Jack-Benny Persson
LX13
Linux Fortsättningskurs
2013-11-10

Fördjupningsarbete/laboration om Git

Av: Jack-Benny Persson
jack-benny@cyberinfo.se

Innehåll

Jack-Benny Persson
LX13
Linux Fortsättningskurs
2013-11-10

Syfte

Lära mig väsentliga grunder i git och dess användningsområden och historia. Efter genomförd laboration klara av att hantera en git repo med commits, grenar m.m., både lokalt med enskild användare och centraliserat med flera användare.

Tillvägagångssätt

Denna laboration utförs enskilt och med hjälp av onlineresurser som git-scm.com, stackexchange.com, diverse guider samt med boken Pro Git.

Vad är git och versionshantering?

Versionshantering, version control, revision control, source code control... Kärt barn har många namn men syftar alla på i princip samma sak, ett system för att hålla reda på ändringar genom tiden i källkod. Det finns en rad andra versionshanteringsprogram än git, så som Subversion, CVS, Perforce m.fl. Mer om skillnaden mellan dessa och git senare. Hela konceptet med versionshantering handlar om att kunna ändra tillbaks till tidigare ändringar i en fil, se vad som ändrats sedan X dagar/veckor sedan, jämföra nuvarande version med tidigare versioner osv. Versionshantering illustreras här nedan i *Illustration 1: versionshantering*. Här har vi vår fil, fil.sh, som finns i tre versioner. Den nuvarande versionen, version 3, och filen, fil.sh är här i samma stadium. Version 2 och version 1 är tidigare versioner av samma fil, alltså fil.sh. Låt oss nu ta som exempel att något inte längre fungerar som det var tänkt i version 3 och denna versionen nu är att anse som buggig. Då kan man enkelt antingen backa tillbaks helt och hållet till version 2, eller jämföra version 2 och version 3 för att hitta felet, alternativt backa tillbaks till version 2, skriva om en ny utgåva utav version 3 och sedan göra denna till version 3.0.1 (en gammal praxis när man arbetar med versioner är att använda första siffra till vänster, alltså heltalet, till stora förändringar eller milstolpar i programmet/koden, andra siffran till större förändringar och sista/tredje siffran till buggfixar). Under arbetets gång i en git repo är det dock inte vanligt att man arbetar med versionsnummer utan man brukar istället skriva ett så kallat ”commit message” till varje ändring man gör i koden. Sedan när ändringen är ok att skeppas ut så taggar man den med en version. Mer om detta senare.

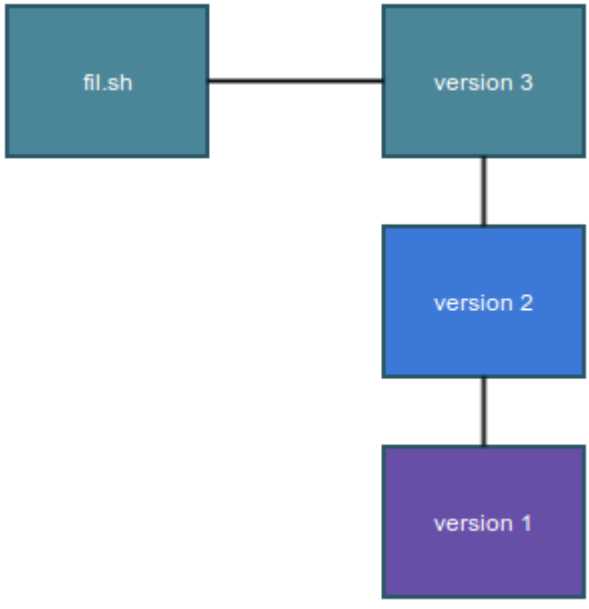


Illustration 1: versionshantering

Med git kan man alltså ångra sina ändringar om man råkar göra något dumt. I och med detta används git idag till en mängd andra områden än bara källkod. T.ex. är det många administratörer som idag använder git på servrar och workstations för att hålla reda på ändringar i /etc katalogen på Unixsystem. Git kan även användas till binära filer, så som bilder, även om git inte från början är skapt för just detta ändamål. Med just bilder kan man t.ex. inte så skillnader (diff) mellan versionerna, men man kan dock fortfarande ångra ändringar och plocka fram tidigare versioner av en och samma fil.

Git är ett Distributed Version Control System (DVCS)

Till skillnad från många andra verktyg för source/revision control så är git ett DVCS, d.v.s. ett distribuerat system. Andra verktyg är ofta antingen lokala (local only) eller serverbaserade (central server). Poängen med att det är distribuerat är att alla filer och all historik finns såväl på servern som på klienterna. Detta medför en hel del fördelar, bland annat att man kan arbeta offline med git repon. Man kan alltså se historik, göra ändringar, ”commita” ändringar m.m. helt offline. När man sedan får en uppkoppling kan man skicka iväg dessa till servern. En annan fördel är att det alltid finns kompletta backuper på flera olika platser. T.ex. om servern skulle krascha finns det kompletta kopior av hela projektet på alla klienterna. Likaså om någons klient skulle krascha kan han eller hon enkelt klona hela repon igen från servern (och skulle servern vara nere kan han eller hon klona repon från en annan klient). Illustration 2 nedan visar hur den kompletta repon finns på både servrar och klienter. Illustration 3 och 4 visar hur lokal respektive central serverbaserad versionskontroll fungerar. Från och med nu kommer vi kalla det VCS istället version/revision kontroll.

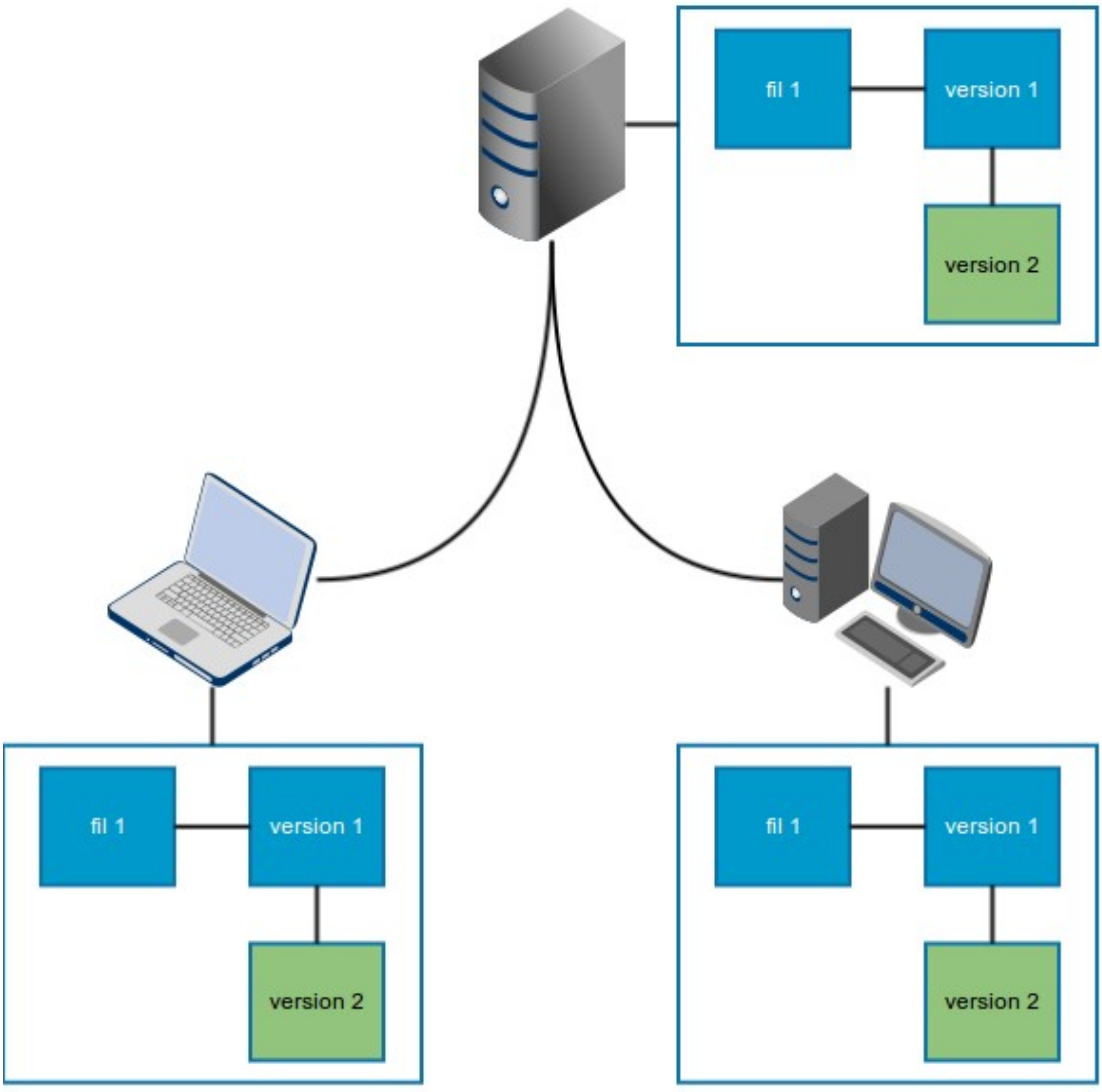


Illustration 2: Distribuerad VCS

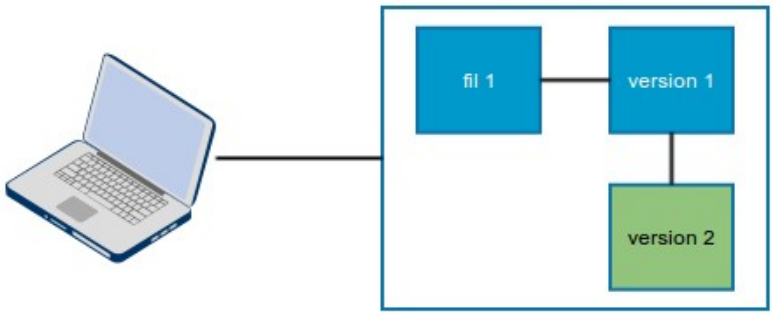


Illustration 3: Lokal VCS

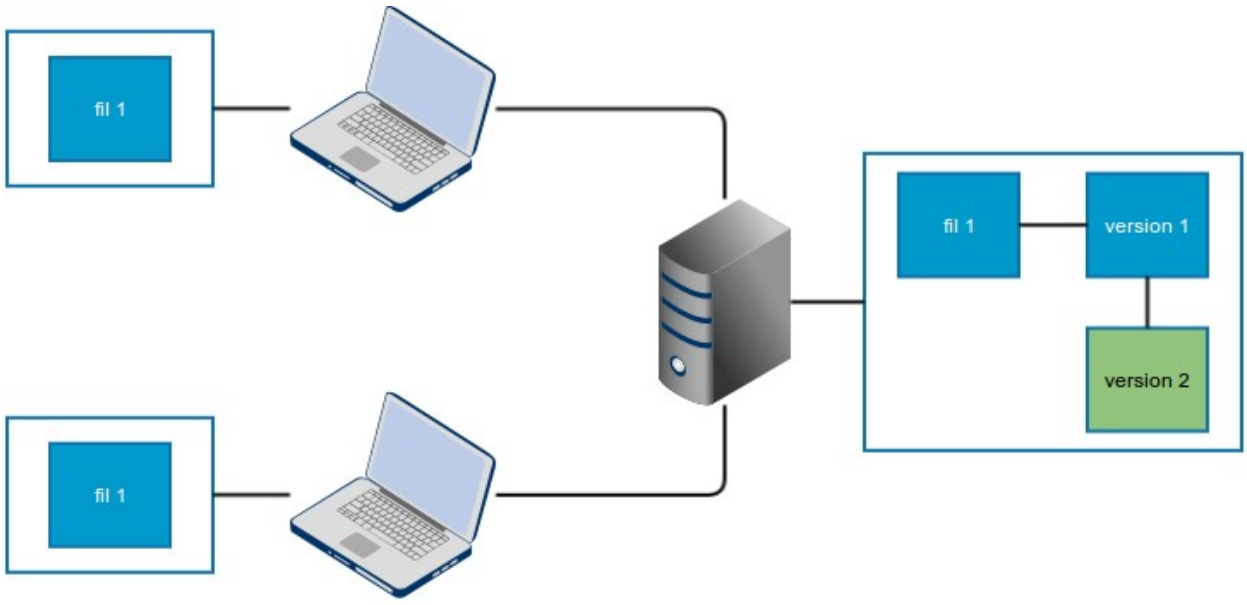


Illustration 4: Serverbaserad VCS

Hur lagrar git data?

Många VCS-system lagrar sin data som diffar (skillnader) mellan olika versioner i historiken. Git fungerar på ett lite annorlunda sätt än detta. Istället för att se det hela som skillnader kan man i princip se en git repo som ett litet mini-filsystem med snapshots. Varje gång man gör en commit i git så tar git en snapshot eller bild av hur alla ens filer ser ut vid just detta tillfälle. För att undvika att icke-ändrade filer tar upp en massa utrymme så länkas dessa bara till hur filen såg ut innan den ändrades (en commit eller 100 commits sedan spelar ingen roll). Detta gör git till ett mycket effektivt system.

Katalogstrukturen

När man listar filerna i en vanlig git repo ser det ut som vilken katalog med filer som helst. Med kommandot *ls -a* ser man dock en dold katalog som heter .git. Det i i denna katalog som git lagrar all data om historiken. Detta är komprimerade binärfiler och går inte att utläsa med vanliga textkommandon som cat/more/tail osv.

Stages

Det finns tre olika ”huvud stages” som en fil kan befinna sig i. Dessa är:

- Committed, som betyder att filen är incheckad i git och allt är frid och fröjd.
- Modified, som betyder att filen är modifierad men ej ännu commitad.
- Staged, som betyder att en fil är kommer att komma med i nästa commit. D.v.s. den är tillagd i git repon (med t.ex. git add) men den är ännu inte commitad.

Grundläggande användning av git

Nu när vi vet vad git är, hur det lagar sin data och vilka de tre huvud-stages:en är så kan vi börja arbeta med git. Vi skapar här först en helt vanlig katalog med några filer i. I detta exempel skapar vi en katalog som heter mygitproj och i denna katalog skapar vi först en fil som heter test.sh med följande innehåll som vi kommer att arbeta med.

```
#!/bin/bash

echo "This is my very first file"
echo "This is the second line of that file"

if [ $? -eq 0 ]; then
exit 0
fi
```

När vi nu står i katalogen mygitproj kan vi skapa vår git repo utav denna katalogen med kommandot *git init.* Om allt gick vägen bör du få ett svar i stil med ”Initialized empty Git repository in /home/user/mygitproj/.git/”. Testa nu att skriva *ls -a* och du bör se följande filer.

```
.  ..  .git  test.sh
```

test.sh är här vår fil vi skapade för exemplet och .git är själva git repon. Om vi nu skriver *git status* kommer vi att få följande meddelande.

```
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#      test.sh
```

Här ser vi alltså att git ser vår fil test.sh men att den är untracked, d.v.s att git inte på något sätt ansvarar för denna fil ännu. Vi kan nu lägga till filen i vår git repo med antingen *gid add test.sh* för att lägga till vår fil eller *git add -A* för att lägga till alla filer i katalogen.

Efter att vi kört *git add -A* eller *git add test.sh* kan vi köra kommandot *git status* igen. Vi får då ett nytt meddelande.

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#      new file:   test.sh
#
```

Här ser vi att git nu anger filen som ”new file”. Filen är alltså nu ”staged”, men ännu inte ”commited”. Vi kan nu raskt gå vidare till att commita med kommandot *git commit -m ”Vår första commit”*. Om man inte anger växeln *-m* får man upp sin editor istället för att skriva in sitt

commit-message.

När vi nu har commitat allting skriver vi återigen *git status* för att kontrollera vad git säger.

```
# On branch master
nothing to commit, working directory clean
```

Nu är vårt ”working directory”, eller på svenska vår arbetskatalog ren. Working directory är den katalog vi arbetar med våra filer i, alltså inte själva .git katalogen då.

Om vi nu ändrar i filen test.txt till att istället innehålla följande

```
#!/bin/bash

echo "This is my very first file"
printf "This is the second line of that file\n"

if [ $? -eq 0 ]; then
exit 0
fi
```

Vi har alltså ändrat echo till printf på fjärde raden samt även lagt till ett \n på samma rad i slutet av raden. Om vi nu skriver *git status* så kommer vi ett få ett meddelande i stil med detta

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   test.sh
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Vi ser nu att filen test.sh är ”modified”, ett av de tre huvud-stages som vi gick igenom tidigare. Vill vi nu se vad som ändrats i denna filen innan vi återigen lägger till filen i vårt ”staging-area” och commitar den använder vi kommandot *git diff* som kommer att svara med något i stil med

```
diff --git a/test.sh b/test.sh
index 89cc5e9..eac3299 100755
--- a/test.sh
+++ b/test.sh
@@ -1,7 +1,7 @@
 #!/bin/bash

 echo "This is my very first file"
-echo "This is the second line of that file"
+printf "This is the second line of that file\n"

 if [ $? -eq 0 ]; then
 exit 0
```

Vi kan nu återigen skriva *git add -A* eller *git add test.sh* för att lägga till filen i vår staging-area. Om vi därefter skriver *git status* igen så ser vi återigen att filen är i stadiet modified. Om vi nu ändrar i filen test.sh igen innan vi har commitat så kommer kommer vi ha två styck modified test.sh. En som är modified i vår staging-area men ytterligare en som är modified men ännu inte befinner sig i staging-arean. Vi testar nu med att lägga till en rad (echo "OMG a third line...?") direkt under printf raden i test.sh. Om vi nu kör *git status* kommer vi få meddelandet

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   test.sh
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   test.sh
#
```

Här ser vi alltså att vi först har en test.sh fil som är modifierad och är på väg att bli commitad (Changes to be committed), men att vi också har en test.sh som ännu inte är tillagd i vår staging-area (Changes not staged for commit). Om vi nu commitar med *git commit -m ”Andra commiten”* och därefter kör *git status* kommer vi att se att filen som fanns under ”Changed to be commited” nu är borta men filen som inte fanns i vår staging-area är fortfarande kvar och står som ”Changes not staged for commit”.

Vi kan nu gå vidare med att lägga till även denna fil samt commita den med *git add -A* och sedan *git commit -m ”Tredje commiten”*. Här kan kan vi faktiskt använda en genväg, istället för de båda kommandona kan man använda *git commit -a -m ”Tredje commiten”* som både lägger till filen och commitar den på samma gång.

Ångra sig i git

En av de största fördelarna med att ha sitt projekt i en git (förutom samarbete vilket vi kommer till senare) är att kunna ångra sig. De finns flera olika sätt att ångra sig i git men det sätt vi kommer att använda här är *git checkout* vilket innebär att vi hämtar ut en tidigare commit från git repon till vår arbetskatalog och lämnar historik och staging-area orörd.

Låt oss ta vårt exempel från ovan med test.sh. Vi ändrar i denna fil och kommer senare på att vi vill ha tillbaks filen så som den var från början, innan vi började redigera. D.v.s. så som filen såg ut i den senaste commiten. Vi lägger till en rad i filen test.sh och kör *git diff* i vår katalog.

```
diff --git a/test.sh b/test.sh
index eac3299..a2865b6 100755
--- a/test.sh
+++ b/test.sh
@@ -2,6 +2,7 @@

 echo "This is my very first file"
 printf "This is the second line of that file\n"
+printf "I don't really want this line at all....\n"

 if [ $? -eq 0 ]; then
 exit 0
```

Vi kan nu enkelt få tillbaks filen från git repon till vår arbetskatalog med kommandot *git checkout -- test.sh.* Om vi nu kollar i filen test.sh kommer den vara återställd igen, d.v.s. sakna den sista raden vi lade till. Kommandot *git checkout* gör precis som det låter, checkar ut en fil från repon. Det dubbel-bindestrecken talar om att vi vill check ut filen från den senaste commiten. Istället för två bindestreck kan man här istället check ut en fil från en tidigare commit, och då här ange checksumman för den commiten. Vi testar. Först kör vi en *git log* så att vi ser checksummorna

för våra commits. När man ska ange checksummor behöver man bara ange de första tecknen i summan för att slippa ange alla tecknen. T.ex. 5 tecken fungerar bra. Nedan visas utdata från *git log*.

```
commit 46ac775441ce21ac21ba728b89d7251690ec4a55
Author: Jack-Benny Persson <jack-benny@cyberinfo.se>
Date:   Fri Nov 22 13:06:44 2013 +0100

    Tredje commiten

commit 81fbb618535eb2aab68cd145ef090702d1a496a8
Author: Jack-Benny Persson <jack-benny@cyberinfo.se>
Date:   Fri Nov 15 20:26:09 2013 +0100

    Andra commiten

commit 8d9e9ba37d3909306443c9118b102c107d787bad
Author: Jack-Benny Persson <jack-benny@cyberinfo.se>
Date:   Fri Nov 15 19:59:40 2013 +0100

    Vår första commit
```

Om vi nu vill återställa filen test.sh till commiten som heter ”Vår första commit” anger vi kommandot *git checkout 8d9e9 test.sh.* Om vi nu kollar i filen test.sh ser vi att den är återställd till hur den såg ut vid den första commiten.

Vill man istället återställa hela sin arbetskatalog till en viss commit kan man hoppa över att ange filnamnet och bara ange checksumman, t.ex. *git checkout 8d9e9.* Vi får då också ett annorlunda meddelande mot när vi endast checkade ut en enskild fil. Utdatan från git blir nu istället i stil med

```
Note: checking out '8d9e9'.

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

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 8d9e9ba... Vår första commit
```

Vad som händer här är att vi checkar ut den tidigare commiten ”Vår första commit” och git placerar oss då i något som kallas för ”detached HEAD state”, vilket innebär att vi inte längre befinner oss i grenen master. Vi kan verifiera detta med med kommandot *git status* som kommer att svara med ”Not currently on any branch”. Vitsen med detta är kan man säga är att man inte ska råka förstöra något medan man testar och experimenterar med en tidigare commit.

Ändra ett commit-message

Skulle vi vilja ändra vårt senaste commit-message så kan vi göra detta väldigt enkelt genom att skriva *git commit --amend.* Man får då upp sin editor och kan redigera det senaste commit-meddelandet (översta raden i editorn). Spara och avsluta editorn för att spara commit-meddelandet.

Vill man redigera ett commit-meddelande som är längre tillbaks i historiken blir det några kommandon till men det är fortfarande relativt enkelt.

Grenar och konflikter

Om vi nu ändrar filen test.sh medan vi befinner oss i detached head kan vi då lägga in detta i en ny gren som vi döper till test_fran_forsta_commit med kommandot
git checkout -b test_fran_forsta_commit. Flaggan -b står för branch. Vi har nu två grenar i vår git repo och om vi vill se dessa grenar skriver vi *git branch.* Vi kan nu commita vår ändringar medan vi står i grenen test_fran_forsta_commit med *git commit -a -m ”Test från Vår första commit i detached state”.* Om vi nu skulle vilja få med oss dessa ändringar till grenen master får man göra en så kallad merge. Vi kan testa detta genom att första byta till grenen master igen med *git checkout master.* Om vi kollar i filen test.sh ser vi att den åter är i det skede den var när vi var på vår tredje commit i grenen master. Vi kan nu testa att merga med *git merge test_fran_forsta_commit.*. Troligtvis kommer nu git att varna för att vi har en konflikt. Vi kan nu öppna upp test.sh i vår editor och fixa till konflikten. Filen i mitt exempel ser ut enligt följande nu:

```
#!/bin/bash

echo "This is my very first file"
<<<<<<< HEAD
printf "This is the second line of that file\n"
printf "Den trejde commiten\n"
=======
echo "This is the second line of that file"
echo "The new line from our detached state"
>>>>>>> test_fran_forsta_commit

if [ $? -eq 0 ]; then
exit 0
fi
```

Jag vill behålla ändringen jag gjorde som innehåller ”The new line from our detached state”. För att åstadkomma detta tar jag bort alla rader med <<<<< och ==== samt >>>> och då de rader från HEAD som jag inte vill ha. När jag är klar ser filen ut enligt följande

```
#!/bin/bash

echo "This is my very first file"
echo "This is the second line of that file"
echo "The new line from our detached state"

if [ $? -eq 0 ]; then
exit 0
fi
```

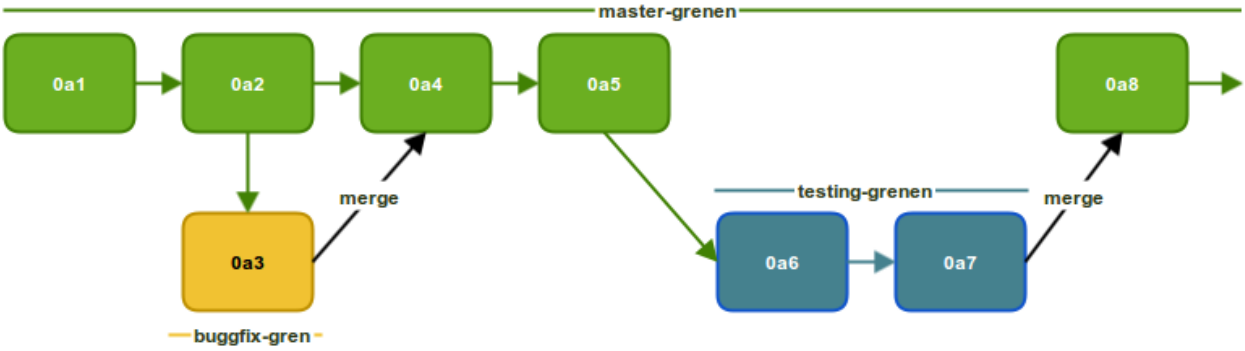
Nu markerar vi att vi har löst konflikten genom att lägga till filen till staging-area med
git add test.sh. Vi commitar sen som vanligt med *git commit -m ”Vår fjärde commit”.* Om ni nu kollar *git log* ser vi att vi alla commits finns här och kollar vi filen test.sh ser vi också att denna fil nu är precis så som vi gjorde den när vi löste konflikten.

Grenar i git är snabbt och enkelt att arbeta med och anses vara git's starka sida jämfört med många andra versionshanteringsprogram. Git är uppbyggd för att det ska gå snabbt att byta mellan olika grenar i projektet. Man kan därför skapa en gren för varje buggfix man gör och sedan mergea in i den i master när väl buggfixen är klar och fungerar felfritt. Man kan också fritt växla mellan olika grenar i projektet. Varje gång man växlar gren med *git checkout <gren>* ser man genast filerna i den grenen i sin arbetskatalog. Ett vanligt sätt att arbeta med grenar är att master-grenen används för

stabil kod, d.v.s väl testad kod som man vet fungerar. Sedan sker all utveckling på andra grener, t.ex. testing och development-grenar. Detta medför att master-grenen alltid ligger efter dessa andra grenar. När man står i en viss gren säger man att det är där HEAD är. HEAD är en fil i git-katalogen som innehåller sökvägen till den grenen man befinner sig i.

```
jake@olivia:~/lx13/Experimentfiler/mygitproj$ git branch testing
jake@olivia:~/lx13/Experimentfiler/mygitproj$ git checkout testing
Switched to branch 'testing'
jake@olivia:~/lx13/Experimentfiler/mygitproj$ vi test.sh
jake@olivia:~/lx13/Experimentfiler/mygitproj$ git add -A
jake@olivia:~/lx13/Experimentfiler/mygitproj$ git commit -m "Gör en commit på testing-grenen"
[testing bf2900a] Gör en commit på testing-grenen
 1 file changed, 1 insertion(+)
jake@olivia:~/lx13/Experimentfiler/mygitproj$ cat .git/HEAD
ref: refs/heads/testing
jake@olivia:~/lx13/Experimentfiler/mygitproj$
```

Här ovan ser vi att HEAD pekar på just den grenen vi står i för tillfället, nämligen testing. Att veta vad HEAD är kan vara viktigt då man ibland kan se meddelanden i git som t.ex. ”HEAD is behind branch”, Fast-forward HEAD” osv osv. Nedanstående flödesschema illustrerar hur man kan arbeta med grenar. I exemplet nedan så fortgår arbetet på master grenen (som här används för stabila utgåvor) fram tills 0a5 där den stabila utgåvan stannar av och utveckling fortgår i testing-grenen istället (för att inte sabba något i master). När testing-grenen är att anse som stabil så förs den samman med master igen och blir commit 0a8. Under tiden master-grenen fortsatte gjordes en snabb buggfix i commit 0a3 (gulmarkerad nedan) och denna fördes sedan snabbt in i a04 igen.



Taggar

Taggar används för att markera viktiga stadier i ett projekt, t.ex. nya versioner. Man taggar då alltså en version. På så sätt kan utvecklarna commita buggfixar och andra förbättringar på koden hur ofta de vill utan att de påverkar själva versionsnumret på projektet. Versionen ändras bara när man då bestämmer sig för att tagga en ny version med tagg-kommandot.

Värt att notera om taggar är att det finns två olika typer av taggar, lightweight och annotated, där den sistnämnda är den som används flitigast. För att skapa en annotated tagg skriver man *git tag -a v1.2 -m ”Version 1.2”* för att tagga en version som ”v1.2” och taggmeddelanden ”Version 1.2”. Tagg-meddelande är detsamma som ett commit-meddelande, en kommentar eller meddelande till just den commiten eller taggen. För att lista alla taggar i projekt används *git tag.* För att visa mer information om en tagg används *git show <tag>*.

```
jake@olivia:~/lx13/Experimentfiler/mygitproj$ git tag -a v1.1 -m "Version 1.1"
jake@olivia:~/lx13/Experimentfiler/mygitproj$ git tag
v1.0
v1.1
jake@olivia:~/lx13/Experimentfiler/mygitproj$ git show v1.1
tag v1.1
Tagger: Jack-Benny Persson <jack-benny@cyberinfo.se>
Date:   Tue Nov 26 20:24:11 2013 +0100

Version 1.1

commit bf2900a5f37de87d82379e7b20dbfe0a5413cdae
Author: Jack-Benny Persson <jack-benny@cyberinfo.se>
Date:   Tue Nov 26 19:36:21 2013 +0100

    Gör en commit på testing-grenen

diff --git a/test.sh b/test.sh
index 72656fe..201c8a8 100755
--- a/test.sh
+++ b/test.sh
@@ -3,6 +3,7 @@
 echo "This is my very first file"
 echo "This is the second line of that file"
 echo "The new line from our detached state"
+echo "Testing-grenen"

 if [ $? -eq 0 ]; then
 exit 0
jake@olivia:~/lx13/Experimentfiler/mygitproj$
```

När man arbetar med versionsnummer brukar man arbeta enligt följande mönster att första siffran anger stora förändringar i programmet, andra siffran mindre förbättringar/förändringar och tredje siffran buggfixar i koden. Exempelvis version 1.3.9 blir då att

1 = Stor versionsförändring, stabil version 1.

3 = En mindre förbättring/förändring, version 3.

9 = Buggfix nr. 9.

Dessa är inga fast bestämda regler men brukar användas flitigt av open-source projekt.

Sammanfattning

Vi har nu sett exempel på när en fil befinner sig i de tredje stadierna/stages modified, staged och commited. Vi har även skapat vår egna första lokala git repo. Kommandon vi har använt oss av visas i nedanstående lista.

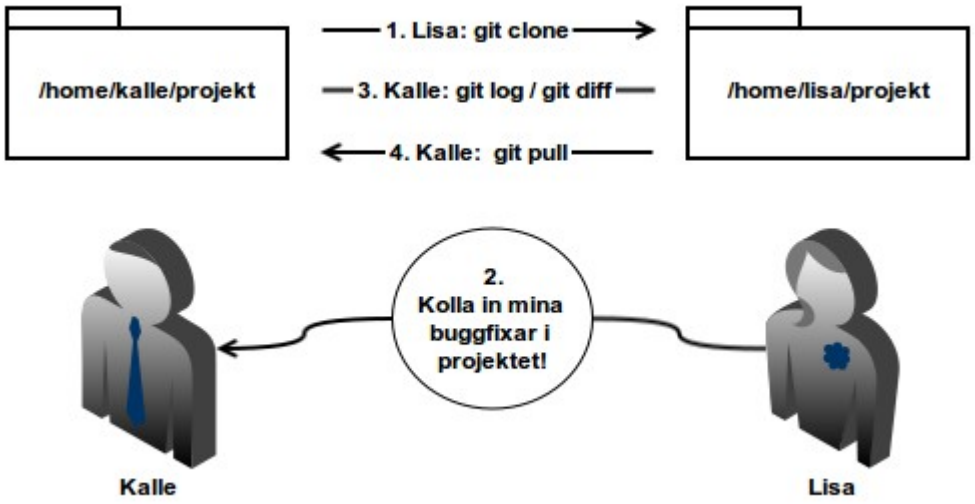
Git kommando	Betydelse
git init	Initialisera katalogen till en git repo
git status	Visas status i katalogen vi står i
git diff	Visa vad som ändrats
git add	Lägga till filer till staging-area
git commit	Commita en ändring
git checkout -- <filnamn>	Check ut en fil sedan senaste commit
git checkout <checksum> <filnamn>	Checka ut en fil från en tidigare commit
git checkout <checksum>	Checka ut en tidigare commit (alla filer)
git checkout -b <gren>	Skapa en ny gren med valfritt namn
git checkout <gren>	Växla till gren
git branch	Visa & skapa grenar
git tag	Skapa och visa taggar

Samarbete i git

Det finns flera olika sätt att samarbeta i git.

Lokalt på samma maskin

Det allra enklaste är att flera användare på samma maskin klonar och ”pullar” kod från varandras hemkataloger. Det enda som krävs för att detta ska fungera är att användarna behåller standardrättigheterna 644 för sin hemkatalog. Användarna kan då fritt klona kod från varandra, kolla på varandras git projekt och använda *git log, git diff* m.m. När en användare vill ta hem buggfixar från en annan användare gör han bara en *git pull* från den andra användaren git repo från dennes hemkatalog. Bilden nedan illustrerar hur detta kan gå till på enklaste möjligaste vis.



Kalle

```
kalle@olivia:~/projekt$ git add -A
kalle@olivia:~/projekt$ git commit -m ”Första commiten”
```

Lisa

```
lisa@olivia:~/$ git clone /home/kalle/projekt
Cloning into 'projekt'
lisa@olivia:~/projekt$ cd projekt
lisa@olivia:~/projekt$ vi test.sh
```

Kalle

```
kalle@olivia:~/projekt$ cd /home/lisa/projekt
kalle@olivia:/home/lisa/projekt$ git log
commit 0e8260ecfde5cc0e91cac01ea9bf05b5f5a567a4
Author: Lisa <lisa@example.com>
Date:   Wed Nov 20 20:06:29 2013 +0100

    Lagt till en rad

commit 88354b268d08121dcb8d298895d6f20986f2e87b
Author: Kalle <kalle@example.com>
Date:   Wed Nov 20 20:01:03 2013 +0100

    Försa commiten
kalle@olivia:/home/lisa/projekt$ cd ~/projekt
kalle@olivia:~/projekt$ git pull /home/lisa/projekt
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Fromt /home/lisa/projekt
 * branch             HEAD       → FETCH_HEAD
Updating 88354b2..0e8260e
Fast-forward
 test.sh | 1 +
 1 file changed, 1 insertion(+)
```

Nu har Kalle fört in Lisas buggfixar i sin egen repo.

Detta sätt att samarbeta kräver att båda användarna befinner sig på samma maskin, vilket inte alltid är fallet. Nästa enkla sätt att samarbeta på är över SSH.

Över SSH (alla har samma rättigheter)

Det enklaste sättet att samarbete mot en fjärrserver är över SSH med en gemensam användare på servern, t.ex. användaren git. De personer som sen ska ha access till git repon på servern skickar in sin SSH-nycklar till administratören av servern som då lägger in nycklarna i git-användarens authorized_keys.

Vi bestämmer oss för att den nya git-användaren ska heta just git och skapar denna på vårt system med *useradd -m -s /usr/bin/git-shell.* Vi anger skalet git-shell för att de användare som sedan får nycklar till systemet inte ska kunna logga in på systemet med ett vanligt shell. Om de försöker logga in när skalet är git-shell kommer de bara få meddelandet ”fatal: Interactive shell is not enabled” eller liknande. Det som krävs nu för att kunna skapa nya git repos är att root skapar en ny ”bare” git repo och sedan ändrar ägaren till git användaren.

Logga in som root på systemet eller som en vanlig användare och använd sedan *sudo -i* för att få root-rättigheter. Gå in i git användarens hemkatalog och skapa en katalog som heter projekt.git och gå in i denna också. Skriv sedan *git init --bare.* Att en repo är ”bare” betyder att den inte har någon arbetskatalog utan bara innehållet i .git (som här istället finns i roten). Backa sedan ur katalogen igen och ange *chmod -R git:git projekt.git* så att alla filerna i katalogen ägs av git-användaren igen.

```
jake@tiger:~$ sudo -i
[sudo] password for jake:
root@tiger:~# cd /home/git/
root@tiger:/home/git# ls
root@tiger:/home/git# mkdir projekt.git
root@tiger:/home/git# cd projekt.git/
root@tiger:/home/git/projekt.git# git init --bare
Initialized empty Git repository in /home/git/projekt.git/
root@tiger:/home/git/projekt.git# cd ..
root@tiger:/home/git# chown -R git:git projekt.git/
root@tiger:/home/git# ls -l
total 4
drwxr-xr-x 7 git git 4096 Nov 23 13:25 projekt.git
```

Nu ber vi användarna Lisa och Kalle att skapa varsin SSH-nyckel med *ssh-keygen* kommandot och sedan maila över sin publika nyckel till administratören för git-servern.

```
kalle@elektra-lab1:~$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/kalle/.ssh/id_rsa):
Created directory '/home/kalle/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/kalle/.ssh/id_rsa.
Your public key has been saved in /home/kalle/.ssh/id_rsa.pub.
The key fingerprint is:
75:34:7d:33:b9:8b:0c:79:4f:d7:f2:f5:78:86:e0:6f kalle@elektra-lab1
The key's randomart image is:
+--[ RSA 2048]----+
|            o.  .|
|           . ..+.|
|          . o  .=|
|         . +...o+|
|        S  .+.+==|
|            .ooo=|
|             . o |
|              E  |
|             .   |
+-----------------+
kalle@elektra-lab1:~$
```

Användaren Lisa gör likadant och de skickar sedan över sina nycklar till administratören av git-maskinen. Root-user lägger sedan till de båda nycklarna i /home/git/.ssh/authorized_keys

```
root@tiger:/home/jake# cat kalle-pub.key > /home/git/.ssh/authorized_keys
root@tiger:/home/jake# cat lisa-pub.key >> /home/git/.ssh/authorized_keys
root@tiger:/home/jake# cat /home/git/.ssh/authorized_keys
```

När allt detta är klart börjar Kalle att pusha kod till projekt.git på maskinen tiger.

```
kalle@elektra-lab1:~$ cd projekt/
kalle@elektra-lab1:~/projekt$ git remote add origin git@tiger:projekt.git
kalle@elektra-lab1:~/projekt$ git push origin master
Enter passphrase for key '/home/kalle/.ssh/id_rsa':
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 261 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@tiger:projekt.git
 * [new branch]      master -> master
```

I *git push* använde vi här origin och master, detta behövs bara första gången för att tala om att origin ska pushas till grenen master.

Nu kan Lisa klona projektet till sin egna dator.

```
lisa@elektra-lab1:~$ git clone git@tiger:projekt.git
Cloning into 'projekt'...
Enter passphrase for key '/home/lisa/.ssh/id_rsa':
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
lisa@elektra-lab1:~$ ls
examples.desktop  projekt
lisa@elektra-lab1:~$ cd projekt/
lisa@elektra-lab1:~/projekt$ git log
commit 364216dab5ffbe379b6345a824e42e63118ca8a3
Author: Kalle <kalle@example.com>
Date:   Sat Nov 23 13:56:08 2013 +0100

    Initial commit
```

Nu har Lisa alltså en egen lokal kopia av hela projektet på sin dator och kan göra ändringar i koden och commita till den. Eftersom alla användare i authorized_keys har fulla rättigheter kan nu Lisa även pusha sina ändringar till servern.

```
lisa@elektra-lab1:~/projekt$ git push
Enter passphrase for key '/home/lisa/.ssh/id_rsa':
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 318 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@tiger:projekt.git
   364216d..fcc0da9  master -> master
lisa@elektra-lab1:~/projekt$
```

Nu kan användaren Kalle använda *git pull* för att få ner alla ändringar som Lisa kan ha gjort på koden.

```
kalle@elektra-lab1:~/projekt$ git branch --set-upstream-to=origin/master master
Branch master set up to track remote branch master from origin.
kalle@elektra-lab1:~/projekt$ git pull
Enter passphrase for key '/home/kalle/.ssh/id_rsa':
Enter passphrase for key '/home/kalle/.ssh/id_rsa':
Updating 364216d..fcc0da9
Fast-forward
 test.sh | 6 ++++---
 1 file changed, 3 insertions(+), 3 deletions(-)
kalle@elektra-lab1:~/projekt$ git log
commit fcc0da9fdb93684d7008d05dc889d74a1659a513
Author: Lisa <lisa@example.com>
Date:   Sat Nov 23 14:06:00 2013 +0100

    Change 'echo' to 'printf' on all lines

commit 364216dab5ffbe379b6345a824e42e63118ca8a3
Author: Kalle <kalle@example.com>
Date:   Sat Nov 23 13:56:08 2013 +0100

    Initial commit
```

Här ser vi både Lisa och Kalles ändringar i koden. Notera att vi bara behöver ange *git branch --set-upstream-to=origin/master master* första gången vi pullar från vår remote-maskin. Detta för att tala om vilken gren på fjärrmaskinen som ska mergas med vår lokala gren, i detta fallet master på både fjärrmaskinen och den lokala maskinen.

Eftersom både Kalle och Lisa använder samma user (git) så har de båda samma rättigheter.

Lägga till public access över HTTP

Att lägga till public access till git repon som finns på servern är inget svårt. Det enda som behövs är att i git repon skapa en så kallad hook som uppdaterar information i repon varje gång någon pushar till den. Det finns en färdig hook för detta som medföljer alla git repos som heter **post-update.sample** och ligger under hooks/ i git repon. Bara döp om denna till att istället heta **post-update** och sätt exekveringsrättigheter på filen om den inte redan har det (vilket den brukar ha i allra flesta fall). Detta behövs för att uppdatera allt i repon för att det ska gå att hämta över HTTP som anses vara en ”dum klient/protokoll”. En sådan hook behövs bara för HTTP. Innehållet i hook:en ser ut enligt följande.

```
#!/bin/sh
#
# An example hook script to prepare a packed repository for use over
# dumb transports.
#
# To enable this hook, rename this file to "post-update".

exec git update-server-info
```

Det enda som körs är alltså kommandot *exec git update-server-info*. Här ser vi också kommentaren vad hook:en gör, uppdaterar och paketerar repon för användning över dumma klienter/protokoll.

Steg två är att sätta upp vår webbserver med ett alias till vår git repo. I Apache görs det enkelt med alias direktivet som nedan.

```
Alias /projekt/ "/home/git/projekt.git/"
```

Därefter är det bara att starta om Apache och alla som nu kan komma åt webbservern kan klona projektet från den. För att klona från HTTP används http:// som protokoll som exemplet nedan.

```
jake@elektra:~/tests$ git clone http://tiger.example.com/projekt
Cloning into 'projekt'...
jake@elektra:~/tests$ ls
projekt
jake@elektra:~/tests$ cd projekt/
jake@elektra:~/tests/projekt$ ls
test.sh
```

GitWeb

I Debian och Ubuntu är det enkelt att sätta upp GitWeb. GitWeb är enkel webbsida som visar och visualiserar git repos. Många open-source projekt använder GitWeb för att låta folk bläddra bland repon utan att behöva klona hela projektet. Bland annat kernel.org använder GitWeb.

Börja med att installera paktet med *apt-get install gitweb.* Konfigurerar därefter vart git repona finns i filen /etc/gitweb.conf. I vårt fall ska filen se ut enligt följande för att kunna visa alla våra projekt under /home/git/

```
# path to git projects (<project>.git)
$projectroot = "/home/git/";

# directory to use for temp files
$git_temp = "/tmp";
```

OBS: Tänk på att alla projekt i /home/git kommer att bli synliga för alla som kommer åt webbservern nu! Även om det inte finns ett alias för att klona repon i Apache kan nu alla bläddra runt och tanka ner filer via GitWeb!

Surfa nu in på http://tiger.example.com/gitweb för att komma åt GitWeb.

Hela vår setup ser nu ut enligt nedanstående illustration med Kalle & Lisa som pullar och pushar över SSH, med public access över HTTP och GitWeb.

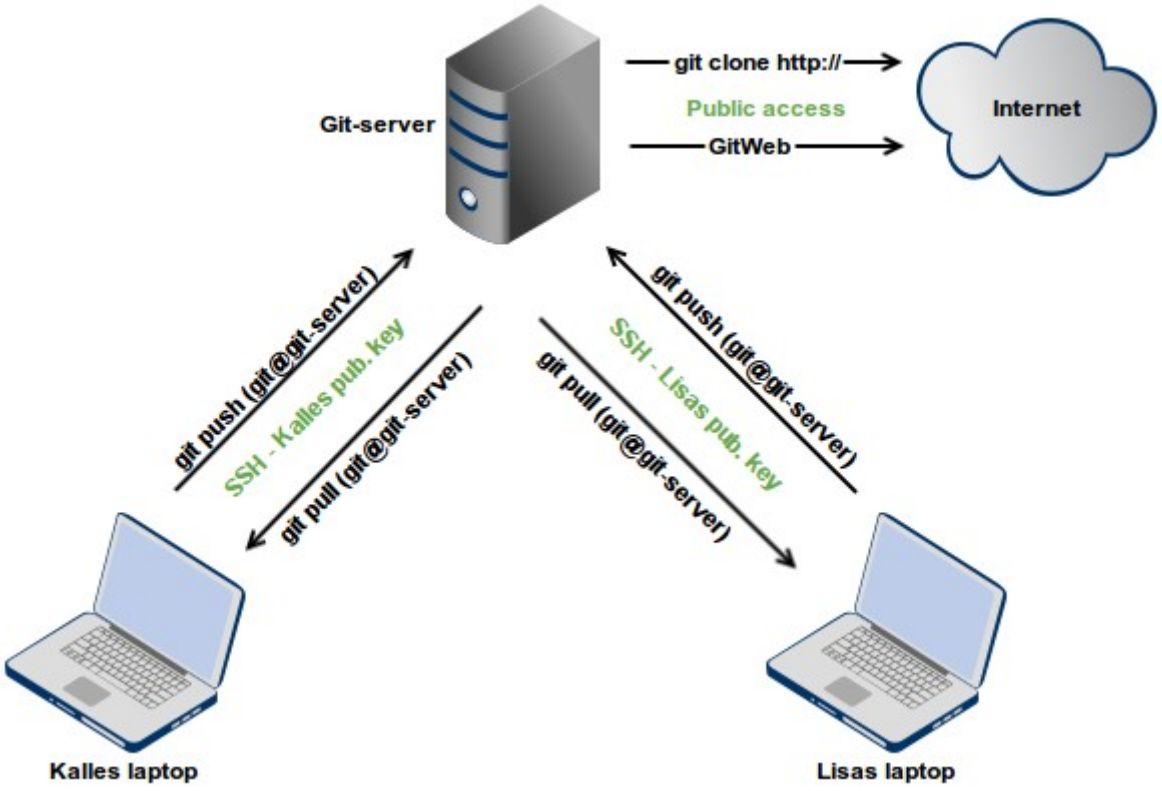


Illustration 7: Git över SSH med public access över HTTP

Gitolite

Gitolite är ett projekt för att underlätta administrationen av git-projekt över SSH. Själva Gitolite är faktiskt en git-repo. För att ändra inställningar, skapa användare, redigera rättigheter, skapa repos m.m. redigeras filer i en repo som sedan pushas till servern. Servern kör då olika hooks för att utföra dessa ändringar.

Gitolite ligger lite utanför ramarna för denna laboration men det är ändå bra att känna till då många företag använder Gitolite för att underlätta administrationen av repos samt att kunna sätta mer precisa rättigheter. Gitolite har t.ex. stöd för att sätta specifika rättigheter för olika grenar för olika användare m.m. Gitolite har även stöd för att skapa så kallade pull-requests där en användare klonar ett projekt och ändringar på detta och sedan skickar en förfrågan till ägaren av projektet att föra samma hans ändringar i koden med originalprojektet.

Gitolite installeras genom att klona projektet som en separat användare på en server. Denna användare ska därför bara användas för git och kan därför heta just git eller liknande. Efter att man klonar projektet skickar man upp den användarens nyckel som ska administrera Gitolite till servern (till användaren git). Därför körs en rad installationskommandon.

```
git@tiger:~$ git clone git://github.com/sitaramc/gitolite
git@tiger:~$ mkdir bin
git@tiger:~$ gitolite/install -ln
git@tiger:~$ gitolite setup -pk $HOME/kalle.pub
```

Därefter kan användaren Kalle från sin egna dator klona repon ”gitolite-admin” med *git clone git@tiger:gitolite-admin.* Från denna repo kan Kalle sedan lägga till nya nycklar för nya användare, skapa nya repos på servern (utan att behöva shell-access till servern, allt sköts via repon) och ställa in rättigheter m.m. I repon gitolite-admin finns två kataloger, en som heter keydir där allas nycklar läggs in och en som heter conf där det finns en konfigurationsfil som heter gitolite.conf. Ett exempel på gitolite.conf kan se ut enligt följande

```
repo gitolite-admin
    RW+     =   kalle

repo testing
    RW+     =   @all

repo epic
    RW+     =   lisa
```

Här ser vi att det är Kalle som får lov att administrera gitolite-admin genom att det är han som har RW+ rättigheter för denna repon (R=read, W=write, +=tillåter borttagning av data från repon). På testing repon får alla lov att både läsa, skriva och ta bort information. På epic-repon får bara Lisa lov att skriva, läsa och ta bort information.

För att skapa en ny repo på servern behöver Kalle bara lägga in en ny rad i config-filen med en ny repo och rättigheterna för denna och sedan pusha den nya konfigurationen till servern och Gitolite skapar då direkt en ny repo med det namn som angavs. Den största fördelen med Gitolite är just det här, att kunna skapa och administrera repos på servern utan att någon behöver ha tillgång till ett skal på servern. Dessutom släpper Gitolite bara in användaren git med sitt lösenord på servern, försöker någon med sin egna nyckel logga in som git på servern kommer inloggningen att nekas.

Gitolite tillsammans med GitWeb

På Debian/Ubuntu system är det relativt enkelt att få Gitolite och Gitweb att fungera tillsammans. Efter att vi installerat GitWeb och Gitolite börjar vi med att ändra i filen .gitolite.rc som ligger i gitolite-användarens hemkatalog. Kompletta sökvägen blir /home/git/.gitolite.rc om vi följt ovanstående exempel. Här ändrar vi umasken till att bli 0027 så att gruppen får läsa alla repos. Raden vi ändrar ska alltså se ut som följande när vi är klara

```
UMASK                           =>  0027,
```

Därefter måste vi lägga till www-data användaren (usern som kör Apache2) till gruppen git, så att www-data får lov att läsa alla repos. Vi måste även ändra rättigheterna på de redan existerande repo-katalogerna till 750.

```
sudo usermod -a -G git www-data
sudo chmod 640 /home/git/projects.list
sudo chmod -R 750 /home/git/repositories
sudo service apache2 restart
```

Det enda som behövs nu för att GitWeb ska kunna se våra repos är att vi i gitolite-admin/conf/gitolite.conf lägger till läsrättigheter för gitweb. När vi gör detta så kommer Gitolite automatiskt att lägga till repon i /home/git/projects.list (filen som GitWeb använder för att lista repos). Rättigheterna för att GitWeb ska kunna läsa våra repos kan alltså se ut enligt följande

```
repo epic
    RW+     =   lisa
    R       =   gitweb
```

Om vi nu surfar in på vår git-server (tiger i det här fallet) till vår GitWeb på URL http://tiger/gitweb kommer vi att se repon epic.git.

Reflektioner

Att få möjligheten att fördjupa sig i git har varit oerhört lärorikt. Jag har länge använt git, men då bara de mest grundläggande saker som *add, commit, pull, push* m.m. Med detta arbete har jag fått en så mycket bättre inblick i grenar och taggar som är ju ändå är ”the killer-feature” när det kommer till git. Dessutom var det riktigt nyttigt att få testa på att sätta upp Gitolite. Gitolite tillsammans med GitWeb är något som är mycket praktiskt att använda. Med denna uppsättning kan man i princip sätta upp sin egen privata Github om man så önskar. Det finns till och med teman för Gitweb för att efterlikna Github.

Det finns även Atlassian Stash som är till för att företag, föreningar och privatpersoner ska kunna köra en egen Bitbucket bakom sin brandvägg i sitt egna nät.

Git är dessutom något som är oerhört användbart till mer än bara kod. Man kan ju t.ex. göra en git-repo av hela sin /etc katalog för att kunna ångra redigeringar om man helt plötsligt inte kan starta en tjänst för att man gjort något fel i en config och inte hittar felet. Då är det bara att check ut en fungerande version av just den filen från tidigare commit.

Likaså kan man använda git till mer än bara rena textfiler, config-filer och kod. Jag använder själv git för alla min arbeten och dokument. På så sätt kan jag klona hela katalogen med alla arbeten till vilken dator som helst. Skriver jag arbetena på min stationära dator kan jag bara klona dem på min bärbara. När jag sedan gjort redigeringar eller fler arbeten på min bärbara kan jag bara köra en *git pull* i katalogen på min stationära och alla de senaste ändringar förs direkt över till min stationära. På så sätt har jag också multipla kopior av katalogen med alla arbeten i. Detta betyder ju att jag faktiskt har en up-to-date backup utspridd över flera datorer.

Men det är trots allt kod som git är skapat för och det är också här som man får full nytta av git.

Referenser

Pro Git av Scott Chacon, ISBN: 978-1-4302-1833-3

http://git-scm.com/documentation

https://www.kernel.org/pub/software/scm/git/docs/gitweb.conf.html

http://gitolite.com/gitolite/master-toc.html

http://blog.countableset.ch/2012/04/29/ubuntu-12-dot-04-installing-gitolite-and-gitweb/

Man-sidor för git i Ubuntu