
  - La commande `git init` ne sert pas à dupliquer un dépôt existant, mais vraiment à créer un dépôt vierge de tout commit



# Visualiser les fichiers modifiés

- Quand ? Avant de préparer un commit
- Pourquoi ? Cela permet de lister les fichiers modifiés. Cela peut permettre de déceler des fichiers non désirés qui se sont ajoutés dans le dépôt
- Comment ?
  - En utilisant la commande `git status`
  - Cette commande affiche les fichiers ajoutés, modifiés ou supprimés

# Préparer un commit

- Quand ? Juste avant de commiter
- Pourquoi ? Comme vu précédemment avant d'être intégrées dans un commit, les modifications doivent être placées dans l'index (une zone d'attente / staging area)
- Comment ?
  - En utilisant la commande `git add fichier.ext`
  - Cette commande va ajouter les modifications du fichier à l'index et seront prêtes à être commitées

# COMMITTER

- Quand ?
  - Très bonne question et primordiale
  - Un commit doit représenter un ensemble cohérent de modifications.
  - Cela peut-être le correctif d'un bug, l'ajout d'une fonctionnalité, etc...
  - Attention toutefois, il est déconseillé de faire des commits de plus de 200 lignes de modifications

# COMMITTER

- Pourquoi ?
  - Le commit sert à enregistrer des modifications au projet.
  - Le commit permet d'intégrer le travail d'un développeur dans une branche du projet
  - En réalité Git récupère les modifications contenues dans l'index et les enregistre dans un commit



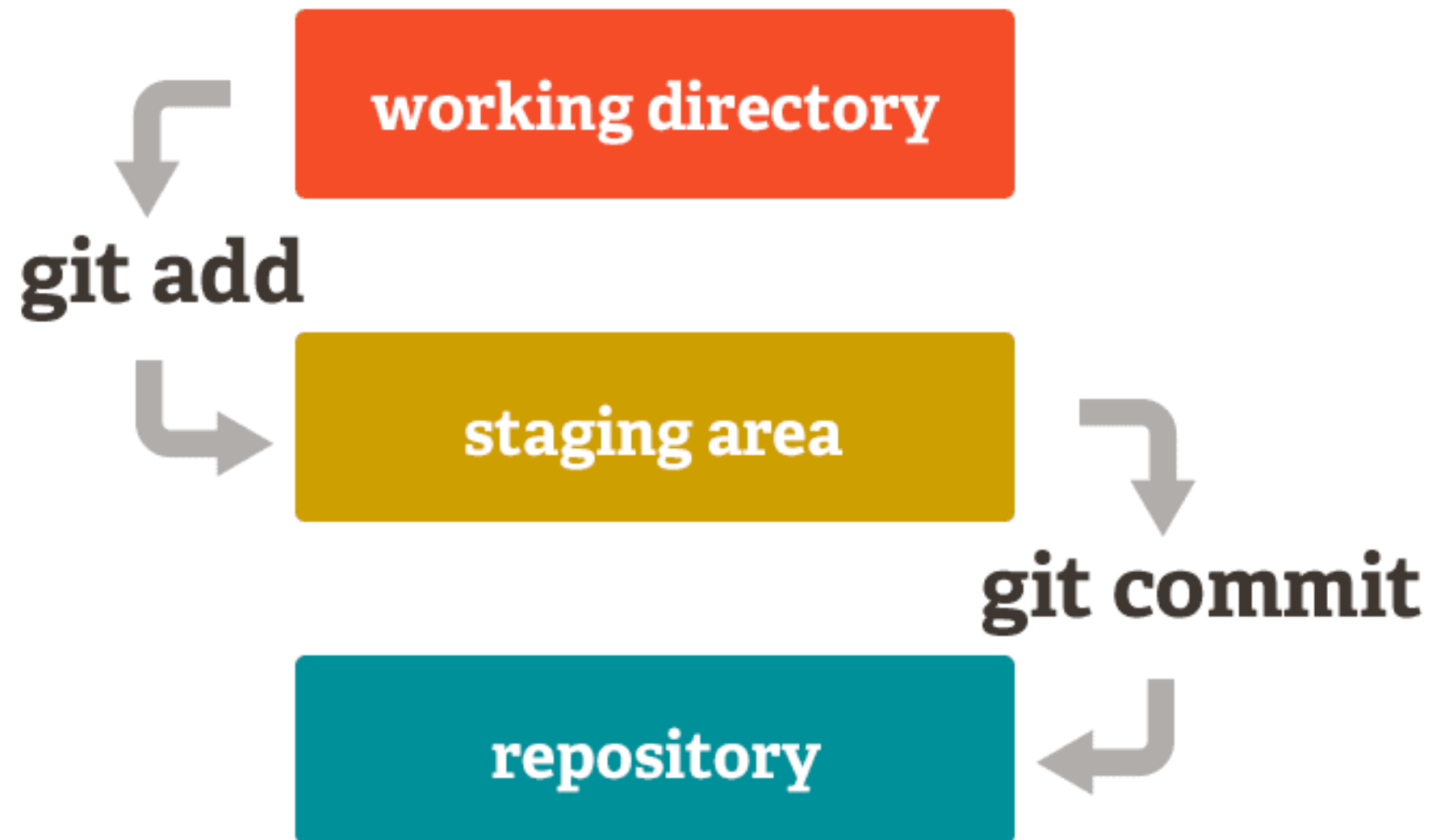
# COMMITER

- Comment ?
  - En utilisant la commande `git commit`
  - Cette commande ouvre un éditeur de texte permettant de donner la nature du commit
  - 49 caractères maximum, essayez d'être le plus clair possible dans l'écriture de ce message :
    - ▶ Correction bug d'affichage navbar en responsive VS  
Bug affichage

# COMMITER

- A savoir ?
  - L'argument `-m` suivi d'un texte permet de spécifier le titre du commit directement dans la commande sans avoir à passer par un éditeur de texte externe
  - Ce qui donnerait par exemple :
    - ▶ `git commit -m "Correction du bug d'affichage de la navbar en mode responsive"`

# En résumé



# Déplacer/renommer un fichier

- Quand ? Quand c'est nécessaire à n'importe quel moment de la vie d'un projet
- Pourquoi ? En passant par Git pour effectuer le déplacement ou le renommage vous évitez de perdre l'historique du fichier
- Comment ?
  - En utilisant la commande `git mv origine.ext destination.ext`

# Supprimer un fichier

- Quand ? Quand un fichier n'est plus utile dans un projet
- Pourquoi ? En passant par Git le fichier est placé dans la zone d'index (staging area) pour que la suppression se fasse lors du prochain commit.
- Comment ?
  - En utilisant la commande `git rm fichier.ext`



# Ignorer DES fichiers

- Quand ? Au début d'un projet mais également au cours de la vie du projet.
- Pourquoi ? Certains fichiers ne doivent pas être versions comme par exemple un fichier généré après compilation, des fichiers contenant une configuration propre à chaque utilisateur, ...
- Comment ?
  - Il faut créer un fichier nommé `.gitignore` à la racine du projet

# Ignorer DES fichiers

- Comment ?

➤ Exemple d'un fichier nommé `.gitignore`

```
# Ignorer le fichier CSS compilé
```

```
/css/main.css
```

```
# Ignorer tous les logs
```

```
/logs/*
```

```
# Ignorer les fichiers système cachés
```

```
.DS_Store
```

# Consulter l'historique

- Pourquoi ? Obtenir des informations sur les différents commit de notre projet (modifications, auteurs, etc...)
- Comment ?
  - La commande qui permet de lister les commits est `git log`
  - Vous l'utiliserez très souvent avec des arguments sinon elle renvoie une liste de tous les commits

# Consulter l'historique

- A savoir : Arguments utiles
  - `--oneline` permet d'afficher l'historique avec une ligne par commit (plus lisible)
  - `-n <nombre>` permet de sélectionner le nombre de commit à afficher
  - `-p <fichier>` permet de voir l'historique des commits affectant un fichier en particulier
  - `--author <auteur>`, permet de voir l'historique par rapport au nom de l'auteur

# Consulter les modifications

- Quand ? Avant d'ajouter les modifications à l'index ou avant de commiter.
- Pourquoi ? Voir les différences qu'il existe sur un fichier entre plusieurs versions. Pratique pour une vérification avant commit.
  - Entre deux commits
  - Entre le répertoire de travail et l'index
  - Entre l'index et HEAD



# Consulter les modifications

- Comment ?
  - `git diff` pour afficher la différence entre le répertoire de travail et l'index
  - `git diff HEAD` pour afficher la différence entre le répertoire de travail et le dernier commit
  - `git diff commit1 commit2` pour afficher la différence entre les deux commits
  - `git diff fichier` pour afficher la différence sur un fichier en particulier



# Branches

# Rappel d'une Branche

- Les branches sont utilisées pour développer des fonctionnalités isolées des autres.
- La branche master est la branche par défaut quand vous créez un dépôt.
- Utilisez les autres branches pour le développement et fusionnez ensuite à la branche principale quand vous avez fini.
- Les branches permettent de garder une version du code source toujours fonctionnelle, tout en travaillant sur plusieurs fonctionnalités

# Créer une branche

- Quand ? Au moment de commencer un nouveau développement pour un correctif ou une nouvelle fonctionnalité
- Comment ?
  - En utilisant la commande `git branch <nomDeLaBranche>`

# Lister Les branches

- Comment ?
  - En utilisant la commande `git branch`



# Renommer une branche

- Comment ?
  - En utilisant la commande `git branch -m <nouveauNomDeLaBranche>`
  - Attention il s'agit de la branche courante qui sera modifié

# Supprimer une branche

- Comment ?
  - En utilisant la commande `git branch -d <nomDeLaBranche>`

# Se placer sur une branche

- Quand ? Au moment de changer de branche
- Pourquoi ? Pour être capable d'aller d'une branche à une autre
- Comment ?
  - En utilisant la commande `git checkout <nomDeLaBranche>`
  - Permet de se rendre sur une branche qui existe

# Se placer sur une branche

- Si vous souhaitez vous pouvez demander à Git de sauter sur une branche qui n'existe pas en la créant au préalable de changer de branche
- Comment ?
  - En utilisant la commande `git checkout -b <nomDeLaBranche>`
  - Qui est équivalent à :  
`git branch <nomDeLaBranche>`  
`git checkout <nomDeLaBranche>`

# Fusionner une branche

- Quand ? Lorsque que le développement sur la branche est terminé
- Pourquoi ? La fusion permet d'intégrer les modifications d'une branche dans une autre.
- Comment ?
  - En utilisant la commande `git merge`  
`<nomDeLaBrancheAFusionner>`

# Fusionner une branche

- Comment ?
  - Attention il faut se positionner sur la branche qui va réceptionner les modifications de la branche avant de lancer le `git merge`
  - La commande crée un nouveau commit qui contient les modifications de la branche à fusionner.
  - La fusion peut entrainer des conflits...



# Résoudre les conflits

- Quand ? Quand une fusion a provoqué un conflit
- Pourquoi ?
  - Un conflit survient quand Git trouve des modifications différentes aux mêmes endroits du fichier entre les deux branches.
  - Il faut donc l'aider à savoir quel code placer dans la version finale.

# Résoudre les conflits

- Comment ?
  - La commande `git merge` place des séparateurs dans les fichiers concernés
  - Le séparateur `<<<<<<<` permet de cibler le code de la branche courante
  - Le séparateur `=====` sert de délimitation entre le code de la branche courante et celui de la branche à fusionner
  - Le séparateur `>>>>>>>` permet de cibler le code de la branche à fusionner

# Résoudre les conflits

- Comment ?
  - Il faut ensuite éditer les fichiers concernés pour supprimer les séparateurs et ne garder que le code désiré.
  - Il arrive que le code conservé soit un mix des deux versions pour avoir une version finale correcte.
  - Il faut ensuite ajouter le fichier à l'index (avec `git add`) et le commuter (avec `git commit`)

# Résoudre les conflits

- Exemple

```
<<<<<< develop:index.html
<h1>Hello !</h1>
=====
<div class="jumbotron">
  <h1>Hello World !</h1>
</div>
>>>>>> feature/test:index.html
```

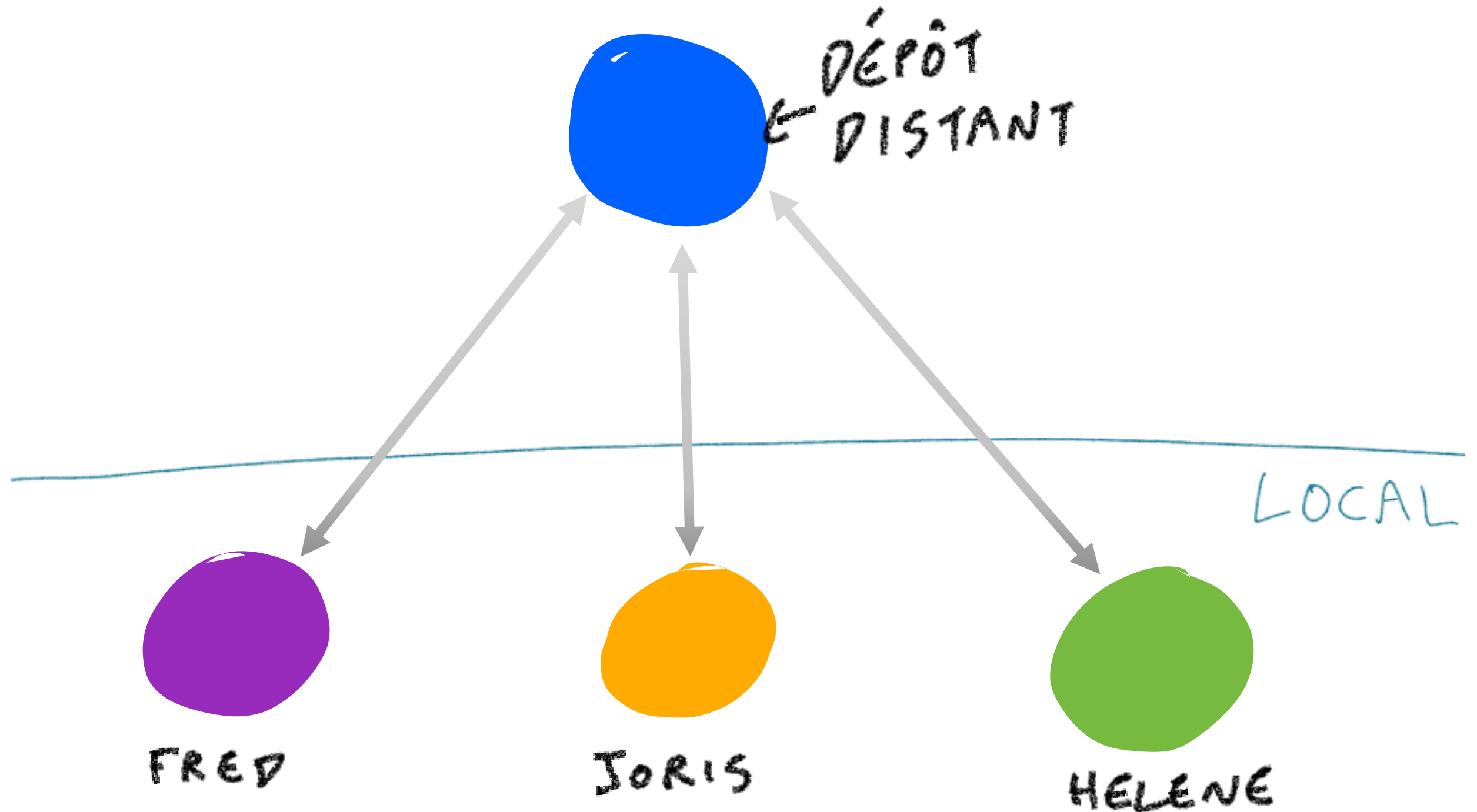


# Travail EN équipe

# Dépôt Distant/Central

- L'intérêt de Git est de permettre à plusieurs développeurs de travailler sur le même projet
- Chaque développeur possède son propre dépôt lié à un dépôt distant commun à tous les développeurs
- Ainsi, chaque développeur pourra envoyer au dépôt distant les commits qu'il a effectué dans son dépôt local
- Mais aussi chaque développeur pourra récupérer les commits partagés par les autres développeurs

# Dépôt Distant/Central





# Dépôt Distant

Les Solutions proposées



# Dépôt Distant

LA Solution étudiée



# Cloner un dépôt

- Quand ? Un développeur souhaite participer à un projet dont le dépôt existe déjà
- Pourquoi ? Pour travailler sur le dépôt local et dans le but d'envoyer ses développements sur le dépôt distant pour que tout le monde puisse en bénéficier

# Cloner un dépôt

- Comment ?
  - En utilisant la commande `git clone <lien vers le dépôt distant>`
  - Le lien peut être un lien HTTP, SSH, FTP, ...
  - Le dépôt généré par la commande contiendra automatiquement un « lien » vers le dépôt d'origine que l'on appellera `remote` par la suite

# Envoyer des commits

- Quand ? Après un commit dans une branche partagée par les développeurs du même dépôt distant.
- Pourquoi ? Pour que chacun bénéficie des développements des autres
- Comment ?
  - En utilisant la commande `git push <nom du remote> <nom de la branche>`
  - Le nom du remote ou des remote disponibles peut être trouvé dans la configuration locale du dépôt

# Mettre à jour le dépôt

- Quand ? Avant de créer une branche pour un nouveau développement ou à tout moment pour mettre sa version à jour des nouveaux développements de l'équipe
- Pourquoi ? Pour recevoir les nouveaux commits du dépôt distant
- Comment ?
  - En utilisant la commande `git pull`
  - Les nouveaux commits seront alors intégrés à la branche actuelle

# GitFlow

- GitFlow est ce que l'on appelle un workflow, ou autrement dit une stratégie d'utilisation de Git
- Il en existe plein d'autre mais GitFlow est l'un des plus connu
- GitFlow est un ensemble de règles simples qui se basent sur le fonctionnement par branche de Git
- Cela permet d'avoir une façon commune de travailler



# GitFlow

- De base vous aurez deux branches :
  - main —> il s'agit de la copie de la version de production du projet
  - develop -> va regrouper tous les développements de la future version qui se retrouvera par la suite sur la branche master
- Il faut se forcer à ne pas envoyer vos modifications directement sur ces deux branches. **C'est à dire uniquement des fusions d'autres branches !**

# GitFlow

- Trois types de branches viennent se greffer aux deux précédentes :
  - feature/XXX —> on développe une fonctionnalité pour la prochaine version
  - release/XXX —> on prépare une version à mettre en production à tester et recetter
  - hotfix/XXX —> correction d'un bug en production

# GitFlow

- Pour mettre en place tout ce workflow il existe encore d'autres commandes à découvrir
- Cependant nous allons découvrir des logiciels qui vont nous permettre de passer par des interfaces graphiques et nous mettrons en pratique GitFlow à ce moment
- En attendant regardons ce site très visuel pour comprendre une bonne fois pour toute la philosophie de GitFlow : <https://danielkummer.github.io/git-flow-cheatsheet/>



# Logiciels

# Sourcetree

- Il s'agit d'un logiciel disponible sur Windows et macOS
- On parle d'un client GIT (client Mercurial également)
- Il propose de tout faire ce que nous avons vu jusqu'à présent via des interfaces graphique
- En bref, Sourcetree va largement nous simplifier la tâche au quotidien
- Cependant l'apprentissage des commandes Git reste un mal nécessaire pour vraiment comprendre la philosophie de Git...

# Sourcetree

- Sourcetree appartient à la société Atlassian tout comme Bitbucket ce qui va nous offrir une intégration aux petits oignons...
- Pour autant vous pouvez utiliser Sourcetree pour interagir avec des dépôts sur GitLab, GitHub, etc...

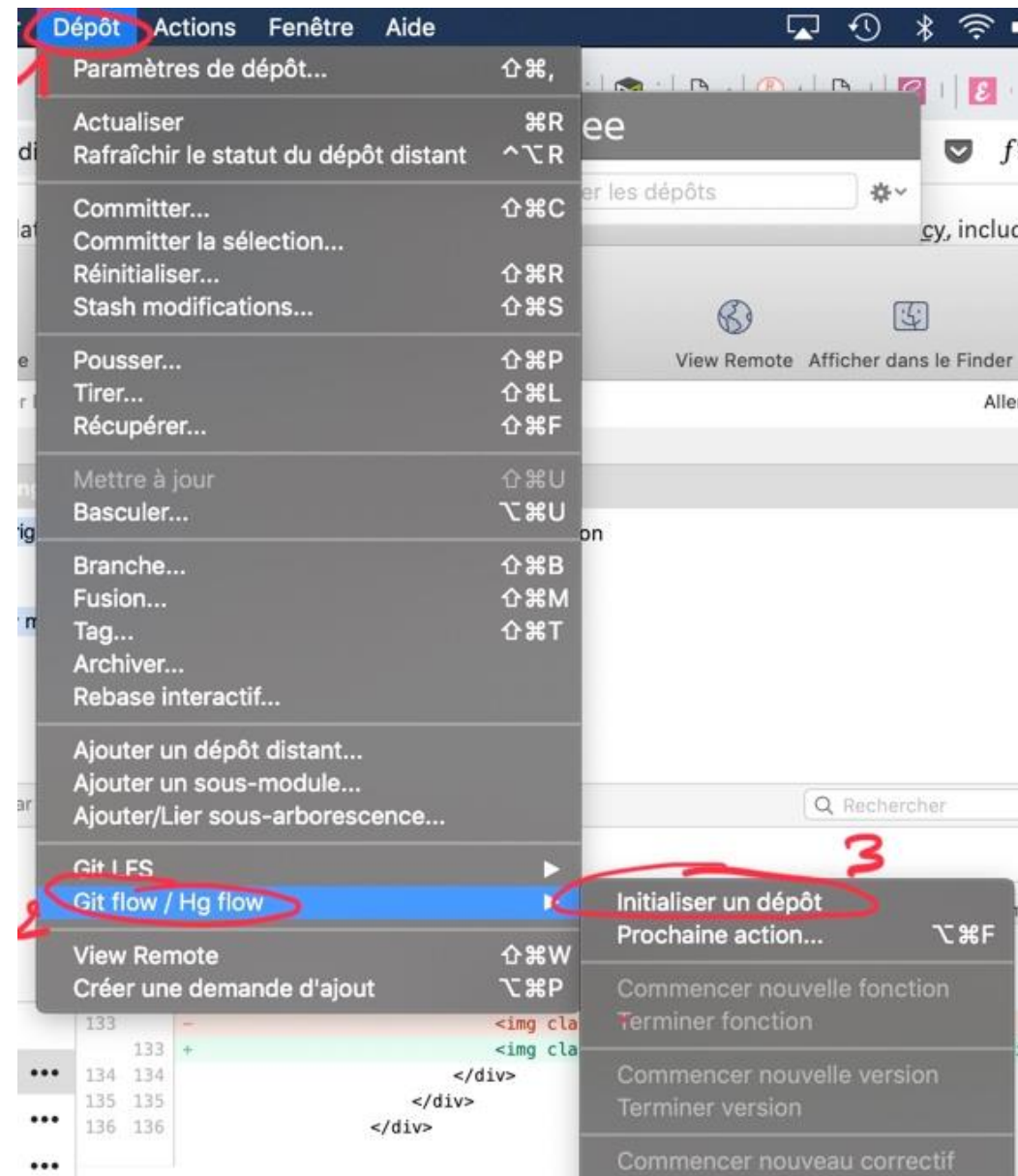
# D'autres logiciels ?

- Il n'existe pas que Sourcetree comme client Git
- Voici deux autres très bons clients Git :
  - Tower
  - GitKraken



# Sourcetree + Gitflow \O/

- Quoi ? On peut implémenter le Workflow GitFlow dans Sourcetree ?
- Oui oui et c'est super !



Je dirais même plus :  
Jira + Bitbucket +  
SourceTree + gitflow





# Pratique

# Pratique

## créer Un projet De Zéro sur Git

- ☞ En utilisant Bitbucket
- ☞ En utilisant Gitflow
- ☞ En intégrant les fichiers de tout le monde
- ☞ Simuler
  - le développement d'une nouvelle fonctionnalité
  - la résolution d'un bug critique
  - le déploiement d'une nouvelle version

*Travail d'équipe et de  
communication  
indispensable...*