Bertsio-kontroleko sistema banatuak: git eta github

1. Aurkezpena

Ikastaro honetan git bertsio-kontroleko sistema banatua eta github zerbitzua azalduko dira. Bertsio-kontroleko sistemak kodea kudeatzeko tresnak dira. Sistema hauek aukera ematen dute iturburu-kodeak denboran zehar jasandako bilakaera aztertzeko, eta nahi izanez gero aldaketaren bat desegiteko. Bertsio-kontroleko sistemak ezinbestekoak dira programatzaile talde batek proiektu berean elkarrekin lan egiteko garaian. Horrela, software libreko proiektu gehienak (ia guztiak), bertsio-kontroleko sistemen bidez kudeatzen dira.

Tradizionalki, bertsio-kontroleko sistemak zentralizatuak izan dira, zeinetan zerbitzari zentral bat dagoen, software-proiektuaren bertsio “ofiziala” gordetzen duena. Bertsio-kontroleko sistema banatuetan, berriz, ez dago zerbitzari zentralizaturik. Horren ordez, garatzaile bakoitzak gordailu osoa gordetzen du, hots, kodea eta bere historia guztia.

Ikastaro honetan *git* bertsio-kontroleko sistema banatua landuko dugu (http://git-scm.com/).

Nahiz eta *git* bertsio-kontroleko sistema banatua izan, software-proiektuaren gordailua zerbitzari batean egoteak abantaila ugari ditu, batez ere proiektua lan talde batek garatu behar badu. *github* (http://github.com) ostalaritza zerbitzua da, *git* bidez sortutako software-gordailuak gordetzen dituena. Horretaz gain, *github* kode irekiko sare soziala dugu, eta haren bidez garatzaileak elkarrekin komunikatzeko hamaika bide eskaintzen ditu.

- Ez dago erreferentzia edo kanonikoa iturburu kopiarik lehenetsia.
- Eragiketa arruntak (*commit, status, revert, eta abar)* oso azkarrak dira, zerbitzari zentral batekin komunikatu ez duelako behar.
- Komunikazioa beharrezkoa da bakarrik aldaketak konpartitu behar denean beste *peer* batekin. *Peer to peer* horrek parekoen arteko komunikazioari egiten dio erreferentzia. Ez daude bezero eta zerbitzari finkorik, bien papera aldi berean egiten duten nodoak baizik. Bezero-zerbitzari ereduaren “kontrako” bezala har daiteke. Hierarkia gabeko sareak dira. Lagun (erabiltzaile edo nodo deritzenak) bakoitza berdinak dira eta ez dago bat bestea baino maila altuagoan.
- Garatzaile bakoitzak gordailu osoa gordetzen du, hots, kodea eta bere historia guztia.

2. Git oinarriak

Atal honetan ikusiko dugu nola sortu git gordailu bat, nola gorde zure proiektuaren egoera-argazkiak (*snaphots)* eta nola begiratu zure proiektuaren historia.

```
$ git config --global user.name "Zure izena"
$ git config --global user.email "zure_helbidea@zerbait.com"
```

Sortu programa bat:

```
$ mkdir kaixo
$ cd kaixo
```

```
$ vim kaixo.pl
        use 5.010;
        use strict;
        use warnings;
        say "Hello World";
$ perl kaixo.pl
```

Sortu zure lehenengo gordailua:

```
$ git init
```

2.1 git init eta git init --bare

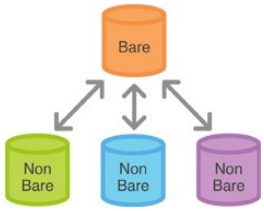
Git init-ek gordailu berri bat sorzen du; init-ek .git direktorio bat sortzen du; direktorio horretan metadatu guztiak daude.

```
$ git init <direktorioa>
```

Komando honek sortzen du direktorio berri bat eta direktorio horretan git gordailu berri bat.

```
$ git init --bare <direktorioa>
```

Komando honek sortzen du gordailu berri bat lan-direktoriorik gabe. Gordailuak konpartitzeko --bare izan behar dira. Normalean, bere izena .git-erekin bukatzen da. Hau da, kaixo-ren izena kaixo.git izango da. Normalean, erabiltzaileek --bare gordailuak sortzen dituzte eta gero *clone* egiten dute kopia bat sortzeko bere ordenagailuan; --bare gordailuetan ezin da aldaketak egin, bakarrik bere lan-kopietan:



2.2 git add

Gehitu zure programa berria gordailuan, hau da, gehitu *staging areara:*

```
$ git add kaixo.pl
```

git add komandoak hartzen du zure lan-direktorioan egindako aldaketak eta grabatzen ditu aldaketa(k). Hala ere, gordailuaren historia eguneratzeko *commit egin* behar da. Bestela, git-ek barrarik daki aldaketa lokalak egin ditugula.

```
$ git add <file>
$ git add <direktorioa>
```

fitxategi edo direktorio bat(en) aldaketak gehitzen ditu.

```
$ git add .
```

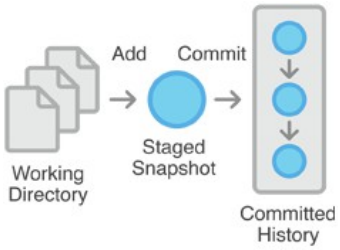
Lan-direktorioaren aldaketa guztiak gehitzen ditu. Kontuz ibili behar da aukera honekin, batzuetan fitxategi/direktorio batzuk nahi gabe gehituko dira.

```
$ git add -p
```

Sortzen du saio interaktibo bat galdetzeko zein aldaketak nahi ditugun gehitu.

Git-erekin software garapena egiteko proiektu batean *add/commit* komandoak erabili behar dira. Hiru pauso dira:

1. Lehenago, fitxategiak/programak aldatu behar dira.
2. Prest zaudenean proiektuaren egoera-argazkia gordetzeko (*snapshot*) *git add* erabili behar da.
3. Ados bazaude egindako uneko-argazkiekin, *git commit* proiektuaren historian uneko-argazkia gordetzen ditu:



2.3 git commit

Hitzartuta (*committed*) dauden snapshot-ak bertsio seguruak dira. Git-ek inoiz ez ditu aldatuko erabiltzeak ez baditu esplizituki aldatzen.

```
$ git commit
```

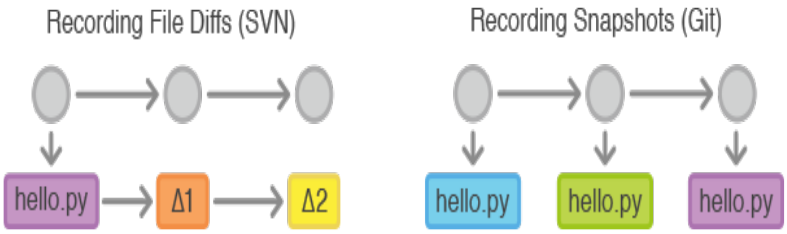
Editore bat irekiko da commit-en mezua idazteko.

```
$ git commit -m "<mezua>"
```

Egin commit bat mezu bat gehitzen:

```
$ git commit -a
```

Egindako aldaketa guztiak gehitzen dira egoera-argazki (*snapshot*) batean eta proiektuaren historian grabatzen dira.



```
$ git commit -m “Gure lehenengo commit”
[master (root-commit) 7b7661b] my first commit
 1 files changed, 5 insertions(+), 0 deletions(-)
 create mode 100644 kaixo.pl
```

2.4 git status

Komando honek ematen dizu zerrenda bat esaten zein fitxategiak aldatu diren (*staged*), zeintzuk ez dauden *snapshot* batean (*unstaged*) eta zeintzuk ez dauden erregistratuta (*untracked*). Beti begiratzen proiektuaren git *status add* edo *commit* egin baino lehen:

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#         new file:   kaixo.pl
#
$ git commit -m “Gure lehenengo commit”
[master (root-commit) 7b7661b] my first commit
 1 files changed, 5 insertions(+), 0 deletions(-)
 create mode 100644 kaixo.pl
$git status
# On branch master
nothing to commit (working directory clean)
```

2.4 Aldaketak egiten

Atal honetan aztertuko dugu proiektuaren eboluzioa. Aldatu programa:

```
$ vim kaixo.pl
use 5.010;
use strict;
use warnings;
say "zein da zure izena? ";
my $name = <STDIN>;
chomp $name;
```

```
say "Kaixo $name, zer moduz?";
$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   kaixo.pl
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Gehitu aldaketa hau *snapshot* batera:

```
$ git add kaixo.pl
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   kaixo.pl
#
```

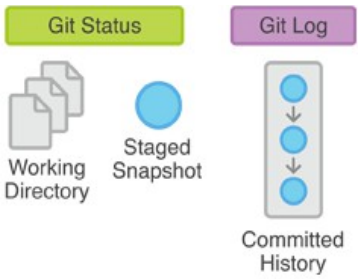
Orain git-ek daki aldaketa bat egin dugula, baina aldaketa ez da oraindik gehitu gordailura izan. Commit ez baduzu egin nahi, *git reset* komandoa erabil dezakezu aldaketa (*snapshot*) desegiteko (*unstaged).*

Praktika ona:

```
$ git add doc1.pl
$ git add doc2.pl
$ git status
$ git commit -m “ezaugarri berri bat gehituta”
$ git status
$ git add doc3.pl
$ git commit -m “ezaugarri2 gehituta”
$ git status
```

3. Historiaren kudeaketa

git log-ek commited egin ditugun snapshot-ak erakusten ditu. Ikusi dugunez, git status-ek erakusten ditu lan-direktorioan egindako aldaketak eta git log-ek bakarrik commit egin dugun historia.




```
$ git log
commit 7b7661b44700a35b75861bd72afd007737866857
Author: Rodrigo Agerri <zure@helbidea.org>
Date:   Mon Jul 14 16:58:02 2014 +0200

    my first commit
$ git log -n <limit>
$ git log --oneline
$ git log --stat
$ git log -p
$ git log --author="<pattern>"
$ git log --grep="<pattern>"
$ git log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
```

--pretty="..." formatua

%h commit hash-a

%d commit etiketak

%ad egilearen data

%s mezua

%an egilearen izena

--graph erakutsi commit-en zuhaitza ASCII grafo batean

--date= data

$ gitk

Sortu ezizen batzuk:

$ vim ~/.gitconfig

```
[alias]
  ci = commit
  st = status
  hist = log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
```

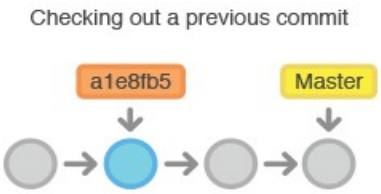
```
$ git hist
$ git hist --all
```

3.1 git checkout

Komando honek hiru funtzionalitate ditu: begira (*checkout*) daitezke fitxategiak, *commit-ak*, eta adarrak. Atal honetan fitxategiak eta adarrak aztertuko ditugu.

git checkout commit bat:

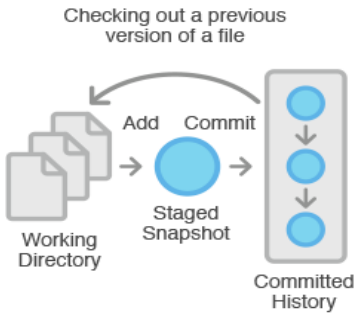
```
$ git checkout <hash>
Note: checking out '7b7661b44700a35b75861bd72afd007737866857'.

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 7b7661b... my first commit
$ less kaixo.pl
use 5.010;
use strict;
use warnings;

say "Kaixo denoi";

$ git checkout master
$ less kaixo.pl
```



git checkout fitxategi bat:

```
$ git checkout 7b7661b kaixo.pl
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#         modified:   kaixo.pl
#
$  git checkout HEAD kaixo.pl
$ git status
```

git checkout-ek fitxategi batekin aldatzen du proiektuaren snapshot-a eta commit egiten bada, proiektuaren historia aldatuko da.



git checkout etiketaren bidez:

Commit bati etiketa ematen badiogu, etiketa hori erabil daiteke etiketaren bidez *checkout* egiteko:

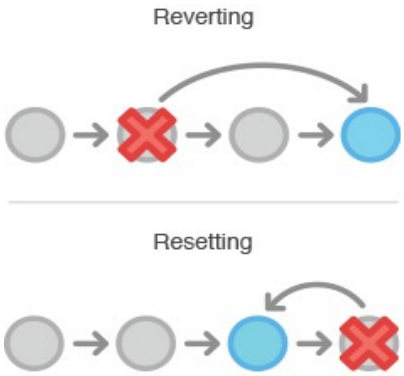
```
$ git tag v1
$ git checkout v1~1
$ less kaixo.pl
$ git tag v1-beta
```

```
$ git checkout v1
$ git checkout v1-beta
$ git tag
$ git hist master --all
```

4. Aldaketak desegin

Atal honetan aztertuko ditugu nola begiratu zure proiektuaren aurreko bertsioak.

1. **git revert**: Proiektuaren historian aurreko hitzartuak (*commit*-ak) bihurtzeko.
2. **git reset**: Berrasieratzeko (*staged) snaphot-ak*, hitzartu (commit) egin baino lehen.

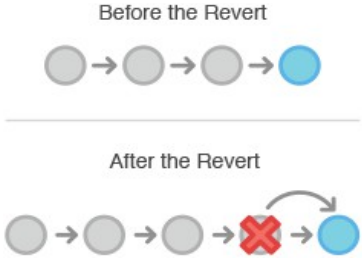


4.1 git revert

Komando honek desegiten du hitzartutako *snapshot* bat. Hala ere, ez du ezabatzen aurreko hitzartua. Aurreko egindako aldaketak desegin ditu eta gehitzen ditu hitzartu berri bat proiektuaren historian; *git revert* erabil daiteke gordailu publikoetan:

Aldatu kaixo.pl:

```
$ git add kaixo.pl
$ git commit “nahi gabe commit”
$ git hist
$ git revert <commit>
$ git revert HEAD
$ git hist
```

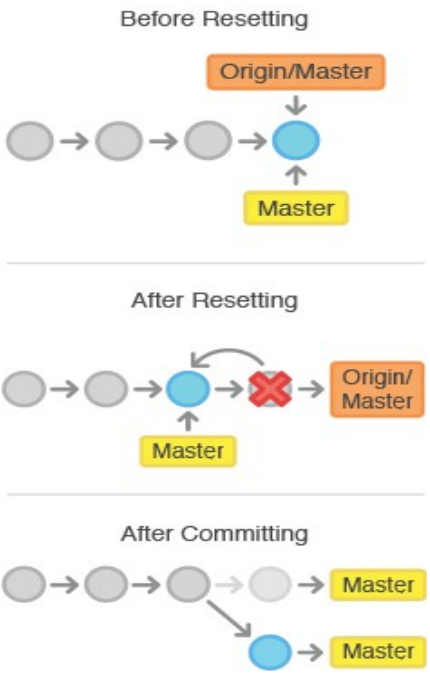


4.2 git reset

Aldaketak *git reset* erabiltzen desegiten direnean, aurreko aldaketak behin-betiko galduko dira. Beraz, ez da batere komenigarria *git reset* erabiltzea hitzartuak desegiteko!! *git reset* normalean erabiltzen da *snapshot*-ak aldaketa-lokalak desegiteko.

Aldatu kaixo.pl

```
$ git add kaixo.pl
$ git status
$ git reset HEAD kaixo.pl
$ git checkout kaixo.pl
$ git status
```



Aldatu kaixo.pl

```
$ git reset --help
$ git reset --hard
$ git clean -f
```

Zer gertatu da?

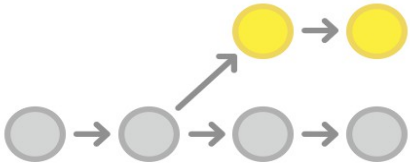
4.3 Commit aldatzen

Batzuetan hitzartu (*commit*) bat aldatu behar da. Adibidez, kode-irekiko proiektu batzuetan, hitzartuen mezuak markatu behar dira *id* batekin. Baina kontuz, **aldatu commit lokalak bakarrik.**

Aldatu kaixo.pl

```
$ git add kaixo.pl
$ git commit -m “egilearen mezua”
$ git hist
$ git commit --amend -m “id-543 egilearen mezua”
$ git hist
```

5. Adarkatzen (Branching)



Atal honetan ikasiko ditugu:

1. Nola sortu adar berri bat.
2. Nola mugitu adar batetik bestera *git checkout*-en bidez.
3. Nola integratu bi adar *git merge* erabiltzen.

Git adarrak oso erabilgarriak dira eta asko erabiltzen dira. Ezaugarri berri bat gehitu nahi duzunean, edo esperimentu berri bat egiteko, *git branch* zure laguna da.

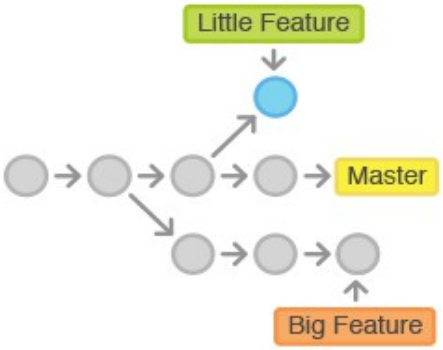
Sortu adar berri bat:

```
$ git branch garapena
$ git checkout garapena
$ git status
edo
$ git checkout -b garapena
$ git status
```

Aldatu zure proiektua:

```
$ vim kaixo.pl
$ git commit -a -m “ezaugarri bat”
$ git status
$ git hist
$ vim kaixo.pl
```

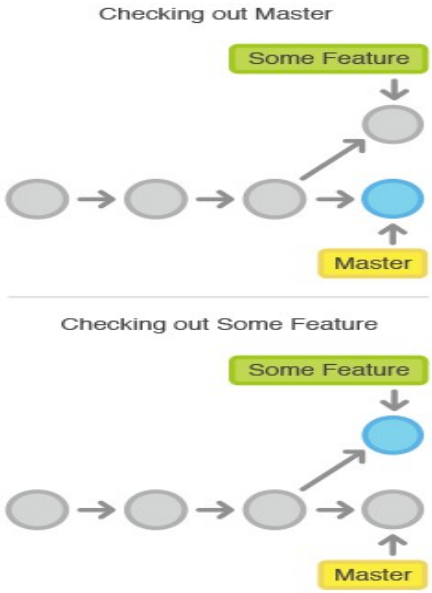
```
$ git commit -a -m “ezaugarri bi”
$ git status
$ git hist --all
$ git checkout master
$ git hist --all
$ gitk
```



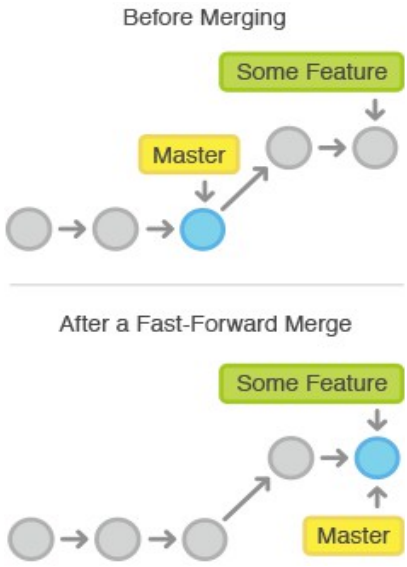

```
$ git branch -d garapena
$ git branch -D garapena
```

…. eta orain sortu berriro beste adar bat … :)

```
$ git checkout master
$ vim README.md
$ git status
$ git add README.md
$ git commit -m “readme added”
$ git status
$ git hist--all
$ gitk
```



5.1 git merge



Git merge-k integratzen du adar bat master-ekin.

```
$ git checkout garapena
$ git merge master
```

Hala ere, aldatzen badugu master adarra eta garapena adarra, gatazka bat sortuko da eta *merge* ez da automatikoa izango.

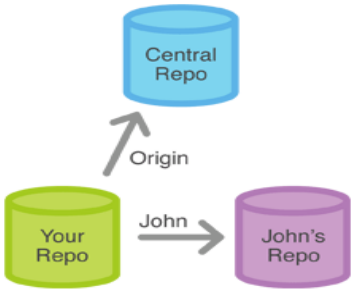
```
$ git checkout garapena
```

```
$ vim kaixo.pl
$ git commit -a -m “ezaugarri 3”
$ git status
$ git checkout master
$ vim kaixo.pl
$ git commit -a -m “ezaugarri master”
$ git merge garapena
```

Zoritxarrez, kasu hauetan gatazkak eskuz konpondu behar dira.

```
$ git add kaixo.pl
$ git commit -m “merge master gatazka”
$ git status
$ git hist --all
```

6. Gordailu anizkunak



Bertsio-kontroleko sistema banatuetan, ez dago zerbitzari zentralizaturik. Horren ordez, garatzaile bakoitzak gordailu osoa gordetzen du, hots, kodea eta bere historia guztia. Gainera, gordailu batetik kopia anizkunak egin ditzakezu:

```
$ git clone kaixo clone-kaixo
$ ls
$ cd kaixo
$ git hist --all
$ cd ../clone-kaixo
$ git hist --all
$ git hist --all
```

Cloned gordailutik ikus dezakegu jatorrizko gordailuaren historia:

```
$ git remote -v
```

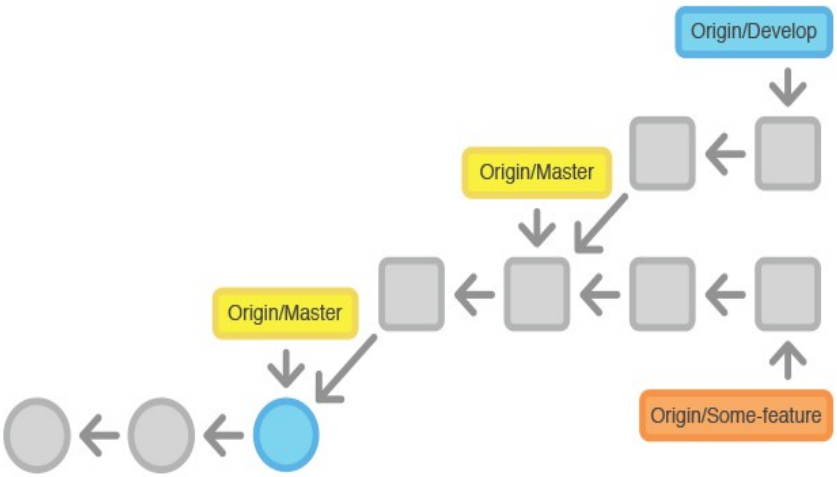
```
$ git remote show origin
$ git remote --help
```

Begiratu orain zenbat adar dauzkagun gure gordailuan:

```
$ git status
$ git branch
$ git branch -a
```

Zer gertatzen da hemen? *Cloned* gordailuan ez daukagu garapena adarra, baina git-ek daki jatorrizko gordailuan (*origin*) adar bat izen horrekin bazegoela. Nahi badugu garapena adarra jaso eta aldatu *git fetch* erabili behar dugu.

6.1 git fetch




```
$ git fetch  <remote>
```

Jatorrizko gordailuaren adar guztiak jasotzen ditu.

```
$ git fetch origin <branch>
```

Jatorrizko gordailuaren adar espezifiko bat jasotzen du:

```
$ git fetch origin
$ git checkout garapena
$ cd ../kaixo
$ vim kaixo.pl
$ git commit -a -m “ezaugarri 5”
$ cd ../clone-kaixo
$ git fetch origin master
$ git checkout origin/master
$ git merge origin master
```

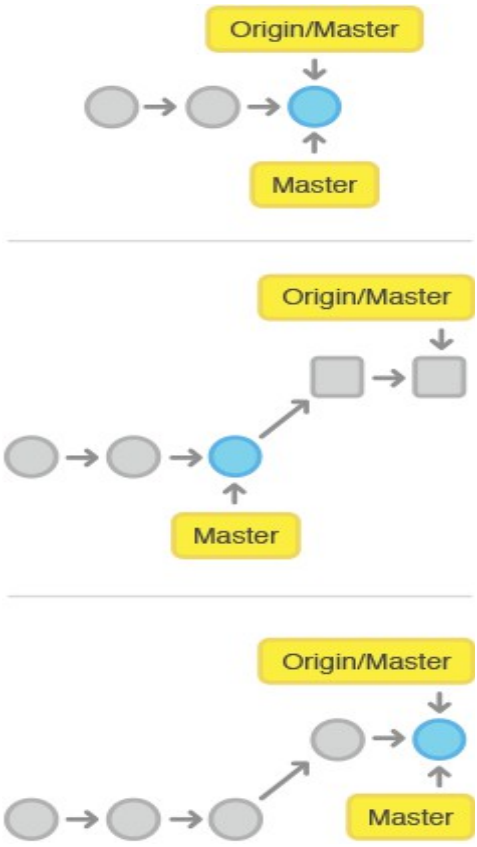
6.2 git pull

Komando honek *git fetch* eta *git merge* egiten du pauso batean.

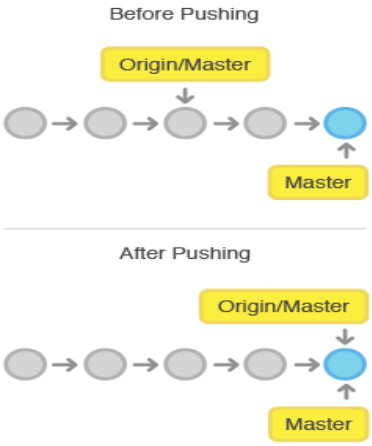
$ git pull <remote> = $ git fetch remote && $ git checkout origin/master $ git merge origin <branch>

Jatorrizko kaixo gordailua aldatu, egin stage eta commit, eta gero *pull* aldaketak cloned gordailutik:

```
$ cd ../kaixo
$ vim README.md
$ git add README.md
$ git commit -m “minor update”
$ cd ../clone-kaixo
$ git pull origin master
```



6.3 git push



Komando honek bidaltzen ditu egindako aldaketak jatorrizko gordailura, hau da, commit bat edo batzuk esportatzen ditu urruneko adar batera (*remote branch*): $ git push <remote> <branch>

```
$ cd clone-kaixo
$ vim README.md
$ git add README.md
$ git commit -m “beste aldaketa”
$ git push origin master
```

Zer gertatu da? Gogoratzen aipatu dugula (atal 2.1-ean) gordailuak--bare izan behar zirela gure aldaketak konpartitzeko?

7. Bare gordailuak

Proiektu bat konpartitzeko, sortu gordailu *bare* bat:

```
$ cd ../
$ git clone --bare kaixo kaixo.git
$ ls kaixo.git
$ cd kaixo
$ git remote add shared-kaixo ../kaixo.git
```

Begiratu git remote --help eta erabili ikusteko zer dagoen kaixo.git-en. Gero joan kaixo proiektura, egin beste aldaketa, egin stage eta commit bat eta push shared-kaixo-ra:

```
$ cd kaixo
$ vim README.md
$ git add
$ git commit -m “yet another minor change”
$ git push shared-kaixo master
```

Orain joan kaixo-clone-ra eta pull egin dugun azken commit-a shared-kaixo-n:

```
$ git status
$ git remote add shared-kaixo ../kaixo.git
$ git fetch shared-kaixo
$ git pull shared-kaixo master
$ vim README.md
$ git commit -a -m “gatazka konponduta”
$ git push shared-kaixo master
$ git hist --all
```

Orain arte, egon gara lanean makina batean, baina benetan kodea konpartitzeko interneten bidez jaso behar dugu (*clone*) proiektu bat. Nahiz eta *git* bertsio-kontroleko sistema banatua izan, software-proiektuaren gordailua zerbitzari batean egoteak abantaila ugari ditu, batez ere proiektua lan talde batek garatu behar badu. *github* (http://github.com) ostalaritza zerbitzua da, *git* bidez sortutako software-gordailuak gordetzen dituena. Horretaz gain, *github* kode irekiko sare soziala dugu, eta haren bidez garatzaileak elkarrekin komunikatzeko hamaika bide eskaintzen ditu.

8. github

1. Sortu kaixo (edo beste bat) proiektua github-en (--bare izango da) eta errepikatu egin duguna lokalean github-eren zerbitzariarekin.
2. Talde batzuk sortuko ditugu proiektu bat konpartitzeko
3. Laguntzaileak gehitzera ikasiko dugu.
4. Eta git pull request egitera.
5. Nola eta zergatik erabili git fork eta mirror.
6. (Hautazkoa) ssh konfiguratuko dugu (ssh-keygen).

9. Kredituak eta baimenak.

Egilea: Rodrigo Agerri (http://www.rodrigoagerri.net)

Data: 2014eko uztailaren 15a.

Aurkibidea