1. ვერსიათა კონტროლის შესახებ

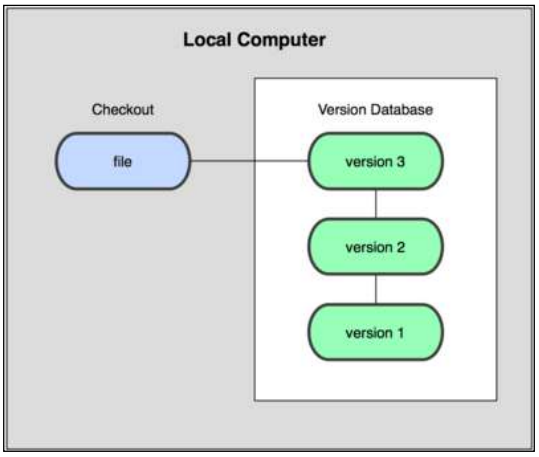
ვერსიათა კონტროლი არის სისტემა, რომელიც საშუალებას გვაძლევს ჩავიწეროთ ფაილების ან ფაილთა ჯგუფების ცვლილებები და მიმდინარე ვერსიები, რათა მოგვიანებით, ნებისმიერ ვერსიაზე დაბრუნების საშუალება გვქონდეს. კომპიუტერში არსებული ნებისმიერი ფაილი შეიძლება იმყოფებოდეს ვერსიათა კონტროლის ქვეშ.

დავუშვათ ვებ-დიზაინერს, პროექტის კეთების პროცესში სურს, რომ შეინახოს სურათის ან დიზაინის ყოველი ახალი ვერსია. ამ შემთხვევაში ჭკვიანურია ვერსიათა კონტროლის სისტემის გამოყენება (ვკს, VCS), რადგან იგი საშუალებას გვაძლევს დავაბრუნოთ ფაილი ან მთლიანი პროექტი, რომელიმე წინა ვერსიაში, მდგომარეობაში.

ვერსითა კონტროლის ლოკალური სისტემა

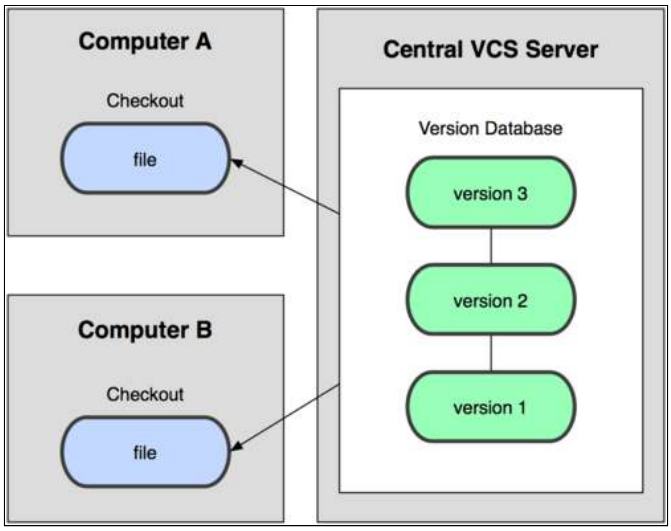
ძალიან ხშირად, ზოგიერთი მომხმარებელი, ვერსიათა კონტროლის შემდეგ მეთოდს ირჩევს : მიმდინარე პროექტს ინახავს ბევრ სხვადასხვა საქაღალდეში, მეთოდი საკმაოდ მარტივია მაგრამ უძლურია სავარაუდო შეცდომების წინაშე, აგრეთვე სავსებით მოსალოდნელია აგვერიოს ეს საქაღალდეები.

ამ პრობლემებთან გასამკლავებლად, კარგა ხნის წინ შეიქმნა ვერსითა კონტროლის ლოკალური სისტემა, მონაცემთა პატარა ბაზით, ამ ბაზაში ინახება ინფორმაცია ფაილთა ცვლილებების შესახებ.



ვერსითა კონტროლის ცენტრალიზებული სისტემა

შემდეგი პრობლემა, რომელსაც დეველოპერები წააწყდნენ გახლდათ ის, რომ აუცილებელი იყო სხვა სისტემაში მომუშავე დეველოპერებთან თანამშრომლობა, ამ პრობლემის მოსაგვარებლად შეიქმნა ვერსითა კონტროლის ცენტრალიზებული სისტემები (CVCS), ასეთ სისტემებს (მაგ: CVS, Subversion, Perforce) აქვთ ერთი სერვერი, სადაც მოთავსებულია ყველა ფაილის ყველა ვერსია, და მომხმარებლები ამ სერვერიდან ამოწმებენ ფაილთა ვერსიებს.

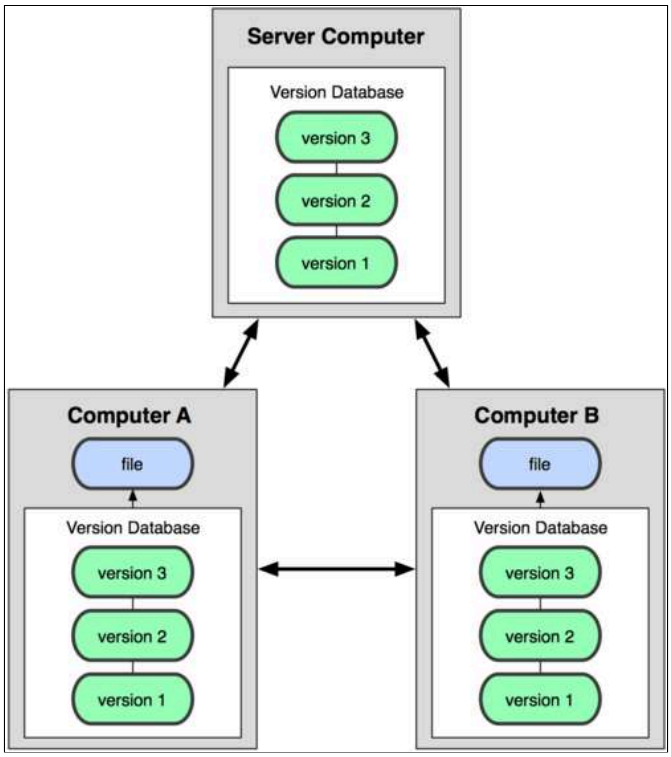


ვერსიათა კონტროლის ასეთ სისტემას ბევრი უპირატესობა აქვს ლოკალურ სისტემებთან შედარებით, მაგალითად ის, რომ ყველამ იცის თუ რას აკეთებენ დანარჩენი მომხმარებლები, ადმინისტრაციას აქვს დაწვრილებითი ინფორმაცია თანამშრომელთა მუშაობისა და პროექტის საერთო მდგომარეობის შესახებ, აქედან გამომდინარე კი გაცილებით ადვილია მუშაობის პროცესის დაგეგმვა და ადმინისტრირება.

თუმცა ამ სისტემასაც აქვს თავისი ნაკლები, მაგალითად თუ სერვერი გამოირთვება ერთი საათით, ამ ხნის განმავლობაში არავის ექნება ცვლილებათა შენახვის და ზოგადად - მუშაობის შესაძლებლობა. ან თუ დაზიანდება სერვერის მყარი დისკი და არ იქნება შენაული პროექტის ასლები, ჩვენ დავკარგავთ აბსოლიტურად ყველაფერს - პროექტის მთელ ისტორიას გარდა იმ ფრაგმენტებისა, რომლებიც თანამშრომლებს ექნებათ თავიანთ ლოკალურ კომპიუტერებში. ზოგადად ეს ასეა: თუ მთელი ისტორია განთავსებულია მხოლოდ ერთ წერტილში, დიდი რისკია ამ ისტორიის დაკარგვისა.

ვერსითა კონტროლის დანაწილებული სისტემა

ვერსითა კონტროლის დანაწილებულ სისტემებში (Distributed Version Control Systems - DVCS), ისეთებში, როგორებიცაა მაგალითად Git, Mercurial, Bazaar ან Darcs, მომხმარებლები არამარტო ფაილთა ბოლო ვერსიებს ადევნებენ თვალს, არამედ ახდენენ მთლიანი საცავის (ინგლ: Repository - საწყობი; საცავი, სათავსი)კოპირებას, კლონირებას, კონტროლს. ასე რომ, თუ რომელიმე სერვერი დაზიანდება, შესაძლებელი იქნება ნებისმიერი მომმხმარებლის საცავის დაკოპირება და ამით სისტემის აღდგენა.

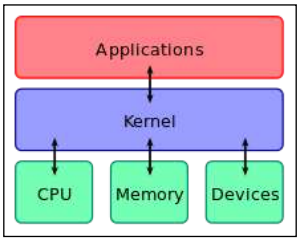


2. Git-ის მოკლე ისტორია

როგორც ცნობილია, ოპერაციული სისტემის - Linux-ის ბირთვი ანუ კერნელი (kernel) გავრცელებულია ღია წყაროთი.

ორიოდ სიტყვით კერნელის შესახებ

კერნელი არის კომპიტერული პროგრამა, ოპერაციული სისტემის ბირთვი, ძირი, რომელიც სისტემაში აკონტროლებს აბსოლიტურად ყველაფერს, სისტემის გაშვებისას პირველად სწორედ კერნელის ჩატვირთვა ხდება, ამის შემდეგ იგი ახდენს დანარჩენი პროცესებისა და პროგრამული უზრუნველყოფების დამუშავებას, სპეციალური ინსტრუქციების



მიხედვით მიღებული ინფორმაციები გადაჰყავს ცენტრალური პროცესორისათვის გასაგებ ენაზე, კერნელი აგრეთვე აკონტროლებს მეხსიერებას და პერიფერიულ მოწყობილებებს (პრინტერი, კლავიატურა, მაუსი და ა.შ)

დავუბრუნდეთ ისევ **Git**-ს :)

Linux კერნელის არსებობის მანძილზე, საკმაოდ დიდი ხნის განმავლობაში, პროგრამული უზრუნველყოფის ცვლილებები ვრცელდებოდა დაარქივებული ფაილების სახით, 2002 წელს Linux კერნელის პროექტმა დაიწყო დაპატენტებული DVCS სისტემის გამოყენება, რომლის სახელიც იყო BitKeeper.

2005 წელს უთანხმოება მოხდა Linux-ზე მომუშავე საზოგადოებასა და BitKeeper-ის შემქმნელ კომერციულ კომპანიას შორის და BitKeeper-ის სტატუსი - "უფასო" გაუქმდა ! :)). ამ ფაქტმა Linux-ის მწარმოებელი კომპანია და მისი ხელმძღვანელი **ლინუს ტორვალდსი** აიძულა, BitKeeper-ისაგან მიღებული გაკვეთილების საფუძველზე შეექმნათ ინფორმაციის გავრცელების საკუთარი წყარო. ამ ახალი ხელსაწყოს მახასიათებლები და მიზნები უნდა ყოფილიყო:

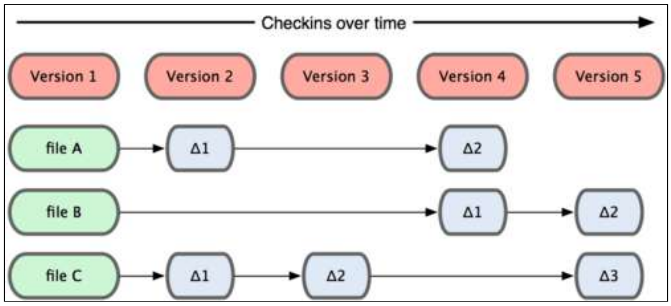
- სისწრაფე.
- მარტივი დიზაინი.
- ფართომაშტაბიანი მწარმოებლობის მხარდაჭერა (ათასობით განშტოების მქონე წარმოება).
- სრულიად ხელმისაწვდომი.
- დიდ პროექტებთან მუშაობის შესაძლებლობა, ისეთებთან როგორიცაა Linux კერნელი.

შექმნის წლიდან (2005) მოყოლებული Git დაიხვეწა და განვითარდა ისე, რომ ადვილად მოსახმარია ყველასათვის, იგი არის ძალიან სწრაფი და ეფექტურად მუშაობს დიდ პროექტებთან.

3. რა არის Git ?

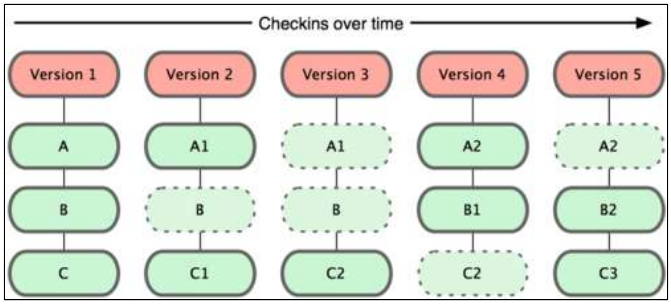
Git არის ვერსიათა კონტროლის **ერთ-ერთი** სისტემა.

ვერსიათა კონტროლის სისტემების მუშაობის ზოგადი სქემა ასეთია



ანუ სისტემების უმრავლესობა ინფორმაციას ინახავს შეცვლილი ფაილების დონეზე: შეიცვალა ფაილი ? გაჩნდა პროექტის ახალი ვერსია, რომელშიც შეტანილია ინფორმაცია **მხოლოდ** შეცვლილი ფაილების შესახებ.

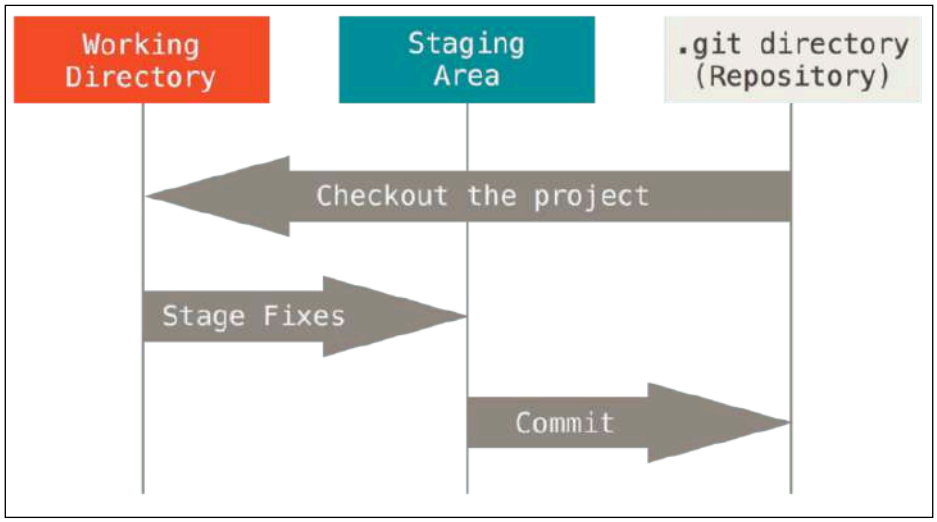
Git კი ინფორმაციას ინახავს არა შეცვლილი ფაილის, არამედ **ფაილის შეცვლიდან გამომდინარე შეცვლილი მთლიანი სისტემის დონეზე**. ყოველი ახალი ცვლილების შეტანისას Git აფიქსირებს სისტემის მიმდინარე მდგომარეობას და ინახავს მას.



სამი საფეხური

ჩვენი პროექტის ფაილები ვერსიათა კონტროლის სისტემასთან მიმართებაში შეიძლება იმყოფებოდეს შემდეგი სამი გარემოდან - მდგომარეობიდან ერთ-ერთში :

1. სამუშაო ხე (Working tree) - **მხოლოდ** სამუშაო დირექტორიაში მყოფი, ვერსიათა კონტროლის ქვეშ **არმყოფი.**
2. გამზადებული ფაილების არე (staging area) - სამუშაო დირექტორიაში **და ასევე გამზადებული ფაილების არეში მყოფი**, შენახვისთვის ანუ დაკომიტებისათვის გამზადებული (staged).
3. ვერსიათა კონტროლის სისტემის საცავი ანუ 'რეპოზიტორი' (Git directory) - სამუშაო დირექტორიაში **და ასევე საცავში მყოფი** ანუ დაკომიტებული (committed).



შეცვლილი ფაზა გულისხმობს, რომ ფაილი განახლდა მაგრამ ჯერ არ არის დაფიქსირებული მისი ეს ცვლილება, იგი არ არის მონიშნული შემდეგ კომიტში შესატანად.

გამზადებული კი გულისმობს, რომ ჩვენ უკვე მოვნიშნეთ მიმდინარე ვერსიის შეცვლილი ფაილები და მზად ვართ ამ ცვლილებების ისტორიაში შესატანად.

დაკომიტებული ფაზა გულისხმობს, რომ მონაცემები შენახულია ჩვენს ლოკალურ მბ-ში.

ამ ყველაფრის საფუძველზე Git-ის მუშაობის პრინციპი შეიძლება დაიყოს შემდეგ საფეხურებად

1. ვცვლით პროექტის ამა თუ იმ ფაილს.
2. ვახდენთ ამ ფაილის მონიშვნას, ინდექსაციას, სხვა სიტყვებით : ვანიჭებთ სტატუსს **'შევიდეს შემდეგ კომიტში'**.
3. ამის შემდეგ ვაკეთებთ კომიტს, ეს კი ნიშნავს, რომ ლოკალურ ბაზაში შეიქმნება პროექტის კიდევ ერთი ახალი ვერსია.

ეს ყველაფერი ხდება **ლოკალურად**, ჯერ არ ვართ დაკავშირებული **დისტანციურ საცავთან** (მაგ: Github, Bitbucket ...)

4. Git ინსტალაცია

Windows

Windows ოპერაციულ სისტემაში Git-ის ინსტალაცია საკმაოდ მარტივია, საინსტალაციოს გადმოწერა შესაძლებელია შემდეგი მისამართიდან

> `http://msysgit.github.io`

დაინსტალირების შემდეგ ხელმისაწვდომი იქნება Git-ის როგორც ბრძანებათა-ველის ვერსია (command-line), ასევე სტანდარტული სამომხმარებლო ინტერფეისი (GUI - Graphical user interface).

5. Git სამუშაო გარემოს მოწყობა

ინსტალაციის შემდეგ საჭიროა შევქმნათ სამუშაო გარემო, რისთვისაც უნდა მოვაგვაროთ რამოდენიმე საკითხი, ეს მოგვიწევს ერთჯერადად, მხოლოდ დასაწყისში.

კონფიგურაციული პარამეტრების სანახავად, Windows ოპერაციულ სისტემაში, Git-ს ჭირდება **.gitconfig** ფაილი, რომელიც, როგორც წესი განთავსებულია ხოლმე ან აქ

```
C:\Documents and Settings\$USER
```

ან აქ

```
C:\Users\$USER
```

მომხმარებლის იდენტურობა

პირველი, რაც ინსტალაციის შემდეგ უნდა გავაკეთოთ ეს არის, რომ განვსაზღვროთ Git-ის მომხმარებლის სახელი და ელ_ფოსტა, ამის გაკეთება აუცილებელია რადგან ყველა Git კომიტი (ანუ ცვლილებების საცავში ასახვის დავალება) იყენებს ამ ინფორმაციას. სისტემამ უნდა იცოდეს თუ ვინ გააკეთა კონკრეტული ცვლილება. ამის გაკეთება შესაძლებელია შემდეგი ბრძანებებით

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

ყველა ბრძანება იწყება **git** სიტყვით. თუ ბრძანებაში არ მივუთითებთ **--global** პარამეტრს, მაშინ იდენტურობის განსაზღვრა მოგვიწევს სისტემის ყოველი ახალი გამოყენებისას. მონაცემების განსაზღვრის შემდეგ თუ გვსურს, რომ კონკრეტული სეანსის დროს, სხვა ელ_ფოსტითა და სახელით ვიმუშაოთ მაშინ იგივე ბრძანებები უნდა გამოვიყენოთ **--global** პარამეტრის გარეშე და მივუთითოთ სასურველი მონაცემები (ელ_ფოსტა, სახელი).

პარამეტრების გადამოწმება

პარამეტრების გადასამოწმებლად გამოიყენება **git config --list** ბრძანება, ის გამოიტანს დაახლოებით ასეთ შედეგს

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

6. საცავი (repository)

რა არის 'საცავი' ?

მარტივად, რომ ვთქვათ საცავი (რეპოზიტორი) არის ადგილი, სივრცე, სადაც ინახება ჩვენი პროექტის სხვადასხვა მდგომარეობები ანუ ვერსიები (ინგ: repository - საცავი, შემნახველი).

როგორ შევქმნათ საცავი ?

მანამ სანამ გავარკვევთ, თუ როგორ შეიძლება შევქმნათ საცავი, ჩამოვაყალიბოთ თუ რას ნიშნავს საცავის შექმნა. საცავის შექმნა გულისხმობს, რომ შეიქმნება სივრცე სადაც გაერთიანდება ჩვენი პროექტი და ვერსიათა კონტროლის სისტემა Git, ანუ ამ სივრცეში ერთმანეთთან დაკავშირებული იქნება ეს ორი რამ.

Git საცავის მიღება შესაძლებელია ორნაირად:

1. შეგვიძლია მოვახდინოთ ჩვენი პროექტის Git სისტემაში იმპორტი, ანუ პროექტის საქაღალდეში გავაკეთოთ Git სისტემის ინიციალიზაცია.
2. მოვახდინოთ სხვა სერვერზე არსებული საცავის კლონირება.

Git-ის ინიციალიზაცია მიმდინარე დირექტორიაში

თუ გვსურს რომ პროექტი დავაკავშიროთ Git-თან, ბრძანებათა ველის (ტერმინალის) მეშვეობით უნდა გადავინაცვლოთ პროექტის დირექტორიაში და ავკრიფოთ ბრძანება

```
$ git init
```

ეს ბრძანება დირექტორიაში შექმნის ახალ დირექტორიას სახელად - **.git**, რომელიც შეიცავს ვერსიათა კონტროლის სისტემის მუშაობისთვის საჭირო ფაილებს, თუმცა ჩვენი პროექტი **ჯერ კიდევ არ იმყოფება** ვერსიათა კონტროლის ქვეშ. ამისათვის უნდა გავაკეთოთ საინიციალიზაციო კომიტი, კომიტის გასაკეთებლად კი, როგორც პირველ თავში აღვნიშნეთ, ჯერ უნდა მოვახდინოთ პროექტის ფაილების ინდექსაცია, გამზადება. კომიტში შესატანი ფაილების განსაზღვრა ხდება **git add** ბრძანებით.

```
$ git add *
$ git commit -m '- საინიციალიზაციო კომიტი'
```

ახლა უკვე გვაქვს Git საცავი გამზადებული ფაილებითა და საინიციალიზაციო კომიტით (ამ ბრძანებათა შესახებ ვისაუბრებთ ოდნავ მოგვიანებით).

საცავის კლონირება

უკვე არსებული საცავის ასლის მისაღებად გამოიყენება **git clone** ბრძანება. ამ ბრძანების გაშვებისას ხდება ყველა ფაილის ყველა ვერსიის გადმოწერა. git clone ბრძანებას უნდა მიეთითოს გადმოსაწერი საცავის URI მისამართი

```
$ git clone [url]
```

```
$ git clone git://github.com/schacon/grit.git
```

ეს ბრძანება შექმნის დირექტორიას სახელად **grit**, მასში მოახდენს ვერსიათა კონტროლის ინიციალიზაციას და გადმოწერს საცავში არსებულ ყველა ფაილს. თუ გვსურს რომ საცავის გადმოწერა მოხდეს სხვა საქაღალდეში და არა grit-ში, მაშინ ბრძანებაში უნდა მივუთითოთ ამ საქაღალდის სასურველი სახელი

```
$ git clone git://github.com/schacon/grit.git mygrit
```

7. ცვლილებების შენახვა საცავში

მაშ ასე, უკვე გვაქვს Git-საცავი. როგორც კი დავარედაქტირებთ პროექტის რომელიმე ფაილს, გამომდინარე იქიდან, რომ ფაილი შეიცვალა ბოლო კომიტის გაკეთების შემდეგ, Git მას აღიქვამს **შეცვლილად**, ახლა უნდა მოვახდინოთ ამ ცვლილებების ფიქსირება, ინდექსაცია, დაბოლოს - შენახვა, დაკომიტება.

ფაილის მდგომარეობის განსაზღვრა

იმის გასარკვევად თუ პროექტის რომელი ფაილი რა მდგომარეობაშია ვერსიათა კონტროლის თვალსაზრისით, გამოყენება **git status** ბრძანება. თუ საცავის კლონირების შემდეგ ამ ბრძანებას გავუშვებთ, ასეთ შეტყობინებას ვიხილავთ

```
$ git status
On branch master
nothing to commit, working directory clean
```

ეს ნიშნავს, რომ სამუშაო დირექტორია სუფთაა - არც ერთი ფაილის მოდიფიცირება არ მომხდარა. ეს ბუნებრივიცაა - ჩვენ ხომ ახლახანს გადმოვწერეთ საცავი.

პროექტის საქაღალდეში დავამატოთ ახალი ფაილი მაგალითად **README**, და ისევ გავუშვათ **git status** ბრძანება. ვიხილავთ შეტყობინებას

```
$ vim README
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        README

nothing added to commit but untracked files present (use "git add" to track)
```

ფაილის მოქცევა ვერსიათა კონტროლის ქვეშ

ვერსიათა კონტროლის ქვეშ ფაილის მოსაქცევად გამოიყენება **git add** ბრძანება. ჩვენი ფაილის შემთხვევაში ეს ბრძანება ჩაიწერება ასე

```
$ git add README
```

ხოლო თუ გვინდა, რომ შემდეგ კომიტში შევიდეს ყველა შეცვლილი ფაილი, მაშინ კონკრეტული ფაილის სახელის ნაცვლად უნდა ავკრიფოთ **'*'** სიმბოლო

```
$ git add *
```

თუ ახლა ისევ გავუშვებთ **git status** ბრძანებას ვიხილავთ შეტყობინებას

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

        new file:   README
```

ცვლილებების დაკომიტება

ცვლილებების დაფიქსირებისა და ფაილების ვერსიათა კონტროლის მოქცევის შემდეგ საჭიროა ამ ცვლილებების დაკომიტება. ამისათვის გამოიყენება ბრძანება **git commit**, თუ ბრძანებას ასეთი სახით ავკრეფთ:

```
$ git commit
```

ვიხილავთ დაახლოებით ამდაგვარ შეტყობინებას

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       new file:   README
#       modified:   benchmarks.rb
#
~
~
~
```

```
".git/COMMIT_EDITMSG" 10L, 283C
```

ანუ სისტემა ითხოვს ჩვენს მიერ გაკეთებულ ცვლილებების, **სათაურს, აღწერას**, ეს შეგვიძლია გავაკეთოთ **git commit** ბრძანებაში **-m** ოპერატორის დახმარებით (სიტყვა 'message'-ს პირველი ასო). **სასურველია** კომიტს სახელი მივცეთ გაკეთებული ცვლილებების შინაარსიდან გამომდინარე.

```
$ git commit -m "პირველი კომიტი"
```

ამ ბრძანების გაშვების შემდეგ ბრძანებათა ველში ვიხილავთ ინფორმაციას კომიტის შესახებ: რომელ განშტოებაზე (ანუ 'branch'-ზე) გაკეთდა კომიტი, რამდენი ფაილის ცვლილება მოხდა, რამდენი ფაილი დაემატა და ა.შ.

მუშაობის სქემა

თუ გავაერთიანებთ იმ ყველაფერს რაც აქამდე ვთქვით, მივიღებთ Git სისტემის მუშაობის შემდეგნაირ სქემას

მე-5-ე და მე-6-ე საფეხურებზე შემდეგში ვისაუბრებთ.

8. დისტანციური საცავები (gitlab, github ...)

იმისათვის, რომ პროექტზე ვიმუშაოთ სხვებთან ერთად ანუ გუნდურად, საჭიროა გამოვიყენოთ **დისტანციური საცავი**, დისტანციური საცავი, მარტივად რომ ვთქვათ, არის ინტერნეტში შენახული, ჩვენი პროექტის ვერსიების ნაკრები, რომელიც ხელმისაწვდომია სხვებისათვისაც.

გვაქვს თუ არა დამატებული დისტანციური საცავი ?

იმის გასარკვევად, არის თუ არა ჩვენი ლოკალური პროექტი დაკავშირებული, რომელიმე დისტანციურ საცავთან, გამოიყენება **git remote** ბრძანება (ინგ: remote - დისტანციური; დაშორებული). ეს ბრძანება გამოიტანს იმ დისტანციური საცავების 'მეტსახელების' ჩამონათვალს, რომელიც დამატებული გვაქვს.

```
$ git remote
origin
```

თუ ბრძანებას **-v** ოპერატორთან ერთად გამოვიყენებთ, აგრეთვე ვიხილავთ საცავის URL-ებსაც

```
$ git remote -v
origin  https://gitlab.com/VasilNadiradze/git.git (fetch) #წაკითხვის რეჟიმი
origin  https://gitlab.com/VasilNadiradze/git.git (push)  #ჩაწერის რეჟიმი
```

დისტანციური საცავის დამატება

დისტანციური საცავის დამატების სინტაქსი ასეთია

```
$ git remote add [მეტსახელი] [ბმული]
```

კონკრეტული მაგალითი კი ასეთი

```
$ git remote add origin https://gitlab.com/VasilNadiradze/git.git
```

ინფორმაციის ჩამოტვირთვა დისტანციური საცავიდან

ამისათვის გამოიყენება **git fetch** (ინგ: fetch - ამოღება, მიღება) ბრძანება. მისი გამოყენების სინტაქსი ასეთია

```
$ git fetch [საცავის-მეტსახელი]
```

კონკრეტული მაგალითი კი ასეთი

```
$ git fetch origin
```

ინფორმაციის ატვირთვა დისტანციურ საცავზე

ამისათვის გამოიყენება **git push** (ინგ: push - ბიძგი, იმპულსი, ზემოქმედება) ბრძანება. მისი გამოყენების სინტაქსი ასეთია

```
$ git push [საცავის-მეტსახელი] [განშტოების-მეტსახელი]
```

კონკრეტული მაგალითი კი ასეთი

```
$ git push origin master
```

9. განშტოებები (branching)

განშტოებების შექმნა საშუალებას გვაძლევს, მუშაობა გავაგრძელოთ პროექტის ძირითადი ხასისაგან დამოუკიდებლად, ვიმუშაოთ ისე, რომ არ შევეხოთ მას.

განშტოების შექმნა

განშტოების დამატება ხდება **git branch** ბრძანებით (ინგ: branch - ტოტი, განტოტება). მისი გამოყენების სინტაქსი ასეთია

```
$ git branch [განშტოების-სახელი]
```

კონკრეტული მაგალითი კი ასეთი

```
$ git branch testing
```

ეს ბრძანება ახალ განშტოებას შექმნის **ლოკალურ საცავში**. იმისათვის რათა განშტოება შეიქმნას დისტანციურ საცავშიც, ჯერ ლოკალურ სისტემაში უნდა გადავინაცვლოთ ახლადშექმნილ განშტოებაზე

```
$ git checkout testing
```

ახალი განშტოების შექმნა და ავტომატურად მასზე გადართვა შესაძლებელია ერთი ბრძანებითაც

```
$ git checkout -b testing
```

დისტანციურ საცავზე კი ახალი განშტოება შეიქმნება ავტომატურად, მას შემდეგ რაც გავუშვებთ მასზე ინფორმაციის ატვირთვის ბრძანებას

```
$ git push [დისტანციური-საცავის-მეტსახელი] [ახალი-განშტოების-სახელი]
```

ანუ

```
$ git push origin testing
```

შეიქმნა ახალი განშტოება და აიტვირთა ფაილებიც.

განშტოებების სიის სანახავად გამოიყენება შემდეგი ბრძანება

```
$ git branch
```

თუ ამ ყველაფერს გავაერთიანებთ გამოვა, რომ განშტოებათა სისტემის ძირითადი დადებითი მახასიათებელი არის შემდეგი:

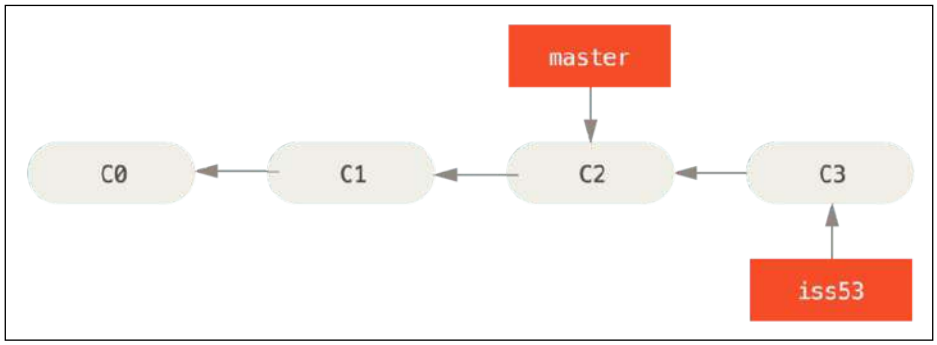
- შესაძლებელია ერთ პროექტზე იმუშავოს რამოდენიმე ადამიანმა ისე, რომ არც ერთმანეთს შეუშალონ ხელი მუშაობის პროცესში (გადამეწერა, გადმომეწერა და ა.შ .. :) და პროექტის ძირითადი ხაზის დაზიანების რისკიც მინუმუმამდე იყოს დაყვანილი.

10. განშტოებათა შერწყმა (merging)

ვთქვათ პროექტზე მუშაობისას გადასაწყვეტი გვაქვს პრობლემა #53 (issue #53). შევქმნათ მისთვის ახალი განშტოება

```
$ git checkout -b iss53
```

თუ ვიმუშავებთ ამ განშტოებაზე და **გავაკეთებთ კომიტებს** (ეს აუცილებელია განშტოებათა შერწყმამდე), გამოვა, რომ განშტოება **iss53** კომიტების თვალსაზრისით წავა უფრო წინ, ვიდრე პროექტის ძირითადი განშტოება - **master**. გვექნება ამდაგვარი სურათი



ახლა საჭიროა, რომ ეს ცვლილებები აისახოს ძირითად საცავშიც (master), ამისათვის უნდა მოხდეს ამ ორი საცავის **შერწყმა** (merging). პირველ რიგში უნდა გადავერთოთ ძირითად საცავზე

```
$ git checkout master
```

ვინაიდან გადავერთეთ პროექტის ძირითად ხაზზე, ამ მომენტისათვის ჩვენს სამუშაო დირექტორიას აქვს ზუსტად ის სახე რაც პრობლემა #53-ზე მუშაობამდე და განშტოება iss53-ის შექმნამდე ჰქონდა, რადგან განშტოებათა შერწყმა ჯერ არ გაგვიკეთებია.

> ეს მნიშვნელოვანია !
>
> როდესაც გადავდივართ ერთი რომელიმე განშტოებიდან მეორეზე, Git სისტემა, სამუშაო კატალოგის მდგომარეობას აბრუნებს იმ სახით, რაც კატალოგს გააჩნდა გადასულ განშტოებაზე ბოლო კომიტის გაკეთების მომენტში. სისტემა ავტომატურად შლის, ამატებს, არედაქტირებს ფაილებს, სწორედ ამიტომ სამუშაო კატალოგის მდგომარეობა ყოველთვის შეესაბამება ბოლო კომიტის გაკეთების მომენტის მდგომარეობას.

ახალ განშტოებაზე მოვახდინეთ ცვლილებები და დავაკომიტეთ ისინი, შემდეგ გადავედით ძირითად განშტოებაზე, ახლა ისღა დაგვრჩენია გავაკეთოთ განშტოებათა შერწყმა. განშტოებათა შერწყმა კეთდება **git merge** ბრძანების მეშვეობით, მისი გამოყენების სინტაქსი ასეთია

```
$ git merge [ახალი-განშტოების-მეტსახელი]
```

ჩვენს შემთხვევაში კი ბრძანებას ექნება ასეთი სახე

```
$ git merge iss53
```

განშტოების წაშლა

ახლა დროა **წავშალოთ** განშტოება **iss53**, რადგან ის უკვე აღარ გვჭირდება, მასში ასახული ცვლილებები უკვე გადმოტანილია ძირითად განშტოება **master**-ზე.

განშტოების წასაშლელად **git branch** ბრძანება უნდა გამოვიყენოთ **-d** პარამეტრთან ერთად (სიტყვა 'delete'-ს პირველი ასო), შემდეგ კი მივუთითოთ წასაშლელი განშტოების სახელი

```
$ git branch -d iss53
```

ეს ბრძანება განშტოებას წაშლის **ლოკალურ საცავში, დისტანციურ საცავზე განშტოების წასაშლელად** კი უნდა ავკრიფოთ შემდეგი ბრძანება

```
$ git push [დისტანციური-საცავის-მეტსახელი] --delete [განშტოების-მეტსახელი]
```

ანუ

```
$ git push origin --delete iss53
```

თუ **-d** ოპერატორის დახმარებით დავაპირებთ ისეთი განშტოების წაშლას, რომელზეც ბოლო ცვლილებები ჯერ არ შეგვინახავს ანუ არ დაგვიკომიტებია, მაშინ სისტემა Git არ მოგვცემს განშტოების წაშლის უფლებას. ასეთ ემთხვევაში **-d** ოპერატორი უნდა მივუთითოთ ასოთა დიდ რეგისტრში :

```
$ git branch -D iss53
```

11. კონფლიქტები განშტოებათა შერწყმისას

არის შემთხვევები, როდესაც განშტოებათა შერწყმისას ყველაფერი მარტივად არ გვარდება: **თუ შევცვალეთ ერთი და იგივე ფაილის, ერთი და იგივე ფრაგმენტი ორ სხვადასხვა განშტოებაში**, მაშინ სისტემა Git ვერ გააერთიანებს მათ, დაფიქსირდება ე.წ **'კონფლიქტი'** და ვიხილავთ ამდაგვარ შეტყობინებას:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git-მა არ შექმნა შერწყმის კომიტი ავტომატურად, მან შეაჩერა პროცესი მანამ სანამ **ჩვენ არ მოვაგვარებთ ამ კონფლიქტს**. იმის სანახავად თუ რომელ ფაილებში შეტანილმა ცვლილებებმა

წარმოქმნა კონფლიქტი, კონფლიქტის წარმოქმნის შემდეგ უნდა გავუშვათ **git status** ბრძანება, შედეგად ვიხილავთ ამდაგვარ შეტყობინებას

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add ..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

ამასთანავე Git სისტემა, შეცვლილი ფაილების კონფლიქტურ ფრაგმენტებს მონიშნავს **სპეციალური ნიშნულებით**

```
<<<<<<< HEAD:index.html
<div id="footer">კონტაქტი : info@goodweb.ge</div>
=======
ჩვენი ელ_ფოსტა : info@goodweb.ge
>>>>>>> iss53:index.html
```

ეს ნიშნავს, რომ, **HEAD**-ის ანუ **master** (ამ შემთხვევაში სწორედ master განშტოებას აქვს HEAD ნიშნული რადგან მასზე ყოფნის დროს გავუშვით შერწყმის ბრძანება) განშტოების რომელიღაც ხაზზე წერია შემდეგი :

```
<div id="footer">კონტაქტი : info@goodweb.ge</div>
```

ხოლო **iss53** განშტოების **ამავე ხაზზე** წერია შემდეგი :

```
ჩვენი ელ_ფოსტა : info@goodweb.ge
```

ვიცით, რომელ ფაილებს შორისაა კონფლიქტი და რომელ ფრაგმენტებში, ახლა უნდა გავაკეთოთ შემდეგი:

- მოვაგვაროთ კონფლიქტები, რაც გულისხმობს შემდეგს: გადავწყვიტოთ თუ რომელი ფაილის ფრაგმენტია კორექტული და სწორი, შემდეგ ეს ფრაგმენტი ჩავსვათ მეორე ფაილშიც.
- დავაფიქსიროთ, ცვლილებები, გავაკეთოთ კომიტები ორივე განშტოებაზე.
- ხელახლა გავუშვათ განშტოებათა შერწყმის ბრძანება.

12. სამუშაო სქემა

საწყის ეტაპზე **ლოკალურ საცავში** (საცავში) გვაქვს კონკრეტული პროექტის **ბოლო ვერსია** - Myproject (ის ვერსია, რომელიც ატვირთულია 'გაშვებულზე').

ამის შემდეგ **ადმინისტრატორი** აკეთებს **დახურულ** დისტანციურ საცავს :

```
https://gitlab.com/Administrator/Myproject
```

და მასზე ტვირთავს ლოკალურ საცავში არსებულ **Myproject** საქაღალდეს, ასევე gitlab.com-ზე აკეთებს სატესტო განშტოებას **'testing** (ე.ი ამ მომენტისათვის პროექტის ბოლო ვერსია უკვე გვაქვს **gitlab.com**-ზე არსებულ ორ განშტოებაზე - **master** და **testing**).

დავუშვათ საჭირო გახდა საიტზე ორი ახალი ფუნქციის დამატება. შესაბამისად **ადმინისტრაორი gitlab.com**-ზე ატვირთულ პროექტში რთავს, შესაბამისი უფლებებით (permissions) აღჭურვილ ორ პროგრამისტს.

თითოეული პროგრამისტი :

1. ლოკალურ საცავში იწერს gitlab.com-ზე არსებულ პროექტის ვერსიას.
2. ლოკალურ საცავშივე აკეთებს შესაბამის განშტოებას.
3. ამ განშტოებაზე ასრულებს დასახულ ამოცანას.
4. აკეთებს შერწყმას თავისსავე, ანუ ლოკალურ საცავში არსებულ ძირითად განშტოებაზე (master)

ამის შემდეგ ორივე პროგრამისტი ტვირთავს პროექტის მისეულ ვერსიას **დისტანციური საცავის ანუ gitlab.com**-ის **testing განშტოებაზე**. თუ ამ მომენტში წარმოიქმნება კონფლიქტები, პროგრამისტები **ადმინისტრატორთან ერთად** წყვეტენ ამ კონფლიქტს, რის შედეგადაც gitlab.com-ზე, განშტოებების თვალსაზრისით გვაქვს ასეთი ვითარება

1. განშტოება master - პროექტის იმ ვერსიით, რომელიც 'გაშვებულზეა'.
2. განშტოება testing - პროექტის იმ ვერსიით, რომელიც შეიქმნა ახალი ამოცანების გადაჭრის შემდეგ.

ამის შემდეგ **ადმინისტრატორი 'testing'** განშტოების ვერსიას ტვირთავს პროექტის **სატესტო** ჰოსტინგზე.

ამის შემდეგ პროექტი სატესტო ჰოსტინგზე იტესტება **ხარისხის მართვის დეპარტამენტის მიერ**

თუ ყველაფერი რიგზეა **ადმინისტრატორი** gitlab.com-ზე აკეთებს master და testing განშტოებების შერწყმას და საჭიროების შემთხვევაში, პროგრამისტებთან ერთად აგვარებს კონფლიქტებს. ასევე, საიტის ბოლო ვერსიას, ანუ იმ ვერსიას რომელიც ახლა გვაქვს gitlab-ის master განშტოებაზე, ტვირთავს პროექტის ძირითად ჰოსტინგზე ('გაშვებულზე').

ბრძანებები

პროექტის მიღება და შექმნა

1. init

git init ბრძანება, **ძირითად კატალოგში** ქმნის ცარიელ Git საცავს (ანუ ძირითადი კატალოგის ქვე-საქაღალდეს) სახელად - **.git** ან ახდენს უკვე არსებულის რეინიციალიზაციას. **.git** საქაღალდეში, **ანუ საცავში**, განთავსებულია ვერსიათა კონტროლის სისტემის მუშაობისათვის აუცილებელი ქვე-საქაღალდეები და ფაილები.

```
$ git init
```

2. clone

git clone ბრძანება აკოპირებს **უკვე არსებულ საცავს ახლად შექმნილ** საქაღალდეში. კოპირდება დისტანციური საცავის ყველა განშტოება და ფაილი. ამ ბრძანების **აუცილებელი** პარამეტრი არის **იმ საცავის მისამართი**, რომლის კლონირებაც გვინდა, ბრძანებას გააჩნია სხვა პარამეტრებიც, რომლებზეც ქვემოთ ვისაუბრებთ.

```
$ git clone [გადმოსაწერი-საცავის-მისამართი]
```

```
$ git clone https://gitlab.com/VasilNadiradze/Git.git
```

ძირითადი ბრძანებები

3. add

git add ბრძანება გამოიყენება სამუშაო დირექტორიაში არსებული **შეცვლილი** ფაილების, **დასაკომიტებლად გამზადებულ ფაილთა არეში** გადასაყვანად. შეიძლება მოხდეს ისე, რომ არ გვჭირდებოდეს ყველა შეცვლილი ფაილის ამ არეში გადატანა, ამისათვის განვიხილოთ **git add** ბრძანების ძირითადი **დამხმარე პარამეტრები**, რომლებიც სწორედ იმას განსაზღვრავენ თუ რომელი ფაილი გადავიდეს ზემოხსენებულ არეში და რომელი არა.

	ახალი ფაილები	შეცვლილი ფაილები	წაშლილი ფაილები	
git add -A	✔	✔	✔	ყველა (ახალი, შეცვლილი, წაშლილი) ფაილის მონიშნვა
git add .	✔	✔	✔	ყველა (ახალი, შეცვლილი, წაშლილი) ფაილის მონიშნვა
git add --ignore-removal	✔	✔	✘	მხოლოდ ახალი და შეცვლილი ფაილის

	ახალი ფაილები	შეცვლილი ფაილები	წაშლილი ფაილები	
				მონიშნვა
git add -u	✖	✔	✔	მხოლოდ შეცვლილი და წაშლილი ფაილის მონიშნვა

როგორც ვხედავთ, '**git add -A**' და '**git add .**' ბრძანებები ერთმანეთის სინონიმებია.

მოვიყვანოთ **git add** ბრძანების გამოიყენების კონკრეტული მაგალითები

```
$ git add Documentation/\*.txt
```

Documentation საქაღალდეში არსებული ყველა ისეთი ფაილის მონიშნვა, რომლის გაფართოებაცაა '**.txt**"

```
$ git add git_*.php
```

.php გაფართოების იმ ფაილთა მონიშნვა, რომელთა დასახელებაც იწყება '**git_**'-ით.

4. status

git status ბრძანება გვიჩვენებს **სამუშაო დირექტორიასა** და **გამზადებული ფაილების არეს** შორის სხვაობას **ფაილების დონეზე**, მისი დახმარებით ვგებულობთ, თუ რომელ ფაილებს აქვთ სტატუსი '**შეცვლილი**', რომლებს სტატუსი '**შენახვისთვის გამზადებული**' და ა.შ

```
$ git status
```

5. diff

git diff ბრძანება გვიჩვენებს **სამუშაო დირექტორიასა** და **გამზადებული ფაილების არეს** შორის სხვაობას **ფაილების შიგთავსის დონეზე**, მისი დახმარებით ვგებულობთ, თუ რომელი ფაილების, რომელ ფრაგმენტებში მოხდა ცვლილებები, ასევე შეტყობინების სახით ვხედავთ თავად ცვლილებებსაც.

```
$ git diff
```

6. commit

git commit ბრძანება ახდენს **გამზადებული ფაილების არის** მიმდინარე მდგომარეობის შენახვას, დაკომიტებას. ამ ბრძანების ძირითადი პარამეტრი არის '**-m**' რომლის შემდეგაც უნდა ავკრიფოთ კომიტის სათაური, აღწერა. მაგალითად:

```
$ git commit -m "პროექტის ბოლო ვერსია"
```

განშტოებები და შერწყმა

7. branch

git ranch ბრძანება გამოიყენება განშტოებების შესაქმნელად ან წასაშლელად. შევქმნათ განშტოება სახელად '**newBranch**'

```
$ git branch newBranch
```

განშტოებათა სიის სანახავად უნდა ავკრიფოთ ბრძანება **$ git branch**

```
$ git branch
* master
newBranch
```

ეს ჩანაწერი ნიშნავს, რომ სულ გვაქვს ორი განშტოება, მიმდინარე კი არის '**master**' განშტოება.

დისტანციურ საცავზე არსებული განშტოებების სიის სანახავად უნდა გამოვიყენოთ **-r** ოპერატორი

```
git branch -r
```

თუ ახალი განშტოების დამატებისას უკვე არსებული განშტოების დასახელებას მივუთითებთ, სისტემა გამოიტანს შეტყობინებას, რომ ასეთი განშტოება უკვე არსებობს, მიუხედავად ამისა, თუ მაინც გვსურს განშტოების შექმნა, მაშინ უნდა გამოვიყენოთ **-f** ოპერატორი

```
git branch -f newBranch
```

განშტოების წასაშლელად უნდა გამოვიყენოთ **-d** ოპერატორი. წავშალოთ ახლადშექმნილი '**newBranch**' განშტოება

```
git branch -d newBranch
```

თუ ისეთი განშტოების წაშლას დავაპირებთ, რომელზეც ცვლილებები არ არის შენახული, სისტემა არ წაგვაშლევინებს მას. ასეთ შემთხვევაში უნდა გამოვიყენოთ **-D** ოპერატორი.

```
git branch -D newBranch
```

8. checkout

git checkout ბრძანება გამოიყენება ერთი განშტოებიდან მეორეზე გადასასვლელად

```
$ git checkout newBranch
```

შესაძლებელია ახალი განშტოების შექმნა და ავტომატურად მასზე გადართვაც

```
$ git checkout -b newBranch
```

9. merge

git merge ბრძანება გამოიყენება განშტოებების შერწყმისათვის. დავუშვათ გვინდა, რომ მოვახდინოთ ძირითადი განშტოების (**master**) და ახალი განშტოების (**newBranch**) შერწყმა. ამისათვის **ჯერ უნდა გადავინაცვლოთ** ძირითად განშტოებაზე

```
$ git checkout master
```

შემდეგ კი გავუშვათ **merge** ბრძანება და მივუთითოთ იმ განშტოების დასახელება, რომლის შერწყმაც გვსურს ძირითად განშტოებასთან

```
$ git merge newBranch
```

პროექტების გაზიარება და განახლება

10. pull

git pull ბრძანება გამოიყენება დისტანციური საცავის (საცავის) ბოლო ვერსიის გადმოსაწერად ლოკალურ საცავში. ბრძანებას პარამეტრად უნდა გადაეცეს დისტანციური საცავის მეტსახელი და და ასევე დისტანციური იმ განშტოების მეტსახელი, რომლის გადმოწერაც გვსურს

```
$ git pull origin master
```

ბრძანება გადმოწერს დისტანციური საცავის ბოლო მდგომარეობას და მოახდენს მის **ავტომატურ შერწყმას ლოკალური საცავის მხოლოდ იმ განშტოებასთან, რომელშიც მიმდინარე მომენტში ვიმყოფებით, დანარჩენი განშტოებები უცვლელი დარჩება**.

11. fetch

თუ გვსურს, რომ გადმოვწეროთ დისტანციური საცავის ბოლო ვერსია და ლოკალურ საცავთან შერწყმამდე ჯერ გადავხედოთ მას, ან შერწყმა მოვახდინოთ მოგვიანებით, მაშინ უნდა გამოვიყენოთ **git fetch** ბრძანება. ბრძანებას პარამეტრად უნდა გადაეცეს დისტანციური საცავის მეტსახელი

```
$ git fetch origin
```

ბრძანება გადმოწერს დისტანციურ საცავზე ასახულ ბოლო ცვლილებებს და შეინახავს მას, მაგრამ **'pull' ბრძანებისაგან განსხვავებით არ მოახდენს ავტომატურ შერწყმას მიმდინარე სამუშაო განშტოებასთან**. ისმის კითხვა : როგორ მოვახდინოთ ამ ბრძანებით გადმოწერილი ცვლილებების ლოკალურ საცავთან შერწყმა ? ამისათვის ჯერ უნდა შევამოწმოთ თუ რა განშტოებები გადმოწერა დისტანციური საცავიდან **git fetch** ბრძანებამ, ამისათვის **git branch** ბრძანება უნდა გამოვიყენოთ **-r** პარამეტრთან ერთად

```
git branch -r
origin/master
```

ეს ნიშნავს, რომ დისტანციური საცავიდან გადმოიწერა მხოლოდ ერთი განშტოება '**master**', ლოკალურ განშტოებასთან შერწყმა კი მოხდება ამ ბრძანებით

```
git merge origin/master
```

12. push

git push ბრძანება გამოიყენება ლოკალურ საცავზე განხორციელებული ცვლილებების დისტანციურ საცავზე ასატვირთად,

```
$ git push origin master
```

დისტანციურ საცავზე სიახლეების ატვირთვამდე სასურველია დავრწმუნდეთ, რომ მანამ ჩვენ ლოკალურ საცავში ვმუშაობდით ამასობაში დისტანციურ საცავზეც ხომ არ დაფიქსირდა ჩვენგან დამოუკიდებელი ცვლილებები (მაგალითად სხვა პროგრამისტმა ატვირთა რაიმე), ამისათვის **push** ბრძანებამდე უნდა გავუშვათ **pull** ბრძანება, თუ სხვაობა დაფიქსირდა, სისტემა გვეტყვის ამას. თუ ასე არ მოვიქცევით შესაძლებელია ვიხილოთ შეტყობინება

```
Updates were rejected because the tip of your current branch is behind
```

თუ, მიუხედავად იმისა შეიცვალა თუ არა რაიმე დისტანციურ საცავზე, მაინც გვინდა ჩვენი ლოკალური საცავის ვერსიის ატვირთვა, მაშინ **push** ბრძანება უნდა გამოვიყენოთ **-f** პარამეტრთან ერთად

```
$ git push -f origin master
```