Git tutorial

Door Marijn Stollenga (m.stollenga@gmail.com)

Git is een gedistribueerd source code management systeem, gemaakt door Linus Torvalds. Het is gedistribueert in tegenstelling tot bijvoorbeeld SVN en CVS. Wat dit in de praktijk betekend dat je geen centrale repository hebt waar alle code in zit, maar dat iedereen zelf de volledige branch bij zich heeft.

Indeling

"Oops ik was nog iets vergeten te committen" ***2***

Eerst wat links

- Ik raad aan eerst deze presentatie van Linus te bekijken, http://www.youtube.com/watch?v=4XpnKHJAok8 . Duurt ruim een uur en geeft je een idee van de voordelen van git.
- Hier is nog een presentatie, deze gaat wat praktischer in op Git: http://video.google.com/videoplay?docid=-1019966410726538802

- En hier is de volledige documentation, http://www.kernel.org/pub/software/scm/git/docs/user-manual.html

Ik geeft hier eerst een kleine inleiding en daarna uitleg over hoe je git daadwerkelijk kunt gebruiken:

Inleiding

Dit betekend als eerste dat je lokaal al je veranderingen kunt committen, zo vaak als je wilt. Dus geen commits meer uitstellen omdat je veranderingen meteen door iedereen te zien zijn of de code kunnen breken.

Een direct gevolg is ook dat je geen internet verbinding hoeft te hebben om te committen. Dit doe je immers lokaal.

Belangrijke eigenschappen

- Het principe van git is 'alles is een branch en alle branches zijn gelijk'; er is geen super-branch (of trunk) die in het centrum staat en waar alles heen moet.

- En alle branches bevatten hun volledige geschiedenis, dus die branch die je lokaal hebt is niet zomaar een checkout van een tijdelijke staat, het bevat de volledige geschiedenis van branch tot aan het begin van het project. Nu denk je misschien, dat wordt heel groot, maar git is heel efficient met ruimte (gemiddeld een factor 10 kleiner dan svn, volledige geschiedenis van linux-kernel bijvoorbeeld past in ~1,2 gig)!. Door de volledige geschiedenis aan een branch te koppelen is het mergen in Git vele malen beter dan SVN bijvoorbeeld. Linus Torvalds (maker van git) geeft hier een mooie uitleg van git (http://www.youtube.com/watch?v=4XpnKHJAok8 , ruim 1 uur) en legt daar onder andere uit dat je zonder de volledige geschiedenis gewoon geen goede beslissingen kunt maken wat betreft mergen.

Workflow

Door deze gedecentraliseerde manier van werken zijn hele andere werktypes mogelijk. Piet kan gaan werken aan zijn eigen functie, zonder dat hij daarvoor toegang moet hebben tot een of andere centrale server, want die is er gewoon niet). Stel dat die Piet dan die functie mooi heeft afgewerkt en getest, dan kan hij dat tegen Jan zeggen. Jan kan dan de veranderingen van Piet "pull-en". Dat betekend dat alle veranderingen van Jan lokaal bij piet worden opgeslagen (onder een aparte branch, zodat Jan zijn eigen branches enzo veilig kan houden). Dan kan Jan op zn gemak de veranderingen bekijken en testen, en als hij tevreden is merged hij de veranderingen.

Stel dat in deze situatie Jan aangesteld is tot maintainer, dan kan hij de nieuwe branch publiceren zodat andere coders de nieuwe code kunnen gebruiken.

Installeren

Voor Ubuntu en dergelijke:

```
apt-get   install   git-core   git-doc
```

Anders moet je de source downloaden van http://git.or.cz/, het pakketje uitpakken en vervolgens zoals gebruikelijk:

```
./configure
make
make  install
```

Dan heb je wel meteen de nieuwste versie :)

Nu moet je jezelf ook even introduceren aan Git (naja moet niet, maar het is voor iedereen handig als je naam bij je commit staat). Dit kun je het makkelijkst doen door de file .gitconfig aan te passen in je 'home' directory. Net zoals dat gaat met alle linux programma's staan in deze config files de instellingen die voor je hele account gelden.

```
emacs ~/.gitconfig
```

en voeg de volgende regels toe, of pas ze aan:

```
[user]
        name = Your Name Comes Here
        email = you@yourdomain.example.com
```

Wat je ook waarschijnlijk wilt is kleurtjes in de output van git. Dit staat standaard uit omdat veel mensen er een hekel aan hebben ofzo. Voer de volgende commando's uit:

```
git config  --global  color.diff auto
git config  --global  color.status auto
git config  --global  color.branch auto
```

Een project beginnen

Er zijn twee mogelijkheden om een project te beginnen (natuurlijk zijn er wel meer, maar het gaat om het idee;)

Mogelijkheid 1 (een nieuw project)

Om een volledig nieuw project te beginnen typ je:

```
git init
```

Dit maakt een mapje ".git" in de directory waarin je zit. In dit mapje wordt ALLES wat git gebruikt bewaard, er zijn geen stomme submapjes in alle directories die je hebt, alles wordt in dit mapje bewaard.
Nu wil je eerst aangeven welke bestanden niet meegenomen moeten worden. Dit wil je meteen in het begin doen zodat git geen onnodige bestanden gaat bijhouden. In source-code wil je bijvoorbeeld meestal geen *.o bestanden bijhouden, en zo zijn er meer voorbeelden.
De bestanden die je niet wilt bijhouden zet je in een apart bestand: .gitignore. Deze maak je aan en edit je als volgt:

```
emacs .gitignore
```

Daarin zet je bijvoorbeeld

```
*.o
*~
```

Vervolgens wil je (aangenomen dat je al bestanden in dit project hebt) een 'initiele commit' doen, dus alle bestanden die je hebt wil je committen zodat ze bijgehouden worden. Dit doe je als volgt:

```
git add .
git commit -m "Hier komt de commit message"
```

Het add commando zorgt ervoor dat de bestanden die je meegeeft (in dit geval ".", ook wel de volledige inhoud van de huidige map) worden meegenomen in de commit die je gaat doen. Het commit commando doet de daadwerkelijke commit met een bepaalde message. Als je het argument -m "blabla" niet meegeeft wordt een texteditor geopend waarin je je message als nog in kunt zetten.
In de .git map zit de file "description" waarin je even een beschrijving van de repository kunt zetten, ik heb ervaren dat dit op een bepaald moment inhoud MOEST bevatten, dus vandaar dat ik het even zeg. Deze file kun je gewoon aanpassen met

```
emacs .git/description
```

Mogelijkheid 2 (klonen van een bestaand project)

Dit is voor ons de way to go, omdat we aan hetzelfde project willen werken. Als we allemaal ons eigen project zoals hierboven aanmaken, dan weet ik niet hoe we die projecten aan elkaar moeten plakken (zou vast wel kunnen, maar wordt ingewikkeld).
Een project klonen doe je zo:

```
git clone  <te klonen project>   <map waar het project in moet>
```

Ik heb een start project gemaakt zodat we allemaal hetzelfde start punt hebben. Die kloon je als volgt:

```
git clone http://www.ai.rug.nl/~mstollen/littlegreenbats.git <mapnaam,
bijv 'littlegreenbats'>
```

Visualisering

Je kunt meteen een mooie visualisering van de geschiedenis opvragen met

```
gitk
```

Gitk laat een boom zien van de geschiedenis, dit commando zul je vaak gebruiken om te zien wat er zoal is veranderd.

Beetje veranderingen maken en committen

Nu hebben we een project waarin we kunnen gaan veranderen. Dus
Beetje editen in humanoidbat.cc
Stel dat we deze verandering willen committen. We hebben een bestaand bestand committen. We moeten eigenlijk typen:

```
git add humanoidbat.cc
```

Om de veranderingen van het bestand te 'registreren' of hoe je het wilt noemen, en ze vervolgens committen.
Het kan echter ook sneller met een argument aan het commit commando:

```
git commit -a -m "blabla"
```

De toevoeging van het argument "-a" zorgt ervoor dat alle veranderingen in alle bestanden meegenomen worden, behalve de bestanden die nieuw zijn. Bestanden die nieuw zijn moeten eerst met ‘git add’ worden toegevoegd.
Als je niet wilt dat alle veranderingen worden gecommit, moet je geen "-a" gebruiken en alle bestanden die je wilt meenemen handmatig toevoegen, en ze vervolgens committen:

```
git add <bestand>
git commit -m <message>
```

Stel dat we ook een nieuw bestand hebt aangemaakt, "blabla.txt". Dit bestand wordt dus niet automatisch meegenomen door het "-a" argument, we moeten ook hier dus eerst git-add gebruiken.

```
git-add blabla.txt
```

Nu wordt het bestand blabla.txt ook meegenomen in de eerst volgende commit.

Branches

Nu kun je veranderingen maken en deze committen. Dan is het tijd om branching te leren. In Git is alles branches (niet een rare indeling van trunk en branches en tags), je project begint dan ook met de branch 'master'. Stel je wilt werken aan een nieuwe functie x, dan is het aan te bevelen om dit te doen in een aparte branch. Het is namelijk veel makkelijker om nieuwe functionaliteit onafhankelijk van elkaar te ontwikkelen en ze samen te voegen wanneer ze er klaar voor zijn.
Een branch maak je als volgt:

```
git branch functie_x
```

Aannemende dat je in de branch 'master' zat, is er nu een branch 'functie_x' gemaakt die de inhoud van 'master' heeft. Je kunt je branches bekijken met:

```
git branch
```

Die bij mij de output geeft:

```
* master
  functie_x
```

De nieuwe branch check je uit door:

```
git checkout functie_x
```

git branch geeft nu de output:

```
  master
* functie_x
```

En je commits gaan nu dus naar de functie_x

LET OP !
Bij het uitchecken van een (nieuwe) branch kun je problemen krijgen wanneer je veranderingen hebt gemaakt in de huidige branch, die nog niet gecommit zijn. Git wil deze veranderingen natuurlijk niet zomaar weggooien en zal ze proberen toe te passen in de nieuwe checkout. Wanneer hierdoor echter conflicten ontstaan weigert git te wisselen van branch. Je kunt dan drie dingen doen:

Je veranderingen eerst committen, zoals eerder uitgelegd, en dan pas de checkout doen
Je veranderingen terugzetten tot de laatste commit, waardoor je veranderingen dus verdwijnen, en dan de checkout doen:

git reset --hard (in sommige gevallen moet je ook "git clean" doen)

Je veranderingen "stash-en" waardoor je veranderingen opgeslagen worden (maar niet gecommit), je checkout en je ding doen, en vervolgens weer de veranderingen toepassen:

git stash

of

git stash "<even uitleg van je huidige veranderingen>"

Nu heb je weer een 'schone' branch en kun je je checkout doen. Wanneer je je gestash-de veranderingen weer wilt toepassen doe je:

git stash apply

Je kunt een lijst opvragen van gestash-de veranderingen

git stash list

En ook kiezen welke stash je wilt toepassen door bepaalde argumenten (die ik zo niet weet).

Mergen

Het maken van branches heeft natuurlijk geen zin als je niet kunt mergen. Laat git hier nou net heel goed in zijn. In git heeft elke branch ook kennis van zijn volledige geschiedenis, dit in tegenstelling tot CVS en SVN. Hierdoor kan git zinnige beslissingen maken die SVN nooit zou kunnen maken; renamen en deleten van files, veranderingen van dezelfde files, het werkt vrijwel altijd zoals je wilt. Git zal geen conflicts geven wanneer ze niet nodig zijn en niet gaan mergen als er een conflict is.

Zo kun je mergen, aangenomen dat je in de branch zit waar je naartoe wilt mergen:

```
git merge   <te mergen branch>
```

Als er eventuele conflicts ontstaan moet je deze oplossen. Je kunt de status van git bekijken om de conflicterende bestanden te zien:

```
git status
```

Er staat een 'c' voor de bestanden die een conflict hebben (ze hebben ook een rode kleur als je kleur hebt aangezet in je output zoals eerder is aangegeven).

Verander de bestanden zodat ze weer kloppen (in de bestanden staan de verschillen aangegeven met >>>>'s en <<<<<'s, ook wel conflict markers) en verwijder dus de regels die verwijderd moeten worden.

Voeg de veranderingen aan de volgende commit toe d.m.v. "git add" of doe dit automatisch door "-a" toe te voegen aan het commit commando (er zijn genoeg situaties waarin je niet alle veranderingen wilt committen, dan moet je dus handmatig 'git add' aanroepen en geen '-a' aan commit meegeven).

Wanneer alle veranderingen opgelost zijn kun je ze committen zoals je dat gewoonlijk zou doen.

```
git commit   <eventueel -a en andere argumenten>
```

Fetching

Fetching is het ophalen van een branch uit een andere repository (ook wel remote repository). Je wilt bijvoorbeeld de nieuwste versie van de littlegreenbats code bijhouden. Ik neem aan dat je de clone-methode hebt gebruikt om je project te starten.
Tijdens het clonen zijn de gegevens van de repository die je kloonde onthouden onder de naam 'origin', omdat dit de oorsprong is. Ook is de standaard repository waarvan je fetch't ingesteld op 'origin'. Dit is handig want om de nieuwe code op te halen hoef je alleen:

```
git fetch
```

aan te roepen. De nieuwe code wordt niet meteen in je huidige branch (bijvoorbeeld 'master') gestopt, daar kan immers allemaal code in zitten waaraan jij werkt. De nieuwe staat wordt onder de naam 'remote/origin/<branchnaam> gestopt. De master branch (als die bestaat) van origin heet dus 'remote/origin/master'. Als je na het fetch-en de veranderingen wil mergen van de branch 'master' van 'origin' kun je simpelweg typen:

```
git merge remote/origin/master
```

Omdat de combinatie 'fetch' - 'merge' vaak voorkomt kun je dit tegelijk doen met git-pull:

```
git pull
```

Dit fetch't EN merged de branch van origin.

Je kunt zelf remotes toevoegen, instellen van welke remote je standaard wil fetchen en instellen welke branch van deze remote je automatisch wilt mergen met 'git pull'. Dit doe je met git-config commando's die ik nu niet zal uitleggen maar het staat allemaal in de git-documentatie. Het toevoegen van een remote kan ook met een commando dat ik wel zal uitleggen:

git remote add <de referentie-naam die je gebruikt> <url naar de remote>

Hierna kun je naar de url verwijzen d.m.v. de referentienaam. Zo heb ik mijn usb-stick 'USB' genoemd en die heeft de url '/Volumes/USB_STICK/littlegreenbats'.

Rebase

Wanneer je gaat mergen mergen breng je eigenlijk twee branches, die ooit zijn gesplitst, weer samen:

```
           /--branch2--\ -------branch2 (leeft vrolijk verder)
--(splitsen)          (merge) >------branch1
           \--branch1--/
```

Nu is het goed mogelijk dat je dit helemaal niet wilt. Als een functie die je aan het maken was nog niet klaar is en je dus nog niet wilt mergen, maar je wilt wel de nieuwste code van branch2, dan gebruik je rebase:

git rebase branch2

Hier neem ik aan dat je in branch1 zit. Wat dit commando doet is de veranderingen op branch2 sinds de splitsing, toepassen TUSSEN het punt van de splitsing en het begin van branch1:

```
           /--branch2--\ -------branch2 (leeft vrolijk verder)
--(splitsen)          (rebase)>--branch1>-------branch1
```

Op deze manier blijft de geschiedenis van branch1 netjes met uiteindelijk 1 splitsing en 1 punt van samenkomen van de twee branches, in plaatst van meerdere knooppunten door merges, die eigenlijk nergens voor nodig zijn.

Je repository op een USB-stick

Het is heel handig om je repository te backuppen op een USB-stick. Het is ten eerste veilig, mocht je harde schijf crashen of iets dergelijks.
Ten tweede kun je je repository overal mee naartoe nemen:
- Je steekt de usb-stick in een pc die git heeft
Je kloont vanaf je usb stick naar een lokale repository met ‘git clone’
Je veranderd hier een daar wat, en je commit je veranderingen
Vervolgens doe je een ‘git push’ om je veranderingen naar de usb te gooien, omdat je hebt gekloond vanaf de usb is dit de standaard plek waar naartoe wordt gepush-ed.
Je haalt de usb-stick er weer uit en gaat vrolijk verder

Je moet een paar dingen doen om je repository op de usb-stick te krijgen. Maar eerst moet je een regel leren: een repository waar je naar push-ed (als backup zoals bij de usb-stick, of als publish methode zodat iedereen je veranderingen kan zien) mag je nooit gebruiken als een repository waar je zelf in gaat veranderen. Je mag er ALLEEN NAAR PUSH-EN! Dit is heel belangrijk omdat pushen niet gemaakt is om te mergen, dat hoort alleen te gebeuren als iemand pull-ed van een ander.

Dan kunnen we nu een repository clonen. Omdat er niet in de usb-repository veranderd gaat worden, heb je alleen een zogenaamde ‘bare’ repository nodig. Dit is als het ware alleen het .git mapje, en verder niks. Ik neem even aan dat je USB-stick gemount is onder ‘/media/USB_STICK/’ en je repository in het mapje ‘~/littlegreenbats/’ zit. Dan clone je als volgt:

```
git clone --bare ~/littlegreenbats   /media/USB_STICK/
littlegreenbats.git
```

Omdat je alleen een ‘bare’ repository wilt hebben, noem je de map bij conventie <repository naam>.git

Nu moet je eerst even wat invullen in de description file:

```
emacs /media/USB_STICK/littlegreenbats.git/description
```

Typ hier gewoon een korte uitleg van je repository in, als je dit niet doet kan git gaan zeuren bij het push-en.

Nu staat de repository op je usb stick, maar er moeten nog een paar dingen gebeuren zodat je straks kunt pushen. Eerst moet je update-server-info uitvoeren, zodat bepaalde dingen (weet niet wat) upgedate worden en je repository gesynchroniseerd wordt:

```
cd /media/USB_STICK/littlegreenbats.git
git --bare update-server-info
```

Dit voert het commando uit. Nu moeten we er nog voor zorgen dat update-server-info iedere keer wordt aangeroepen na een push. Hier is een scriptje voor dat je even aan moet zetten:

```
cd /media/USB_STICK/littlegreenbats.git
chmod +x hooks/post-update
```

Dit activeert het scriptje dat update-server-info uitvoert. Nu is de usb-stick klaar. Je wilt de usb nog even toevoegen aan je remotes zodat je je usb makkelijk kan aanspreken:

```
git config remote.USB.url   /media/USB_STICK/littlegreenbats.git
```

Dit voegt de remote met de naam ‘USB’ toe (je kunt hier elke naam invullen die je makkelijk vindt) die een url heeft die naar de repository op de usb-stick verwijst.

Stel je hebt nu wat veranderingen gecomitted, dan kun je typen

```
git push USB
```

En je veranderingen staan op je usb.

Je repository delen

Het delen van je repository is natuurlijk heel belangrijk. Anders kunnen andere coders geen regel code van jou te zien krijgen. Je kunt in principe een webserver opzetten waar je je repository naar push-ed. Maar ik leg hier alleen de makkelijkste manier uit om je code te delen, namelijk hoe je je code met een directe verbinding met iemand anders deelt:

Stel je wil je code met iemand anders delen omdat je iets gaafs hebt gemaakt. Je hebt bijvoorbeeld een msn-gesprek o.i.d. met diegene en je verteld hem dat hij je nieuwe code moet fetchen.

Nu moeten er een paar dingen gebeuren:

Een bare repository maken

Om je code te delen heb je een bare-repository nodig, net zoals uitgelegd onder de kop ‘zet je repository op een usb-stick’. Gelukkig hoeft je dit maar één keer te doen:
Eerst clone je je huidige repository naar een handige map, bijvoorbeeld:

```
git clone --bare   ~/littlegreenbats   ~/delen/littlegreenbats.git
cd ~/delen/littlegreenbats.git
git --bare update-server-info
chmod +x hooks/post-update
```

Je hebt nu een kant en klare bare repository onder ~/delen/littlegreenbats.git. Ga nu terug naar je normale repository en stel een remote in zodat je deze deel-repository makkelijk up-to-date kunt houden:

```
cd ~/littlegreenbats
git config remote.deel.url   ~/delen/littlegreenbats.git
```

Je hebt nu een snelle manier om je deel-repository aan te roepen. De volgende keer dat je wilt delen staat alles klaar en hoef je alleen het volgende commando te geven om de deel-repository up te daten:

```
git push deel
```

De git daemon draaien

Jij gaat nu de ‘git-daemon’ draaien, een simpel programma dat andere mensen de mogelijkheid geeft om van jou een repository te downloaden:

```
git-daemon --reuseaddr --verbose --base-path="~/delen/" --export-all
--  ~/delen/littlegreenbats.git
```

Zoals je ziet worden er een paar argumenten meegegeven:

--reuseaddr : die zorgt ervoor dat een verbinding eventueel hergebruikt wordt, dit voorkomt problemen als de verbinding op de een of andere manier al open staat.
--verbose : laat alles zien wat er gebeurd, dit is handig om te zien of de ander überhaupt verbinding maakt en laat eventuele andere errors zien.
--base-path=<pad> : Deze path zorgt ervoor dat de toegang tot git-repositories beperkt blijft. Het stelt eigenlijk het mount-point in, zodat de andere gebruiken niet de volledige path-name hoeft te weten.
--export-all : Deze optie zorgt ervoor dat je niet handmatig hoeft in te stellen welke repositories gedeeld mogen worden. Eigenlijk moet je eerst het bestand ‘~/delen/littlegreenbats.git/git-export-daemon-of’ moeten aanmaken om hem te delen.
‘-- ~/delen/littlegreenbats.git’ : achter de ‘--’ kun je de te delen repositories instellen, dit is niet nodig maar het beperkt de andere gebruiker alleen tot die repository. Het is dus iets veiliger om wel te doen.

Wat de ander moet doen

Nu kan de ander van jou gaan fetchen. Daarvoor moet hij wel eerst je ip-adres weten natuurlijk. En het is ook handig om je router goed in te stellen mocht je daar achter zitten (dat heb ik ervaren, je moet de port die git gebruikt, 9418, doorverwijzen naar je lokale ip-adres).
De ander moet eerst jouw deel repository als remote opgeven, zodat hij in het vervolg geen ip-adressen hoeft te vragen of onthouden:

```
git config remote.marijn.url   git://123.123.123.123/littlegreenbats.git
```

Dit voegt onder de naam ‘marijn’ de url ‘git://123.123.123.123/littlegreenbats.git’ toe. Nu is hij klaar om te fetch-en:

```
git fetch marijn
```

Als het lukt om verbinding te maken geeft de git-daemon output over dat er verbinding is gemaakt met een ip-adres erbij. De bestanden die nodig zijn worden gedownload naar de ander en komen onder de naam ‘refs/remote/marijn/<branch>’ te staan. Hij kan jouw branches ook bekijken door te typen:

```
git ls-remote marijn
```

En als hij bijvoorbeeld jouw master branch wil mergen met de branch waar hij op dat moment in zit moet hij typen:

```
git merge refs/remotes/marijn/master
```

Git-daemon afsluiten

Nu kun je de git-daemon gewoon weer afsluiten met een <ctrl>-<c> commando. En de andere persoon is weer helemaal up-to-date met jou code en kan jou nieuwe functionaliteit gebruiken.

Handige dingen

Git log

Laat me alle commits zien die een regel x hebben toegevoegd dan wel weggehaald:

```
git log -S"<regel x>"
```

Signature achterlaten

Door "-s" toe te voegen aan het commit commando laat je je naam en email achter in de commit:

```
commit -s <en eventuele andere argumenten>
```

"Oops ik was nog iets vergeten te committen"

Je kunt aan een vorige commit toevoegen. Er wordt dan dus niet een nieuwe commit toegevoegd, maar de vorige commit krijgt er dan wat bestanden bij.

```
commit --amend
```

LET OP! Doe dit alleen als je niks hebt gemerged of gepubliseerd van deze commit. Het veranderd namelijk de history van de repository (een commit verdwijnt) en dat zorgt voor problemen als ergens ander die commit nog wel bestaat.