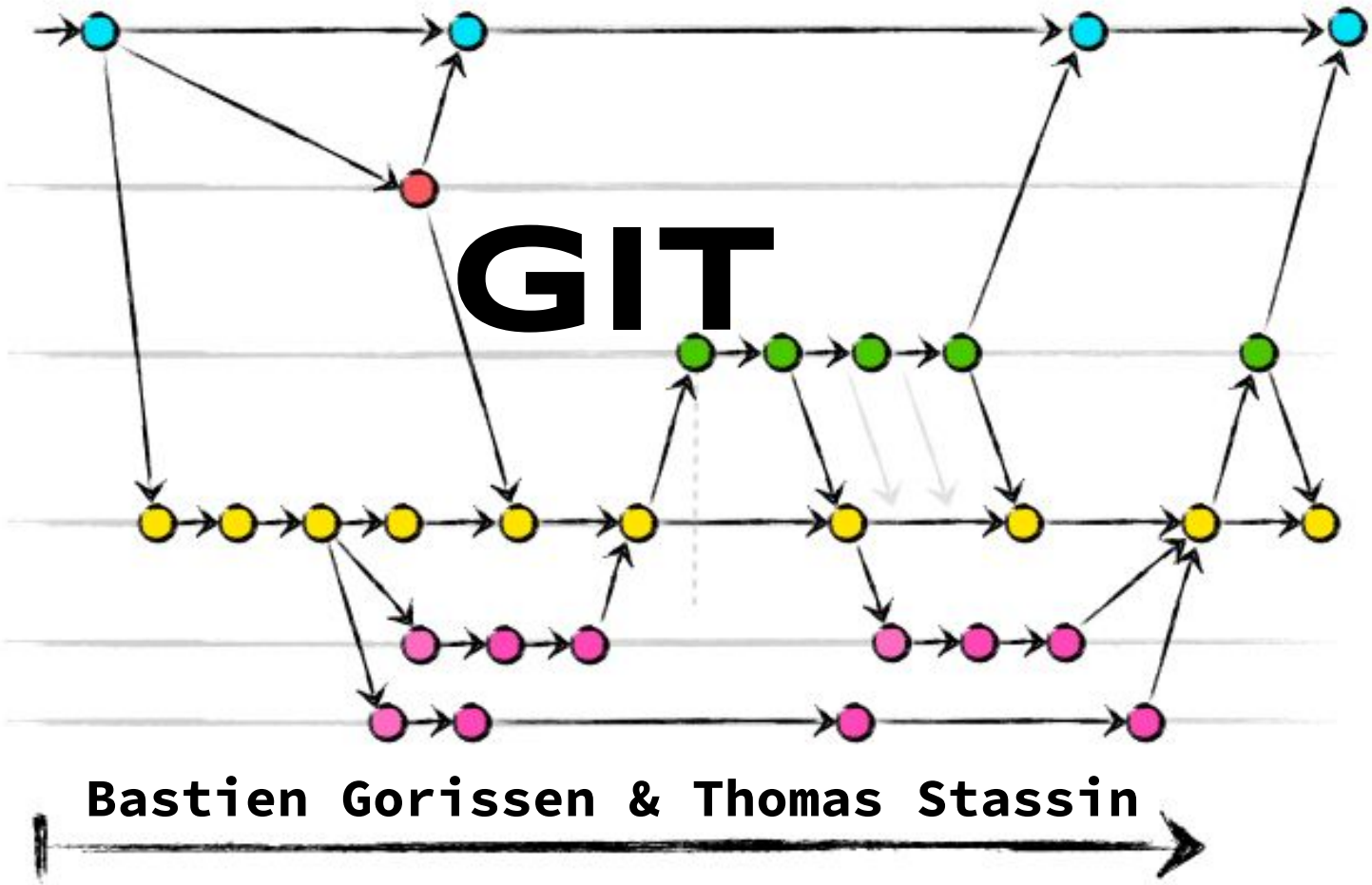
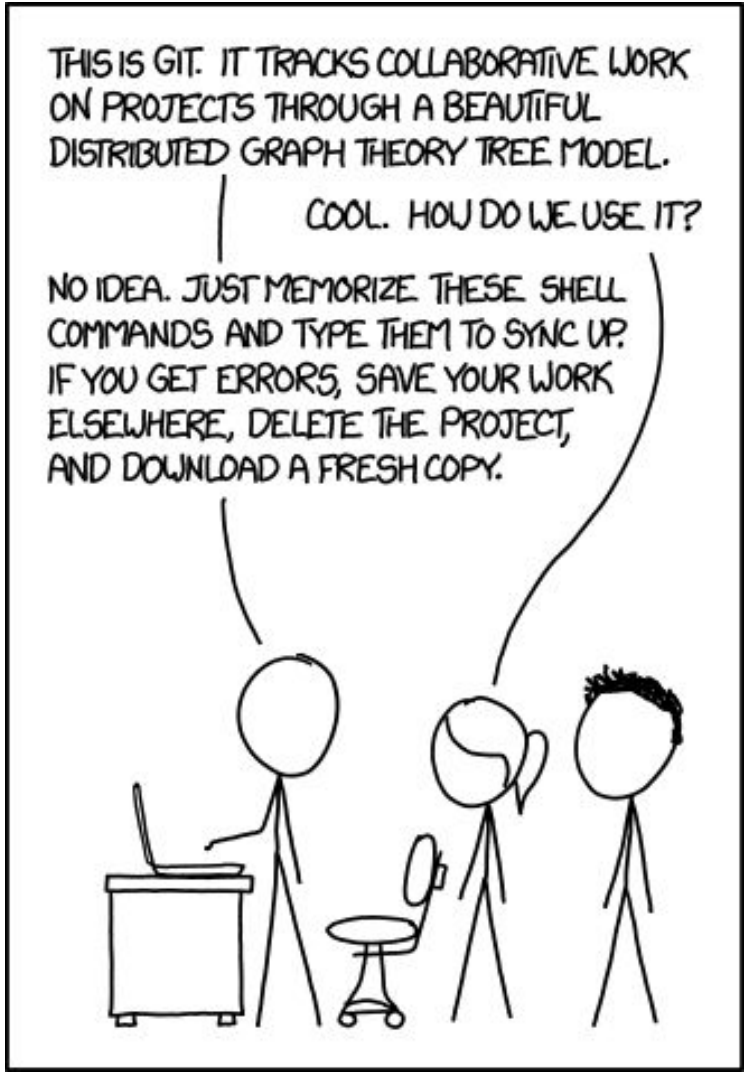

GIT
Bastien Gorissen & Thomas Stassin

Prologue

Où nos héros découvrent une contrée inconnue...


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Petit sondage...

Avant de plonger dans le vif du sujet, voyons quelle est la situation :D

Levez la main si…

- ...vous avez entendu le nom Git.
- ...vous avez une idée de ce que Git fait.
- ...vous pensez en avoir besoin.
- ...vous avez déjà perdu des changements dans votre code (dans un crash disk ça compte :D).
- ...vous n’avez pas peur de la ligne de commande.

Au menu...

Qu'allons-nous faire ?

Nous n'avons pas beaucoup de temps, mais nous allons voir:

- Git : Qui ? Quoi ? Comment ?
- Les commandes de base.
- Jardinage: Git & les branches.

Ensuite, si le temps le permet, nous verrons quelques outils qui simplifient l'interaction avec Git, et comment organiser son travail efficacement !

Level 1-1

Git, les gens.
Les gens, Git.

Git, ça sert à quoi ?

Git est un système de contrôle de version distribué.

Son rôle est de vous aider à garder une trace des versions successives de votre code, et de vous assister quand vous devez collaborer sur des projets.

Idéalement, TOUS vos projets devraient utiliser Git.

Non seulement pour vous entraîner, mais pour éviter toute catastrophe (“Oh mon chat a marché sur le clavier et tout effacé !”, “Mon petit frère a versé son verre d’eau sur mon laptop !”, “Mon disque dur est tombé dans un volcan !”).

Et concrètement ?

Git garde en mémoire tous les changements successifs que vous effectuez sur votre code, en faisant une “photo” de votre code quand vous lui demandez de le faire.

Vous pouvez donc à tout moment annuler un changement problématique, ou retrouver un morceau de code que vous auriez supprimé.

Git construit un historique de vos changements:

Ok. Et comment on utilise ça ?

Nous allons utiliser 2 choses :

1) Git-bash

Une ligne de commande qui permet d'interagir avec Git.
Oui, il existe des GUI, mais non, on ne va pas les utiliser tout de suite.

2) http://github.com/

C’est un site qui vous permet de sauvegarder sur “le cloud” vos projets Git, pour que vous y ayez toujours accès.

Level 1-2

Back to basics

Git et les "repo"

Git fonctionne par "projets", qu'on nomme **dépôts** (ou **repository/repo** en anglais).

Un repo est simplement un dossier sur votre disque dur qui contient un sous-répertoire caché nommé ".git" dans lequel Git va organiser ses informations.

Vous allez ensuite dire à Git quelles parties du contenu du repo il doit surveiller, et quelles parties il peut ignorer. Et à partir de là, Git va prendre note des changements qui surviennent dans vos fichiers.

La règle d'or

...enfin une des règles d'or du moins.

Ne pas mettre un repo à l'intérieur d'un autre repo! (même dans un sous-sous-sous dossier du premier repo!)

Dans certains cas très particuliers, quand vous savez exactement ce que vous faites, ça peut passer, mais en général évitez, ça vous évitera des migraines :)

La genèse d'un "repo"

Avant d'utiliser GitHub, nous allons voir comment utiliser Git en local, sur votre PC.

- Ouvrez git-bash (touche windows, tapez "**git bash**", [Enter], ou bien clic droit dans l'explorateur, "Git Bash Here")
- **git init**

Et voilà ! Votre répertoire courant est connu par Git, et Git a créé un "repository" pour traquer les changements. Il s'agit du sous-dossier .git que Git-bash mentionne.

Se présenter par politesse

La première chose à faire (et à ne faire qu'une seule fois), est de nous présenter à Git. Pour ce faire, on peut utiliser les commandes suivantes:

- **git config --global user.name "My Name"**
- **git config --global user.email "email@provider"**

Git va maintenant attacher notre identité à tous nos changements, ce qui pourra être pratique en cas de collaboration avec d'autres.
Notez que si vous ne mettez pas **--global**, vous changez la config pour le repo dans lequel vous êtes.

Les commandes de base

Nous allons maintenant voir les commandes principales de Git, celles que vous allez utiliser tous les jours, plein de fois par jour.

- **git status**
- **git add**
- **git commit**

Une première chose à savoir : Git peut faire plein de choses, mais au moins, il vous donne en général pas mal d'info sur ce qu'il se passe. Il faut donc prendre l'habitude de bien **lire** ce qu'il vous dit :)

Status quo

Testons directement la première commande :

- **git status**

On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)

Résumé : il n’y a rien, nada, et Git vous donne comme conseil d’utiliser **git add** pour changer ça.

Status quo

git status permet donc de voir l'état actuel de votre repo.

Par état, on veut dire quels changements ont été opérés depuis le dernier "snapshot" de votre code, quels fichiers ont été supprimés, etc.

Prenez l'habitude de l'utiliser tout le temps pour voir où vous en êtes.

C'est la commande la plus utile de votre arsenal.

Mais donc, ce fameux **git add**...

ADDicted to Git

git add permet, comme son nom l'indique, d'ajouter un fichier, ou les changements d'un fichier, dans le prochain paquet de changements traqué par Git.

Comme il vaut toujours mieux essayer par soi-même…

- Créez un nouveau fichier ("test.txt" par exemple) dans le répertoire.
- **git status**
- **git add test.txt**
- **git status**

Mais c'est qu'il est gentil...

Remarquez que Git continue de vous donner des conseils sur les probables prochaines étapes que vous pourriez vouloir suivre.

Mais donc, qu'avons-nous fait ?

- Nous avons dit à Git "test.txt fait partie des fichiers que tu dois tenir à l'oeil"
- Git en a pris note, et fera donc à l'avenir mention des changements qui sont effectués sur le fichier.
- Git est prêt à "commiter" le fichier.

Commit ?

Depuis le départ, nous parlons de faire des “snapshots” ou des “photos” de notre code. En réalité, on nomme ça des “commits”.

Chaque commit est donc une étape dans votre développement, et représente un ensemble de changements sur vos fichiers sources.

Git vous laisse ajouter (**git add**) tout ce que vous voulez avant de finalement fermer le commit et le considérer comme un paquet isolé (**git commit**).

Commit d'office

Dans notre exemple, nous avons donc ajouté notre fichier “test.txt” au commit en cours.

Git appelle cette “zone” intermédiaire la “staging area”. Vous pouvez voir ça comme une boîte en carton ouverte. Chaque **git add** rajoute quelque chose dans la boîte, et quand vous voulez la fermer :

- **git commit -m “Added our first file to the repo.”**

Conseil : utilisez toujours **-m “message”**, sinon...

Level 1-3

Push me, pull you!

C'est pas Folichon

Bon, utiliser Git sur son PC, c'est bien, mais ça ne nous sauve pas d'un crash de disque dur.

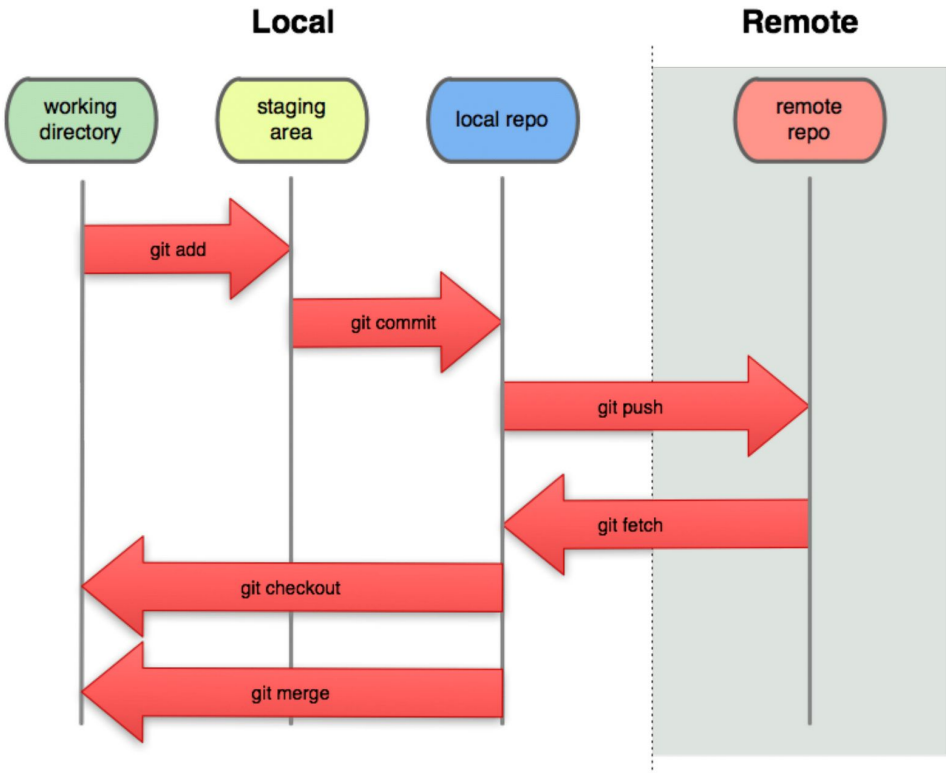
Pour ça, on peut envoyer son code ailleurs, par exemple sur Github (ou sur un autre serveur).

Après avoir créé un projet sur Github, voyons les commandes dont nous allons avoir besoin pour synchroniser notre travail en local avec un serveur distant.

Et plutôt que vous donner 50 slides en plus…
https://git-scm.com/doc

Remote control

- **git push** permet d'envoyer son travail (après l'avoir commité) sur le serveur.
- **git fetch** ramène les changements depuis le serveur (mais ne change pas vos fichiers locaux)
- Il existe **git pull** qui fait **fetch + merge**.



A day in the life...

Au jour le jour, la séquence classique quand vous travaillez avec un “remote” sera :

- **git pull**
- **git add**
- **git commit -m “Super commentaire”**
- (répéter ci-dessus)
- **git push**

Après, évidemment, tout n’est pas toujours si simple :)

Level 1-4

Jardins & Loisirs

On peut faire une micro-révision ?

Nous avons vu les commandes de base :

git status pour savoir où on en est

git add <mon_fichier> pour ajouter mon_fichier au commit

git commit -m “Mon message” pour créer un commit

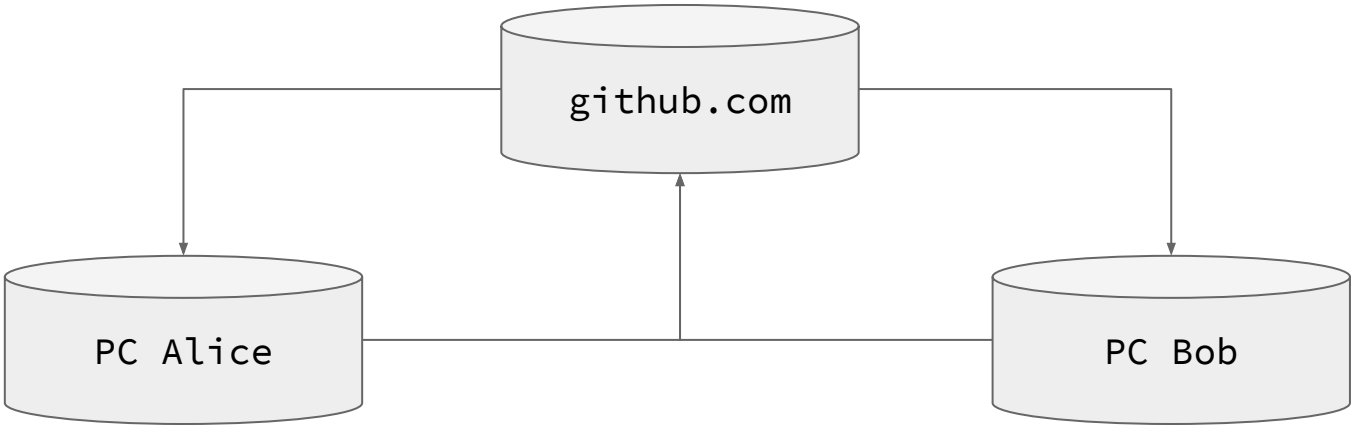
git push pour envoyer tout ça vers BitBucket

Rappel : un commit est un “snapshot” de votre code à un moment donné.

A quoi d'autre sert Git ?

Git est un très bon outil de collaboration et d'organisation du travail.

Imaginez la situation suivante :



Git s'y retrouve ?

Git est capable de travailler avec plusieurs “historiques” qui varient entre eux et est capable de les “merger” (rassembler).

On peut voir ça comme des espèces d'univers parallèles.

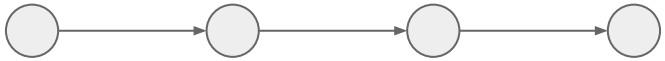
Le nom technique est “branche”.

Et Git vous donne le pouvoir de créer et manipuler ces univers parallèles comme bon vous semble !

Voyons comment ça se passe quand vous travaillez seules.

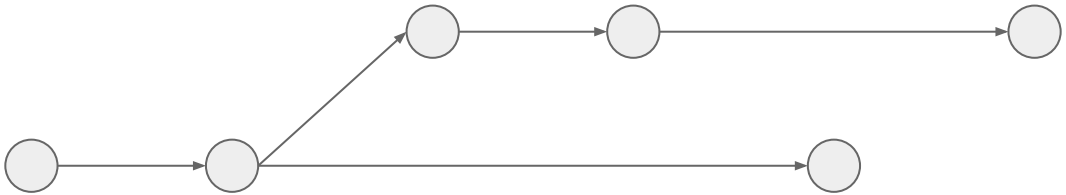
Branchements

Nous avons vu le cas “linéaire” simple :



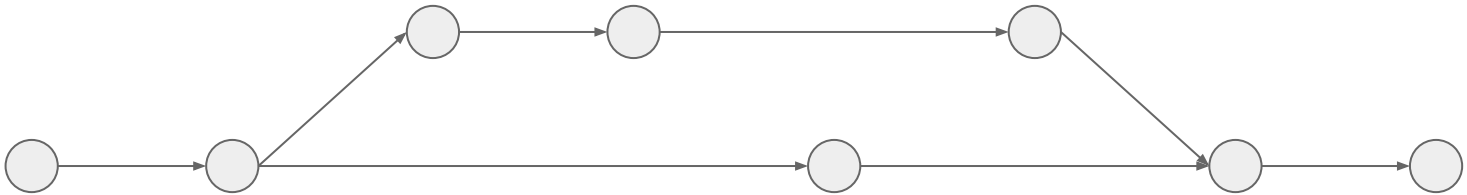
Mais imaginez que vous ayez deux fonctionnalités à implémenter.

Vous pouvez partir d’un point donné et aller dans deux directions différentes.



Croisements

Et bien entendu, les branches peuvent se rejoindre (on appelle ceci un “merge”).



Git va donc créer un nouveau commit au moment du merge, et va ensuite vous laisser continuer à travailler à partir de là.

Voyons rapidement comment créer et merger des branches !

Commandes !

Voici les commandes principales pour les branches

git branch <ma_branche> pour pour créer une nouvelle branche

git checkout <ma_branche> pour passer sur la branche ma_branche

git checkout -b <ma_branche> pour créer la branche et passer dessus directement

git merge <ma_branche> pour merger la branche ma_branche dans la branche actuellement active

Waiiiit a minute...

Que se passe-t-il si on a modifié le même code sur 2 branches ?

Il se peut que vous ayez des conflits entre plusieurs branches au moment d'un merge.

C'est l'une des situations les plus "stressantes" en travaillant avec Git.

Nous allons donc tenter d'en créer un pour voir comment nous en dépêtrer...

Conflict cheat sheet

Donc en cas de conflits :

1) Regarder quels fichiers posent problème
2) Choisir quoi garder
3) Sauvegarder les fichiers
4) Les ajouter à un nouveau commit
5) Commiter
6) Et voilà !

Maintenant que nous savons comment gérer les branches, voyons comment trouver notre “flow”.

Level 1-5

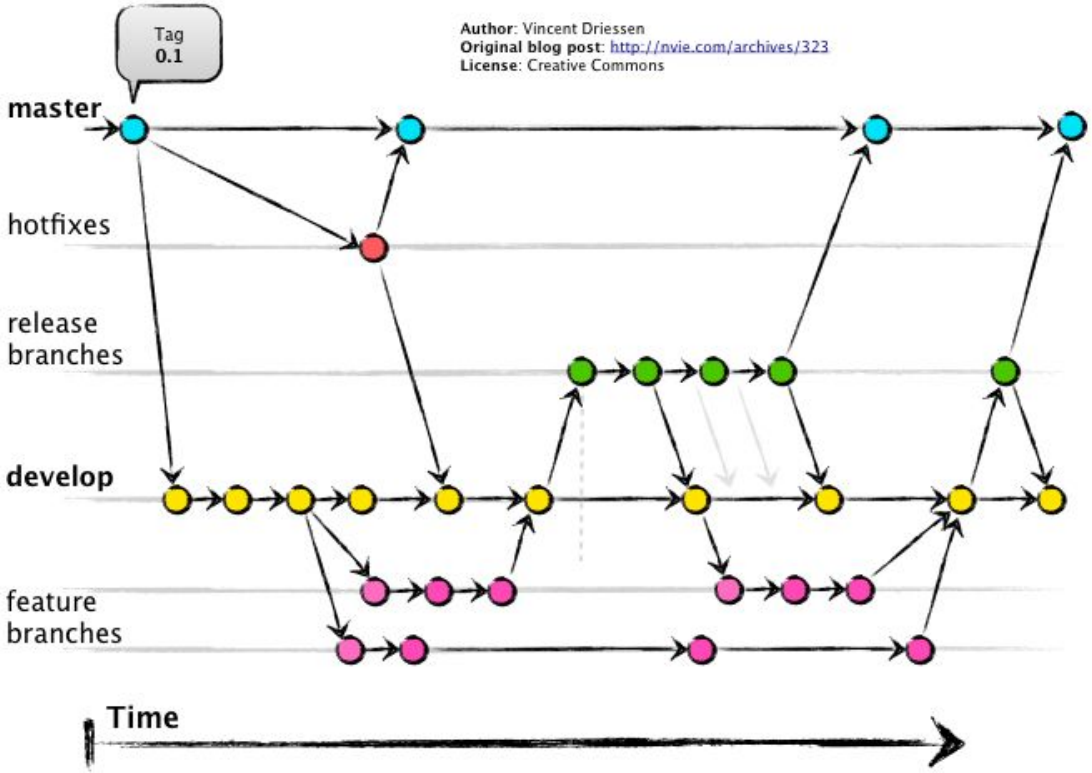
Go with the flow...

Qu'est-ce que c'est encore que ce truc ?

Git-flow est simplement une proposition de façon de travailler qui permet de rester organisé et clair. Le tout basé sur un certain nombres de branches:

- **master** : Contient les versions “officielles”, considérées comme finies.
- **develop** : Branche principale, qui contient cependant toujours du code qui tourne.
- **feature/*** : Branches de travail. Une par fonctionnalité à implémenter.
- **release/*, hotfix/*** : Branches temporaires.

En une image…


Tag
0.1
Author: Vincent Driessen
Original blog post: http://nvie.com/archives/323
License: Creative Commons
master
hotfixes
release
branches
develop
feature
branches
Time

Et donc on fait plein de "git branch" ?

La beauté de Git-flow est de vous mâcher le travail en proposant des commandes "toutes faites". Essayons !

- **git flow init**: Initialise le repo en tant que repo utilisant Git-flow.
- **git flow feature start <my_feature>** : Pour commencer une nouvelle feature.
- **git flow feature finish** : Pour clôturer une feature.
- **git release start <my_release>** : Commencer une release.
- **git flow release finish**
- Bonus: **git flow feature publish**