

**GIT : gerer le versionning et utiliser Egit**

# **CFD**

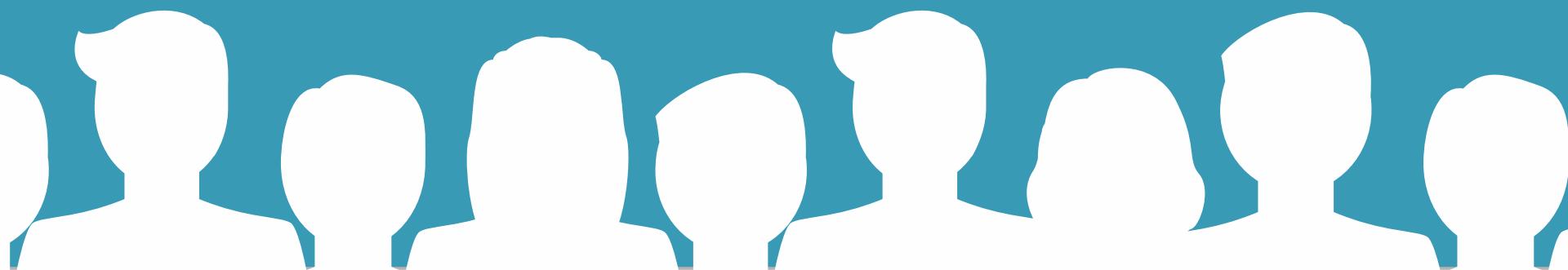
Du Lundi 14 au Mardi 15 d'Ã©cembre 2020



**ORSYS**  
formation



**Vous souhaite la bienvenue**



# L'expertise ORSYS

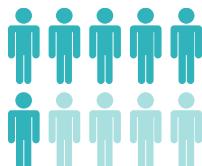
**40** ANS D'EXPÉRIENCE



Plus de **1 800 FORMATIONS** au  
**MANAGEMENT, MÉTIERS ET TECHNOLOGIES**



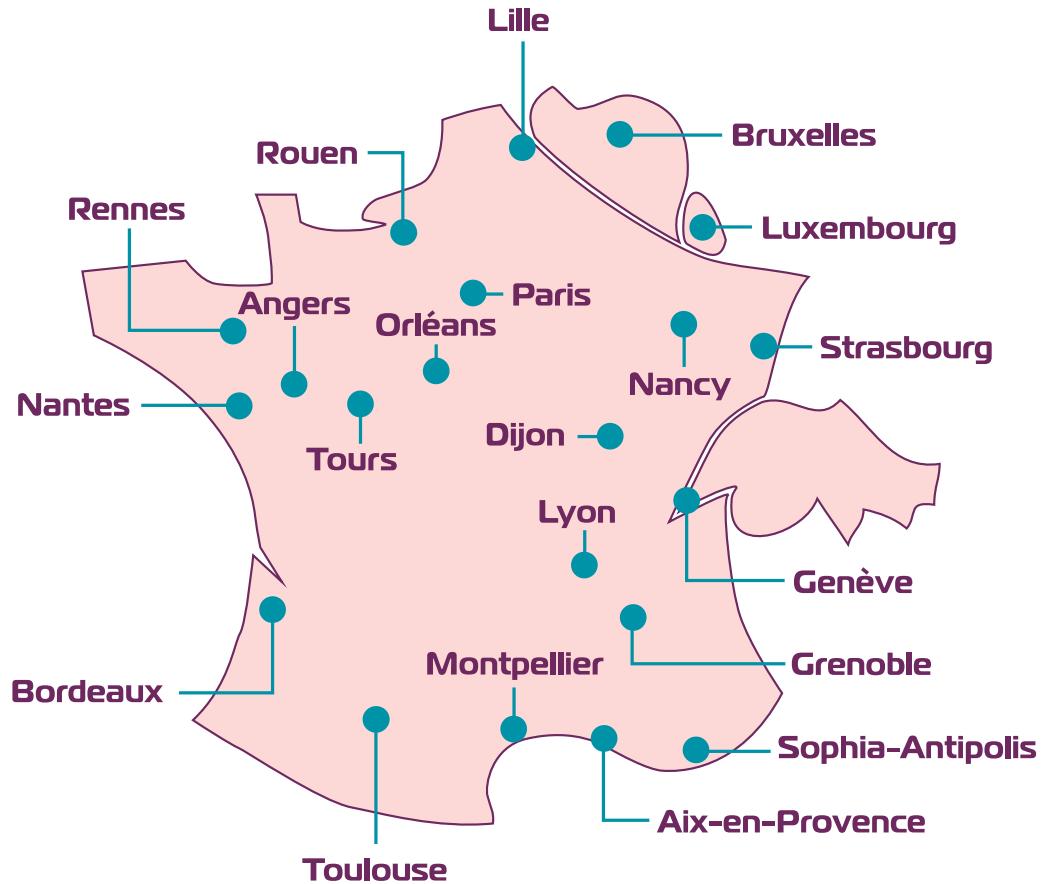
**UNE DÉMARCHE QUALITÉ** (validation intervenants, OPQF)



Près de **50 000 PARTICIPANTS** en 2015  
avec un **TAUX DE SATISFACTION 97,4%**

# Nos centres : près de chez vous

Une présence dans  
4 capitales européennes  
et 18 villes en France



# La méthode ORSYS

## Les animateurs

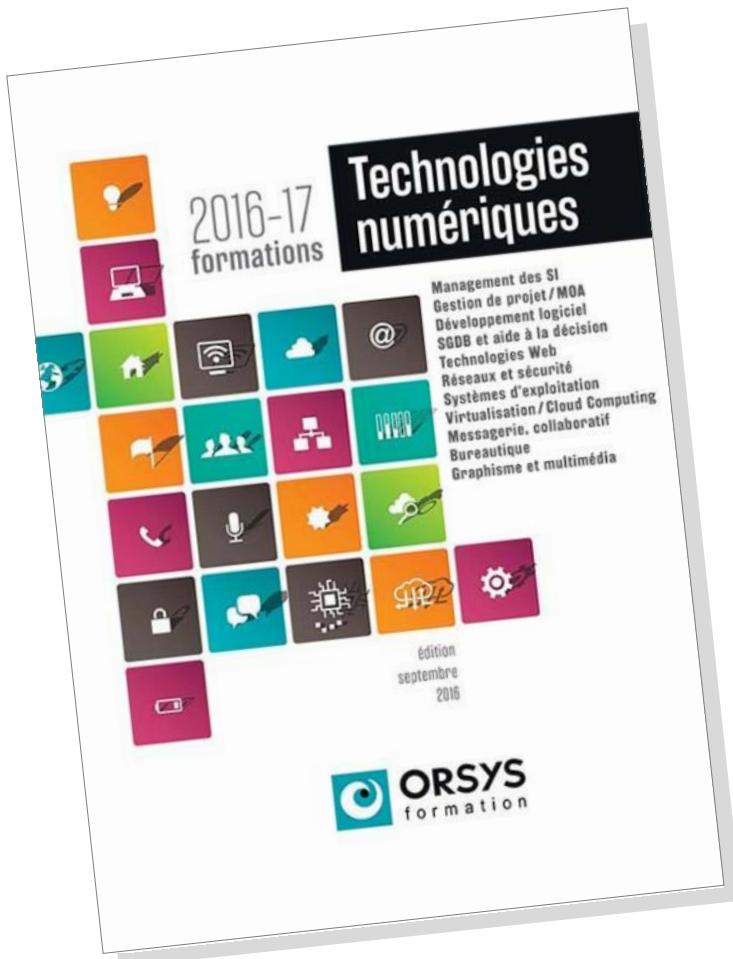
- Plus de 900 spécialistes reconnus et validés sur leur pédagogie, expertise et expérience,
- Responsables de projets d'avant-garde, hommes et femmes de terrain.

## Une orientation opérationnelle

- Une conception originale Orsys,
- Des formations directement applicables,
- Des exercices, des études de cas et des jeux de rôles,
- Blended, E-learning, Full-learning, serious games,
- Des certifications et des préparations aux certifications.



# Les domaines abordés



# Votre journée chez ORSYS



Les cours débutent à **9h** et se terminent aux alentours de **17h30**. Les déjeuners ont lieu entre 11h45 et 14h,

**9h – 17h30**

Pour les stages pratiques de 4 ou 5 jours, les sessions se terminent à **15h30** le dernier jour.



**Les salles de pause sont équipées** en matériel informatique et d'un accès Wi-Fi,

ORSYS a le plaisir de vous **inviter à déjeuner** et vous offre les collations lors des pauses jusqu'à la fin de votre formation.

# Avant de partir...

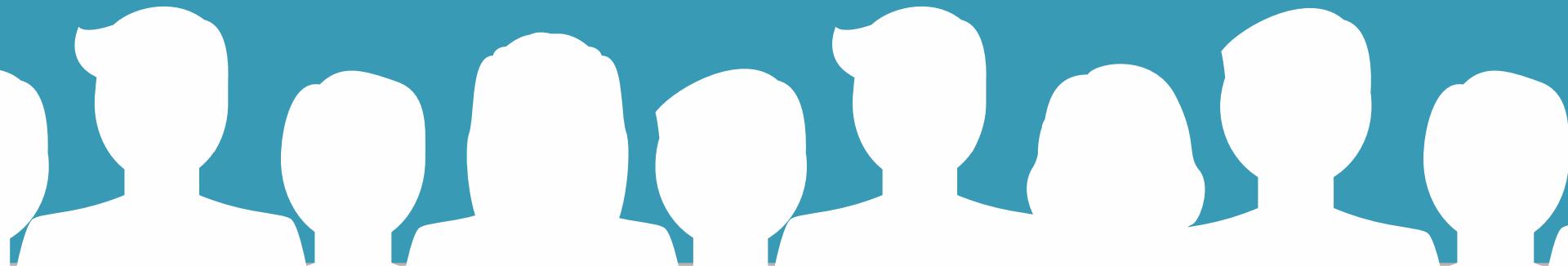
**N'oubliez pas de compléter la feuille d'évaluation !**



- C'est une **obligation légale** qui permet à votre service formation d'avoir un retour sur la qualité du cours suivi.
- Cela nous permet également de **mieux cerner vos attentes** concernant le contenu des cours et les formations que nous pourrions mettre en place.

**ATTENTION ! un badge** vous a été remis ce matin : il est indispensable pour l'accès à La Grande Arche, le matin ou lors des pauses. **Prenez-le toujours avec vous** et remettez-le aux hôtesses du rez-de-chaussée à la fin de votre formation.

**Après votre formation,  
gardez le contact !**





**Merci de votre attention**

Nous vous souhaitons une bonne formation



# Présentation

Romain THERRAT

POCKOST

28 janvier 2020

# Sommaire

## 1 Les SCMs

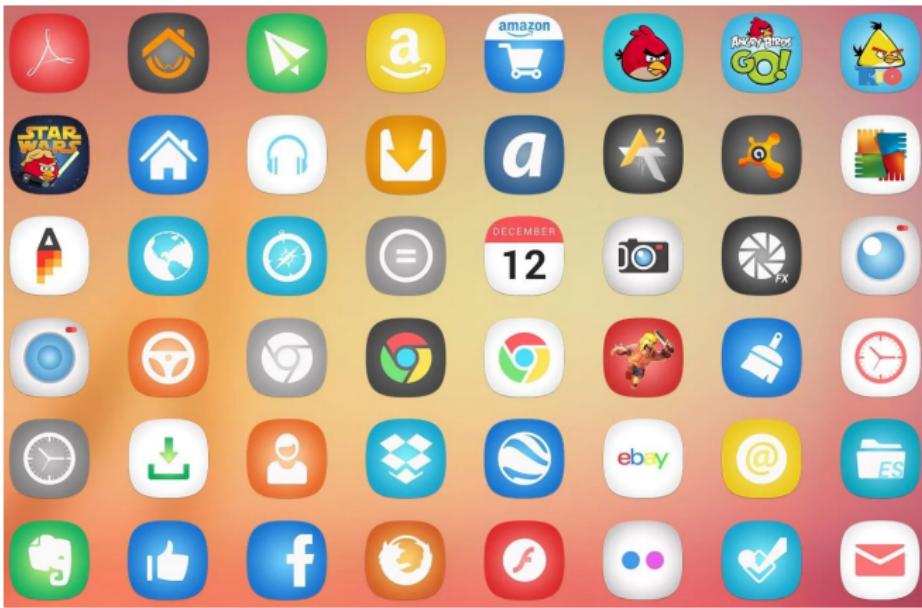
- L'application
- Sauvegarde et partage

## 2 Les acteurs

## 3 Gestion centralisée vs décentralisée

# Qu'est-ce qu'une application

Qu'est-ce qu'une application ?



# Qu'est-ce qu'une application

Qu'est-ce qu'une application ? . . . Un exécutable !

- Des instructions CPU
- De la mémoire
- Des 1 et des 0

Nous en utilisons tous les jours

- La Calculatrice
- Word
- Antivirus
- Serveur HTTP
- . . .

## Comment créer une application ?

- Code source
- Compréhensible
- Un langage
  - Français/Anglais/Allemand/...
  - C/Java/PHP/Python
- Une recette de cuisine

# Le code source élément central

O'l'good Hello world

```
<?php
    echo 'Hello World !';
?>
```

# Compiler ou interpréte

Il existe deux types de langages . . . et demi !

- Compilés
  - Nécessite un compilateur
  - Conversion du code source en code machine
  - C/C++/Brainfuck
- Interprétés
  - Compilateur JIT
  - Conversion en live
  - PHP/Python/Ruby
- Pseudo compilés
  - Java
  - DotNet

Dans tous les cas le code sources est l'élément central pour la construction et l'exécution de l'application

# Problématique

Si le code source est important, plusieurs questionnements doivent attirer notre attention.

- Comment sauvegarder son code source ?
- Comment le partager ?
- Comment historiser les modifications ?

# Problématique : Sauvegarde

Ce qui vient tout de suite à l'esprit

- Au final le code n'est que de simples fichiers
- Pourquoi ne pas le sauvegarder comme mes photos de vacances ?
  - Sur son disque dur
  - Sur une Clé USB
  - Dans le cloud Dropbox/Google Drive/ ...

# Problématique : Sauvegarde

Quels avantages ?

- Simple à utiliser
- Tout le monde connaît ces outils

Mais il y a des inconvénients

- Risque de perte (Matériel)
- Risque de perte (Humaine)
- Encodage
- Le partage n'est pas toujours simple

## Problématique :partage

Ce qui vient à l'esprit est donc de déporter ces fichiers sur un serveur

- Dropbox/Google Drive/OneDrive/...
- FTP
- Partage SMB
- mail (patch noyau Linux)

# Problématique :partage

Avantages :

- Simple à utiliser
- Tous le monde connais

Inconvénients :

- Encodage
- FTP :Édition en production...
- Google/Dropbox :Vendeur externe
- Qui a fait quoi ?

# Problématique : historisation

Encore une solution ?

- Dropbox
- Un nouveau dossier/fichier pour chaque modification
- Archive journalière par mail

# Problématique : historisation

Avantages :

- Simple à réaliser (pas forcément à utiliser)

Inconvénients :

- Par de traçabilité automatique
- Il faut être assidus à la tâche
- Duplication des ressources (espace disque)
- Retour arrière difficile

# Solution

## Les SCM

### Source Control Manager

# Les SCM

Qu'est-ce que c'est ?

- Partage optimisé pour le code source
- Historisable
- Traçable
- Permet un instantané du code
- Des fonctions avancées de collaboration

# Sommaire

- 1 Les SCMs
- 2 Les acteurs
- 3 Gestion centralisée vs décentralisée

# Les acteurs

- CVS
- SVN
- git
- Mercurial
- TeamFoundationServer
- ...

# Les acteurs

Ici nous présenterons

- SVN
- git

Les autres SCM ont un système de fonctionnement équivalent.

# Subversion

- Successeur de CVS
- SCM centralisé
- La référence . . . dans les années 2000
  - Apache foundation
  - Écrit en C
  - Une partie client
  - Une partie serveur (svnserv)
  - Intégrable avec la majorité des produits du marché (Trac, Redmine, IDE, . . . )

# Installation

- Multi plateforme
- En CLI commande svn
- Des clients graphiques
  - TortoiseSVN
  - Fonctionnement équivalent

- Crée en 2005
- Linus Torvald
- ... d'où son nom
- Besoin du noyau Linux
- Idée d'un système de fichier
- Une référence aujourd'hui

# Sommaire

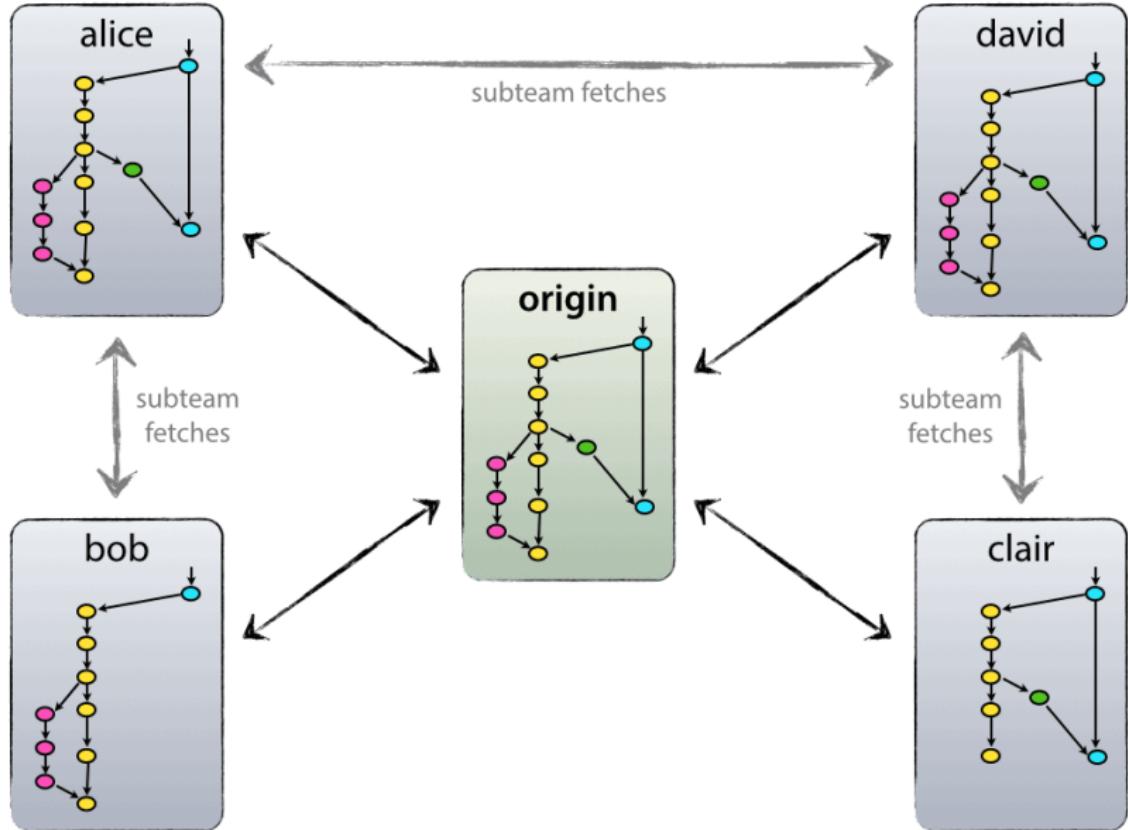
- 1 Les SCMs
- 2 Les acteurs
- 3 Gestion centralisée vs décentralisée

# Centralisation ou distribution

Nous avons deux types de stockage du code

- Centralisé
  - SVN
  - Un serveur central
  - Tout doit toujours être envoyé sur ce serveur
- Décentralisé
  - git
  - Zéro, un ou plusieurs serveurs
  - Plus moderne
  - Nous y reviendrons ...

# Décentralisation



Merci de votre attention.

On continue avec l'installation de git ?

# Installation et configuration

Romain THERRAT

POCKOST

28 janvier 2020

# Sommaire

1 CLI et GUI

2 Installation

3 Configuration

4 Labs

- Crée à l'origine pour Linux
- Donc : CLI <3
- Commande : git
- CLI vs GUI

CLI Command Line Interpreter

GUI Graphical User Interface

Nous présenterons les exemples en CLI mais des alternatives graphiques existent

- La CLI couvre 100% des possibilités
- Les interfaces graphiques peuvent :
  - Masquer des fonctionnalités
  - Complexifier la gestion (afficher trop de possibilités)
- Nous présenterons tous de même quelques solutions graphiques :).

# Utilisation de la CLI

Les commandes git sont toujours composées d'une commande et peuvent avoir un ensemble d'arguments

```
romain@laptop:~$ git command arg1 arg2 ...
```

Par exemple si vous souhaitez créer un dépôt vide dans un dossier

```
romain@laptop:~$ git init ./myproject
Initialized empty Git repository in
/home/romain/myproject/.git/
```

# Sommaire

1 CLI et GUI

2 Installation

3 Configuration

4 Labs

# Installation sous Linux

- Sous Linux installation simple

Debian `apt-get install git`  
RH,CentOS `yum install git`  
Gentoo `emerge install git`

...

- La commande git disponible dans le shell

```
romain@laptop:~$ git --version
git version 2.20.1
```

- Sous Windows plusieurs solutions
  - msysgit Obsolète
  - git for windows
  - Intégration à des IDEs
  - TortoiseGIT

- Git For Windows

- Basé sur le code source officiel de git
- Installation simple

- 

- <https://github.com/git-for-windows/git/releases/latest>

- Version 32 ou 64bit

- Contient aussi

- git Bash : Émulation de bash pour Windows
- git GUI : Deux interfaces graphiques pour la visualisation et l'exécution d'actions
- Intégration dans le menu contextuel (clique droit)

# Git For Windows

The screenshot shows a terminal window titled "MINGW32:~/git". The window contains the following text:

```
Welcome to Git (version 1.8.3-preview20130601)

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

Bacon@BACON ~
$ git clone https://github.com/msysgit/git.git
Cloning into 'git'...
remote: Counting objects: 177468, done.
remote: Compressing objects: 100% (52057/52057), done.
remote: Total 177468 (delta 133396), reused 166093 (delta 123576)
Receiving objects: 100% (177468/177468), 42.16 MiB | 1.84 MiB/s, done.
Resolving deltas: 100% (133396/133396), done.
Checking out files: 100% (2576/2576), done.

Bacon@BACON ~
$ cd git

Bacon@BACON ~/git (master)
$ git status
# On branch master
nothing to commit, working directory clean

Bacon@BACON ~/git (master)
$
```

# Git For Windows

The screenshot shows the Git Gui interface with the following details:

- Repository Bar:** Repository, Edit, Branch, Commit, Merge, Remote, Tools, Help.
- Current Branch:** master
- Staged Changes (Will Commit):** A list containing `compat/winansi.c`.
- File Content:** The code for `compat/winansi.c` is shown in the main pane, with changes highlighted in red and green. The code includes functions like `warn_if_raster_font`, `is_console`, and `write_console`.
- Commit Message:** A text input field for the commit message, currently empty.
- Buttons:** Stage Changed (highlighted), Rescan, Sign Off, Commit, Push.
- Status:** Ready.

- Pour SVN on avait TortoiseSVN
- Équivalent
- Peut être utile si on sait ce qu'on fait
- Intégration au menu contextuel

# Sommaire

1 CLI et GUI

2 Installation

3 Configuration

4 Labs

Nous avons trois types de configuration

- Pour tout le monde (`/etc/gitconfig`)
- Par utilisateur (`/home/user/.gitconfig`)
- Par dépôt (`.git/config`)

# Configuration

- Toutes les configurations sont réalisées via la commande `git config`
- Arguments
  - `-system` Configuration pour tous les utilisateurs
  - `-global` Configuration pour tous les projets de l'utilisateur courant
  - `rien` Configuration pour le projet dans lequel nous sommes actuellement
- Exemples

```
$ git config --global user.name "Romain THERRAT"  
$ git config --global user.email "romain@pockost.com"
```

Voici quelques exemples de configuration

`user.name` Nom de l'utilisateur (Affiché dans l'historique)

`user.email` Email de l'utilisateur (Affiché dans l'historique)

`commit.template` Template de base pour les messages d'historique

`core.excludesfile` Liste de fichier à ne jamais versionner (MacOS ?)

`core.autocrlf` Convertir les CRLF en LF

L'ensemble des options peuvent être visualisées via `man git-config`

- Une exclusion par projet
- Versionné avec celui-ci
- Le fichier `.gitignore`
- Une ligne par exclusion (avec ou sans \*)
- À la base du projet ou par dossier
- Liste de fichier `gitignore` par type de projet  
<https://github.com/github/gitignore>

En plus des configurations standards il est possible d'appliquer des scripts custom : Les hooks (crochets en français ...)

- Côté client
- Côté serveur
- dossier .git/hooks
  - Des exemples : type.sample
  - Doivent être exécutables
- Exemples
  - Valider la forme d'un commentaire de commit
  - Empêcher certaines actions
  - Valider le contenu des fichiers (plus de TODO, ...)

# Sommaire

1 CLI et GUI

2 Installation

3 Configuration

4 Labs

Nous allons procéder à notre premier Labs !

Dans celui-ci nous allons bien évidemment commencer par procéder à l'installation de git sur votre poste. Nous configurerons ensuite notre compte utilisateur et réaliserons quelques commandes git de base.

- Si vous êtes sous Windows installer Git for Windows (<https://github.com/git-for-windows/git/releases/latest>)
- Si vous êtes sous Linux installer git via votre gestionnaire de paquet.

Une fois l'installation réussis, valider le bon fonctionnement de celle-ci via la commande `git --version`. Cette commande est à exécuter dans le shell et doit vous retourner la version de git installée.

Nous allons maintenant configurer votre compte utilisateur. Il conviendra donc d'utiliser la commande `git config`. Nous souhaitons modifier :

- Le nom de l'utilisateur
- Son adresse mail

De la même manière nous allons réaliser des configurations spéciales pour un dépôt.

- Initialiser un projet vide via la commande `git init <path/to/project>`
- Déplacer vous dans ce dossier
- Exécuter la commande `git config -l` pour afficher la configuration actuelle
- Modifier la valeur de `core.autocrlf` à `true` pour votre projet
- Constater le changement via la commande évoquée précédemment

Pour finir nous allons créer un fichier `.gitignore` dans notre projet. Pour constater son fonctionnement nous utiliserons la commande `git status` (ou un affichage graphique). Cette commande sera décrite dans le prochain chapitre.

- Créer un fichier `identifiants.txt` à la racine du projet
- Afficher le résultat de la commande `git status`. Noter l'état du fichier `identifiants.txt`
- Ajouter un fichier `.gitignore` à la racine de votre projet
- Dans ce fichier ajouter la ligne `identifiants.txt`
- Afficher le résultat de la commande `git status`. Constater que le fichier `identifiants.txt` n'apparaît plus
- Rendez-vous sur [github.com/github/gitignore](https://github.com/github/gitignore) et consulter des exemples de fichiers `.gitignore`

# Les fondamentaux

Romain THERRAT

POCKOST

28 janvier 2020

# Sommaire

1 Les objets

2 Le staging area et index

3 Visualisation

4 Labs

- Git est avant tout un système de fichier
- Au-dessus de celui-ci a été construit le SCM
- Avant la version 1.5 la commande git était centrée sur le système de fichier
- git était peut-être complexe . . . avant !
- Pas trop peur ? On y va !

## Le dossier “.git”

Après un `git init` un dossier `.git` est créé dans le dossier parent du projet. Certains fichiers et dossiers sont assez explicites

```
romain@laptop:~/myproject$ ls .git
branches config description HEAD hooks info objects
```

`config` Ce qu'on a configuré avec `git config`

`hooks` Dossier contenant les hooks git pour ce projet

`info` Diverses informations (exclusion propre à l'utilisateur par exemple)

Les éléments les plus importants sont

HEAD La référence vers l'état actuel

objects Réel stockage des données

refs Dossier des différentes branches (wait and see)

index Éléments en attentes de versionnement

## Stockage de données interne

A l'initialisation le dossier objects est vide. Nous pouvons ajouter des données via git hash-objects.

```
$ echo "Hello ;)" | git hash-object -w --stdin  
2183f45093dfa506941dfffc62ab9cf352c2915
```

- Retourne un id (hash)
- Présent dans le dossier objects (chiffré)
- Récupérable via git cat-file -p <id>

Note : On utilise jamais cette commande !

# Stockage interne et version

- Pourquoi faire ça ?
  - Avec un simple référence on pointe un fichier
  - Possibilité de gérer plusieurs versions

```
$ echo "version 1" > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
$ git hash-object -w test.txt
$ echo "version 2" > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30
version 2
```

- objects = Contenu d'un fichier
- blob : Binarie Large OBject
- Aucune notion de nom de fichier
- Aucune notion d'arbre (Hiérarchie)
- Comment grouper les object entre eux
- Solution :

Les arbres (tree)

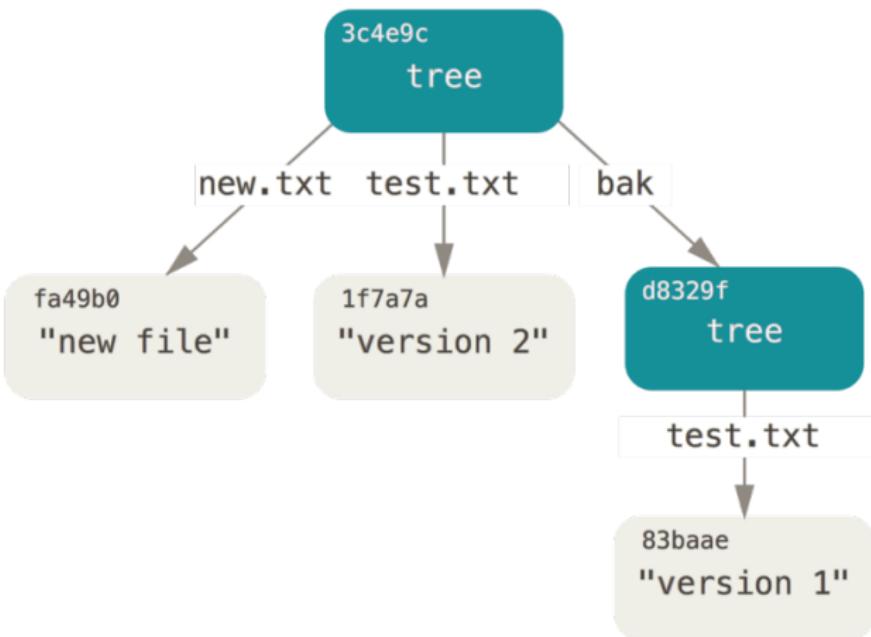
# Les arbres

Les arbres :

- Correspondance entre un object et un nom
- Contient le mode (droits)

```
$ git update-index --add --cacheinfo 100644 83baae61804e  
$ git write-tree  
c1113de8ce1a603f7fdff0149c4ccfcdbfc823c3  
git cat-file -p c1113de8ce1a603f7fdff0149c4ccfcdbfc823c3  
100644 blob 83baae61804e65cc73a7201a7252750c76066a30
```

# Les arbres



Avec les object et les tree nous avons encore un problème

- Comment avoir un référentiel des différentes versions d'un fichier ?

Solution

- Les commits

# Les commit

Qu'est-ce qu'un commit ?

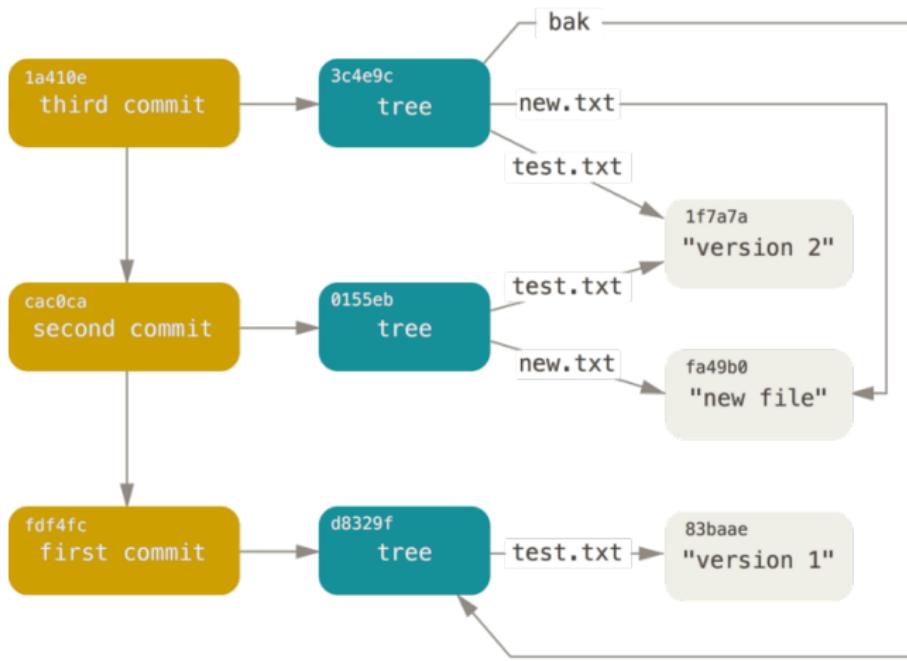
- Un object
- Référence un arbre
- Et un ancien commit

```
$ echo 'first commit' | git commit-tree c1113de8ce1a603f  
f5c427827e1cb89ad7de521448cad48de6c2d264  
$ git cat-file -p f5c427827e1cb89ad7de521448cad48de6c2d2  
tree c1113de8ce1a603f7fdff0149c4ccfcdbfc823c3  
author Romain TERRAT <romain@pockost.com> 1579270598 +0  
committer Romain TERRAT <romain@pockost.com> 1579270598
```

first commit

```
$ echo "second commit" | git commit-tree <new-id> -p f5c427827e1cb89ad7de521448cad48de6c2d264
```

# Les commits



# Sommaire

1 Les objets

2 Le staging area et index

3 Visualisation

4 Labs

On utilise jamais ces commandes vues précédemment

- Trop complexe
- Trop de risque d'erreur
- Pas de fonctions avancées

On va utiliser des commandes de plus haut niveau mais le cœur de git reste le même.

# Créer un commit

Pour créer un commit il est nécessaire de

- Créer des fichiers (=votre code source)
- Ajouter ces fichiers pour le prochain commit (=staging) via la commande `git add <files>`
- Créer un commit via `git commit -m "comment"`

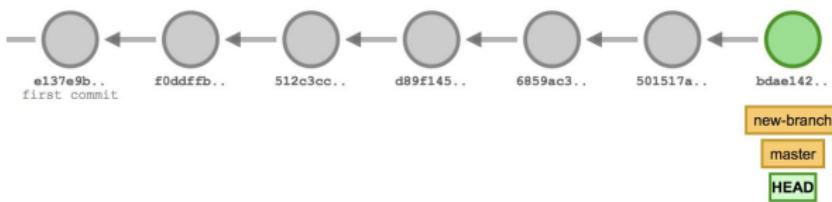
Note : Vous pouvez, à tout moment, voir l'état de votre dépôt via la commande `git status`.

# Créer un commit

```
$ touch fichier1.txt
$ git status
...
Untracked files:
    fichier1.txt
...
$ git add fichier1.txt
$ git status
...
Changes to be committed:
    new file:   fichier1.txt
...
$ git commit -m "First commit"
[master (root-commit) 9a3e68e] First commit
 1 file changed, 0 insertions(+), 0 deletions(-)
```

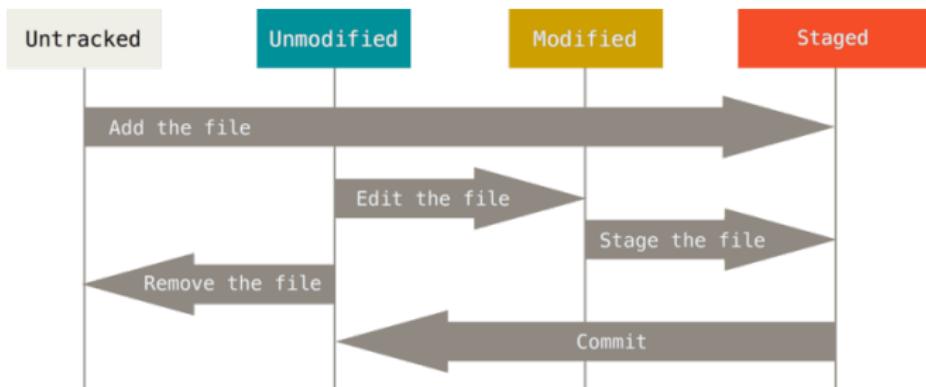
# Créer un commit

Les commits se suivent



- Idem pour les
  - Modifications de source
  - Suppressions de fichier

# État des fichiers



# Sommaire

1 Les objets

2 Le staging area et index

3 Visualisation

4 Labs

- Que pouvons-nous voir ?
  - Historique des commits
  - `git log`
  - Fichier modifié et/ou à commiter
  - `git status`
  - Changement à commiter
  - `git diff`
  - Changement réalisé par un commit
  - `git show` et `git blame`

# git log

```
$ git log
commit d515e5c2b0b2b2ba5797cbfd996f0da2a4f72c6d
Author: Romain THERRAT <romain@pockost.com>
Date:   Sun Jan 19 16:14:28 2020 +0100

    Second commit
```

```
commit 9a3e68e69a077711bb1465967f1f771bd3846633
Author: Romain THERRAT <romain@pockost.com>
Date:   Fri Jan 17 15:26:05 2020 +0100

    First commit
```

- Option --stat
- Option -p -2

# git status

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   fichier2.txt

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

        fichier3.txt

no changes added to commit (use "git add" and/or "git commit")
```

# git diff

```
$ git diff fichier2.txt
diff --git a/fichier2.txt b/fichier2.txt
index e69de29..e965047 100644
--- a/fichier2.txt
+++ b/fichier2.txt
@@ -0,0 +1 @@
+Hello
```

## git show

```
git show d515e5c2b0b2b2ba5797cbfd996f0da2a4f72c6d
commit d515e5c2b0b2b2ba5797cbfd996f0da2a4f72c6d
Author: Romain THERRAT <romain@pockost.com>
Date:   Sun Jan 19 16:14:28 2020 +0100
```

Second commit

```
diff --git a/fichier1.txt b/fichier1.txt
index e69de29..8bdf806 100644
--- a/fichier1.txt
+++ b/fichier1.txt
@@ -0,0 +1 @@
+changement
diff --git a/fichier2.txt b/fichier2.txt
new file mode 100644
```

# git blame

```
git blame fichier1.txt
```

```
d515e5c2 (Romain THERRAT 2020-01-19 16:14:28 +0100 1) change  
bebc26f9 (Romain THERRAT 2020-01-19 16:32:54 +0100 2) change
```

# Visualisation gitk

Nous pouvons aussi utiliser l'utilitaire gitk

The screenshot shows the gitk application window titled "myproject: All files - gitk". The interface displays a commit history with three entries:

- Local uncommitted changes, not checked in to index
- Second commit (highlighted in yellow)
- First commit

Details for the second commit:  
Author: Romain THERRAT <romain@pockost.c>  
Parent: d035e5c2bfb2b3a5787cfdf99f0da2a4f72cd6 (Second commit)  
Branch:   
Followed:  
Preceded:  
Local uncommitted changes, not checked in to index  
-----  
index a9de29... e9e5047 109644  
R0 -0.0 +1 00  
mello

# Sommaire

1 Les objets

2 Le staging area et index

3 Visualisation

4 Labs

Nous allons réaliser un labs pour mettre en œuvre toutes ces nouvelles connaissances. L'idée de celui-ci est de présenter la création d'un projet git local, l'ajout et la modification de fichier ainsi que la gestion des commits.

- Placer vous dans un nouveau dossier vide. Celui-ci contiendra à terme le code source de votre projet.
- Créer un nouveau projet git via la commande `git init`
- Vous pouvez constater via la commande `git status` que votre dépôt a bien été créé et que celui-ci ne contient aucune donnée.
- Ajoute un fichier nommé `index.php` vide
- Constater le changement d'état de la commande `git status`
- Utiliser les commandes `git add` et `git commit` pour versionner ce premier fichier.
- Exécuter les commandes `git status` et `git log` pour constater les changements.

- Ajouter dans votre fichier index.php du code source (<?php echo "hello world"; par exemple)
- Créer un nouveau dossier vide nommé static
- Créer un nouveau fichier vide nommé .htaccess
- Exécuter les commandes git status et git diff pour constater les changements.
- Créer un nouveau commit avec ces modifications
- Ajouter une image dans le dossier static et utiliser la commande git rm pour supprimer le fichier .htaccess. Commiter le tout
- Exécuter les commandes git status et git log pour constater les changements.
- Visualiser le résultat en GUI avec l'utilitaire gitk
- Réaliser un git blame sur le fichier index.php

Si vous souhaitez aller plus loin vous pouvez

- Réaliser quelques commit sur le site [git-school.github.io/visualizing-git/](https://git-school.github.io/visualizing-git/) pour constater l'évolution de votre arbre
- Vous pourrez utiliser ce site par la suite
- Analyser les fichiers créés dans l'exercices dans le dossier .git
- Récupérer le contenu d'un fichier à l'aide de la commande git cat-file

# Utilisation locale

Romain THERRAT

POCKOST

28 janvier 2020

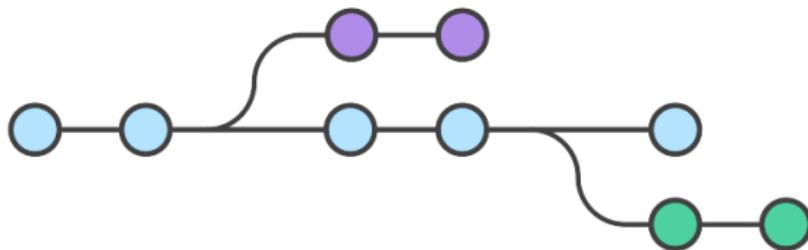
# Sommaire

- 1 Les branches et les Tags
- 2 Historique et modification
- 3 Labs

# Qu'est-ce qu'une branche ?

- Divergence dans le code
- Nouvelle fonctionnalité = copie du code
- C'est le cas de SVN
- git apporte une autre approche
  - Simple
  - Évite la duplication
  - Beaucoup plus léger (espace disque)
- À utiliser le plus possible !

# Les branches



# Comment fonctionnent les branches ?

Une branche c'est

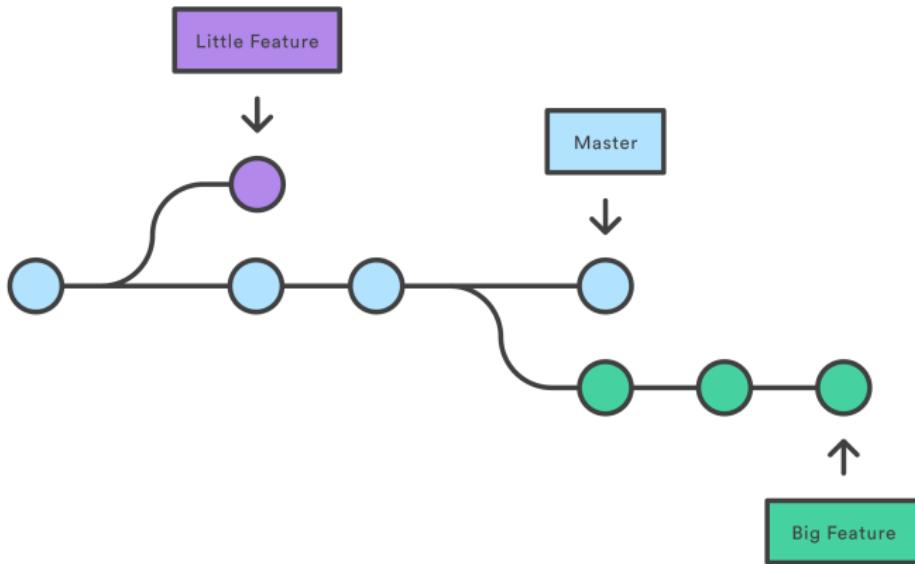
- Un nom
- Un numéro de commit

Et par défaut

- Nom `master`
- Une nouvelle branche = commit courant

Donc passer d'une branche à l'autre ne revient qu'à afficher un commit particulier

# Les branches



- Voir la branche courante
- `git status`
- Créer une nouvelle branche
- `git branch nom` ou `git checkout -b nom`
- Changer de branche
- `git checkout nom`
- Note : Il peut être impossible de changer de branche si vous avez des modifications non commitées qui entre en conflit avec la branche de destination.
- Nous verrons en détail les fusions et cie dans le prochain épisode.

# git branch

```
$ git branch dev
$ git checkout dev
Switched to branch 'dev'
$ git status
On branch dev
nothing to commit, working tree clean
$ git checkout master
Switched to branch 'master'
```

# Qu'est-ce qu'un tag

- Version d'un site
  - v1.0
  - v2.0
  - v2.1
  - ...
- Un nom sur un commit
- Commande git tag

# git tag

## Voir les tags

```
$ git tag  
v1.0  
v1.1
```

## Créer un tag pour le commit courant

```
$ git tag v2.1
```

## Se placer sur un tag

```
$ git checkout v1.1
```

# Sommaire

- 1 Les branches et les Tags
- 2 Historique et modification
- 3 Labs

Nous avons vu dans le chapitre précédent qu'il était possible

- De consulter l'historique des changements
- Se déplacer d'une branche à l'autre

Ceci est possible grâce au fonctionnement des commits. Aussi il est aussi possible de

- Se placer dans un état particulier (=sur un commit)
- Inverser un commit
- Et même ... supprimer un commit !

## Se placer sur un commit

Pour se placer sur un commit nous utilisons la commande

```
git checkout <id>
```

```
git checkout d515e5c2b0b2b2ba5797cbfd996f0da2a4f72c6d
```

Note: checking out 'd515e5c2b0b2b2ba5797cbfd996f0da2a4f72c6d'

You are in 'detached HEAD' state. You can look around, make changes and commit them, and you can discard any commits you state without impacting any branches by performing another command.

If you want to create a new branch to retain commits you created do so (now or later) by using -b with the checkout command as

```
git checkout -b <new-branch-name>
```

HEAD is now at d515e5c Second commit



## Revert

Pour inverser un commit nous utilisons git revert <id>. Ceci crée un nouveau commit

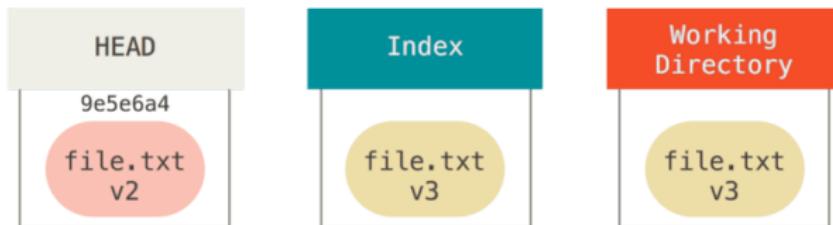
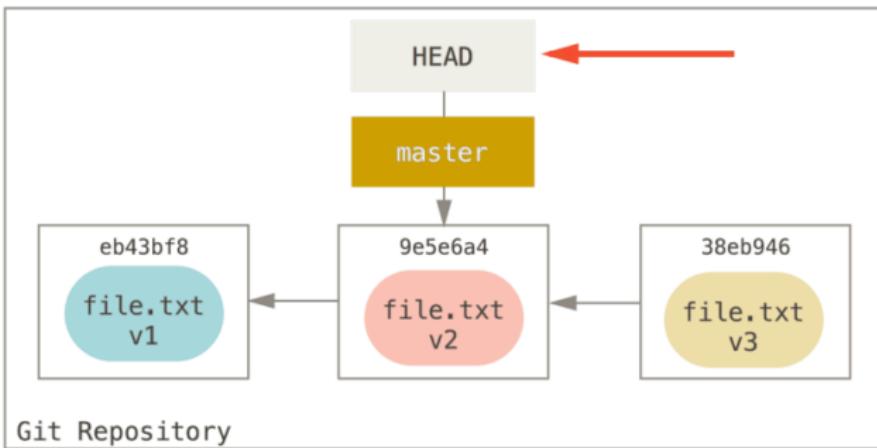
```
git revert bebc26f99eeca85f8b4d24ac6802b82d4ec24dca
[master 0ea76d3] Revert "3de commit"
 3 files changed, 2 deletions(-)
 delete mode 100644 fichier3.txt
```

## reset

Pour supprimer un ou plusieurs commits nous pouvons utiliser la commande `git reset`. On déplace HEAD

```
$ git reset d515e5c  
HEAD is now at d515e5c Second commit
```

`git reset` est aussi utilisé pour sortir un fichier du staging



`git reset --soft HEAD~`

- Un reset soft
  - Ne déplace que HEAD
  - Les modifications sont dans l'index
  - `git reset -soft <id>`
  - Option par défaut
- Un reset hard
  - Déplace HEAD
  - Remet les fichiers de l'index dans l'état de HEAD
  - `git reset -hard <id>`

# Sommaire

- 1 Les branches et les Tags
- 2 Historique et modification
- 3 Labs

Nous allons continuer notre projet précédent en ajoutant une notion de branche à celui-ci.

- Placer vous sur la branche master
- Créer une branche develop basé sur la dernière version de la branche master
- Placer vous dans cette branche et confirmer que vous vous situez bien dans celle-ci via `git status`
- Ajouter un fichier `contact.php` à votre projet et commiter le
- Constater la présence de ce fichier dans l'historique
- Déplacer vous sur la branche `master`
- Constater que votre commit n'est plus présent
- Afficher l'arbre git avec `gitk`

Nous allons maintenant utiliser des tags

- Tagguer le commit de la branche master en tant que v1.0
- Modifier le fichier index.php pour ajouter à la fin une ligne contenant ?>
- Commiter cette modification
- Tagguer ce nouveau commit sous le nom v1.1
- Afficher la liste des tags
- Placer vous sur le commit taggué en v1.0
- Consulter l'historique en CLI et en GUI

Nous allons maintenant supprimer un commit

- Visualiser l'historique
- Faire un revert du commit précédent (ajout d'une ligne dans index.php)
- Constater qu'un nouveau commit a été créé dans l'historique
- Afficher l'état de l'arbre dans gitk
- Utiliser la commande git reset pour annuler les deux précédents commit
- Afficher l'état de l'arbre dans gitk
- Constater que les tags existent toujours

Tous comme dans l'exercice précédent vous pouvez reproduire ces actions sur le site [► git-school.github.io/visualizing-git/](https://git-school.github.io/visualizing-git/).

# Les branches

Romain THERRAT

POCKOST

28 janvier 2020

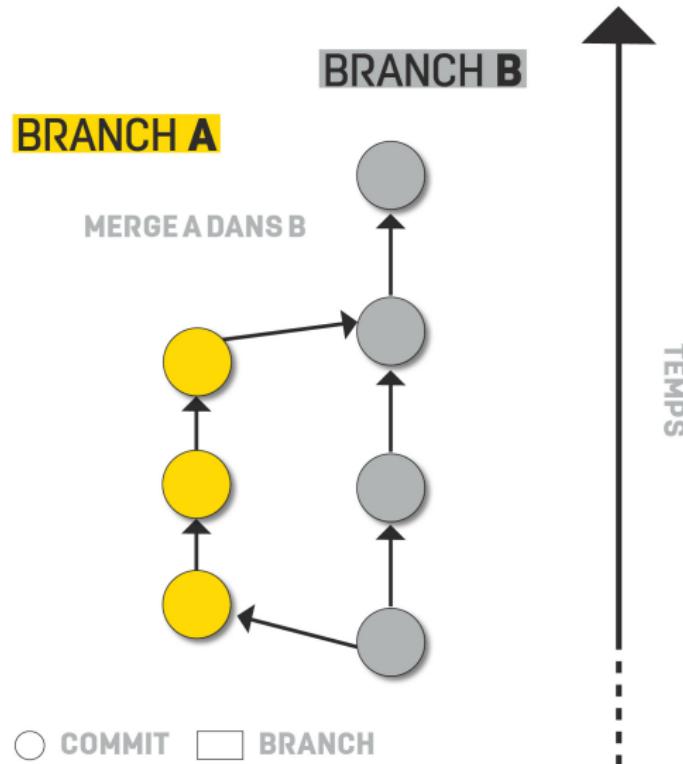
# Sommaire

1 Fusion

2 Rebase

3 Labs

- Les branches peuvent être
  - fusionnées (`merge`)
  - rebasés (`rebase`)
- La fusion est utile pour
  - Intégrer une nouvelle fonctionnalité en préprod
  - Passer la préprod en production
  - Appliquer un correctif urgent en production
  - ...



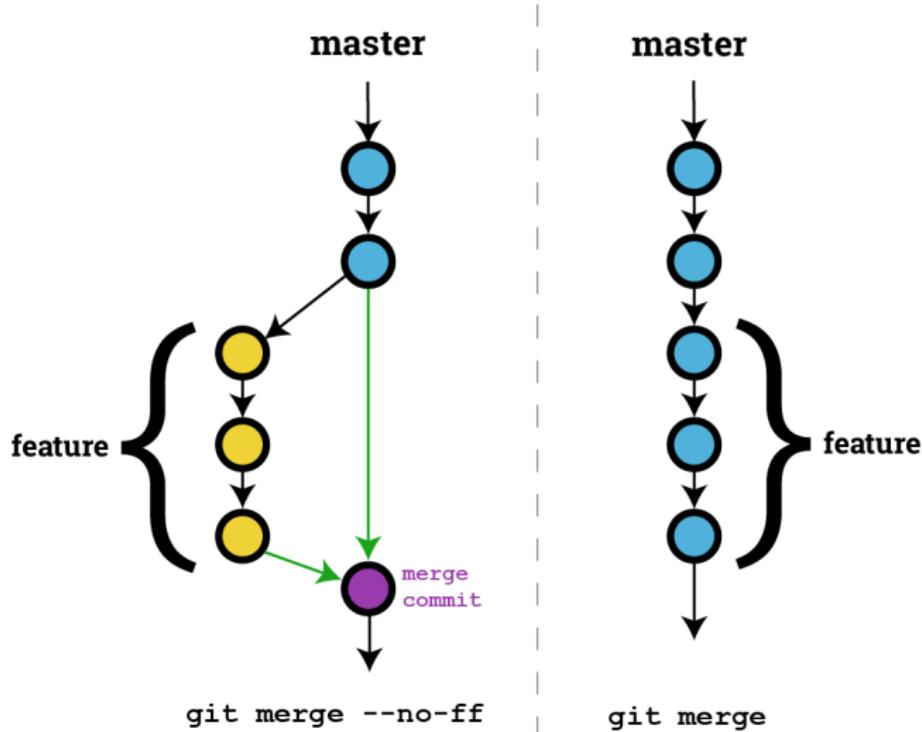
# git merge

```
$ git branch
  dev
* master
  testing
$ git merge dev
Merge made by the 'recursive' strategy.
 new-file-dev.txt | 0
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 new-file-dev.txt
```

Il existe deux type de merge

- Avec un **fast-forward** (défaut)
- Appliquer tous les changements sur la branche de destination
- Sans **fast-forward**
- Créer un commit avec toutes les modifications

# Type de merge



# Sommaire

1 Fusion

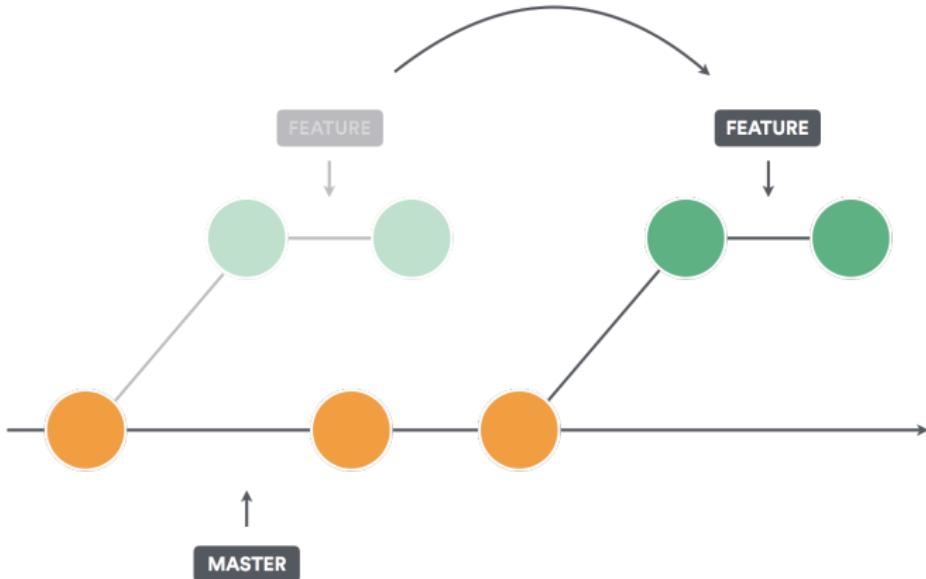
2 Rebase

3 Labs

Pourquoi utiliser le rebase ?

- la branche de base (master?) a évolué
- Récupérer une nouvelle fonctionnalité

L'ensemble de nos commits vont être réappliqués



# Rebase

```
$ git branch |tee  
* dev  
  master  
  testing  
$ git rebase master
```

```
First, rewinding head to replay your work on top of  
Applying: Some comment (dev)
```

# Sommaire

1 Fusion

2 Rebase

3 Labs

Nous allons reprendre notre exercice précédent pour merger nos branches. Nous avons actuellement deux branches :

- master
- develop

Le fichier contact.php n'est présent que sur la branche develop

- Merger votre branche develop dans master
- Constater dans l'historique que le commit ajoutant contact.php a été réintégré
- Observer l'arbre sous gitk

Nous allons maintenant créer deux nouvelles branche pour les faire évoluer indépendamment. L'objectif étant de pouvoir utiliser un rebase

- Créer une branche `feature-about` basée sur `master`
- Créer une branche `feature-contact` basée sur `master`
- Placer vous dans la branche `feature-about` et ajouter un fichier `about.php`
- Placez-vous maintenant dans la branche `feature-contact` et ajouter quelques lignes au fichier `contact.php`
- Constater que vos commits ne sont présent que dans les branches où ils ont été créés
- Merger `feature-about` dans `master`
- Constater que le fichier `feature.php` n'est présent que dans `master` et pas dans `feature-contact`
- Noter le numéro du dernier commit de `feature-contact`
- Utiliser un `rebase` pour rebaser la branche `feature-contact` sur `master`
- Comparer le numéro du dernier commit de `feature-contact`

Nous allons refaire le même exercice mais en utilisant l'option `--no-ff`.

- Créer deux branches `feature-design` et `feature-logo`
- Faites des modifications dans ces deux branches
- Merger la branche `feature-design` dans `master` avec l'option `--no-ff`
- Constater que le numéro de commit de votre modification est le même et qu'un commit de merge a été créé
- Faire un rebase de `master` dans `feature-logo`

Est ce que je vous ai déjà parlé de [git-school.github.io/visualizing-git/](https://git-school.github.io/visualizing-git/) ?

# Le travaille en équipe

Romain THERRAT

POCKOST

28 janvier 2020

# Sommaire

1 Présentation

2 Tirer et pousser

3 Authentification

4 Services en ligne

5 Les conflits

6 Labs

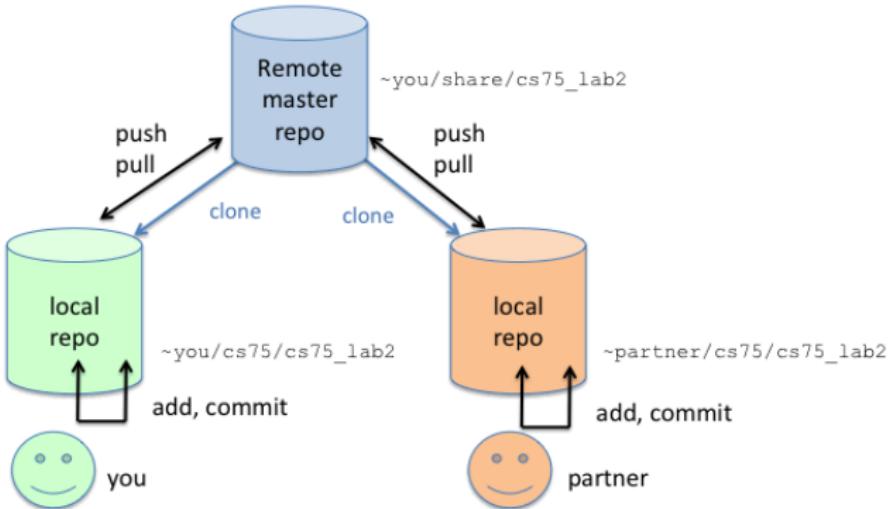
# A quel problème on répond

Actuellement on ne travaille qu'en local

- Comment partager les données
- Serveur centrale ?
- Risque de conflit

Nous utiliserons donc les `remote`.

# Les remote



## git remote

```
$ git remote add origin <url>
$ git remote rm origin <url>
$ git remote show origin
$ git remote set-url origin git@github.com:myteam/myproj
```

Le nom du remote par défaut est master mais vous pouvez en avoir plusieurs

# Sommaire

1 Présentation

2 Tirer et pousser

3 Authentification

4 Services en ligne

5 Les conflits

6 Labs

# push

Un push permet de pousser ses modifications sur un serveur git distant.

- Envoyer ses modifications
- Sur une branche en particulier
- `git push <remote> <branch>`

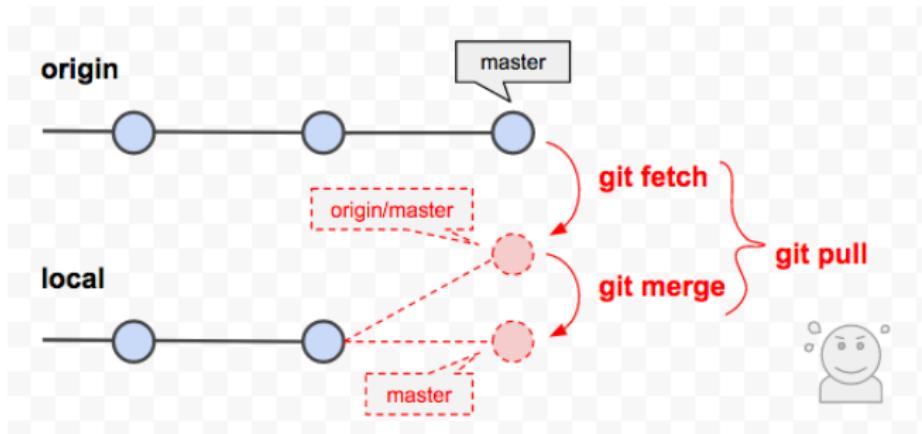
Par exemple `git push origin master` Si vous avez fait des modifications du passé (fast forward) il faudra utiliser l'option `-f`

Contrairement au push, le pull permet de tirer (récupérer) une modification et les appliquer localement.

- `git pull <remote> <branch>`
- Récupérer les modifications et les appliquer
- Équivalent à
- `git fetch` et `git merge`
- `git fetch` = Récupérer les modifications distantes (Nommé `remote/banch`, exemple : `origin/master`)
- `git merge` = Appliquer les nouvelles modifications en local

Par exemple `git pull origin master`

# pull



# Sommaire

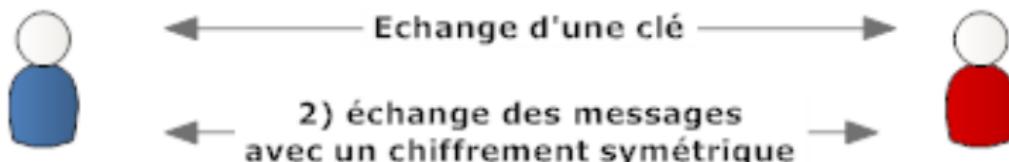
- 1 Présentation
- 2 Tirer et pousser
- 3 **Authentification**
- 4 Services en ligne
- 5 Les conflits
- 6 Labs

Plusieurs modes de connexion à un serveur git sont possibles

- HTTPS
  - Simple
  - Pas de configuration spécifique
  - Auth HTTP username/password
- SSH
  - Utilise les certificats SSH
  - Doivent être générés via ssh-keygen
  - Copier sa clé PUBLIQUE dans GitHub/GitLab
- Démonstration

## Le chiffrement symétrique

# Chiffrement symétrique



Un message chiffré avec la clé est déchiffré avec la même clé.

Le problème : comment transmettre la clé de façon sécurisée ?

copyright Kitpages <http://www.kitpages.fr>

## Le chiffrement asymétrique

### Chiffrement assymétrique

**Principe : On génère 2 clés. Ce qui est encodé avec une clé est décodé par l'autre et réciproquement**



# Les certificats SSH

Les clés privées doivent ... rester privées !!

# Sommaire

- 1 Présentation
- 2 Tirer et pousser
- 3 Authentification
- 4 Services en ligne
- 5 Les conflits
- 6 Labs

- Nombreux acteurs
  - GitHub
  - GitLab
  - Bitbucket
  - ...

- GitHub
- Crée en 2008
- Service en ligne (SaaS)
- Gratuit pour les dépôts publics
- Des options payantes
- Rachat par Microsoft 2018

- Public first = Favorise l'OpenSource
- Les concurrents à l'époque
  - Google source
  - SourceForge
- Une interface simple et lisible
- Le fork and merge multi projet (wait and see)
- Adopté par les principaux projets OpenSource
  - Symfony
  - Twitter Bootstrap
  - JQuery
  - ...

## Utilisation de base

- Après avoir créé un compte
- Cliquer sur “New Repository”
- Choisir un nom et une description
- Équivalent à un git init sur un remote distant
- Démonstration

# Pusher sur mon dépôt

- Si j'ai déjà un dépôt avec des commits
  - \$ git remote add origin <url>
  - \$ git push origin master
- Si je n'ai pas encore de source
  - git clone <url>

# Les forks

- Copie de l'arbre git d'un dépôt sous un autre nom
- Cycle de vie indépendant
- Projet le plus forké ?  
<https://github.com/jtleek/datasharing>
- Vrais projets les plus forkés
  - TensorFlow
  - Twitter Bootstrap
- <https://bit.ly/2M3dw6g>

# Les forks

Exemple d'un dépôt forké

<https://github.com/raphapr/ansible-statuscake/network>

# Les pull requests

- Gestion des issues
- Proposition de solutions
- Peut être automatiquement intégrée
- Comme un merge inter-repository
- Favorise la collaboration

# Les pull requests

Démonstration

# Sommaire

1 Présentation

2 Tirer et pousser

3 Authentification

4 Services en ligne

5 Les conflits

6 Labs

# Les conflits

Qu'est-ce qu'un conflit et pourquoi arrivent ils ?

- Travail à plusieurs sur le même fichier
- Merge de deux branches éditant le même fichier
- ...
- Que devons-nous garder ?

Les conflits sont courant et "normaux".

Le conflit peut se produire de deux méthodes

- J'ai des modifications locales non commitées
- J'ai des commits qui modifient le même fichiers

- Le cas le plus simple
- Mettre les modifications en attente
- git stash et git stash pop
- Pas besoin de créer un commit de merge
- L'arbre reste propre
- Peut être automatiquement géré par git (modifications simples)

# Conflit sur les modifications locales

```
$ git merge create-conflict
Updating 9ac6888..7c498aa
error: Your local changes to the following files would be overwriting
      fichier1.txt

Please commit your changes or stash them before you merge.

Aborting
$ git stash
Saved working directory and index state WIP on master: 9ac6888
$ git merge create-conflict
Updating 9ac6888..7c498aa
$ git stash pop
Auto-merging fichier1.txt
CONFLICT (content): Merge conflict in fichier1.txt
```

# Conflit sur les modifications locales

```
$ git status
On branch master
Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add <file>..." to mark resolution)
    both modified:  fichier1.txt

...
$ git diff fichier1.txt

...
++<<<<< Updated upstream
+Je cherche à créer un conflit
+=====
+ J'ai une modif ici aussi
++>>>>> Stashed changes
```

- Création d'un commit de merge
- Peut être automatiquement géré par git
- Faire les modifications nécessaires à la main
- Ajouter modifications : `git add`
- Commiter le résultat : `git commit`
- Possibilité d'annuler le merge : `git merge --abort`
- Pour se simplifier la vie il est possible d'utiliser l'option  
`--merge` à `git log`

## Conflit sur les modifications committées

```
$ git merge create-conflict
Auto-merging fichier1.txt
CONFLICT (content): Merge conflict in fichier1.txt
Automatic merge failed; fix conflicts and then commit the re...
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)
```

Unmerged paths:

(use "git add <file>..." to mark resolution)

both modified: fichier1.txt

# Sommaire

1 Présentation

2 Tirer et pousser

3 Authentification

4 Services en ligne

5 Les conflits

6 Labs

Dans ce Labs nous allons chercher à utiliser un serveur git central.

- Cloner un dépôt  
<https://github.com/nicholaskajoh/faceclone-html-template>
- Déplacer vous dans ce dossier
- Éditer le fichier index.html pour modifier le titre de la page en “MyFaceClone”
- Commiter la modification
- Réaliser un git push origin master et constater que ceci n'est pas possible

Nous allons maintenant travailler sur un serveur git où nous pourrons réaliser des push. Nous créerons aussi un couple de clés pour l'authentification git.

- Connecter vous à `git.pockost.com` avec les identifiants fournis par votre formateur
- Créer un dépôt vide sur l'interface web
- Dans `git bash` entrer la commande `ssh-keygen` pour générer une paire de clé
- Récupérer le contenu de la clé publique pour l'ajouter dans votre compte GitLab
- Initialiser un dépôt vide en local et commiter un fichier
- A l'aide de `git remote` ajouter l'URL de dépôt créé précédemment (`git@git.pockost.com`)
- Pousser vos modifications `git push`

Après s'être familiarisé avec la notion de push et clone nous allons traiter les cas des pull des conflits et pull request

- Crée un nouveau dépôt vide nommé FaceClone sous GitLab
- Ajouter un second remote dans le projet FaceClone. Vous pouvez le nommer gitlab.
- Pousser les sources de FaceClone dans ce remote
- Constater sur l'interface de GitLab que tous les commits de GitHub sont bien présents.
- Créer une branche nommée new-feature et commiter une modification sur le fichier index.html
- Pousser la modification sur GitLab
- Dans l'interface Web de GitLab créer une nouvelle Merge Request vers la branche master
- Valider la
- En local retournez sur la branche master, récupérer les modifications distantes et observez les commits (`git log`)

Sur la même base précédente nous allons volontairement généré un conflit

- En local créer une branche `feature42` basée sur `master`
- Modifier le fichier `index.html` pour modifier le titre en `FaceClone42`
- Commiter cette modification
- Créer une branche `feature43` basée sur `master`
- Modifier le fichier `index.html` pour modifier le titre en `FaceClone43`
- Commiter cette modification
- Merger la branche `feature42` dans la branche `master`
- Merger la branche `feature43` dans la branche `master`
- Constater le conflit et modifier le titre `FaceClone42` et `43`
- Commiter la correction

Si vous souhaitez aller plus loin vous pouvez tester les méthodes de partage de dépôts sur GitLab.

A vous les studios !

## D'autres outils

Romain THERRAT

POCKOST

28 janvier 2020

# Sommaire

1 Git Flow

2 Labs

- Des règles simples
- Limite les conflits
- Favorise un historique propre
- Peut être comparé aux fonctionnalités de pull requests
- Permet de structurer son organisation

## Deux branches pour les gouverner toutes

En suivant le principe git-flow nous n'avons que deux branches principales

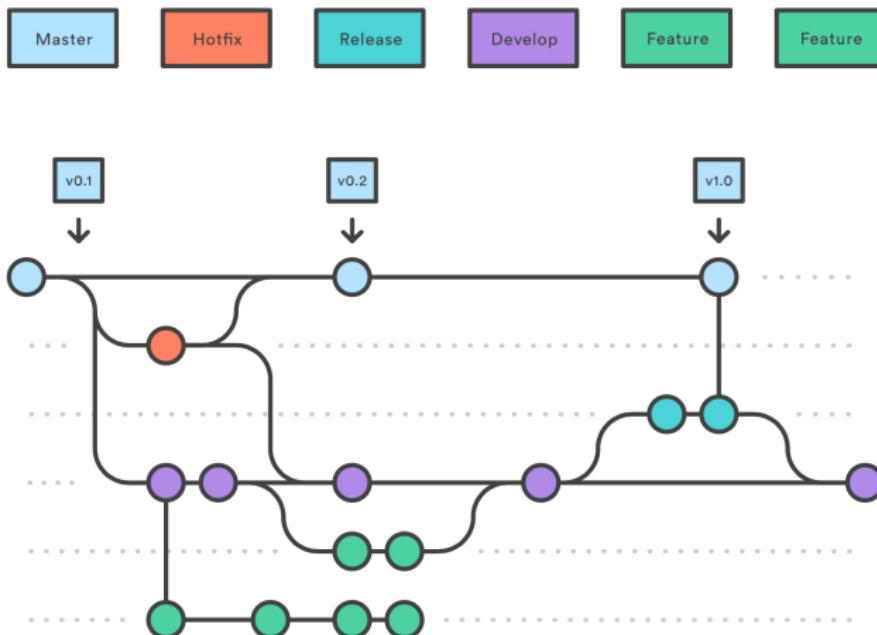
- master : Version du code en production
- develop : Version du code en développement

Ces deux branches sont protégées en écriture.

Nous pouvons créer autant de branches que nous voulons suivant trois types différents

- feature : Travail sur une nouvelle fonctionnalité
- release : Préparation d'une nouvelle version (recette)
- hotfix : C'est le caca . . . je patche vite !

# Fonctionnement



# feature

Je dois travailler sur une nouvelle fonctionnalité.

```
$ git checkout develop
$ git checkout -b feature/my_feature
```

Quand j'ai fini

```
$ git checkout develop
$ git merge feature/my_feature --no-ff
$ git branch -d feature/my_feature
```

# hotfix

Je dois corriger un bug en production

```
$ git checkout master
$ git checkout -b hotfix/my_bug
```

Une fois la correction faite

```
git checkout develop
git merge hotfix/my_bug --no-ff
```

```
git checkout master
git merge hotfix/my_bug --no-ff
git tag v1.42.1
```

```
git branch -d hotfix/my_bug
```

# release

Je dois préparer une nouvelle version de l'application.

```
$ git checkout develop
$ git checkout -b release/v2.42
```

Je fais ensuite passer l'ensemble des tests sur le code de cette branche. Si tout se passe bien.

```
$ git checkout develop
$ git merge release/v2.42 --no-ff
$ git checkout master
$ git merge release/v2.42 --no-ff
$ git tag v2.42
$ git branch -d release/v2.42
```

C'est pas un peu compliqué tout ca ?

- Beaucoup de commandes
- Risque d'erreurs
- Une extension git
- ajoute les commandes git flow

# Installation de git-flow

Désolé c'est écrit tout petit mais ça rentrait pas...

```
$ curl -OL \
  https://raw.github.com/nvie/gitflow/develop/contrib/gitflow-installer.sh
$ chmod +x gitflow-installer.sh
# ./gitflow-installer.sh
```

# Configuration du projet

Il faut tout d'abord convertir notre dépôt au format git-flow.

```
$ git flow init
```

Which branch should be used for bringing forth production releases?

- cookie
- master

Branch name for production releases: [master]

Démonstration

# Utilisation

## Créer une nouvelle branche

```
$ git flow feature start my_feature  
$ git flow release start 1.42  
$ git flow hotfix start 1.42.1
```

## Merger mes modifications

```
$ git flow feature finish my_feature  
$ git flow release finish 1.42  
$ git flow hotfix finish 1.42.1
```

# Démonstration

Démonstration

# Sommaire

1 Git Flow

2 Labs

- Installer git-flow
- Convertir notre projet au format git-flow
- Créer une nouvelle feature nommée change-title
- Changer le titre d'index.html et commiter le résultat
- Terminer la feature
- Créer une nouvelle release nommé 1.42
- Si tout vous semble bon terminer la release
- Penser à mettre un message pour le tag
- Créer un hotfix nommé 1.42.1
- Modifier le titre de la page pour ajouter un smiley dedans
- Terminer votre hotfix