



**Légende :**

* *En jaune clair italic :* l’histoire
* *En gris clair :* du code à copier/coller; des extraits de console
* En gris foncé sur gris foncé : les corrections (gris foncé sur gris foncé)
* Certaines étapes ne sont pas indispensables ; elles sont marquées (**bonus**) ou (**conseil**)

****

1. Exercice 1 : git stash

<https://git-scm.com/docs/git-stash/2.16.0>

***Lundi matin 7h30*** *(Oui oui, en Lorraine on commence tôt !)*

*Autour d’un café / croissant, le business analyst arrive et demande une nouvelle fonctionnalité de clic partout sur tous les champs cliquables d’un écran. Cette fonctionnalité doit-être implémentée d’ici 10 min, GO!!*

* *Nouvelle API Ecran#clicAleatoire()*
* *Nouvel utilitaire : UtilitaireDeClicPartout*
* *Et création de la classe de TU associé*
  1. Récupérez le dépôt suivant via le terminal (git bash sous windows) : [https://github.com/efluid/](https://github.com/efluid/junit4.git)jeTestDoncJeSuis-hol (Ou via la clé USB)

git clone [https://github.com/efluid/](https://github.com/efluid/junit4.git)jeTestDoncJeSuis-hol

* 1. Vérifiez que ça compile avec maven : mvn clean install
  2. Ouvrir **Eclipse** / **Intellij (ou l’IDE de votre choix)** :
* Importez le projet
* Lancez le test UtilitaireDeClicAleatoireTest et vérifier qu’il passe
  1. Ajoutez une nouvelle fonctionnalité : "**clic partout**"
* Proposition d’ajout dans la classe js.jetestdoncjesuis#Ecran

*public void clicPartout() {  
 for (Bouton bouton : boutons) {  
 bouton.clic();  
 }  
}*

* Proposition de classe à ajouter : js.jetestdoncjesuis#UtilitaireDeClicPartout

*package js.jetestdoncjesuis;*

*import java.util.Collection;*

*public class UtilitaireDeClicPartout {*

*public void clicsPartout(Collection<Ecran> ecrans) {*

*System.out.println("\n\nDémarrage des clics partout :");*

*for (Ecran ecran : ecrans) {*

*ecran.clicPartout();*

*}*

*System.out.println("Fin des clics partout\n\n");*

*}*

*}*

* Proposition de classe de test associé : js.jetestdoncjesuis#UtilitaireDeClicPartoutTest

*package js.jetestdoncjesuis;*

*import static org.junit.Assert.fail;*

*import java.util.Collection;*

*import org.junit.\*;*

*public class UtilitaireDeClicPartoutTest {*

*private Collection<Ecran> ecrans = null;*

*@Before*

*public void init() {*

*ecrans = ContexteTest.creerEcrans();*

*}*

*@Test*

*public void quand\_j\_ouvre\_un\_ecran\_je\_clique\_partout\_et\_rien\_ne\_bug() {*

*UtilitaireDeClicPartout utilitaireDeClic = new UtilitaireDeClicPartout();*

*try {*

*utilitaireDeClic.clicsPartout(ecrans);*

*} catch (Exception ex) {*

*fail("Bug en approche !!");*

*}*

*}*

*}*

***5 min plus tard*** *: l’expert test fonctionnel demande une correction en urgence sur une branche de maintenance (bugfix) sur la fonctionnalité de test aléatoire qui ne lui semble pas vraiment aléatoire.*

*Une fois le bug fixé, il faut retourner sur la fonctionnalité pour la terminer car le chef de projet vient d’arriver dans le bureau avec son café et met la pression.*

* 1. Stashez toutes les modifications avec un message "mon premier stash"

(dos2unix <nomDuFichier> en cas de problème de CRLF)

Commande la plus simple :

git stash push

ou git stash (alias de git stash push)

ou git stash save (déprécié, il faut maintenant utiliser git stash push)

* 1. Regardez l’état du dépôt et l’état de la liste des stash :

git status

puis git stash list

* L’ensemble du contenu est-il bien stashé ?
* Le message du stash est-il propre ?

⇒ Ne passez pas à la suite tant que vous n’avez pas un **stash unique** avec l’ensemble du contenu et un message “mon premier stash”

(dos2unix <nomDuFichier> en cas de problème de CRLF)

Pour revenir en arrière sur l’étape §1.6 : git stash pop

Option 1 : git stash push -u -m "mon premier stash"

Option 2 : git add --all puis git stash push -m "mon premier stash"

* 1. Consultez le contenu du stash (En visualisant les lignes de code Java modifiées)

git stash show -p stash@{0} ⇒ avec l’option “-p”

git diff stash@{0}

* 1. Remettre le contenu du stash dans le dépôt **tout en conservant le stash dans la liste**

git stash apply

ou git stash apply stash@{0}

* 1. Créer **2 nouveaux stashs** (avec une partie des modifications dans chaque stash) :
* Un premier avec la modification dans Ecran nommé “modification Ecran”
* Et un deuxième avec UtilitaireDeClicPartout et UtilitaireDeClicPartoutTest nommé “ajout utilitaire de clic partout”

git stash push -m "modification Ecran" -- <pathFichierEcran>

git stash push -m "ajout utilitaire de clic partout" -- <pathUtilitaireDeClicPartout > <pathUtilitaireDeClicPartoutTest>

* 1. Vérifiez l’état de la liste des stash ; il en contient normalement 3

stash@{0}: ajout utilitaire de clic partout

stash@{1}: modification Ecran

stash@{2}: mon premier stash

git stash list

* 1. Dépiler le contenu du stash “modification Ecran” dans le répertoire de travail

git stash pop stash@{1}

* 1. Supprimez le stash nommé “mon premier stash”

git stash drop stash@{1}

* 1. Oooops ; je ne voulais pas tout perdre. Je souhaite récupérer le contenu du stash “mon premier stash” et le mettre dans un stash nommé “oops”

⇒ le numéro de commit est indiqué en résultat de la commande précédente (§1.13)

La commande précédente (git stash drop stash@{1}) a affiché un numéro de commit en résultat ; il faut l’utiliser en entrée dans la commande suivante :

git stash store -m "oops" <mon numéro de commit>

(**Option :** Une commande “mystique” est dispo dans l’aide pour retrouver le numéro de commit)

* 1. Créez une branche “feat-clic-partout” à partir du stash nommé "oops" et commiter le code avec le message “ajout de l'utilitaire clic partout”

git stash branch feat-clic-partout stash@{0}

git commit -a -m "ajout de l’utilitaire clic partout"

* 1. Nettoyez la liste des stash

git stash clear

****

1. Exercice 2 : git worktree <https://git-scm.com/docs/git-worktree/2.16.0>

***Mardi matin*** *:*

*On a de nouveau une correction à réaliser en parallèle de la réalisation d’une nouvelle fonctionnalité.*

*Mais cette fois-ci, on veut être plus productif !!*

* 1. Positionnez-vous sur “master” proprement : git reset --hard; git clean -fd; git checkout master
  2. Créez un worktree nommé “jeTestDoncJeSuis-hol-bugfix” (Ici, un nouveau “répertoire” va être créé ; positionnez-le à côté de l’actuel : “../jeTestDoncJeSuis-hol-bugfix”)

git worktree add ../jeTestDoncJeSuis-hol-bugfix

* 1. Consultez la liste des worktree ; voici le résultat que vous devez avoir :

../jeTestDoncJeSuis-hol ⇒ main worktree

../jeTestDoncJeSuis-hol-bugfix

git worktree list

* 1. Comparez l’espace occupé par ces 2 répertoires (commande "du -hs <path>")

du -hs ../jeTestDoncJeSuis-hol ⇒ 46 Mo

du -hs ../jeTestDoncJeSuis-hol-bugfix ⇒ 3,2 Mo

* 1. Dans votre IDE : ouvrir le worktree “nouveau worktree” (jeTestDoncJeSuis-hol-bugfix) via un import de projet
  2. Réalisez la correction (décommentez le code dans Ecran#clickAleatoire()) dans le worktree jeTestDoncJeSuis-hol-bugfix
  3. Ouvrir 2 terminaux (git bash sous windows) (un worktree différent sur chaque fenêtre) : vous pouvez lancer les compilations / tests sur chaque worktree en même temps
  4. Commitez la correction sur le worktree "jeTestDoncJeSuis-hol-bugfix"

git commit -a -m "modification de la gestion du caractère aléatoire du clic"

* 1. Depuis le worktree master (jeTestDoncJeSuis-hol) :
* Est-ce que le commit est visible ? (git show par exemple)
* Est-ce qu’on peut faire un cherry-pick dessus ?

Il faut changer de répertoire pour changer de worktree.

Oui le commit est visible et Cherry-pick possible.

* 1. (**Bonus**) Lancez une commande sur les différents worktree en même temps à partir du même terminal (git bash sous windows) ?

Sous windows : git worktree list --porcelain | awk '/^worktree/ {print "/" $2 "/pom.xml"}' | xargs -I '{}' mvn clean install -f '{}'

* 1. Le travail est terminé, supprimez le worktree "jeTestDoncJeSuis-hol-bugfix"

Se placer dans le répertoire “jeTestDoncJeSuis-hol” et lancer les commandes suivantes.

rm -rf ../jeTestDoncJeSuis-hol-bugfix && git worktree prune

ou “rm -rf ../jeTestDoncJeSuis-hol-bugfix” puis “git worktree prune”

git worktree remove (A partir de la version 2.17)

<https://stackoverflow.com/questions/44109234/how-to-delete-a-git-working-tree-branch-when-its-working-directory-has-been-remo/49331132#49331132>

* 1. Consultez la liste des worktree pour vérifier que le worktree est bien supprimé

git worktree list

* 1. Positionnez vous sur le worktree principal (../jeTestDoncJeSuis-hol) ; est-ce que vous arrivez à retrouver la branche “jeTestDoncJeSuis-hol-bugfix” associé au worktree supprimé à l’étape 2.11 ?

Oui, elle est bien toujours présente malgré la suppression du worktree.

**Complément d’info** : on peut créer un worktree à partir d’une branche existante

****

1. Exercice 3 : git rebase interactif <https://git-scm.com/docs/git-rebase/2.16.0>

***Mardi après-midi*** *: le Business Analyst souhaite, pour une raison inconnue, traduire les messages de log en anglais (oui oui, les messages de log …).*

**Un rebase tout court pour commencer :**

* 1. Positionnez-vous sur “master” proprement : git reset --hard; git clean -fd; git checkout master
  2. Récupérez localement la branch “rebase” qui doit suivre la branche distante “rebase”

git checkout rebase

* 1. Créez une branche “exercice\_3a” à partir de la branche “master” et placez-vous dessus.

git checkout -b exercice\_3a master

* 1. Traduisez le contenu des messages de log de UtilitaireDeClicAleatoire#clicsAleatoires.
  2. Commitez.

*Le Business Analyst souhaite que cette feature englobe les modifications faites par une équipe off-shore sur la branche ‘rebase’.*

* 1. Effectuez un rebase avec la branche “rebase”

git rebase rebase

* 1. Que se passe-t-il ?

on a un conflit avec un commit de la branche ‘rebase’

* 1. Il va falloir résoudre le conflit avec classe, c’est à dire dans un outil fait pour ça car dans l’IDE ça donne quelque chose comme ça :

package js.jetestdoncjesuis;

import java.util.Collection;

public class UtilitaireDeClicAleatoire {

public void clicsAleatoires(Collection<Ecran> ecrans) {

<<<<<<< HEAD

System.out.println("\n\nDémarrage des clicks aléatoires :");

=======

System.out.println("\n\nStarting random clicks :");

>>>>>>> [feat] traduction en anglais des messages de log.

for (Ecran ecran : ecrans) {

ecran.clicAleatoire();

}

System.out.println("Random clicks done\n\n");

}

}

* Ajoutez l’outil de votre choix dans votre .gitconfig (PS : ça marche aussi pour diff/difftool) :

[merge]

tool = bc3

[mergetool "bc3"]

path = D:/Programs/beyondcompare/bcomp.exe

* Résolvez le conflit (avec classe) git mergetool; git rebase --continue

Pour info, quelques outils : Beyond Compare, kdiff3, meld, TortoiseMerge

* (**Bonus**) Des fichiers ont été créés, lesquels ? Que contiennent-ils ?

Des fichiers .orig. Ils contiennent le contenu du fichier avant la résolution de conflit.

* + (**Conseil**) Ok c’est cool mais comment on fait pour que git ne les génère pas parce qu’en fait je les supprime tout le temps : git config --global mergetool.keepBackup false
* Lors d’un rebase, si jamais vous vous êtes mis les pieds dans le tapis, il y a une commande qui permet de revenir dans l’état pré-rebase. Quelle est-elle ?

git rebase --abort

* 1. (**Bonus**) Vous êtes fatigué et vous supprimez votre branche “exercice\_3a”. Zut, il faut tout recommencer, même la résolution du conflit … sauf si git se souvient de votre résolution et l’applique tout seul : nous allons donc activer le ***Re****use* ***Re****corded* ***Re****solution.* git config --global rerere.enabled true
* Si vous avez le temps, faites le test :)
* Un peu plus de documentation : <https://delicious-insights.com/fr/articles/git-rerere/>
  1. **Et maintenant le rebase intéractif**

Vous devez développer 2 features :

* Calcul du temps des tests effectués
* Génération des données au format texte des temps des tests (pour être consommer par un grafana ou équivalent)

Ici, vous avez 2 solutions :

* Soit vous lancez le script magique exercice\_31.sh qui va vous créer une branche locale “exercice\_rebase\_interactif” et qui va cherry-picker les bons commits à partir de la branche “rebase\_interactif”
  + Ils sont aux nombres de 8 : 4 commits par feature
    - 2 pour les implémentations
    - 2 pour les tests
* Soit vous êtes aventurier et vous codez dans votre IDE préféré les 2 features

(3.11 Mode aventure)

* 1. (**Bonus**) Mode aventure
* Un rapide guide d’implémentation est disponible ci-dessous
* Attaquez le développement de la première feature (en faisant 2 commits si possible)
* Développez les TUs de la première feature (en faisant 2 commits si possible)
* Attaquez le développement de la seconde feature (en faisant 2 commits si possible)
* Développez les TUs de la seconde feature (en faisant 2 commits si possible)
* Guide d’implémentation :
  + un package *calcultemps* dans jetestedoncjesuis, une interface *ICalculDeTemps* et une implémentation qui porte 2 long (un temps de début et un temps de fin), une interface *ICalculateurDeTempsDexecutionService* avec 3 signatures (*lancerCalculTempsExecution(ICalculDeTemps)*, *arreterCalculTempsExecution(ICalculDeTemps)* et *calculerTempsExecution(ICalculDeTemps))* et une implémentation simple.
  + Un TU du service de calcul
  + Un package *export* dans jetestedoncjesuis, une interface *IServiceDExportTempsExecution* avec une seule signature qui prend en paramètre un *ICalculDeTemps* et qui utilise le service de calcul pour exporter ça dans un format quelconque (JSON, texte plat, sortie standard, etc.)
  + Un TU du service d’export

*Nous sommes désormais le soir 18h00 et y’a un tournoi de babyfoot dans votre bar préféré du coup vous partez sans merger votre travail sur master, il fera jour demain.*

*Nous sommes maintenant* ***mercredi matin****,* ***9h00*** *(bah oui on a bu des bières aussi en plus de jouer au baby) et on rallume le PC. On s'apprête à pousser et tout d’un coup le CTO débarque dans l’open space :*

*“Et les krombolz, dans 15 minutes je check la master pour voir à qui je donne les primes sur objectifs alors s’il vous reste un truc à pousser vous avez 15 minutes”.*

*Et là, vous vous rappelez les règles de qualité de dév. qu’il y a sur le Wiki interne.*

*Dont deux :*

* *“Découper au maximum ses commits”*
* *“Effectuer les features en TDD”.*

*Et là, c’est le drame : sur nos 2 dernières features, on a fait le dev et ensuite on a fait les tests. Alors certes la première règle est respectée (peut-être même un peu trop) mais pas la deuxième. Et le CTO, c’est un robot : bye bye la prime. Bon on a 15 minutes comment je fais pour avoir ma prime → on envoie le rebase interactif (git rebase -i) pour remettre les commits au propre et remettre ceux de tests avant ceux des implémentations, et on gagne notre prime.*

* 1. (**Conseil**) Si vous n’aimez pas vi, vous pouvez configurer un éditeur de texte pour réaliser les modifications lorsque git le demande (rebase interactif, message de commit, etc.) ; dans votre gitconfig

[core]

editor = 'D:/Programs/Sublime Text 2.0.2 x64/sublime\_text.exe' --wait

* 1. Commencez par regrouper vos commits par feature via le rebase intéractif :

Faites en sorte de n’avoir plus que 4 commits (1 d’implémentation, 1 de tests,1 d’implémentation, 1 de tests)

git rebase -i HEAD~8

* 1. Vous arrivez sur un écran comme ci-dessous

pick 04f192ac premier commit pour calcul (interfaces).

pick a55627ee implémentation d'un calculateur via interpolation quantique.

pick 65aada58 test unitaire pour CalculateurDeTempsDexecutionViaInterpolationQuantiqueService.

pick aa95588b fix TU CalculateurDeTempsDexecutionViaInterpolationQuantiqueService.

pick 665a5307 dépendance Jackson + interface service d'export.

pick d7d6cf05 service d'export à fichier plat.

pick e8b33f68 service d'export sous forme de fichier.

pick 9b50792b TU de ServiceDExportTempsExecutionPlainTextTest.

# Rebase 3d30c6db..9b50792b onto 3d30c6db (8 commands)

#

# Commands:

# p, pick = use commit

# r, reword = use commit, but edit the commit message

# e, edit = use commit, but stop for amending

# s, squash = use commit, but meld into previous commit

# f, fixup = like "squash", but discard this commit's log message

# x, exec = run command (the rest of the line) using shell

# d, drop = remove commit

#

# These lines can be re-ordered; they are executed from top to bottom.

#

# If you remove a line here THAT COMMIT WILL BE LOST.

#

# However, if you remove everything, the rebase will be aborted.

#

# Note that empty commits are commented out

* 1. Ici, le but est de regrouper les commits : quelle commande utiliser ? Sachant que la commande mise devant l’id de commit s’applique ‘sur’ le commit du dessus.

squash

**Rappel :** Faites en sorte de n’avoir plus que 4 commits (1 d’implémentation, 1 de tests,1 d’implémentation, 1 de tests)

**Soyez cool avec votre communauté :** renommez proprement les nouveaux commits fusionnés :)

Résultat à obtenir à ce stade :

tambori@PC2215 MINGW64 /d/java/workspaces/developpement\_dev/jeTestDoncJeSuis-hol (rebase\_interactif u+4-8)

$ git log -5

commit 85dfb369f2678940f8fbe4564fea254e84e12509 (HEAD -> rebase\_interactif)

Author: Jerome Tamborini <j-tamborini@efluid.fr>

Date: Mon Apr 2 11:08:35 2018 +0200

TU de ServiceDExportTempsExecutionPlainTextTest.

commit f526315a111b4e128f6a136bc302303937fd1372

Author: Jerome Tamborini <j-tamborini@efluid.fr>

Date: Wed Mar 28 11:25:18 2018 +0200

[feat] service d'export sous forme de fichier plat.

commit 2dce2556ea2951db5351de2083967a276d42c08d

Author: Jerome Tamborini <j-tamborini@efluid.fr>

Date: Tue Mar 27 13:16:55 2018 +0200

[test] TU du service de calcul de temps.

commit f4f7a446d96c717dc97adb8da57de2b6ef03b8b4

Author: Jerome Tamborini <j-tamborini@efluid.fr>

Date: Mon Mar 26 17:10:06 2018 +0200

[feat] Implémentation des beans et des services pour le calcul de temps.

commit 3d30c6db3b9f450bde23a589ca2feea9f938e567

Author: DMYTRYK Alexis <a-dmytryk@efluid.fr>

Date: Sun Mar 25 22:05:28 2018 +0200

* 1. Amendez le dernier commit avec un message propre

git commit --amend

* 1. Inversez le sens des commits lors d’un second rebase interactif

*(Parce que le but c’est quand même de réussir à avoir la prime :))*

Résultat à obtenir à ce stade :

tambori@PC2215 MINGW64 /d/java/workspaces/developpement\_dev/jeTestDoncJeSuis-hol (rebase\_interactif u+4-8)

$ git log -5

commit 2c0911989f5b3f2a1bae69b64043bca8736c376a (HEAD -> rebase\_interactif)

Author: Jerome Tamborini <j-tamborini@efluid.fr>

Date: Wed Mar 28 11:25:18 2018 +0200

[feat] service d'export sous forme de fichier plat.

commit bf4fe2ad386e933e4a123bbaeaccce0acec299e5

Author: Jerome Tamborini <j-tamborini@efluid.fr>

Date: Mon Apr 2 11:08:35 2018 +0200

[test] TDD->TU de ServiceDExportTempsExecutionPlainTextTest.

commit 587be14cf8f37451b4b18986bfb1add9894af0cd

Author: Jerome Tamborini <j-tamborini@efluid.fr>

Date: Mon Mar 26 17:10:06 2018 +0200

[feat] Implémentation des beans et des services pour le calcul de temps.

commit 8af60264e2078c193aa997e8e6ef5a17a8d350ec

Author: Jerome Tamborini <j-tamborini@efluid.fr>

Date: Tue Mar 27 13:16:55 2018 +0200

[test] TDD->TU du service de calcul de temps.

commit 3d30c6db3b9f450bde23a589ca2feea9f938e567

*Et voilà la prime est pour nous :).*

****

1. Exercice 4 : git reflog

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

***Jeudi Matin*** *:*

*La veille au soir, on est sorti au bar (normal)*

*On attaque un nouveau développement.*

*Puis coup de téléphone !! On switch sur autre-chose (sur un commit précis par exemple) et on ne se rappelle plus de l’emplacement du nouveau développement qu’on était en train de réaliser.*

* 1. Positionnez-vous sur “master” proprement : git reset --hard; git clean -fd; git checkout master
  2. Faites une modification de code
* Par exemple : “touch jeCommenceParEcrireUneDoc.adoc”
  1. Commitez cette modification de code
  2. Supprimez votre commit

git reset --hard HEAD^

* 1. Vous consultez l’historique et vous vous positionnez sur votre précédent commit (en mode "A l’arrache") :
* git log --oneline -n 10 ⇒ sélectionner un précédent commit
* git checkout -f <mon commit>
  1. Consultez le reflog
* Filtrez par branche (ici par la branche master)
* Affichez la date et/ou la date relative sur chaque ligne de log
* Identifiez votre commit

git reflog show

git reflog show master

git reflog show --date=relative

git reflog show --date=relative master

git reflog show --date=relative <ma branche>

git reflog show --date=local master

git reflog show --date=local <ma branche>

* 1. Revenez sur master et récupérez le code via un cherry-pick sans le commiter directement (car il n’est pas terminé)

git checkout master

git cherry-pick -n <numéroCommit>

ou git cherry-pick -n HEAD@{2} (A voir en fonction du résultat de la commande git reflog)

* 1. Commitez le code

git commit -a -m "XXX"

* 1. Oups, une petite coquille : faites une modification (ex : echo “toto” >> jeCommenceParEcrireUneDoc.adoc) et amendez le commit

git commit --amend

* 1. Consultez le reflog

****

1. Exercice 5 : git bisect

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

***Vendredi matin :***

*Le chef de projet débarque en furie car un bug vient d’être détecté en production. Personne ne l'avait vu avant. Ah bon ?*

*Vous devez trouver au plus vite le commit à l’origine de ce bug en utilisant la commande git bisect.*

* 1. Positionnez-vous sur “master” proprement : git reset --hard; git clean -fd; git checkout master ou si vous voulez supprimer le commit de la fin de l’exercice précédent : git reset --hard HEAD^1; git clean -fd; git checkout master
  2. Se placer sur la branche de production : “production”
  3. Lancez les tests avec maven => quel est le constat ?

mvn clean test

* 1. Quel est le commit qui pose problème ?

Lancez un bisect automatique entre le commit courant et le commit 6d140c7c0176c683269dc9d897dbdb1a6b4b7554 car on sait que celui-la fonctionne.

Notez le temps global qu’a pris le mode automatique. Sortez du mode bisect

git bisect start HEAD 6d140c7c0176c683269dc9d897dbdb1a6b4b7554

git bisect run mvn clean test

Ou pour avoir le temps global : time git bisect run mvn clean test

git bisect reset

* 1. (**Bonus**) Comment, de manière plus visuelle, voir le commit qui a fait planté les tests ?

git bisect visualize

* 1. Effectuez la même manipulation mais en mode manuel (mode basic dans la documentation de git bisect)

git bisect start HEAD 6d140c7c0176c683269dc9d897dbdb1a6b4b7554

//vérifications manuelles

git bisect good

//vérifications manuelles

git bisect good

//vérifications manuelles

git bisect bad

//vérifications manuelles

git bisect good

//vérifications manuelles

git bisect good

//vérifications manuelles

git bisect good

(**Bonus**) **On voudrait améliorer le processus, car c’est assez long, pour cela on propose d’utiliser Gradle.**

* 1. Convertir le projet au format Gradle

gradle init

=> vérification du fichier build.gradle généré

=> ajout de :

test {

filter {

//specific test method

includeTestsMatching "org.junit.tests.AllTests"

}

}

Et si nécessaire en cas de problème de récupération des dépendances

repositories {

jcenter()

}

* 1. Lancez un build avec Gradle

gradle build

* 1. Relancez un bisect automatique. Les performances sont-elles meilleures ?

git bisect start HEAD 6d140c7c0176c683269dc9d897dbdb1a6b4b7554

git bisect run gradle build

Temps d’execution : 1m33.578s => gain significatif

* 1. Tentez d’améliorer les performances avec les possibilités offertes par Gradle (build cache, parallélisme, etc…)

Utilisation du gradle build cache : <https://docs.gradle.org/current/userguide/build_cache.html>

git bisect start HEAD 6d140c7c0176c683269dc9d897dbdb1a6b4b7554

git bisect run gradle build --build-cache

Temps d’execution : 1m23.764s

Tuning de build.gradle

test.maxParallelForks = Runtime.runtime.availableProcessors().intdiv(2) ?: 1

tasks.withType(JavaCompile) {

options.incremental = true

}

Utilisation du mode parallel

git bisect start HEAD 6d140c7c0176c683269dc9d897dbdb1a6b4b7554

git bisect run gradle build --build-cache --parallel

Temps d’execution : 1m15.642s

Récapitulatif :

|  |  |
| --- | --- |
| Maven | 2m23s |
| Gradle | 1m33s |
| Gradle avec build cache | 1m23s |
| Gradle avec tuning test et parallélisation | 1m15s |

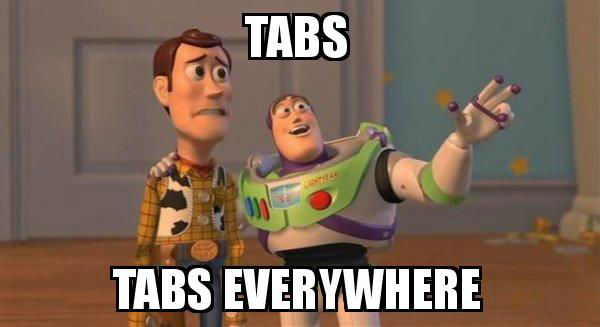
****

1. Exercice 6 : hook / alias <https://git-scm.com/book/en/v2/Git-Basics-Git-Aliases> <https://git-scm.com/book/uz/v2/Customizing-Git-Git-Hooks>

***Samedi matin (oui dans une startup c’est 6 J / 7)***

*Débat interminable au café sur le Space vs Tabs car dès qu’on fait un commit on arrête pas de devoir changer le formatage du code. Y'a pourtant bien un fichier de*

*formatage éclipse à la racine du projet mais personne ne l'utilise. Pour solutionner le problème la décision est prise d'automatiser le formatage.*



* 1. Créez un hook client permettant de remplacer automatiquement tous les tabs des fichiers modifiés par des spaces lors du commit.

Le principe  
  
L’objectif des hooks est simple : il s’agit de fournir des enrobages à certaines commandes pour permettre l’injection de scripts (Bash, Node.js, Perl, Python, PHP…). De cette manière nous pouvons automatiser et donc fiabiliser une partie du travail côté client (machine de l’utilisateur) mais aussi côté serveur.  
  
 Ils sont "par défaut" présents dans chaque répertoire projet dans le sous-répertoire .git/hooks.  
 Leur nom est conventionnel et chaque fichier de script doit être exécutable (chmod +x .git/hooks/…).  
 Leur emplacement fait qu’ils peuvent être supprimés ou désactivés par l’utilisateur sur un poste local.  
  
Ainsi les hooks locaux n’ont pas une vocation de contrôle, mais plutôt de soutien.  
  
À cela s’ajoute le fait qu’un utilisateur peut demander à ce que certains hooks ne soient pas déclenchés, avec l’option --no-verify (pre-commit et commit-msg seulement).  
  
Git fournit à chaque initialisation de projet un ensemble de scripts de démonstration possédant l’extension .sample (ex. : .git/hooks/pre-commit.sample). Pour les activer, il vous suffit de retirer cette extension (ex. : .git/hooks/pre-commit).

* Dans quelle moment du process git le mettre ?

Pre-commit

* Créer le fichier de hook au bon endroit en vous basant sur les exemples qui sont fournis dans le référentiel git avec l’extension .sample

Un exemple de hook est présent dans le dossier : src/test/resources/hks/pre-commit

Vous pouvez le copier dans le dossier .git/hooks

* 1. Testez sur une branche locale avec un fichier source contenant des tabs !

Modification d’un fichier en y ajoutant des tabs

git add .

git commit -m “test hook”

=> Vous devriez voir votre fichier modifié (les tabs remplacés par des espaces)

* 1. Comment faire coté serveur pour s’assurer que les fichiers ne contiennent aucun tabs et rejeter les commits si ce n’est pas le cas ?

Si l’on utilise Git directement (sans application supplémentaire de type github)

Il faut créer des hooks dans le référentiel distant (central) de type :

* Pre-receive
* Update
* Post-receive

Et ajouter le code que l’on souhaite à l’intérieur tout comme pour les hook clients.

Lorsque l’on utilise un application qui wrap Git alors ce type de Hook n’est généralement plus possible pour des raisons de sécurité, il faut alors se référer aux outils que proposent chaque applications, et notamment aux webhooks mais ces derniers peuvent uniquement appeler des services extérieurs mais ne peuvent pas par exemple rejeter un commit s’il n’est pas conforme.

* Github : Enterprise version <https://developer.github.com/v3/repos/pre_receive_hooks/> sinon : <https://developer.github.com/webhooks/>
* GitLab : <https://docs.gitlab.com/ee/user/project/integrations/webhooks.html>
* BitBucket : <https://confluence.atlassian.com/bitbucket/manage-webhooks-735643732.html>
* Gerrit : <https://gerrit-review.googlesource.com/admin/repos/plugins%2Fhooks> ou alors créer un plugin custom

***Ambiance détendue l’après-midi :*** *c’est le week end après tout ! Réflexion autour de l'efficience de développement , et notamment “comment améliorer les outils du développeur ?” En effet aujourd'hui les développeurs font souvent la même chose mais n'ont pas automatisé tout cela.*

**Aliases**

* 1. Quels sont les différentes façons/emplacements d’ajouter un alias git

~/.gitconfig pour en bénéficier dans tous vos dépôts.

.git/config d’un projet pour restreindre son accès à cet unique projet.

Il est également possible de recourir à la commande git config :

git config --global alias.st 'status'

* 1. Créez un alias git ‘git oneline’ qui log proprement en une ligne les derniers commit

alias oneline='git log --oneline'

* 1. Créez un alias git ‘git st’ pour écrire plus vite le status

alias st='git status'

* 1. Créez un alias git ‘git logme’ qui log tous ses commits

alias logme='git whatchanged --author=monmail@mail.com'

* 1. Créez un alias git ‘git listfiles’ prend en entrée un id de commit et qui liste seulement les fichiers du commit

alias listfiles='git diff-tree --no-commit-id --name-only -r'

**Bonus**’

* En modifiant une seule classe (modifiez plusieurs lignes non consécutives), comment faire pour ne choisir que certaines modifications pour faire un premier commit ?

(A l’aide de la commande magique (git help) et de la commande usuellement utilisée pour ajouter les fichiers modifiés dans l’index vous devriez pouvoir vous en sortir ;)! )

git add -p

* Modifiez l'invite de commande git pour qu’elle affiche le statut courant en permanence

Ajouter cette ligne dans votre bash\_profile

export GIT\_PS1\_SHOWDIRTYSTATE=1 GIT\_PS1\_SHOWUPSTREAM=verbose

Astuce sous windows (remplacer %INSTALL\_GIT% par l’emplacement de votre installation git):

Créer le fichier %INSTALL\_GIT%/etc/bash\_profile

Ajouter cela à la fin du fichier suivant %INSTALL\_GIT%/etc/profile.d/aliases.sh

source %INSTALL\_GIT%/etc/bash\_profile

* Ajoutez la complétion des commandes maven dans un terminal (git bash sous windows)

Ajouter le fichier suivant src/test/resources/sh/maven\_bash\_completion.bash

dans %INSTALL\_GIT%/etc/maven\_bash\_completion.bash

Puis ajouter cette ligne à la fin de votre bash\_profile

. %INSTALL\_GIT%/etc/maven\_bash\_completion.bash

* Vous recherchez des métriques sur votre dépôt ? <https://github.com/github/git-sizer>
* Testez une interface graphique sympa :
  + Git up : [*http://gitup.co/*](http://gitup.co/)
  + Atlassian sourcetree : <https://fr.atlassian.com/software/sourcetree>
* Pour s'entraîner à Git : <https://learngitbranching.js.org/>
* Ajouter des emoji dans les messages de commit (en utilisant par exemple [*https://gitmoji.carloscuesta.me*](https://gitmoji.carloscuesta.me/))
* Tapez une commande qui compiler le code, et si tout est ok, qui va créer une pull request github

Cf : <https://git-scm.com/book/en/v2/GitHub-Scripting-GitHub>

Il est également possible d’utiliser des outils tels que hub : <https://hub.github.com/>

On effectue une modification de code sur une branche particulière “test\_01” qui a déjà été poussée sur le repo distant.

Puis on lance notre nouvelle commande :

compileAndSendPR

Cette nouvelle commande est un alias vers le script shell suivant :

src/test/resources/sh/compileAndSendPR.sh

Puis on peux lancer notre commande :

compileAndSendPR "titre du commit" "contenu de la pull request"

****

1. Annexe
   1. Configuration gradle pour un utilisateur d’artifactory

Fichier build.gradle :

maven {

url "${artifactory\_contextUrl}/${repo\_key\_resolve}"

credentials {

username = "${artifactory\_user}"

password = "${artifactory\_password}"

}

}

Fichier gradle.properties dans le projet ou .gradle dans le dossier de l’utilisateur

artifactory\_user=user

artifactory\_password=password ou encrypted password

artifactory\_contextUrl=http://server:port/artifactory

repo\_key\_resolve=libs-release

* 1. Trucs et astuces
* Première installation de Git ?

Lorsque l’on installe Git depuis le début, il faut penser à faire les configs du mail et name. Sinon au premier stash on a un message d'erreur :

git stash push -m "mon premier stash"

**\*** Please tell me who you are.

Run

git config --global user.email "you@example.com"

git config --global [user.name](http://user.name) "Your Name"

