Intr-un comit pot fi pastrate mai multe modificari facute in mai multe fisiere. Depinde cate fisiere erau salvate in staging area.

**Head Alias**

Cand scriem git log, putem vedea ca sunt afisate toate commiturile, dar numai unul din el e Head. Default, Head e mereu commitul cel mai current, dar putem modifica asta. Head ne arata ca pozele ce programul le-ar putea folosi in diferite operatii sunt cele din commitul in care el apare.Cand folosim diff, branch sau status, git foloseste anume acel commit marcat ca head pentru a face operatii cu modificarile noi facute.

**.ignore file**

Fisierul .ignore face ca git sa nu colecteze absolut niciun fel de date despre fisierele specificate in el.

Acest fisier e vazut de git si poate fi pus in staging area sau commit.

**ID comitturilor**

* ID al commiturilor e un String din 40 de caractere(litere si numere)
* Acest String din litere si numere se numeste Hash
* Hash este calculat de un algoritm SHA1
* SHA1 ia multe date ca input si produce un String de 40 de caractere ca output. In cazul la GIT, aceste date sunt luate din continutul la commit(autor,data,commit message,poze luate ...)
* In asa fel, se aigura ca fiecare commit sa fie unic. Chiar de vom copia commituri de pe alte computere, oricum sansele ca ele sa aiba acelasi ID sunt practic 0
* Cand folosim git commit --amend, nu se va modifica doar continutul ultimului commit, dar si id a lui, de acea comanda data nu e pentru commituri publice.
* Putem folosi git show id si sa punem doar primele 4 caractere din ID, si git ne va afisa acel commit, daca e unu doar.

**Branch**

* **Branch** – pointer catre un commit particular
* **Este o versiune separata/noua din main repository.**
* Reprezinta o linie independenta de dezvoltare dintr-un proiect al carui commit spre care pointeaza este cel mai recent link intr-un canal al istoriei de dezvoltare.
* **master** – default branch pe care git o creaza cand cream un repository
* **master branch** este folosit mai mult pentru a reprezenta partea buna a proiectului, adica ceea ce developerii pun in proiect cand stiu la sigur ca e functional. Daca vrem sa incercam ceva nou in proiectul nostru, ca sa nu il stricam, facem un branch nou, si lucram nu direct asupra lui.

**De ex:**

La scoala avem un caiet de matem(Master Branch). Mai avem alte 3 caiete, insa in ele facem mazgalituri, incercam sa rezolvam exercitii, sa modificam exercitii, si cand totul e gata si frumos, copiem in caietul nostru Master Branch.

* Asa branches ne ajuta sa experimentam noi idei in proiect.
* Cand vrem sa incercam sa modificam ceva in proiect, cream un nou branch si cand suntem deja siguri ca ce am facut e bun, il punem in Master Branch
* HEAD – commit la care suntem conectati acum
* git log mereu ne va arata cel mai recent commit din branch curent.
* **Cand cream noul branch, desi cel mai recent commit al sau va fi exact acelasi care a fost in branch parinte, cu git log vedem ca toate commiturile de la branch parinte tot au fost puse, nu doar cel mai recent, ca ele tot sunt necesare pentru a reconstrui Head**

Master:

commit1

commit2

cream un nou branch Test, deci Test va avea :

Test:

commit1

commit2

Sa zicem ca mai adaugam commituri la Test

Test:

commit4

commit3

commit1

commit2

Acum, daca in master dam merge, master va avea:

Master:  
commit4

commit3

commit1

commit2

* Acum, cand ne mutam la un nou branch, working tree se va modifica si in el vor ramane doar fisierele caracteristice acelui branch.
* Cand cream un nou branch, acesta va avea acces la toate fisierele din branchul in care am fost cand l-am creat,si toate commiturile sale, asa cum are un pointer catre cel mai recent commit din branch parinte,si catre toate cele pe care acest branch le-a avut deci.Asa dar, noul branch va avea acces la cel mai recent commit din branch din care a fost creat,si le va avea si pe celelalte de pana la Head din branch parinte, dar va salva doar modificarile facute in el,nu propriu zis intregul commit din branch parinte.In plus, vor fi pastrate si toate commiturile din acel branch de mai in urma. Insa, fisierele pe care le cream in noul branch nu mai apar in cel din care l-am creat. De ex:

Fie ca suntem in Master si avem fisierele:

File1

File2

Cream un branch test si ne mutam la el.

test are la fel:

File1

File2

Are acces la ele caci are acces la cel mai recent commit din Master. Mai cream un File3, deci test branch va avea:

File1

File2

File3

Insa master nu va avea acces la File3!

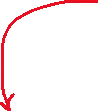
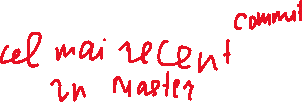
* Cand dam checkout sau commit intr-un branch, aceste modificari vor fi adaugate doar in istoria acelui branch, deci fiecare branch salveaza doar modificarile facute, nu propriu zis copii ale fisierelor.
* Deci, branch nu copie fisierele din branch parinte! Doar le reconstruieste pe baza la commit a lui cel mai recent si pe celelalte si apoi creaza un nou commit cand noi modificam ceva, ce pastreaza doar modificarile facute, si cand reconstruieste fisierele, foloseste commiturile din branch parinte si al lui, caci commitul la acest branch deja va avea o referinta la cel al branch parinte.
* Orice nou branch are acces la cea mai recenta poza a branch din care a fost creat, si deci si la cele din urma. De aceea, test branch poate accesa ce este in master, dar nu poate modifica nimic propriu zis de acolo,ci va salva orice modificari facute doar pentru el
* Cand vom da commit la branch in care suntem, acel commit se va salva doar in acel branch. Tot ce am facut se salveaza doar in acel branch, in cel parinte totul ramane exact asa cum era, fara nici-o schimbare.
* Daca de ex cream un nou branch, modificam ceva si dam commit, apoi ne ducem in branch parinte si modificam iar ceva, branch parinte logic ca va avea un nou commit ca cel mai recent, dar branch copil va continua sa aiba o referinta la fostul commit cel mai recent din branch parinte.Git doar va vedea ca cel mai recent commit din parinte si primul commit din copil(care mereu e cel ce a venit de la parinte cand a fost creat) difera. Aici poate sa apara o problema deja.

**Merge Branch**

* **Merge** – termen folosit de Git pentru a combina datele si istoria din branch in altul
* Dupa ce dam merge, nu se creaza(daca branch copil inca are ref la cel mai actual commit din parent) un nou commit in branch actual! Pur si simplu se va lua cel mai actual commit din branch copil si se va adauga in branch parinte ca cel mai actual. Deci, asa cum branch copil a luat commit principal de la branch parinte cand a fost creat, asa acum dupa merge, branch parinte va lua cel mai recent commit din branch copil.
* Deci, merge face ca ambele branches sa aiba acelasi commit ca cel mai recent, **deci acelasi id la commti**!!!
* **Ceea ce e descris mai sus e valabil cand in child branch se gaseste cel ma recent commit din parent branch**

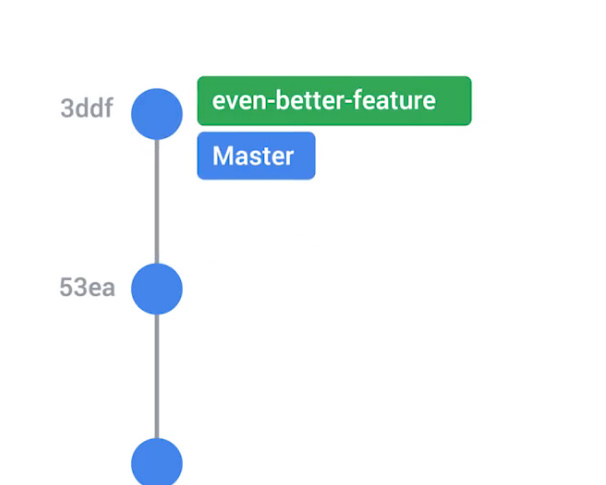
**Pana la merge:**

****



Dupa merge:

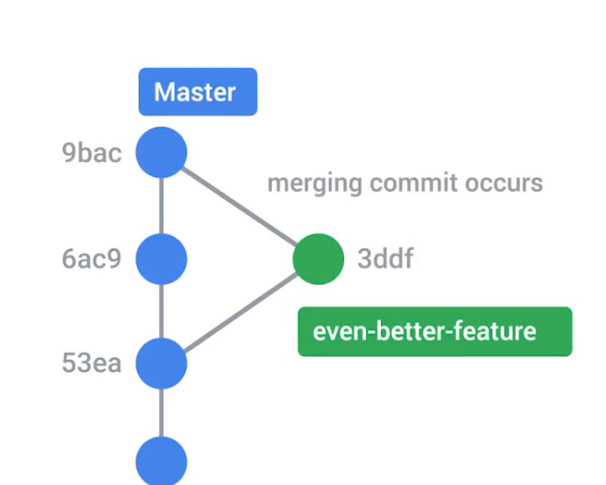


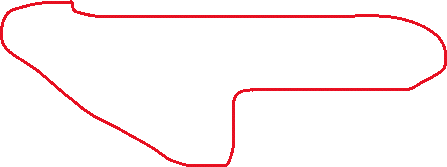




**Merge Conflicts**

* Atunci cand in child branch se gaseste cel mai recent commit din parent branch, e usor sa dam merge in parent branch, deoarece git va trebui pur si simplu sa adauge cel mai recent commit din child branch in parent branch impreuna cu commiturile noi din urma
* Totusi, daca cel mai recent commit din parent branch nu este in child branch, poate sa apara un conflict.
* De ex, dam un commit in parent, apoi cream un child branch, facem modificari, dam commit, si ne ducem inapoi la parent. Insa nu dam merge. In parent branch mai facem ceva si apoi dam commit. Iese ca parent branch are un alt commit cel mai recent, dar la el nu are acces si child branch, el are inca la penultimul, care a fost cel mai recent la momentul crearii la branch copil.
* In asa caz, git va incerca sa uneasca cel mai recent commit din parent cu cel din child si **sa creeze unul nou**. Asta o va putea face daca modificarile sunt in fisiere diferite sau in parti diferite de fisier.Gen, daca in parent am modifica la fisierul test.txt ceva la inceput, si in child ceva de la final, noul commit dupa merge va putea usor contine ce am modificat la inceput in fisier din parent si ce am modificat la final in child. **Insa, daca modificarile sunt in acelasi fisier in aceeasi parte, aici apare conflict. Git nu va sti ce sa faca.**





* Daca scriem Git status, vom vedea **both modified** ca status
* Putem anula merge cu git merge –abort
* **Pentru a rezolva conflictul, trebuie sa alegem noi in file manual ce vrem sa pastram si ce nu, caci git ne va marca in fisier zona cu conflict**.

Linia1

<<<<<<< HEAD

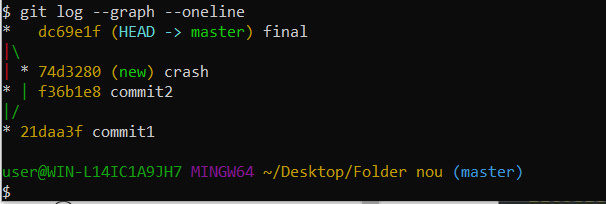
Linia2

=======

ERROR

>>>>>>> new

* Apoi,dupa ce am sters ce nu ne trebuie manual, dam **git add numeFile** pentru a arata ca am rezolvat noi conflctul si **git commit**
* **Apoi, pe langa noul commit creat, se vor copia toate celelalte din branch copil, caci el are nevoie de ele doar.**
* **deci, dupa manual merge, in parent cream un nou commit, dar acest nou commit nu va fi pus si in child branch. Va fi necesar sa dam noi git merge master ca sa se copie acolo ca Head.**
* git log --graph [--oneline] ne poate arata cum au interactionat branches cu commits:



Deci, vedem ca intai(jos), ambele au avut acelasi commit. Apoi, master a creat un nou commit(commit2) si branch al doilea(cu numele new aici) a creat si el un nou commit, si deci in final s-a rezultat un commit final asa cum noi am rezolvat conflictul.

* Cand \* sta in prima coloana, acel branch se refera la master. Deci, commit1 si commit2 si final apartin lui master, iar commit1,crash si final lui new. Atentie ca se pune si commit din branch de la care a inceput. Linia aia inclinata inseaman ca commit se afla in mai multe brnaches.
* Cand \* sta in a 2 coloana, acel commit se refera la alt branch. Linia verde si rosie cu \* arata cui apartine commit.

**Cand nu apare conflict la merge**

* Fie in master un fisier cu urmatoarele linii:

Line1

Line2

Line3

Line4

* Acum, cream un nou branch test
* Facem in test o modificare la linia 1

azazaza

Line2

Line3

Line4

* Acum, ne ducem in master, dar nu dam merge, modificam o alta linie

Line1

Line2

bbbb

Line4

* Acum, cand vom da merge in master, ca sa il unim cu test, nu vor fi conflicte. Se va crea un nou commit, dar nu vor fi conflicte, caci file va arata asa:

azazaza

Line2

bbbb

Line4

* Dar, de se modifica aceeasi linie, era conflict. Daca un commit modifica o linie si alt commit alta linie, nu pot aparea conflicte. E logic ca daca s-a modificat nu avem cum sa o comparam cu originalul, de altfel nu o mai modificam de nu era necesar.

**Automat Merge**

* Fie master si test branch
* Master are commiturile:

commit 1.2

commit 1.1

Acum cream un test branch de la master, deci e logic ca test are:

commit 1.2

commit 1.1

Mai facem 2 commituri in test:

commit2.2

commit2.1

commit1.2

commit1,1

Acum ne ducem inapoi in master si nu dam merge cu test, ci mai facem si aici 2 commituri, deci avem

master

commit1.4

commit1.3

commit1.2

commit1.1

Sa zicem ca in fiecare commit pur si simplu am creat un fisier nou, sa nu aiba cum fi conclicte. Acum, daca in master vom da git merge test, Git va pune toate noile commituri din test in master, si va crea un nou commit ce va avea legatura si cu commiturile din master noi, si cu cele din test, deci master va fi asa:

Merged Commit

commit1.4

commit1.3

commit2.2

commit2.1

commit1.2

commit1.1

Deci, e asemanator ca la ***rebase***. Commiturile de test branch au fost puse acolo unde ele ar trebui sa stea in master in mod normal. Dar, merge are o diferenta fata de rebase, si anume ca doar Merged Commit va fi legat si cu commiturile venite din test, deci commit2.2 si commit2.1, si deci el e legat si de commit 1.4, commit 1.3, dar commit1.3 si commit1.4 nu vor fi legate de commit2.2 si commit2.1!!! Doar Merged Commit e legat si de ele si de celelalte. In rebase, commit1.4 si commit1.3 ar fi inlocuite de altele ce sunt legate de commit2.2 si commit2.1, ca si cum ele mereu au si fost, desi cu merge nu, ele asa raman cum ar fi daca test branch nu ar fi existat.