Programmation C avancée

Concepts & Outils
pour le développement

maj 03/2024

La gestion de sources

gestion de sources

- Lorsque l'on interagit avec d'autres développeurs, il est indispensable de pouvoir communiquer des propositions de modifications (ajout de fonctionnalité, correctif de bug, amélioration des perfs…) :

gestion de sources

#QDLE#S#ABC#25#

- Lorsque l'on interagit avec d'autres développeurs, il est indispensable de pouvoir communiquer des propositions de modifications (ajout de fonctionnalité, correctif de bug, amélioration des perfs…) :

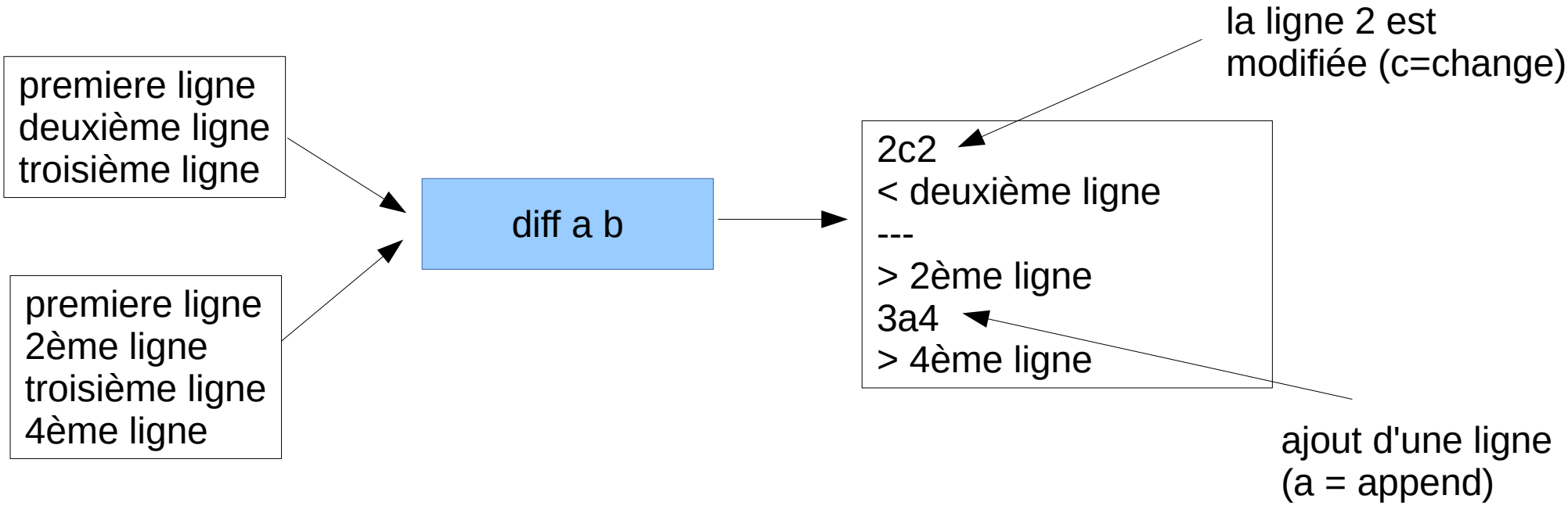
A) j'envoie tout le code en indiquant que c'est une nouvelle version.

B) j'écris un email détaillé des modifications à effectuer.

C) autre approche...

diff

- diff est un outil d'analyse de texte qui compare deux fichiers entre eux et produit le nombre minimum d'éditions de lignes à faire sur le premier fichier pour obtenir le second :



diff : side by side

- on peut afficher les deux fichiers cote à cote :

```
diff -y a b
```

```
premiere ligne                                  premiere ligne
deuxième ligne                                | 2ème ligne
troisième ligne                                 troisième ligne
                                              >     4ème ligne
```

- diff permet la comparaison récursive de deux arborescences.

diff : recursif

- Je souhaite modifier le code source d'un projet :
  1. je fais un copie de sauvegarde des sources d'origine
  2. j'effectue mes modifications
  3. à tout moment, je visualise mes modifications avec diff : `diff -r projet projet_new`

diff : recursif

- Je souhaite modifier le code source d'un projet :
  1. je fais un copie de sauvegarde des sources d'origine
  2. j'effectue mes modifications
  3. à tout moment, je visualise mes modifications avec diff : `diff -r projet projet_new`

```
diff -r GenTaskLib/TaskParser.cpp GenTaskLib_new/TaskParser.cpp
15c15
<         throw gtl::InvalidArgumentException("json parser error: expected a dictionnary at each task");
---
>         throw gtl::InvalidArgumentException("json parser error: expected a dictionnary for each task");
diff -r GenTaskLib/TaskParser.hpp GenTaskLib_new/TaskParser.hpp
36a37,38
>   /*! TaskParser: build a task from a json description.
>    */
```

Ajout d'un commentaire

diff : algorithme

- Le programme diff repose sur un problème classique d'algorithmique du texte : La plus longue sous-séquence commune

  MOTELS ARE NOT HELL!

  MIROIR TU ES LA!

- Sur cet exemple « MO T » est une sous séquence commune.

diff : algorithme

#QDLE#Q#ABC*D#100#

- Le programme diff repose sur un problème classique d'algorithmique du texte : La plus longue sous-séquence commune

  MOTELS ARE NOT HELL!

  MIROIR TU ES LA!

- Taille de la plus longue sous séquence commune ? (les espaces comptes).

  A. 6 B. 8 C. 9 D. 10

diff : algorithme

- Le programme diff repose sur un problème classique d'algorithmique du texte : La plus longue sous-séquence commune

  MOTELS_ARE_NOT_HELL!

  MIROIR_TU_ES_LA!

diff : algorithme

- Chaque fichier est découpé en lignes
- Les lignes sont comparées entre elles
- Puis l'ensemble des lignes sont comparées entre elles (LCS où chaque « symbole » représente une ligne).

diff et communication

- Ainsi, si l'on souhaite communiquer une modification, il suffit d'envoyer le résultat d'un diff récursif : `diff -rupN original new > patch`
- On appel ce fichier un **patch**.
- Le destinataire peut lire ce fichier et comprendre vos modifications
- Il peut également appliquer ces modifications en local grâce au programme patch

- *les options upN sont nécessaire au fonctionnement de patch, elles ajoutent des informations de contexte (u), de fonction (p), d'ajout de fichier (N)*

patch

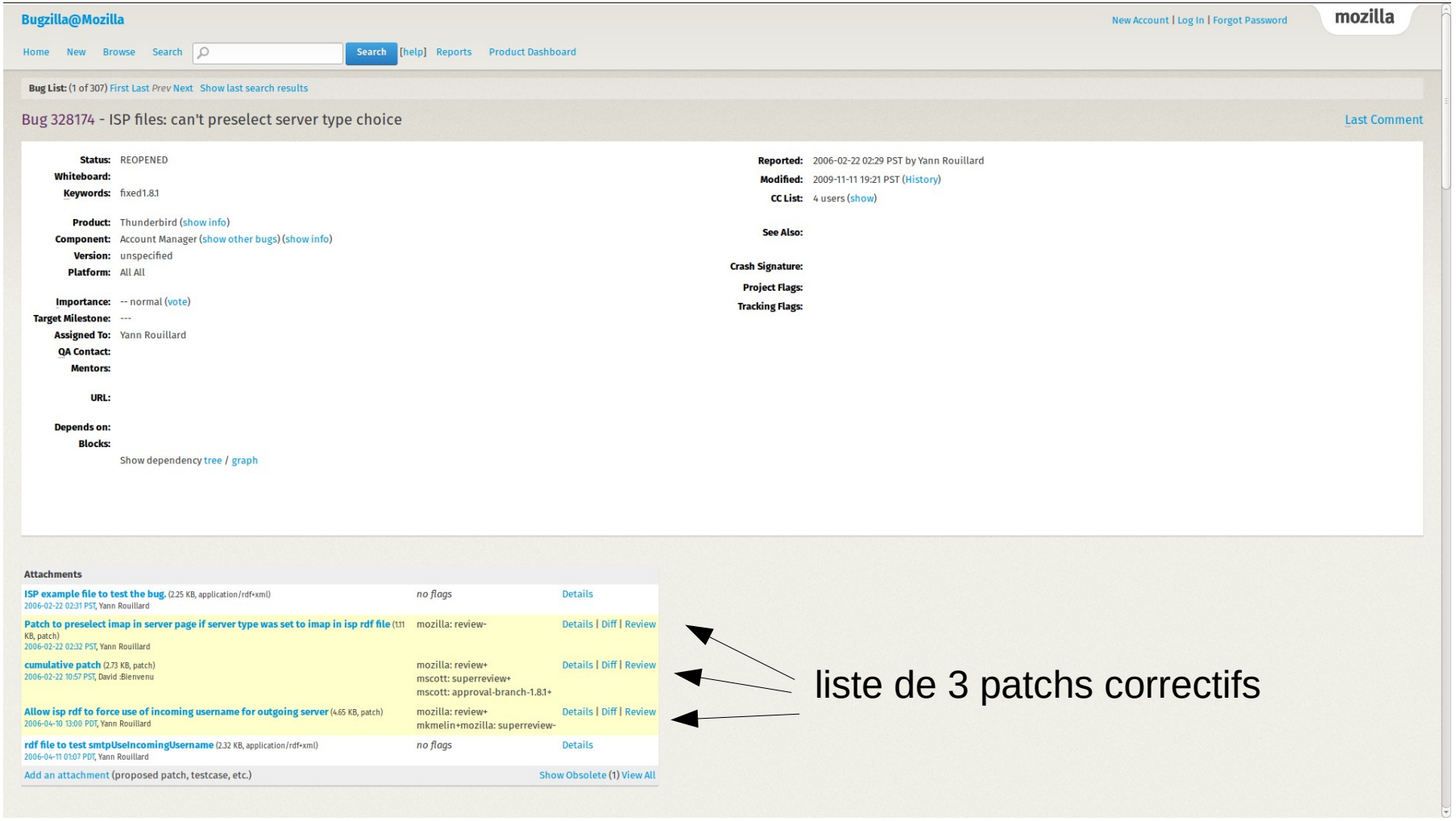
- le programme patch permet d'appliquer les modifications identifiées par diff.
- Ainsi sur l'exemple précédent je peux faire :

```
$ cp -R GenTaskLib GenTaskLibMod
$ cd GenTaskLibMod
$ patch < ../patch
patching file TaskParser.cpp
patching file TaskParser.hpp
$ cd .. ; diff -r GenTaskLib GenTaskLibMod
```

```
diff -r GenTaskLib/TaskParser.cpp GenTaskLibMod/TaskParser.cpp
15c15
<         throw gtl::InvalidArgumentException("json parser error: expected a dictionnary at each task");
---
>         throw gtl::InvalidArgumentException("json parser error: expected a dictionnary for each task");
diff -r GenTaskLib/TaskParser.hpp GenTaskLibMod/TaskParser.hpp
36a37,38
>     /*! TaskParser: build a task from a json description
```

diff & patch

- Dans le monde open source, diff et patch sont très utilisés



gestion de sources

- Un gestionnaire de sources permet de conserver l'historique des modifications apportées à un ensemble de fichiers.
- Il permet à un ensemble d'utilisateurs d’interagir sur un même code source.
- Tous les gestionnaires de code sources sont basés sur les principes de diff et patch
- Il existe deux grandes catégories de gestionnaires :
  - les centralisés
  - les décentralisés

Les gestionnaires centralisés

- Historiquement, cvs (concurent versioning system) et son successeur svn (subversion)
- Les gestionnaires centralisés reposent sur un serveur central qui archive toutes les modifications apportées au code.
- Chaque modification incrémente un numéro de révision
- Les commandes de base de svn :
  - checkout, update, infos, status, commit, diff, revert

svn

- On commence par la création d'un dépôt serveur : celui-ci n'est pas un espace de travail (on ne peut pas coder dedans).
- Chaque utilisateur interagit avec le serveur, il n'y a pas d'échange direct entre deux utilisateurs.
- Une méthodologie est associée à l'utilisation de svn, vous retrouverez cet méthodologie dans tout projet de développement (open source ou entreprise) => « svn red book »
- Cette méthodologie tente de répondre à des problèmes liés à tout développement logiciel (indépendamment de SVN).

svn : méthodologie

- Il est possible d'utiliser SVN juste pour le répertoire de développement principal.
- Seulement, comment faire si on a un « gros » développement à produire: si on transmet les modifications intermédiaires, le programme devient instable (ne compile plus par exemple...)
- Comment faire également pour gérer les versions:
  - On sort une version 1.0 qui évolue en 1.1 puis 1.2
  - On sort la version 2.0 (« casse » la compatibilité avec 1.x)
  - Un bug est trouvé dans la version 2.0: il faut le corriger dans la version courante mais également dans la version 1.x!

Un mot sur les versions...

- Il n'y a pas de « règles » universelles mais un certain nombre de pratiques.
- L'approche par numérotation est la plus répendue :

2.10.5

Major.Minor.Patch

Version.SousVersion.Patch

- Dans le cas d'une bibliothèque, on peut suivre les principes suivant :
  - Tant que l'API i.e. les fichiers d'entête et les fonctions (noms/signatures) ne changent pas, on garde le même numéro de version
  - On peut ajouter des fonctionalités : on incrémente le numéro de sous version
  - On peut corriger des bugs : on incrémente le numéro de patch

→ Un changement de version (Major) implique (sauf exception) une non compatibilité

Un mot sur les versions….

- Il existe d'autres stratégies de nommage/numérotation des versions.
- Par exemple, sur la base d'une date :
  - Ubuntu 18.04 => 2018 / Avril
- Parfois un mix :
  - Version.SousVersion.date(YYYYMMDD)
- On veille cependant à ce qu'il y ai un logique d'ordre
- Il est aussi apprécié que la logique de numérotation suit un ordre (lexicographique et/ou numérique).

Svn red book : les problèmes

- Les problèmes à résoudre sont :
  - Je dois pouvoir consulter les sources correspondantes à une version
  - Je dois pouvoir figer le développement en vue d'une mise en production sans empêcher les développement en cours
  - Je dois pouvoir développer une fonctionnalité sans empêcher d'autres développements tout en restant synchronisé avec ceux-ci
  - Je dois pouvoir intégrer ma fonctionnalité dans la version en cours de développement

svn : méthodologie

- La gestion des versions implique de maintenir plusieurs « répertoires » de développement en parallèle.
- Une méthodologie classique organise les sources ainsi:
  - / « racine »
  - /trunk : contient la version en cours des sources (dev.)
  - /branches/: des copies de trunk (« svn copy »)
  - /branches/1.0 : copie de trunk pour release (tests), retour de modif dans le trunk avec « svn merge » si compatible
  - /tags/1.0.0: version **figée** d'une branche qui est publiée, sert de référence. La branche correspondante est étiquetée (tag). **pas de commit/modifs dans ce répertoire**
  - /branches/modif_allali_155/: copie temporaire pour de « grosses » modifications avec retour dans le trunk par « merge ». Synchronisation régulière depuis le trunk (« merge »)

svn : la création d’un dépôt

- Le création d'un dépôt se fait à l'aide de la commande « svnadmin create nom_de_depot »
- possibilité d'ajouter l’exécution de script avant/pendant et après les « commits »
- possibilité d’envois de mails lors des commits
- facile à mettre en place sur son compte:
- cd ~/.depots/ ; svnadmin create SVN
- cd ~/; svn co file:///$HOME/.depots/SVN
- via ssh:
- svn co svn+ssh://allali@ssh.enseirb.fr/.depots/SVN

contrôle dans svn

- Il est possible d'avoir une approche hiérarchique dans svn en subdivisant le répertoire « branches » en répertoires et en ajustant les droits d’accès (lecture/ecriture) :
  - seuls les masters peuvent commiter dans « trunk »
  - les dev travaillent dans des branches
- Cela permet de faire de la revue de code

Les gestionnaires dé-centralisés

- Dans ce cas, il n'y a pas de dépôt central.
- Chaque utilisateur gère son propre dépôt.
- Un protocole permet l'échange de modifications (commit) entre deux utilisateurs.
- Exemple : git
- Commandes de base :
  - clone, add, commit, push, pull, fetch, merge, branch, checkout

git : les commits, pull et push

- Dans git, les commits sont locaux (il n'y a pas de dépôt central).
- On peut transmettre un ensemble (suite) de *commit* à un autre «dépôt» avec la commande *push*
- On peut réceptionner un ensemble de *commit* depuis un autre dépôt avec la commande *pull*.
- On peut ajouter/supprimer autant de dépôts connectés à notre dépôt avec « remote ». A faire dans les deux dépôts si l’on souhaite une symétrie.

git : les branches

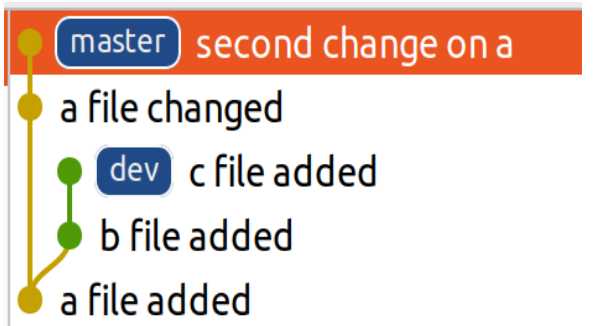
- une branche est une copie du dépôt à un instant t
- par la suite, on peut travailler dans la copie sans que cela affecte l'original de la copie
- si l'original a été modifié, on peut récupérer les modifications dans notre copie
- enfin, on peut propager les modifications faites dans la copie dans l'original.

git

- git intègre une gestion native des branches :
  - création :
    - git branch b2 # on créer la branch b2 par copie de la branche courante
    - git checkout b2 # on bascule dans la branche b2
    - ou bien git checkout -b b2 # creation et basculement
    - ou git **switch** -c b2 # plus « moderne »
  - la fusion :
    - on bascule dans la branche qui doit recevoir les modifs
    - git checkout master (ou git switch master) ; git merge b2
    - Les modifications doivent avoir été *commitées* dans b2
  - la délétion : git branch -d b2
- La branche par défaut s'appelle master ou main ou trunk (paramétrable lors de la création d'un dépôt).

git branches

- *git log* ne montre que les commits de la branche courante : on ajoute --all pour avoir les commits de toutes les branches
- Dans l'exemple à droite (gitg):
  - on a fait un commit dans master
  - créer la branche dev
  - fait deux commits dans dev
  - fait deux commits dans master

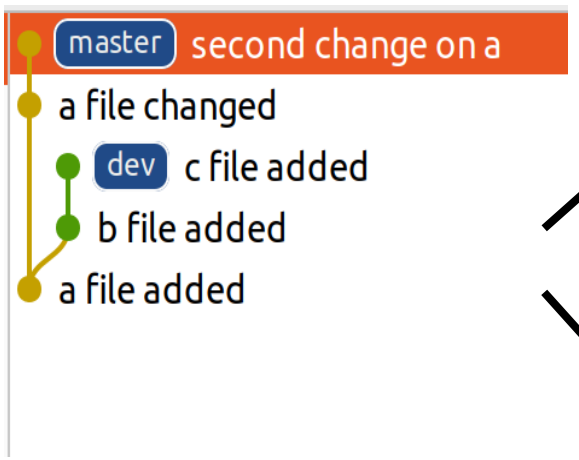


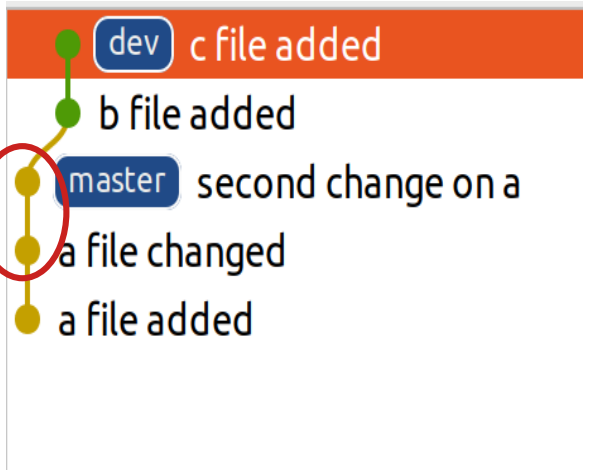
vue obtenue via gitg

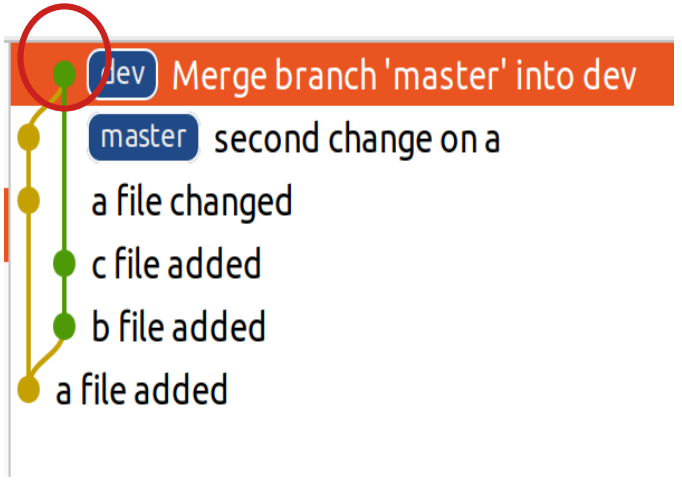
- Attention : vos branches restent locales à moins de les transmettre (push)

rebase vs merge

- git checkout dev
- git rebase -i master







- git checkout dev
- git merge master

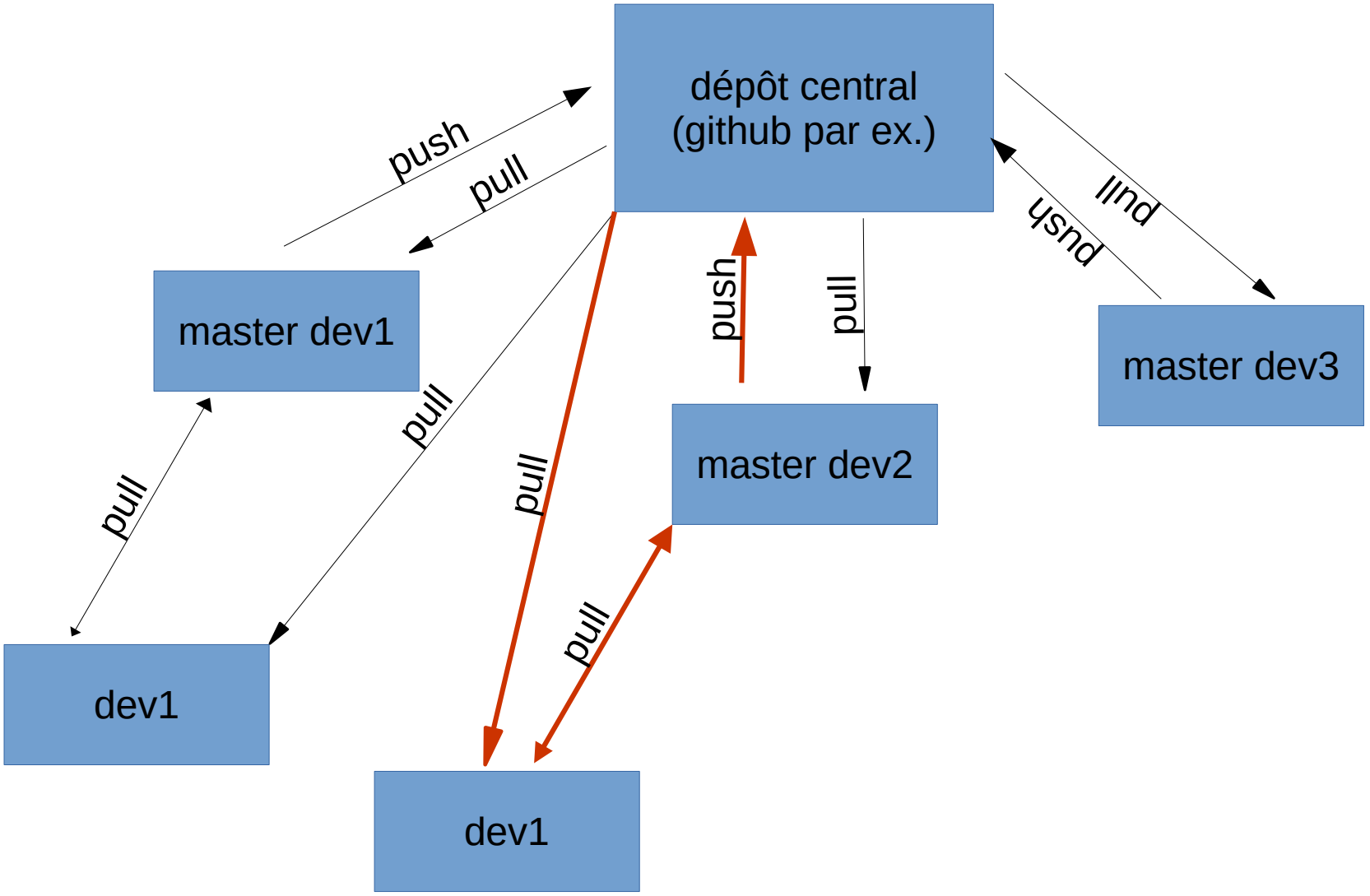
rebase

- avec rebase on perd l'histoire telle qu'elle s'est produite
- on crée une vision « linéaire » : les commits ne sont plus en parallèles mais en séquences, ce sera vrai lors du merge pour fusionner la branche dans master (suite de commit successifs)
- avec merge, on garde une histoire conforme à ce qui s'est passé
- avec merge, la lecture de l'histoire peut devenir complexe s'il y a beaucoup d'aller-retour entre les branches
- ne pas faire de rebase dans master mais dans vos branches

re-centralisation

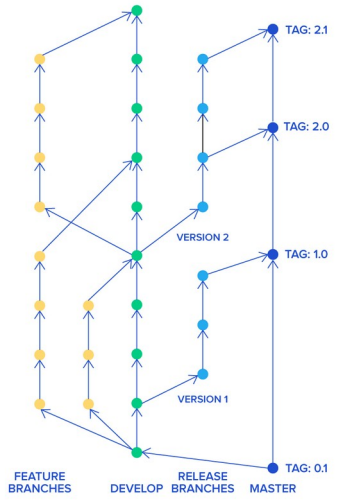
- L'avantage d'être en décentralisé et de pouvoir faire des « commit » sans connexion à un serveur.
- Pour la plupart des projets, il est cependant nécessaire d'avoir une référence : on utilise alors un dépôt git comme tel (github par exemple). Un tel dépôt s'appelle un « bare », il a la spécificité d'accepter les *push*
- On peut ensuite mettre en place un système de propagation hiérarchique des « commits » (des plateforme proposent un système de *pull request* pour cela).

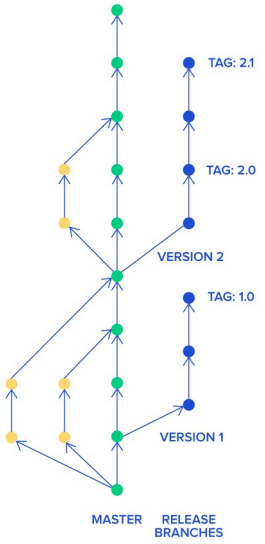
git


dépôt central
(github par ex.)
push
pull
master dev1
pull
pull
dev1
push
pull
master dev2
pull
dev1
llnd
ysnd
master dev3

git: méthodologie

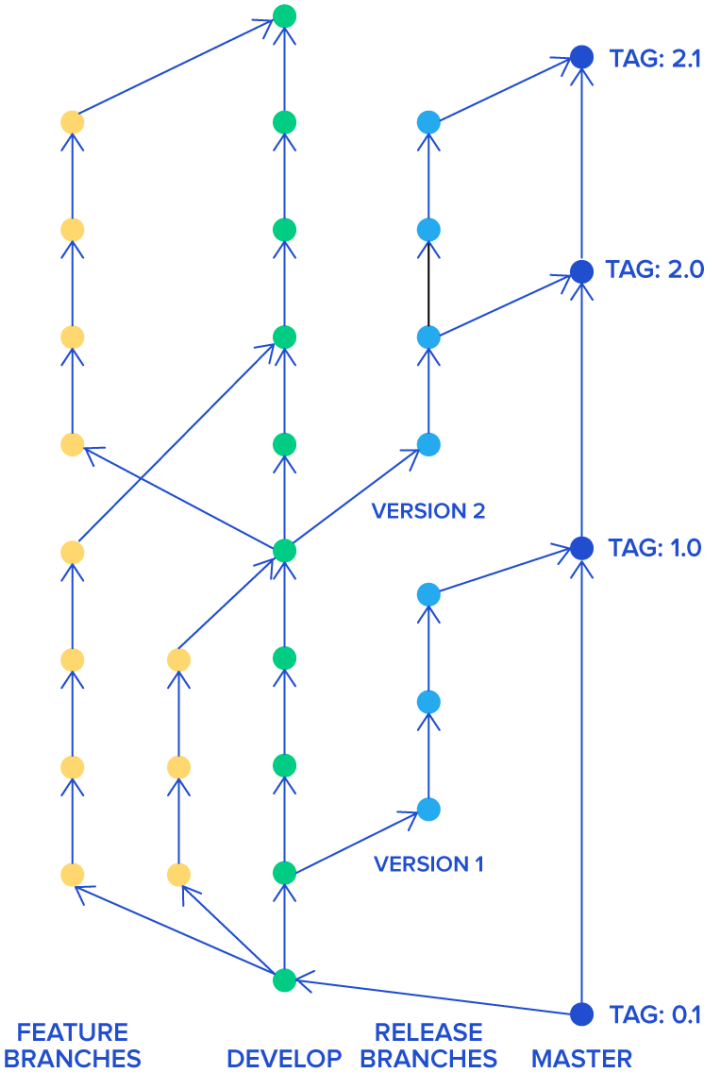
- L'utilisation de git repose nativement sur les branches
- Il y a deux approches actuellement:
  - git flow
  - trunk based





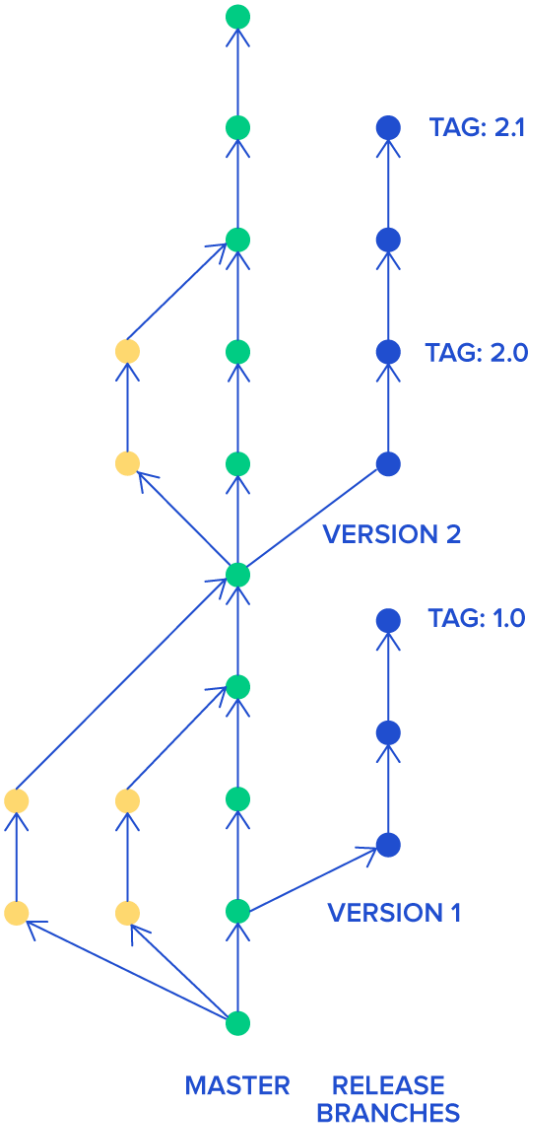
GitFlow

- *GitFlow* suppose des branches avec une durée de vie assez longue: il y a des aller-retour entre ces branches et une branches de développement (et les autres branches), elle même en synchronisation avec une branche de pré-release et la branche master de qui contient le code stable



Trunk Based

- *trunk based* privilégie les branches à durée de vie courte synchronisée avec master/trunk
- les release sont maintenues à coté.



GitFlow ou trunk based

- Les deux méthodologies ont leurs avantages et inconvénients
- L'environnement, le type de projet, le nombre de développeurs sont autant de facteurs qui plaident pour l'un ou l'autre des solutions

git ou svn?

- Différences majeures entre svn et git est :
  - centralisé / dé-centralisé
  - support natif du système de branches dans git
  - commit locaux dans git
- aujourd'hui git est de loin de le gestionnaire de sources le plus utilisé

Les autres gestionnaires

	Open Source	Centralisé	Décentralisé
CVS	•	•	
SVN	•	•	
GIT	•		•
SourceSafe		•	
Mercurial	•		•
Bazaar	•		•
BitKeeper			•
Team Foundation Server		•	

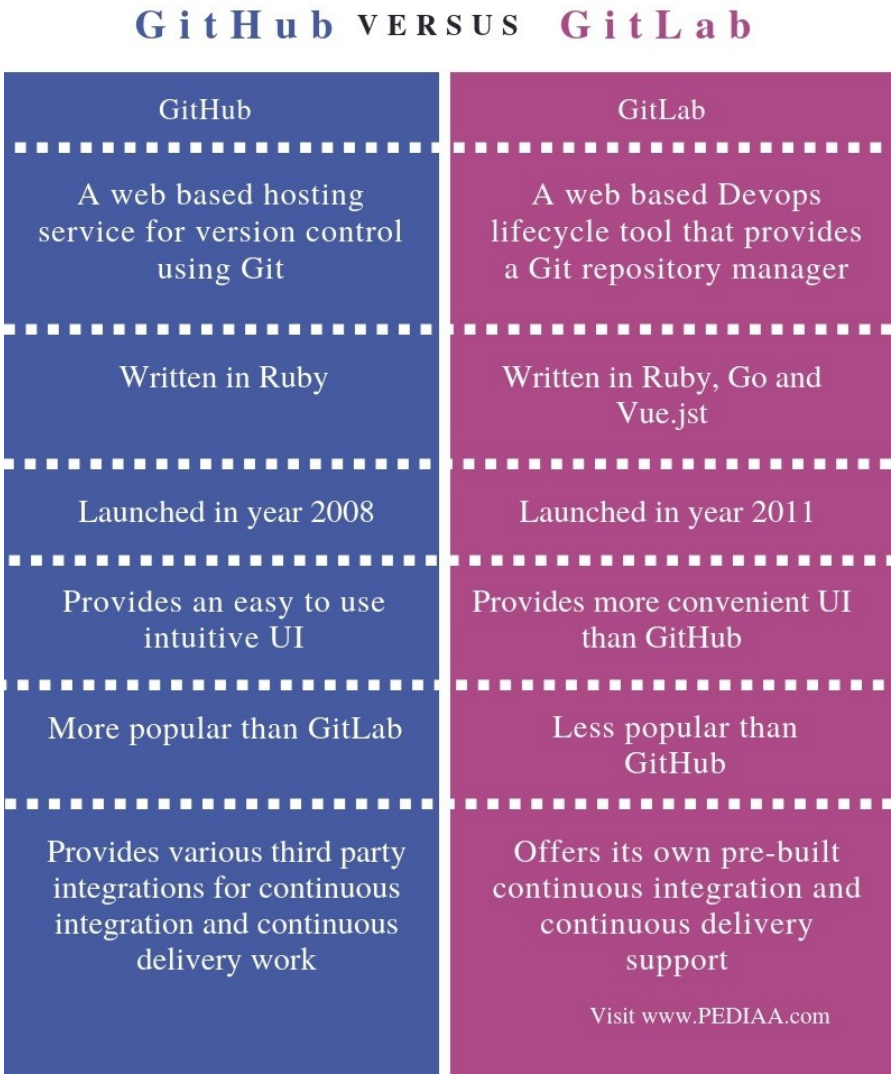
Il en existe bien d’autres (voir : https://en.wikipedia.org/wiki/List_of_version-control_software)

les plateformes

- De nombreuses plateformes proposent l'hébergement de dépôts telles que github et gitlab

  mais aussi bitbucket...

- ajout de services tels que l'intégration continue, suivi de bugs, wiki etc....



Quizz !

Session Quizz sur la compilation

The compiling trivia

#QDLE#Q#ABCD*E#25#

- Quelle commande permet de compiler le fichier toto.c en objet ?
  - A. gcc toto.c
  - B. gcc -E toto.c
  - C. gcc -S toto.c
  - D. gcc -c toto.c
  - E. gcc -fPIC toto.c

The compiling trivia

#QDLE#Q#AB*CD#25#

- Quelle commande permet de récupérer le source après l'étape de pré-compilation ?
  - A. gcc toto.c
  - B. gcc -E toto.c
  - C. gcc -S toto.c
  - D. gcc -c toto.c

The compiling trivia

#QDLE#Q#ABCDEFG*H#25#

- Je compte faire du debug avec gdb sur mon source prog.c, je compile avec les options :
  - A. -gdb
  - B. -g
  - C. -debug
  - D. -DNDEBUG
  - E. -O0
  - F. A & D
  - G. B & E
  - H. C & D

The compiling trivia

#QDLE#Q#A*BCD#25#

- Quelle commande permet de lier toto.o et main.o en un exécutable ?
  - A. gcc toto.o main.o
  - B. gcc -shared toto.o main.o
  - C. gcc -static toto.o main.o
  - D. gcc -exec toto.o main.o

The compiling trivia

#QDLE#Q#ABCD*#25#

- Quelle commande permet de lier toto.o et tata.o en une bibliothèque statique?
  - A. gcc -shared -o libfoo.a toto.o tata.o
  - B. gcc -static -o libfoo.a toto.o tata.o
  - C. gcc toto.o tata.o -lfoo
  - D. ar rcs libfoo.a toto.o tata.o

The compiling trivia

#QDLE#Q#ABCDEF*G#30#

- Pour pouvoir regrouper des objets (.o) dans une bibliothèque dynamique, il faut les compiler avec la ou les options :
  - A. -c
  - B. -E
  - C. -s
  - D. -fPIC
  - E. réponse A&B
  - F. réponse A&D
  - G. réponse B&E

The compiling trivia

#QDLE#Q#A*BCD#25#

- Quelle commande permet de lier toto.o et tata.o en une bibliothèque dynamique?
  - A. gcc -shared -o libfoo.so toto.o tata.o
  - B. gcc -dynamic -o libfoo.a toto.o tata.o
  - C. gcc toto.o tata.o -lfoo
  - D. ar rcsD libfoo.so toto.o tata.o

Intégration Continue

L'Intégration Continue

- Lors que l'on développe, on est sur un système particulier :
  - type du système (unix, linux, windows, macosx, etc.)
  - version du compilateur
  - version des bibliothèques
  - environnement général (ressources…)
- Avant de transmettre une modification (commit), le développeur doit s'assurer que ses modifications fonctionnent pour l'ensemble des systèmes/configurations cibles.

L'Intégration Continue

- Pour cela, on dispose d'un ensemble de machines.
  - Solution 1 : avant de transmettre mes modifications, je me connecte sur chacune des machines et je teste.
  - Solution 2 : j'utilise un système qui fait cela automatiquement pour moi !

    ⇒ C'est ce que l'on appelle l'Intégration Continue.

  - l'IC *garantie* une « stabilité » des développements au fur et à mesure. Cela permet de contrôler certaines dettes techniques.

L'Intégration Continue

- Il existe plusieurs plateformes d'intégration continue :
  - Jenkins (successeur de Hudson, java-open source)
  - TeamCity (JetBrain, commercial)
  - CruiseControl (java-open source)
  - Team Foundation Server (Microsoft, commercial)
  - Travis IC (online IC for github projects).
  - ...

L'Intégration Continue

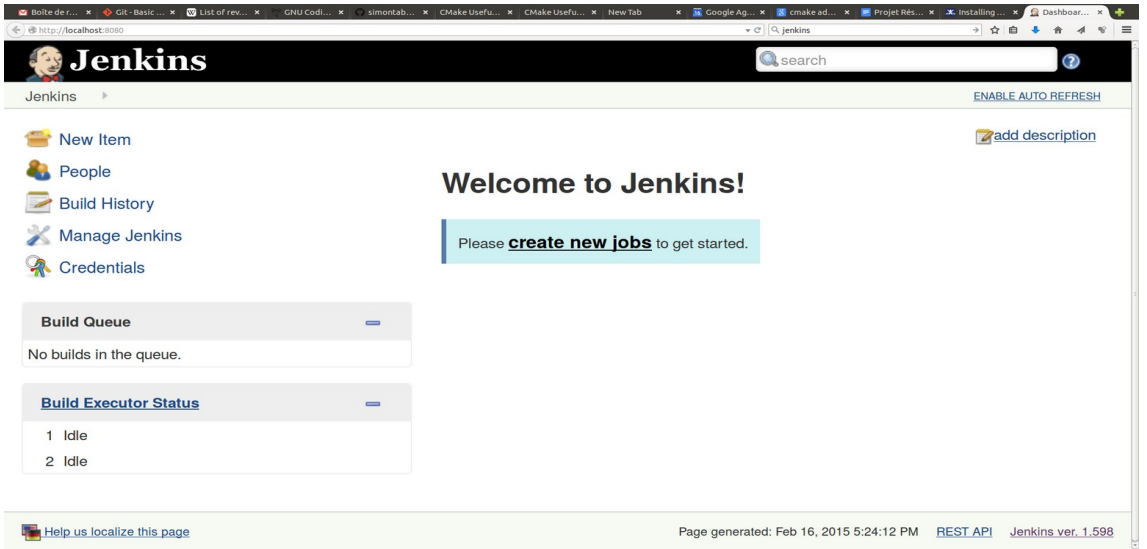
- Un serveur d'intégration continue va se synchroniser avec un dépôt
- A intervalle régulier, il va vérifier que le dépôt est à jour. Si une mise à jour est intervenue, il va effectuer une série de taches (compilation par exemple).
- En fonction du résultat des taches, le serveur va indiquer l'état du projet et possiblement transmettre des alertes.
- Afin de gérer plusieurs environnements, le serveur d'IC va piloter des clients sur lesquels il lancera les taches (via ssh par exemple).

L'Intégration Continue

- La qualité qu'offre l'IC va dépendre principalement de deux facteurs :
  - la nature et la diversité des clients (environnement de validation)
  - la complexité des taches à réaliser :
    - de la compilation
    - à l'exécution de taches complexes de validation
- Le serveur d'IC peut également rendre compte de facteurs comme les ressources utilisées (cpu, temps, mémoire).

Exemple avec Jenkins

- répertoire projet :
  - projet/ :
    - makefile
    - main.c
- installation de jenkins et connexion au port 8080 sur localhost :



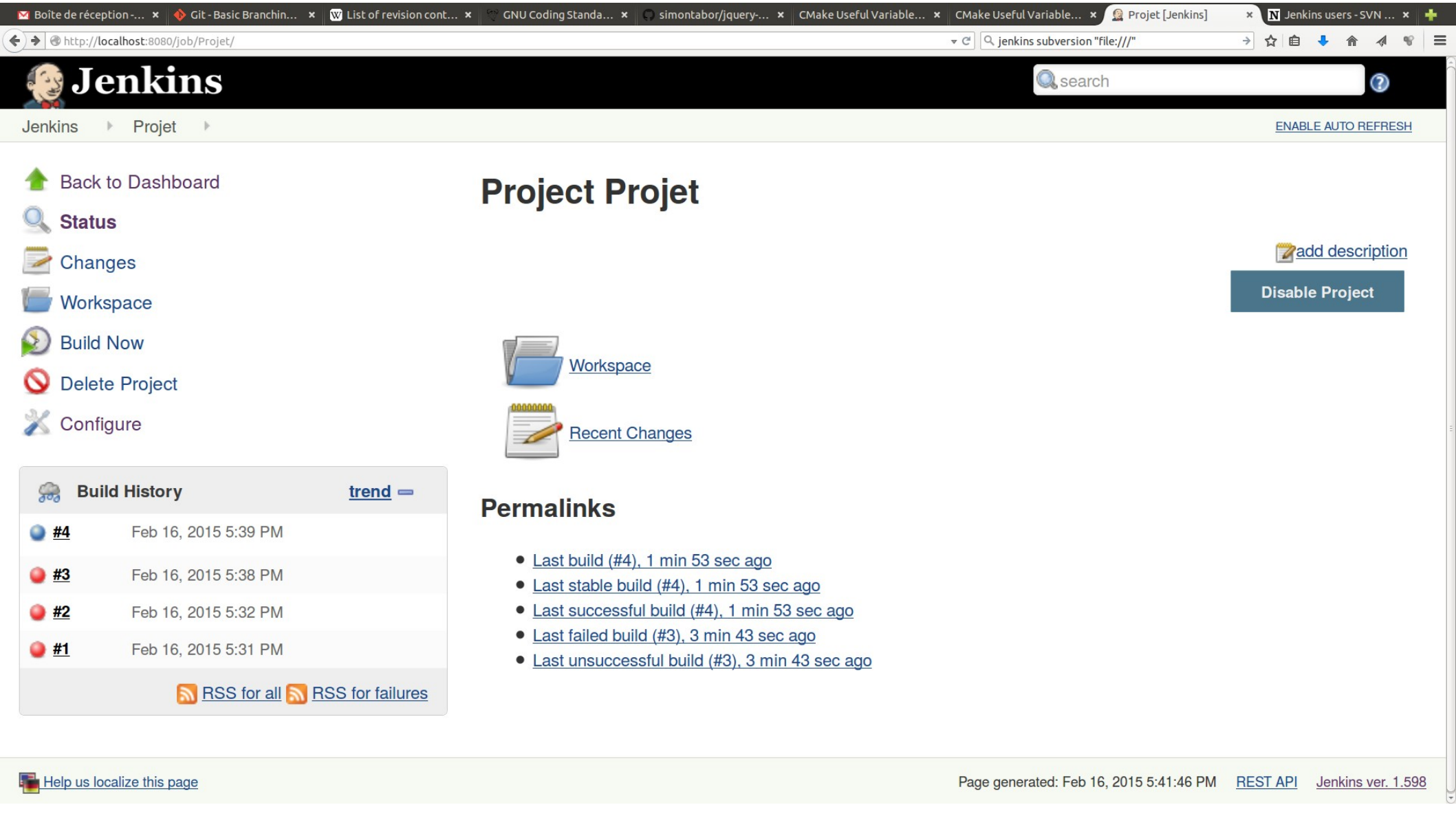
Jenkins : exemple

- création d'un dépôt (svn) local avec les sources :

  svnadmin create /tmp/projet

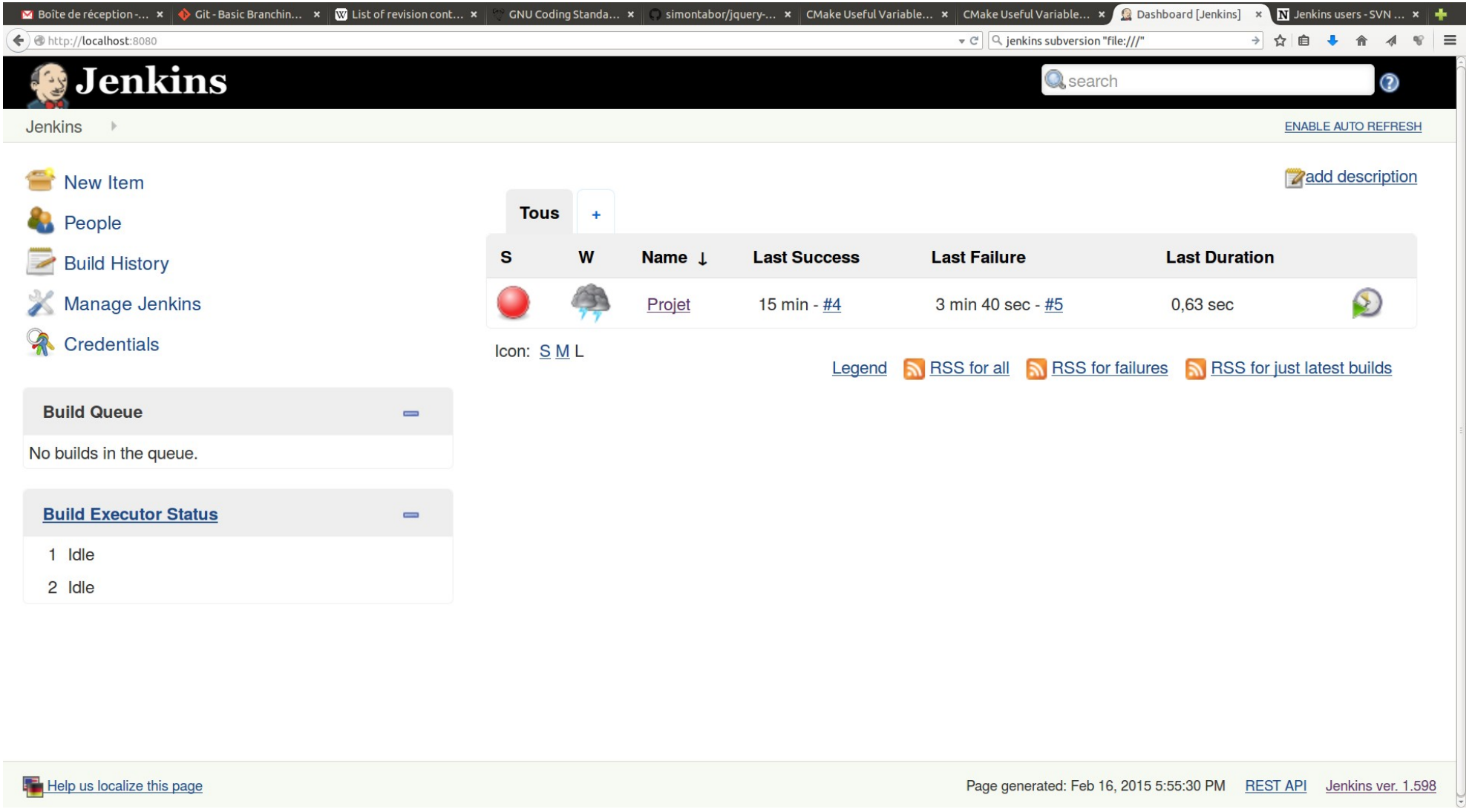
  checkout + ajout des sources + commit
- Ajout du dépôt dans Jenkins en utilisant comme url `svn+ssh://localhost/tmp/projet/`
- Ajout comme commande de build : make
- Lancement d'un build dans jenkins

Jenkins : statut du projet

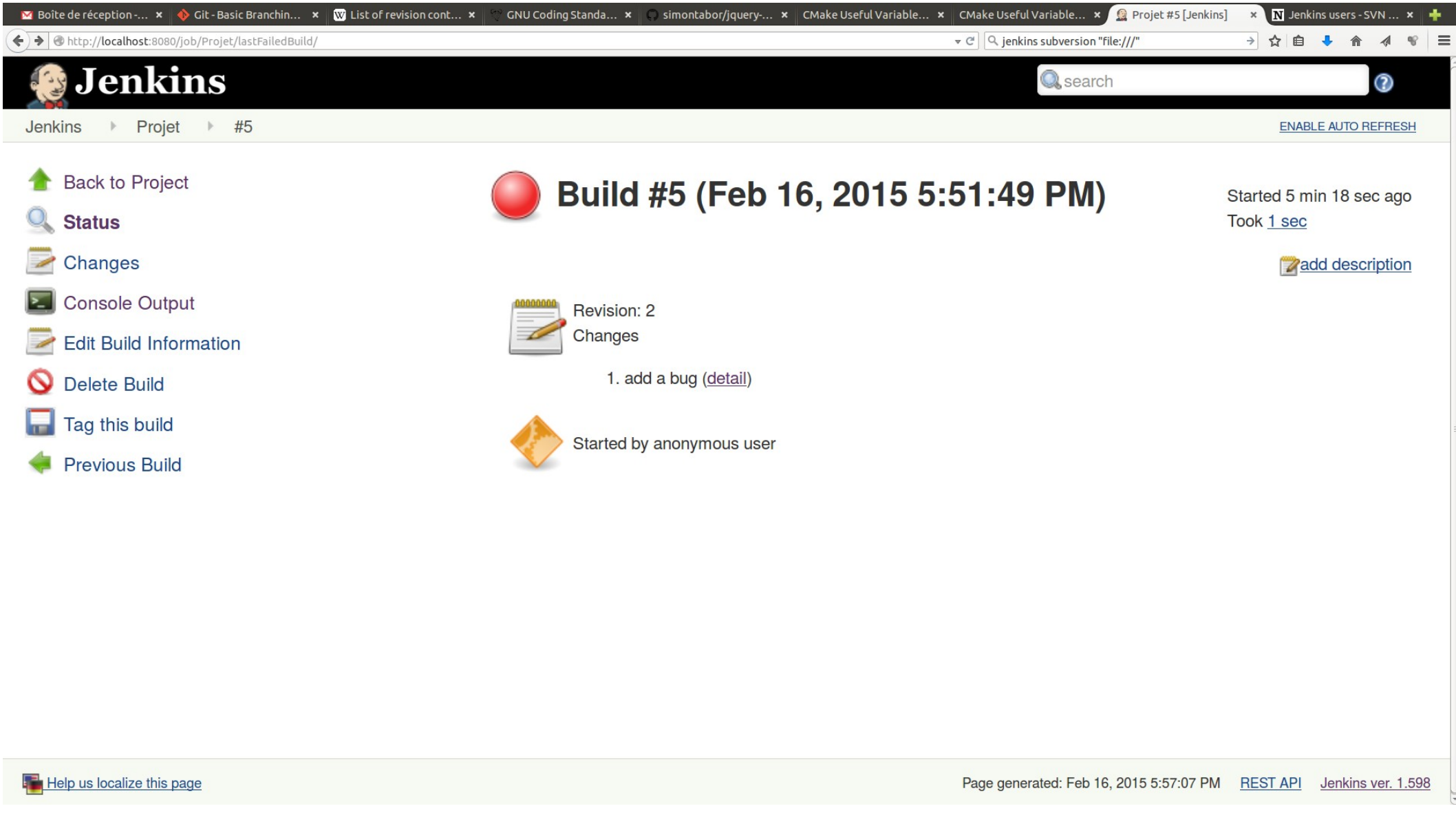


Jenkins : commit

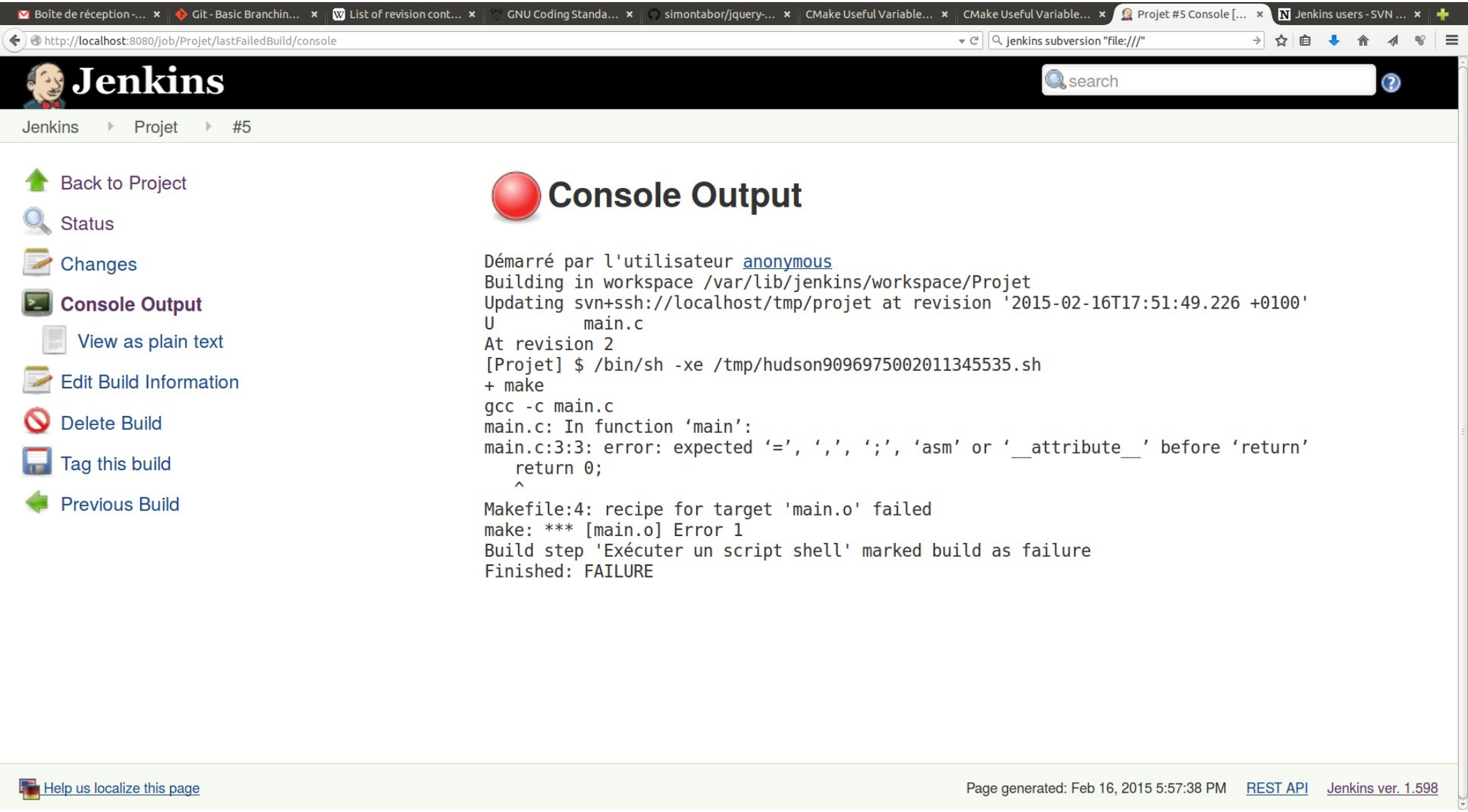
- Ajout d'un bug, commit dans le dépôt



Jenkins : bug


Boîte de réception -...
Git - Basic Branchin...
List of revision cont...
GNU Coding Standa...
simontabor/jquery-...
CMake Useful Variable...
CMake Useful Variable...
Projet #5 [Jenkins]
Jenkins users - SVN ...
http://localhost:8080/job/Projet/lastFailedBuild/
jenkins subversion "file:///"
Jenkins
search
Jenkins
Projet
#5
ENABLE AUTO REFRESH
Back to Project
Status
Changes
Console Output
Edit Build Information
Delete Build
Tag this build
Previous Build
Build #5 (Feb 16, 2015 5:51:49 PM)
Started 5 min 18 sec ago
Took 1 sec
add description
Revision: 2
Changes
1. add a bug (detail)
Started by anonymous user
Help us localize this page
Page generated: Feb 16, 2015 5:57:07 PM
REST API
Jenkins ver. 1.598

Jenkins : bug

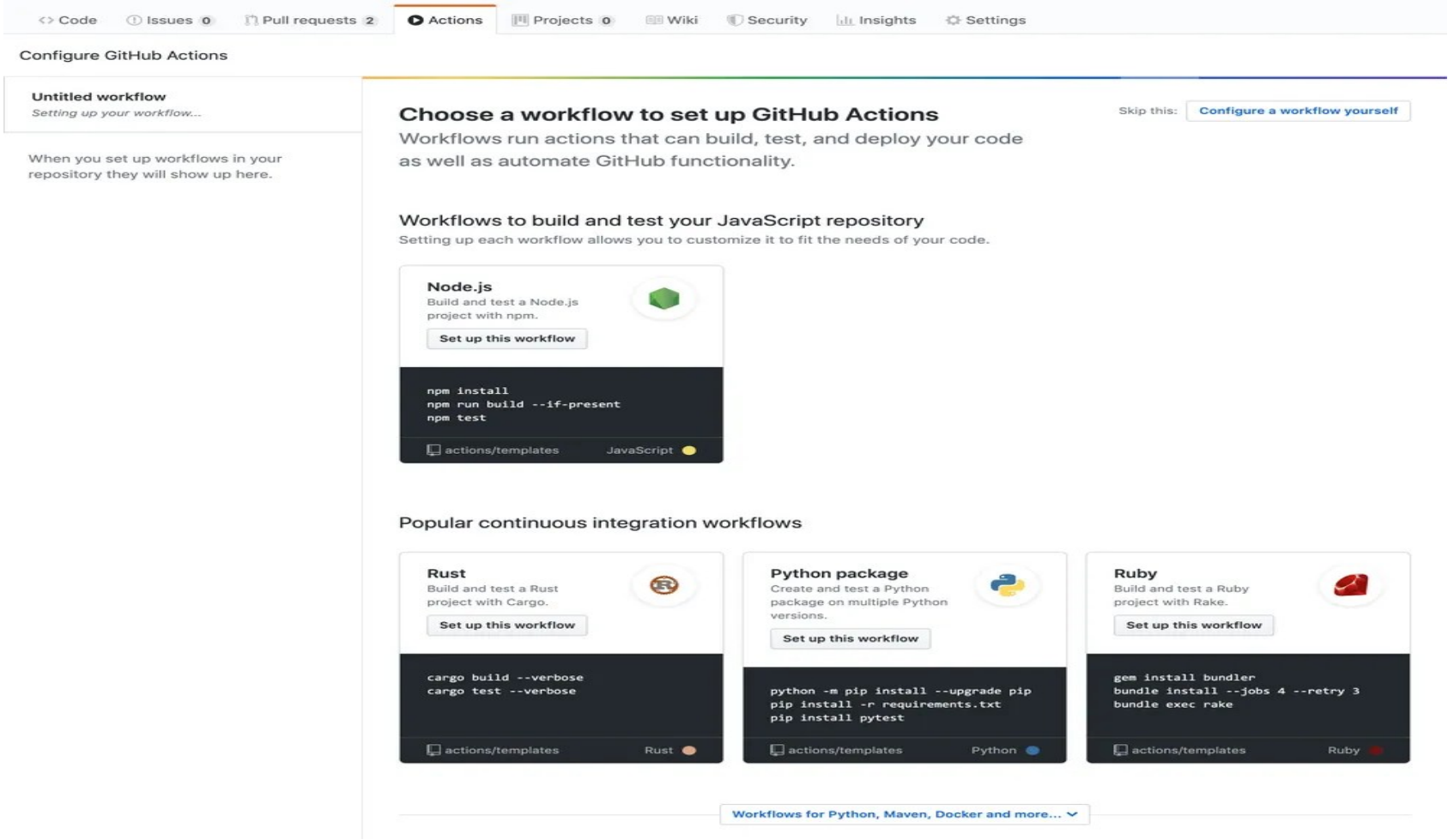

Boîte de réception -...
Git - Basic Branchin...
List of revision cont...
GNU Coding Standa...
simontabor/jquery-...
CMake Useful Variable...
CMake Useful Variable...
Projet #5 Console [...
Jenkins users - SVN ...
http://localhost:8080/job/Projet/lastFailedBuild/console
jenkins subversion "file:///"
Jenkins
search
Jenkins
Projet
#5
Back to Project
Status
Changes
Console Output
View as plain text
Edit Build Information
Delete Build
Tag this build
Previous Build
Console Output
Démarré par l'utilisateur anonymous
Building in workspace /var/lib/jenkins/workspace/Projet
Updating svn+ssh://localhost/tmp/projet at revision '2015-02-16T17:51:49.226 +0100'
U main.c
At revision 2
[Projet] $ /bin/sh -xe /tmp/hudson9096975002011345535.sh
+ make
gcc -c main.c
main.c: In function ‘main’:
main.c:3:3: error: expected ‘=’, ‘,’, ‘;’, ‘asm’ or ‘__attribute__’ before ‘return’
return 0;
^
Makefile:4: recipe for target 'main.o' failed
make: *** [main.o] Error 1
Build step 'Exécuter un script shell' marked build as failure
Finished: FAILURE
Help us localize this page
Page generated: Feb 16, 2015 5:57:38 PM
REST API
Jenkins ver. 1.598

Intégration Continue

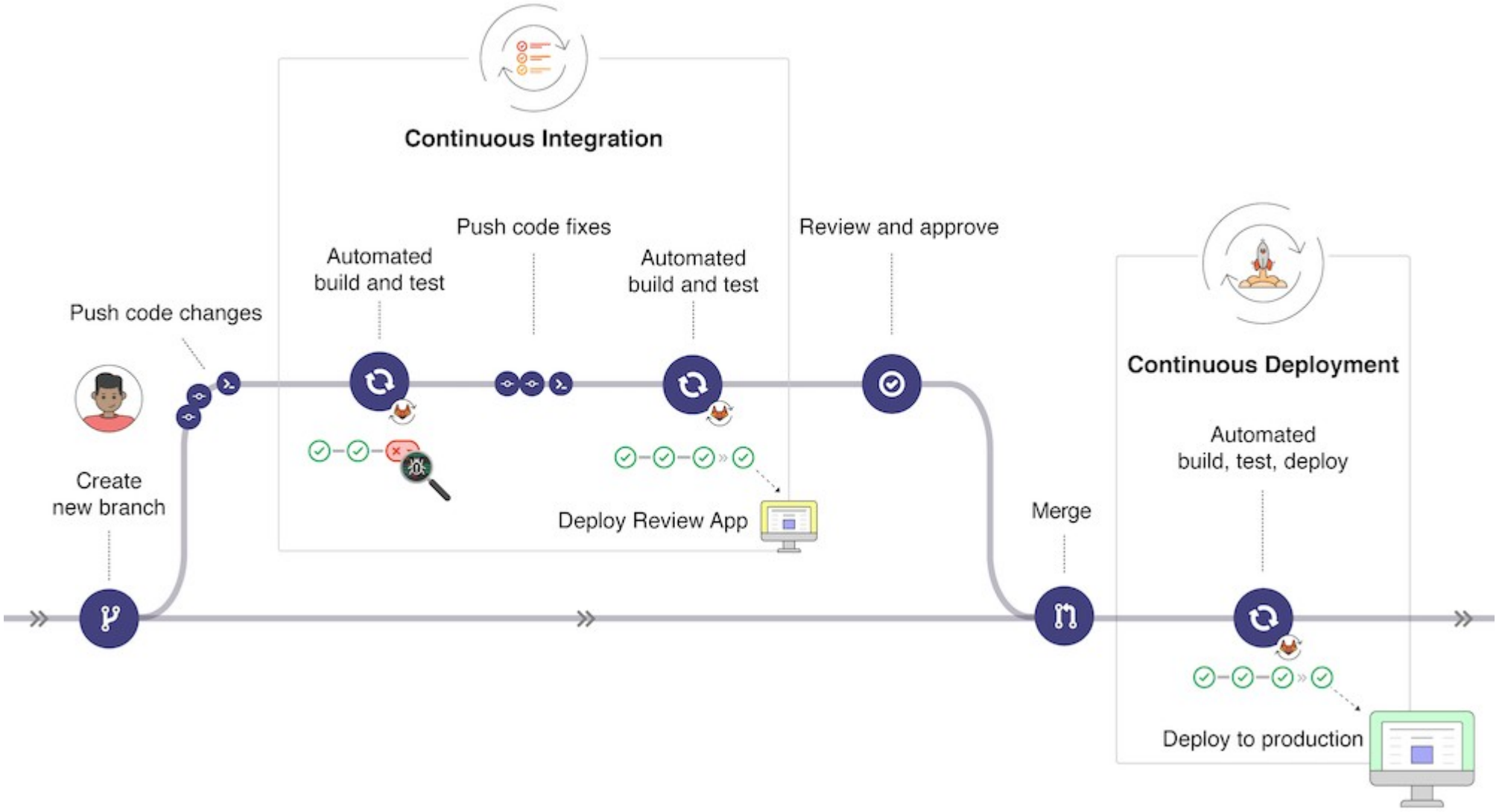
- Tester que le programme compile sous plusieurs environnements est bien mais cela n'offre que peu de garanties quand à l'état **fonctionnel** du projet.
- Pour *garantir* une qualité tout au long du développement, il faut ajouter des **tests**

Intégration Continue & github

- GitHub propose un système d'intégration continue appelé GitHub Action:



Intégration Continue & gitlab



Intégration Continue & gitlab

- l'IC repose sur l'écriture d'un fichier yml:

```yaml
build-job:
  stage: build
  script:
    - echo "Hello, $GITLAB_USER_LOGIN!"

test-job1:
  stage: test
  script:
    - echo "This job tests something"

test-job2:
  stage: test
  script:
    - echo "This job tests something, but takes more time than test-job1."
    - echo "After the echo commands complete, it runs the sleep command for 20 seconds"
    - echo "which simulates a test that runs 20 seconds longer than test-job1"
    - sleep 20

deploy-prod:
  stage: deploy
  script:
    - echo "This job deploys something from the $CI_COMMIT_BRANCH branch."
  environment: production
```