

# Git, CI/CD, GitOps

Farid Zarazvand



# Who am I?

## About me

-  **Education**

- Computer Eng. (B.Sc.) @ AUT
- IT / Computer Networks (M.Sc.) @ SUT

-  **Experience**

- DevOps TechLead @ ParsPack (Abrha)
- PaaS Product Owner @ ParsPack (Abrha)

-  **Personal**

- Linux User (Since 2002)
- Apple Fan (Since 2007)

-  **Contact**

- Email: [zarazvand@abrha.com](mailto:zarazvand@abrha.com)
- Telegram: @faridz88



# Ice Breaking



# Trivia Time

# Git

## Day 1

- Git Fundamentals
- GitLab Install
- Creating Snapshots
- Browsing Project History

## Day 2

- Branching and Merging
- GitFlow
- Remote Collaboration
- Rewriting History

# گیت چیه؟

رایج‌ترین سیستم کنترل ورژن

- سیستم کنترل ورژن یا VCS چیه؟
- نگهداری تاریخچه تغییرات روی سورس کد
- کار گروهی روی پروژه Repository ■
- VCS types ■
- **Centralized:** Microsoft Team Foundation, Subversion ■
- **Distributed:** GIT, Mercurial ■
- مزیت‌ها و معماری گیت ■
- توزیع شده ■
- OpenSource ■
- سرعت بالا ■

# نصب و استفاده از گیت

- CommandLine
  - macOS
  - Linux
  - Windows
- Code Editors & IDEs
  - JetBrains
  - VSCode (+GitLens)
  - ...
- Graphical User Interfaces
  - GitKraken



# تنظیمات گیت



## ■ Levels

- System: تمام کاربران
- Global: تمام پروژه‌های کاربر فعلی
- Local: فقط برای پروژه فعلی

```
git config -l --global # list all global variables  
  
git config --global user.name "Farid Zarazvand"  
git config --global user.email "faridzarazvand@gmail.com"  
  
code  
code --wait  
git config --global core.editor "code --wait"  
  
git config --global -e
```



--distributed-even-if-your-workflow-isnt

# Getting Help

- Web: [doc](#)
- Terminal: `git config --help`
- Terminal Short: `git config -h`

Type / to search entire site...

[Latest version ▾](#) git-config last updated in 2.47.1    [Topics ▾](#) [English ▾](#)

## NAME

git-config - Get and set repository or global options

## SYNOPSIS

```
git config list [<file-option>] [<display->  
option>] [--includes]  
git config get [<file-option>] [<display->  
option>] [--includes] [--all] [--regexp] [--  
value=<value>] [--fixed-value] [--default=-->  
<default>] <name>  
git config set [<file-option>] [--type=-->  
<type>] [--all] [--value=<value>] [--fixed-  
value] <name> <value>  
git config unset [<file-option>] [--all] [--
```



# Creating SnapShot

# Creating Snapshots

## Creating Empty Repository

```
mkdir kelaasor-git
cd kelaasor-git
# Make empty git repository
git init
ls
ls -a
open .git
```

# Git Workflow

## local

فضای لوکال که روی اون توسعه می‌دهیم و تغییرات خودمون رو اعمال می‌کنیم

## staging area (index)

فضای بین لوکال و ریموت - این فضای به ما این امکان را می‌دهد که مدیریت بهتری روی تغییراتی که مایل هستیم به ریموت ارسال کنیم داشته باشیم و پیش از ارسال به ریموت فایل‌هایی که تغییر کردند و باید داخل اسنپ شات قرار گیرند و به ریموت ارسال شوند رو قبل از ارسال ببینیم و بازبینی کنیم

## remote

فضای روی سرور گیت

معمولًاً روند کاری‌مون به این صورت هست که ما می‌خواهیم یک یا چند فایل رو تغییر بدهیم و بعد یک اسنپ شات از وضعیت فعلی آن با عنوان کامیت تهیه نموده و به ریموت ارسال می‌کنیم.

# Workflow Example

```
echo "hello" > file1
echo "world" > file2
git add file1 file2 # adds to staging area

# Create a snapshot, staging becomes the same as snapshot we want to send to remote
git commit -m "Initial Commit."

# We do some changes on file1
git add file1
git commit -m "Fixed the bug that..."

# We noticed that we no longer need file2
rm file2
git add file1 # git knows that file deleted
git commit -m "Removed unused code"
```

# Commit

## Consists of:

- ID
- Message
- Date/Time
- Author
- Complete Snapshot of Project

گیت تغییرات و دلتا رو نگه نمی‌داره. بلکه کل اسنپ‌شات رو نگه می‌داره. این طوری خیلی سریع‌تر می‌تونه بین کامیت‌ها سوئیچ کنه بدون این که نیاز باشه تغییرات رو محاسبه کنه. ممکنه برآتون سوال بشه که خوب این طوری که کل دیتا رو نگه می‌داریم فضای خیلی زیادی مصرف نمی‌شه؟ باید بگم نه. چون به صورت هوشمندانه عمل می‌کنه و یک دور کل محتوا رو compress می‌کنه و حین فشرده کردن محتوا تکراری حذف می‌شه.

# Committing Best Practices

## ■ Commit Size Matters:

سایز کامیت نباید خیلی کوچیک یا خیلی بزرگ باشد. اگر خیلی کوچیک باشه History ما شلوغ و ناکارآمد می‌شده و اگر دنبال یک تغییر خاص باشیم به مشکل می‌خوریم. اگر خیلی تغییرات زیادی هم روی یک کامیت ارسال کرده باشیم اگر نیاز باشد revert اش کنیم تمام اون تغییرات همراهش revert می‌شون. به کامیت می‌شده به دید یک اسنپشات یا یک checkpoint نگاه کرد.

## ■ Commit Often:

عدد دقیقی برای ایده‌آل این مورد نمی‌شده گفت، ولی متوسط ۱۰-۵ کامیت در روز برای یک برنامه نویس یک عدد رایجی هست. البته که این بنا به شرایط پروژه می‌توانه کاملاً متفاوت باشد ولی این ۱۰-۵ تا یک مقدار متوسطش هست.

## ■ each commit, single change

ایده‌آل این هست که هر کامیت فقط شامل یک تغییر مشخص باشد مثلاً وقتی داریم یک باگ رو فیکس می‌کنیم یک اشتباه تایپی هم داخل یک فایل دیگه پیدا می‌کنیم، نباید این دو تا رو داخل یک کامیت قرار بدم و هر کدوم رو داخل کامیت جداگانه‌ای باید push کنیم. اگر هر دو رو داخل یک کامیت پوش کنیم به فرض اگر اون کامیت باگ فیکس ما موفق نباشد و بخواهیم ریورت کنیم، اون اشتباه تایپی هم همراهش برمی‌گردد که خوب نیست. هر کامیتی بهتره که یک checkpoint از جایی باشد که کد ما قابل اجرا هست.

## ■ Informative Commit Message:

معمولًا فعل زمان حال به کار می‌رده. مثلاً Fixed the bug نه این بیشتر حالت فراردادی داره. ولی بهتره کل تیم روی یک حالت قرارداد داشته باشن و ازش پیروی کنن.

# Conventional Commits

A specification for adding human and machine readable meaning to commit messages

[Quick Summary](#)[Full Specification](#)[Contribute](#)

# Conventional Commits 1.0.0

# Staging Files

```
echo hello > file1.txt
echo world > file2.txt
git status
git add file1.txt file2.txt # or "*.txt" or "."
echo hello2 >> file1.txt
git status # We have first version of file1
git add file1.txt
```

# Committing

```
git commit -m "short message"
# or
git commit
```

# Committing without staging

```
echo test >> file1.txt
git commit -am "Fix the bug"
```

# Removing Files

```
rm file2.txt # removes just from local. neither stage nor remote  
git status  
git ls-files # files in staging area. file2 still exists  
git add file2.txt  
git status  
git commit -m "Remove Unused Code"  
  
git rm file2.txt # Removes both working directory as well as staging area
```

# Renaming or Moving Files

```
ls  
# file1.txt  
  
# Method 1  
mv file1.txt main.js  
git status  
git add file1.txt  
git add main.js  
git status # recognizes that file has renamed  
  
# Method 2  
git mv main.js file1.txt  
  
git status  
git commit -m "renamed file2 to main.js"
```

# Ignoring Files

ما روی هر پروژه‌ای ممکنه یک سری فایل داشته باشیم که نخواهیم روی پروژه گیت ما قرار بگیرند. مثلاً فایل کانفیگ که حساسه و ممکنه داخلش پسوردی وجود داشته باشه، یا فایل‌های تست و لاغ و فایل باینری بیلد شده پروژه و ...

```
mkdir logs  
echo hello > logs/dev.log  
echo logs/ > .gitignore  
code .gitignore  
# logs/  
# main.log  
*.log
```

```
git status
```

```
git add .gitignore
```

```
git commit m "Add gitignore"
```

این نکته رو در نظر داشته باشید که این فقط وقتی کار می‌کنه که هنوز فایل رو به گیت اضافه نکرده باشد. اگر اول فایل رو add & commit کنید و بعد داخل gitignore بگذارید، گیت همچنان اون رو track می‌کنه.

# Ignoring Files Demo

```
mkdir bin  
echo hello > bin/app.bin  
git status  
  
git add . # accidentally add all files  
git commit -m "Add bin."  
# everytime we compile, git says app.bin has changes. this doesnt make sense.  
  
code .gitignore # add bin/  
git status  
echo helloworld > bin/app.bin  
  
git ls-files # shows staging area  
git rm -h  
git rm --cached bin/  
git rm --cached -r bin/  
git status  
git commit -m "Remove the bin directory that was accidentally committed."
```

gitignore repo



# Short Status

ستون سمت چپ staging area

```
echo sky >> file1.js  
echo sky >> file2.js  
git status  
git status -s
```

ستون سمت راست working directory

```
git add file1.js  
git status -s
```

```
echo ocean >> file1.js  
git status -s
```

```
git add file2.js  
git status -s  
# A shows newly added
```

چون روی working directory تغییر کرده وضعیتش رو نشون می‌ده و قرمز هست. داخل staging area هم اصلاً نیستش و به خاطر همین سمت چپ اصلاً چیزی نشون نمی‌ده.

برای آن file1 staging مودیفاید سیز شد. سمت راست هم چیزی نداریم چون چیزی نیست که به استیج منتقل بشه.

# Viewing the Staged & Unstaged Changes

مشاهده محتوای تغییر کرده که قرار هست push شود:

```
git diff --staged
```

تغییراتی که روی stage هست و قرار هست پوش بشه رو نشون می‌ده. به صورت دیفالت از ابزار diff لینوکس و با همون فرمت برای خروجی استفاده می‌کنه. البته این ابزاری که برای گیت استفاده می‌کنه قابل تنظیم و تغییره که جلوتر می‌بینیم. توضیحات خروجی — و +++ تغییرات قدیمی با — و جدید با +++ نشون داده می‌شن. یه جورایی — ها قراره حذف بشن و ++ ها اضافه بشن. هر چانک با این قسمت شروع می‌شه.

```
@@ -1,3 +1,5 @@
```

فایل قدیمی از خط ۱ به میزان ۳ خط نمایش داده شده. فایل جدید از خط ۱ ۵ خط بعدی نمایش داده شد.

```
git diff # comparing working directory with staging area
git status -s
echo 'test' >> file1.js
git status -s
git diff
```

کلا دیف با محیط بعدیش به صورت دیفالت مقایسه می‌کنه.

# Diff Tools

- KDiff3
- P4Merge
- WinMerge (Windows only)
- VSCode

از طریق این کانفیگ می‌توانیم تعیین کنیم که گیت از کدوم ابزار برای diff استفاده کنه.

```
git config --global diff.tool vscode # name
# Setting Command to Use for diff
git config --global difftool.vscode.cmd "code --wait --diff $LOCAL $REMOTE"
git difftool
git difftool --staged
```

ادیتورها و ide ها معمولا خودشون قابلیت گیت دارند و diff رو داخل خودشون نمایش می‌دن و معمولا از طریق کامند نیاز نمی‌شه.

# Viewing the History

## مشاهده تاریخچه کامیت‌ها

```
git log
```

.Head: is a reference to the current branch

از طریق پوینتر head هست که گیت می‌فهمه آن روی کدوم برنج هستیم.

```
git log --oneline
```

```
git log --reverse
```



# Viewing a commit

در قسمت قبل دیدیم که چطور می‌توانیم یک لیست از کامیت‌ها را داشته باشیم. اگر لیست تغییرات یک کامیت مشخص رو بخواهیم چطور می‌توانیم؟ با استفاده از کامند `show` می‌توانیم تغییرات یک کامیت مشخص رو لیست کنیم.

```
git show d083 # only first few characters that are unique and there is no ambiguity
git show HEAD
git show HEAD~1

# What if we want the last version of a file, not just diffs?
git show HEAD~1:.gitignore # Exact version of file

# list all the files that are in a commit
git ls-tree HEAD~1 # files are stored as blobs and directories as trees

git show BLOBID
git show TREEID
```

# git show objects

- commits
- blobs (Files)
- Trees (Directories)
- Tags



# Unstaging Files

پیشنهاد: review فایل‌های تغییر کرده در stage پیش از کامیت

```
git status -s
```

مثلاً فرض کنید تغییراتی که روی file1 داشتیم و git add کردیم و روی استیج اضافه شده، نباید روی کامیت بعدی باشن چون مربوط به تسك دیگه‌ای بوده و فقط تغییرات2 باید کامیت بشه. چطوری می‌تونیم اون git add که زدیم رو undo کنیم؟ زمان قدیم از کامند reset استفاده می‌شد که ابهام داشت، ولی روی نسخه‌های جدیدتر گیت کامند restore هم اضافه شد که این برگرداندن فایل رو به شکل شفاف‌تری انجام می‌ده.

```
git restore --staged file1.js # or . or pattern or ...
git status -s
```

کامند restore آبجکت رو از محیط بالایی خودش روی محیطی که تعیین کردیم برمی‌گردونه. مثلاً اینجا که staged گذاشتیم یعنی اینجا می‌خواهیم ریستور بشه یعنی file1 که روی staging هست رو با بالایی خودش (که می‌شه آخرین کامیت برنج فعلی یا همون head) جایگزین کن.

```
git restore file2.js
git status -s
```

اینجا2 جدید هست و قبل از کامیت نشده. اگر اون رو روی staging ریستور کنیم چه اتفاقی می‌افته؟ خوب روی گیت می‌بینیه روی آخرین کامیت اصلاً وجود نداشته، پس پاکش می‌کنه و روی محیط local development هم به عنوان untracked file قرار می‌گیره.



# Discarding Local Changes

شرایطی که از تغییراتی که روی محیط لوکال دادیم منصرف شدیم و بخواهیم اون به شرایط قبلی برگردانیم. مثلاً روی لوکال تغییراتی انجام دادیم و تست کردیم و دیدیم که کار نمی‌کنه. اگر فقط یک تک فایل رو بخواهیم برگردانیم از کامند ریستور به working directory شکل زیر استفاده می‌کنیم. دیفالت کامند ریستور به هست.

```
git restore file.js # copy from stag  
git restore . # copy all files from  
git status -s # file2 still here as  
# how can we tell git to clean envir  
git clean # fatal. because dangerous  
git clean -h  
git clean -fd  
git status -s
```



# Restoring a file to an earlier version

ممکنه شرایطی داشته باشیم که فقط یک تک فایل را بخواهیم برگردانیم به کامیت‌های قبلی و کل پروژه لازم نباشه که برگردد به اون کامیت. یا به فرض فایلی به اشتباه روی کامیت قبل حذف شده باشه و ما نیاز داشته باشیم که از کامیت‌های قبل ترش اون را برگردانیم. تو این شرایط چی کار می‌کنیم؟

```
git rm file1.js
git status -s
git commit -m "Delete file1.js"
git log --oneline
git restore -h
# by default restores from next envir
# Here without specifying source, git
git restore --source=HEAD~1 file1.js
git status -s
```

# Creating Snapshots using VSCode

# Browsing Project History

# Viewing the History

```
# --stat number of lines changed
git log --oneline --stat

# Exactly what lines changed
git log --oneline --patch

# Last three commits
git log --oneline -3

# Filter by author
git log --oneline --author="Farid Zarazvand"

# Filter by date
git log --after="2020-08-17"
git log --after="yesterday"
git log --after="one week ago"
```

# Browsing Project History (Cont.)

```
# Search in commits
git log --oneline --grep="GUI"
git log --oneline -S"hello()" --patch

# all commits from dfa to a34
git log oneline dfa...a34ab

# all commits that modified this file. anti ambiguous
git log --oneline -- toc.txt
git log --oneline --patch -- toc.txt
```

## Formatting log output

```
git log --pretty=format:"%Cgreen%an%Creset committed %h or %H on %cd" # author name
```

# Creating aliases

```
git config --global alias.lg "log --pretty=format:'%an Committed %h'"  
git config --global -e  
git log  
  
git config --global unstage "restore --staged ."  
git unstage
```

# Viewing a Commit

```
# info, diff, ...
git show HEAD~2

# Final version of exact file
git show HEAD~2:sections/changes.txt

# Only name of files
git show HEAD~2 --name-only

# List of changed, modified files
git show HEAD~2 --name-status
```

## Viewing the changes across commits

```
git diff HEAD~1 HEAD
git diff HEAD~1 HEAD fixbug.txt

# list of files
git diff HEAD~1 HEAD --name-only
# list of modified, added files
git diff HEAD~1 HEAD --name-status
```

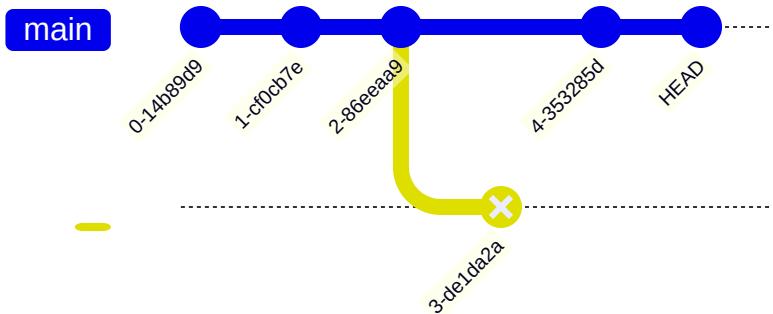
# Checking Out a Commit

زمانی که بخواهیم working directory را به یک اسنپشات (کامیت) مشخص در گذشته برگردانیم.

```
git log --oneline  
git checkout 6ad5b  
# you get a warning saying you are in 'detached HEAD' state
```

با اضافه شدن هر کامیت جدید پوینتر HEAD و master هم یکی به جلو حرکت کرده و به آخرین کامیت اشاره خواهند کرد.

زمانی که به گذشته checkout کردیم، نباید کامیت جدید بزنیم. به دلیل اینکه این کامیت به هیچ برنچی اضافه نمی‌شود و به صورت خودکار توسط garbage collection حذف خواهد شد.



# Finding bugs using bisect

زمانی که یک باگی پیدا می‌شه، اگر بخواهیم پیدا کنیم که کدوم تغییر روی کدام کامیت باعث ایجادش شده، احتمالاً خیلی زمان زیادی بخواهد که دونه چک کنیم. با استفاده از `bisect` می‌توانیم خیلی سریع‌تر پیدا ش کنیم.

```
git bisect start

git bisect bad

git log --oneline
git bisect good 6752f # initial commit (5 revisions to test 3 steps)
git log --oneline # detached head
git log --oneline --all # top is refs/bisect/bad bottom is bisect/good head is middle like quicksort
git bisect good
git bisect good
git bisect good
git log --oneline
git bisect bad
git bisect bad
# shows commid, author, short history changes
# detached head state
git bisect reset # now we are on master
```

# Finding Contributors Using Shortlog

ممکنه نیاز داشته باشیم لیست تمام افرادی که روی پروژه مشارکت داشتن رو استخراج کنیم. یک کامند برای این کار shortlog داریم به اسم

```
git shortlog  
git shortlog -h  
  
# sort by number of commits  
git shortlog -n  
  
# summary  
git shortlog -n -s  
  
# email  
git shortlog -n -s -e  
  
# time based report  
git shortlog -n -s -e --before="" --after=""
```



# Viewing the history of a file

ممکنه نیاز داشته باشیم لیست کامیت‌هایی که یک فایل مشخص رو تحت تاثیر قرار دادن رو در بیاریم.

```
git log toc.txt  
git log --oneline toc.txt  
git log --oneline --stat toc.txt  
git log --oneline --patch toc.txt
```





# Restoring a Deleted File

```
git rm toc.txt  
git commit -m "Removed toc.txt"  
git log --oneline toc.txt # ERROR  
git log --oneline -- toc.txt  
git checkout a87b toc.txt  
git status -s  
git commit -m "Restore toc.txt"
```

# Blaming

زمانی که یک باگی پیش میاد و یک یا چند خط کد درست کار نمیکنن یا مثلاً اشتباه تایپی داشته، ممکنه بخواهیم پیدا کنیم که این خرابکاری کار کی بوده؟ چون ممکنه روی این فایل افراد زیادی کار کرده باشن و کامیت‌های زیادی خورده باشه، از لیست کامیت‌ها و تغییرات که خوب خیلی سخته که پیدا بشه. گیت یک کامند برای این کار داره به نام blame معنیش همون سرزنش می‌شه دیگه.

```
git blame authors.txt

# e-mail address
git blame -e authors.txt

# Limit lines
git blame -e -L 1,3 toc.txt # only first three lines
```

# Tagging

ممکنه نیاز باشه یک سری نقاط خاص یک کامیت‌های به خصوصی رو bookmark کنیم. بیشتر این برای ورژن کارآیی داره. مثلا یک کامیت از main رو بگیم این ورژن v1.0.0 ماست. برای این کار از تگ می‌تونیم استفاده کنیم.

```
git tag v1.0

# Specific Commit
git tag v1.0 ab54

git checkout v1.0
git tag # shows all tags

git tag -a v1.1 -m "My Version 1.1" # Annotated Tag
git tag
git tag -n # for not annotated tags shows commit message

git show v1.1 # Info about tag
git tag -d v1.1
git tag
```

# Viewing History Using VSCode

# Session 2

Git Branching, Collaboration, Rewriting History

# Branching and Merging

# Branches

- انشعاب از محیط فعلی
- ساختار درختی (linked list)

کاربرد:

- ایجاد و توسعه فیچر جدید ایزوله از محیط اصلی و انتقال روی برنج اصلی زمانی که توسعه فیچر انجام شد.

مزایا:

- تمیز و استیبل نگه داشتن برنج اصلی
- سرعت خیلی بالاتر نسبت به scm های دیگر مثل .subversion

در بعضی svn ها برای ایجاد برنج از تمام فایل‌ها یک کپی جدید ایجاد می‌شد که زمان برابر بود. در گیت تنها ID مربوط به آخرین کامیت برنج نگهداری می‌شود. برنج فعلی معادل HEAD است.

# Working with Branches

فرض کنید که یک باگ ریپورت جدید اومده که باید روش کار کنید. خوب نیازه که یک برنج جدید بسازیم.

## ایجاد branch جدید

```
git branch bugfix  
# git checkout -b branch is deprecated
```

## نمایش لیست branch ها

```
git branch # List of branches asterisk shows current branch  
  
# Install ZSH Plus git plugin prevents accidentally committing to wrong branch  
git status
```

تغییر branch: با تغییر هم به برنج مقصد تغییر می‌کند.

```
# in the past: checkout has multiple uses and makes ambiguity  
# git checkout BRANCH  
git switch bugfix  
git switch -c bugfix2 # create and switch to newly created branch
```

# Working with Branches (Cont.)

تغییر نام branch

```
# Rename to more specific name  
git branch -m bugfix bugfix/some-bug
```

حذف branch

```
git branch -d bugfix2
```

مثال

```
vim tox.txt  
git add .  
git commit -m "Fixed some bug"  
git log --oneline  
git switch master  
vim tox.txt # old version  
git log --oneline  
git log --oneline --all --graph  
git branch -d bugfix/some-bug # Delete Branch error: not merged. force with -D
```

# Comparing Branches

```
# show all commits  
git log master..bugfix/some-bug  
  
# Actual Changes  
git diff master..bugfix/some-bug  
  
# short version, compare with current branch which :  
git diff bugfix/some-bug  
git diff --name-status bugfix/some-bug # or name-on
```



# Stashing

همون طور که گفتیم، وقتی branch‌ها را جایه‌جا می‌کنیم، working directory به ای کامیت از آخرین کامیت ای خواهیم بلهش سوئیچ کنیم تغییر پیدا می‌کند.

حالا اگر روی working directory تغییرات کامیت نشده داشته باشیم چی می‌شه؟

این تغییرات خوب ممکنه پاک بشن یا با فایل‌های برجاز بازنویسی بشن. گیت خودش اجازه این کار رو نمی‌ده و زمانی که فایل‌های تغییر کرده روی لوكال داشته باشیم زمان تغییر برجاز ارور می‌ده.

كاربردها:

- تغییر برجاز زمانی که لوكال تغییرات کامیت نشده داریم.
- تست سریع بین دو حالت متفاوت بدون نیاز به کامیت کردن.

code fixbug.txt

```
# Do some edits on file

git switch bugfix/some-bug # error
# We dont want to commit
```

# Stashing (Cont.)

کردن یک قابلیت در گیت هست که یک فضای موقت در اختیار ما می‌گذاره تا بتونیم فایل‌های working directory و همین طور index (staging area, cached) را داخل اون به صورت موقت نگه داریم.

فضای stash کاملاً لوکال هست و داخل تاریخچه گیت هم ثبت نخواهد شد.

**ذخیره کردن:** تغییرات لوکال و استیج داخل stash ذخیره می‌شود و working directory به آخرین کامیت برمی‌گردد.

```
git stash
```

لیست stash ها

```
git stash list
```

آخرین stash را اعمال می‌کند.

```
git stash apply  
# the same as:  
git stash apply 0  
# or  
git stash apply stash@{0}
```



## Stashing (Cont.)

آخرین stash را خارج و اعمال می‌کند.

```
git stash pop  
# the same as:  
git stash pop 0  
# or  
git stash pop stash@{0}
```

آخرین stash یا یک stash مشخص را پاک می‌کند.

```
git stash drop  
# the same as  
git stash drop stash@{0}  
  
# Clear all stashes  
git stash clear
```

# Stashing (Cont.)

## پیش رفته

```
# Stash some files with custom message  
git stash push -m "Message" file1 file2  
# zsh shows status  
  
# include untracked files  
git stash -u  
  
# only changes in the index (staged files)  
git stash --keep-index
```



# Merging

مرج برای این هست که تغییرات یک برنج رو به یک برنج دیگه منتقل کنیم.

- Fast-Forward
- Three-way merging

# Merge Types: 3-way vs FastForward Merging

# Fast-Forward in Action

```
git log --oneline --all --graph # better representation you can use alias  
# on branch master  
git merge bugfix/some-bug # fast forward
```

## Empty Merge (Force No FastForward)

```
git switch -C bugfix/login # Create and Switch  
vim file1.txt  
git add .  
git commit -m "update file1.txt"  
git log --oneline --all --graph  
git switch master  
git merge --no-ff bugfix/login # even if ff is possible, dont do it and create a new merge commit and bring it to master  
git log --oneline --all --graph  
  
# Always force no ff-merge  
git config ff no # which? current you can use --global for user
```

# 3-way merge in action

```
git log --oneline --all --graph
git switch -C feature/change-pass
git log --oneline --all --graph
echo hello > change-pass.txt
git add .
git commit -m "Build change password"
git log --oneline --all --graph
git switch master
code objectives.txt
git add .
git commit -m "Update objectives.txt"
git log --oneline --all --graph
git merge feature/change-pass
git log --oneline --all --graph
```



# Viewing the merged branches

زمانی که main branch با feature branch مرج شد، می‌توانیم برنج را پاک کنیم.

لیست branch های مرج شده و مرج نشده

```
# List of Merged Branches  
git branch --merged
```

```
# Merged branches can be deleted  
git branch -d bugfix/loginform
```

```
# List of not merged branches  
git branch --no-merged
```

# Merge Conflicts

در صورتی که یک فایل داخل هر دو برنج به دو شکل متفاوت ادیت شده باشد. یا زمانی که یک طرف تغییرات داشته و طرف دیگه اصلاً حذف شده. یا فایل روی هر دو برنج اضافه شده باشد ولی با دو محتوای متفاوت

```
git switch -c bugfix/password
code password.txt
git add .
git commit -m "Update Change password"
git switch master
code password.txt
git add .
git commit -m "Update Password.txt"
git merge bugfix/change-passeword # conflict
git status
code password.txt # Accept incoming, Current, Both, manual edit dont add sth new that is not in any of commits
# Accept cuerrent changed
git add password.txt
git commit -m "conflict resolved"
git log --oneline --all --graph
```

you can use p4merge

# Aborting a merge

```
# back to state before merge and resolutions are gone  
git merge --abort
```

# Undoing a Faulty merge

بعد از مرج شدن پشیمان می‌شویم و می‌خواهیم که به حالت قبلی برگردیم.

```
git log --oneline --all --graph
```

- Reseting to last commit

بازنویسی تاریخچه هست و فقط در صورتی که merge commit ما روی local باشد و هنوز push نکرده باشیم مجاز هست. در صورتی که push کرده باشیم، استفاده از این روش بقیه اعضای تیم با مشکل روبرو خواهد شد.

- Reverting last commit

به جای حذف کامیت آخر، یک کامیت جدید اضافه می‌شود که تغییرات کامیت مرج آخر خنثی شود.



# Reseting last commit

```
git log --oneline --all --graph  
# copy merge commit ID
```

```
git reset --hard HEAD~1
```

```
git log --oneline --all --graph  
# commit has been disappeared
```

```
git reset --hard 765ab54
```

```
git log --oneline --all --graph  
# merge commit recovered
```

اینا رو فقط زمانی باید انجام بدین که هنوز به ریموت پوش نکرده‌یم.

# Reverting Last Commit

```
git log --oneline --all --merge  
  
git revert HEAD  
# error: this commit is a merge but no -m was given  
  
git revert -m 1 HEAD  
# first parent
```

اینجا پارامتر m شماره parent ای هست که جایگزین می‌شود.

- 1 بروزرسانی کردن ایجاد merge روی آن قرار داشتیم.  
(اینجا main)
- 2 بروزرسانی کردن اطلاعاتش را با برنج فعلی مرج کردیم.



# Reset Options

- **Soft:** Only HEAD moves to another snapshot. staging area and working dir will be unaffected  
کامیت را undo می‌کند اما تغییرات را در stage نگه می‌دارد تا مجدد بتوانیم آن را کامیت کنیم.
- **Mixed (default):** also puts snapshot in staging area as well working dir will be unaffected  
کامیت را undo می‌کند اما تغییرات در باقی می‌ماند تا در صورت لزوم مجدد کامیت کنیم.
- **Hard:** put new snapshot to working dir as well (working dir and staging and last snapshot)  
کامیت آخر را به کل هم از استیج و هم از working directory حذف می‌کند.

Option	Working Directory	Staging Area
soft	No	No
mixed	No	Yes
hard	Yes	Yes



# Squash merging

کاربرد: زمانی که روی feature branch تعداد زیادی کامیت ریز خورده که نیاز به نگهداری همه اون ها نیست و تاریخچه برنج اصلی را طولانی و کثیف می‌کند. معمولاً برنچ‌های bugfix که کامیت‌های سعی و خطای زیادی دارند این حالت هستند.

در این صورت همه کامیت‌هایی که روی feature branch هستند را با هم اجتماع می‌گیریم و به صورت یک کامیت به main اضافه می‌کنیم.

```
git switch -c bugfix/bugfix2

# First Commit
echo bugfix >> fixbug.txt
git commit -am "some mesage"

# Second Commit
echo bugfix >> toc.txt
git commit -am "update toc.txt"
git log --oneline --all --graph

git switch master
git merge --squash bugfix/bugfix2
git status -s
# both changes combined are in staging area and no new commit. you should commit yourself
```

# Squash merging (Cont.)

کامیت اضافه شده یک کامیت جدید است که شامل تغییرات هر دو کامیت bugfix branch هست و یک کامیت مستقل و تک parent حساب می‌شود و کامیت وابسته به bugfix branch نیست.

```
git commit -m "fix the bug"  
git branch --merged # not in list. because no  
git branch --no-merged # is here  
git branch -d bugfix/bugfix2 # error  
git branch -D bugfix/bugfix2  
git log --oneline --all --graph # single linear hist
```



# Rebasing

کاربرد: زمانی که برنج مبدا (main) کامیت جدید داشته و نیاز داریم تا feature branch نیز با همان تغییرات آپدیت شود تا در زمان مرج از کانفلیکت سنگین جلوگیری کنیم.

همچنین برای جلوگیری از نیاز به 3 way merge و استفاده از FF Merge برای داشتن تاریخچه خطا و تمیز

با توجه به این که تغییر تاریخچه صورت می‌گیرد، فقط باید زمانی انجام شود که روی آن کار می‌کنیم لوكال است و هنوز پوش نشده است.

```
git switch -c feature/shopping-cart
echo hello > cart.txt
git add .
git commit -m "add cart.txt"
git log --oneline --graph --all # shopping cart branch one commit ahead of master

git switch master
echo hello >> toc.txt
git commit -am "Update toc.txt" # branches will be diverged and 3 way merge or rebasing will be needed
git log --oneline --graph --all
```

# Rebasing (Cont.)

روی هر کدام از branch‌ها یک کامیت جدید قرار گرفته و از هم فاصله گرفتند. حالا می‌توانیم rebase کنیم تا تغییرات main به feature branch منتقل شده و ما آپدیت خواهد شد.

```
git switch feature/shopping-cart
git rebase master # change base of this branch with last commit of master
# rebase done successfully but this is not the case in real world. conflicts may occur
git log --oneline --graph --all # now we have a linear clean history and we can do ff merge

git switch master
git merge feature/shopping-cart # fast-forward
git log --oneline --graph --all
```

این حالت بدوین conflict بود. ممکن است در فرآیند rebase نیز به کانفلیکت بخوریم.

# Rebasing - Conflict Resolution

```
echo ocean > toc.txt
git commit -am "Update toc"

git switch feature/shoppingcart
echo mountain > toc.txt
git commit -am "Write mountain"

# now branches have been diverged and we have conflicting changes
git log --oneline --graph --all
git switch feature/shopping
git rebase master

# conflict
git mergetool
git rebase --continue # we go to the next commit
# or
git rebase --skip # we dont want this commit, go to the next commit
# or
git rebase --abort # branches have been diverged alot and we dont have time to resolve all commits
```



# Cherry-Picking

فرض کنید که فیچر برنج ما دو تا کامیت  $f_1, f_2$  رو داره.  
بعد  $f_1$  یه قابلیتی اضافه کرده که ما می‌خواهیم روی  
مستر بیاریم. بدون اینکه بخواهیم کل  $f_2$  اضافه بشه.  
چون اگر  $f_2$  رو هم می‌خواستیم که حل بود. مرج  
می‌کردیم. ولی فقط تک کامیت رو می‌خواهیم بیاریم روی  
مستر. cherry-pick می‌کنیم.

```
git log --oneline  
git cherry-pick COMMITID
```

```
# conflict  
git mergetool  
git status -s  
git commit
```





# Cherry-Picking - Picking files from another branch

با کل یک commit cherry-pick رو می‌تونستیم روی یک branch اپلای کنیم. حالا شرایطی رو فرض کنید که ما بخواهیم تغییرات فقط یک فایل رو از یک کامیت بیاریم روی کامیت دیگه.

```
git switch -C feature/send-email
echo river > toc.txt
git commit -am "Update toc"
git switch master
git restore --source=feature/send-email toc.txt # ...
git status -s
cat toc.txt
```

# Collaboration

Internal Projects vs. Open-Source Projects

# Creating Github Account

# Cloning Repo

```
git clone URL FOLDER_NAME  
cd  
git log  
# origin/master origin/HEAD  
  
# git names the source repository names that origin  
# where is the master branch on our origin repository  
git switch origin/master # fatal error. we can not switch  
git branch # only master  
  
git remote # list of remote repositories  
git remote -v
```



# Fetching

آخرین کامیت‌ها را از remote دریافت می‌کند اما تغییری روی محیط لوکال انجام نمی‌دهد. صرفاً برای دریافت کامیت‌های جدیدی هست که روی لوکال موجود نیست.

```
git fetch # Downloads new commits from master and puts forward origin/MASTER pointer  
# Our working directory wont be updated.
```

بعد از دریافت، برای اعمال کامیت‌های جدید به صورت زیر عمل می‌کنیم. با توجه به این‌که تاریخچه خطی هست ff-merge انجام می‌شود. در غیر این صورت ابتدا باید conflict را حل کنیم.

```
git merge origin/master  
git log --oneline # we dont have commit here  
git fetch origin master # remote name and branch are optional. if we dont specify, it will download all branches on that  
git log --oneline --all --graph # origin/master has moved forward. but master is one commit behind  
git branch -vv # shows how our local and remote branches have diverged  
git merge origin/master # git merge origin/master linear ff merge  
git log --oneline --all --graph
```

# Pulling

برای اینکه تغییرات ریموت رو روی لوکال بیاریم ما اول fetch میکنیم بعد با origin/master مرج میکنیم. یک کامند داریم که ترکیب این دو را همزمان انجام می‌دهد.

بعضی‌ها این مدل رو نمی‌پسندن چون تاریخچه رو به هم می‌ریزه. لوکال و ریموت هم متفاوت می‌شون. اگر از -- pull استفاده کنیم، برجسته رو روی origin/master ریبیس می‌کنیم. این طوری تاریخچه تمیز خطی خواهیم داشت.

کدام روش بهتره؟ نمی‌شه گفت هر کسی و هر تیمی یه جور پیش می‌رده و هم سلیقه‌ای هست و هم باز بسته به team culture یا policy سازمان داره. گیت لب دیفالت rebase می‌کنیم.

```
echo hello > file1.txt
git add .
git commit -m "Add file1"
git log --oneline --all --graph
git pull
git log --oneline --all --graph
# undo last merge commit
git reset --hard HEAD~1
git log --oneline --all --graph
git pull --rebase
git log --oneline --all --graph
```

# Pushing

اول کامیت‌هایی که روی ریموت نیستند رو می‌فرسته. بعد روی ریموت Master رو می‌بره جلو و در نهایت روی لوکال

رو می‌بره جلو

```
git push origin master # remote name and remote branch name are optional  
git push  
# verify on github
```

فرض کنید شما کامیت B رو روی لوکال اضافه کردین یکی از افراد تیم هم C رو اضافه کرده و push کرده. ولی شما هنوز پول نکردین و روی a هست هنوز اینجا git push بزنید ارور می‌ده چون تاریخچه ریموت و لوکال با هم فرق داره و کامیت آخرشون یکی نیست که این b رو بفرسته اون طرف یک کاری که یک عده انجام می‌دن استفاده از آپشن force -f می‌زن git push هست که اشتباهترین کار ممکنه است. گیت لب دیفالت جلوش رو می‌گیره. این چی کار می‌کنه؟ می‌گه چیزی که روی ریموت هست رو و لش کن، چیزی که من پوش می‌کنم رو قرار بده. کار اون هم تیمی رو کلا پاک می‌کنه و ریموت رو عین لوکال ما می‌کنه. به خاطر همین وقتی چند نفر روی یک برنج کار می‌کنیم. حتماً حتماً قبل از push باید یک pull بزنیم تا آخرین تغییرات رو بگیره و مرج یا ریبیس کنیم بعد تغییرات خودمون رو روی اون بفرستیم به ریموت.

# Storing Credential

اگر بخواهیم گیت پوش بکنیم روی ریموت باید هر دفعه یوزرنیم و پسورد بزنیم. خوب این خیلی سخته. گیت یک کانفیگ داره که کردنشیال رو برای تایمی که مشخص می‌کنیم نگه داره که هر دفعه مجبور نباشیم پسورد بزنیم

```
git config --global credential.helper cache
```

برای ویندوز و لینوکس و مک حالت گرافیکی هم پلاگین‌هایی هست که به صورت دائمی پسورد رو داخل keychain و ... نگه می‌داره.

# Pushing Tags

به صورت دیفالت push تگها ارسال نمی‌شوند. به صورت زیر می‌توانیم تگها را حذف یا اضافه کنیم.

ایجاد تگ جدید

```
git tag v1.0  
git log --oneline
```

ارسال به remote، local و سپس حذف از remote

```
git push origin v1.0  
git push origin --delete v1.0 # delete tag from origin. still in repository  
git tag -d v1.0 # from local
```

# Release Management

on GitHub

# Sharing Branches

برنج هم مثل تگ دیفالت private و local هست. یعنی گیت فرض می‌گیره این برج باشد local خودتون باشه و شما فقط دارین روی اون کار می‌کنید. اگر لازم هست که remote روی branch بفرستید و بقیه اعضای تیم هم به اون دسترسی داشته باشند و بتونن روی اون کار کنند، باید این رو تعیین کنید.

```
git switch -c feature/change-pass  
git push # no upstream branch  
git branch -vv  
git branch -r # remote branches
```

تنظیم branch برای upstream

```
git push --set-upstream origin feature/change-pass  
# or  
git push -u origin feature/change pass  
git branch -vv  
git branch -r # remote branches new remote
```

# Removing Branch

حذف از remote

```
# when we are done, we want to remove this branch  
git push -d origin feature/change-pass  
git branch -vv  
# we still have local branch -> remote is gone
```

حذف از local

```
git switch master  
git branch -d feature/change-pass
```



# Collaboration Workflows

فرض کنید من و علی روی یک فیچر برنج با هم کار می‌کنیم. یک راه اینه که یکیمون برنج رو بسازه و پوش کنه به ریموت. یک راه دیگه هم اینه که روی ریموت مثلاً گیت‌هاب برنج رو بسازیم با `ui` و روی اون کار کنیم.

```
git fetch # gets new branch
git branch # only shows master
git branch -r # we have a remote tracking branch
# we should create a local branch that maps to remote branch
git switch -C feature/password origin/feature/password

cd /home/amy
git clone hjhkjhjlhl
git branch # only master. amy also should create local branch that points to remote branch
git switch -C feature/password origin/feature/password
echo 'pass' > file1.txt
git commit -am "Update file1"
git push
# check on github page
```

# Collaboration Workflows (Cont.)

```
# my machine
git pull
git log ... # feature branch one commit ahead of master
git switch master
git merge feature/change-pass # ff
git log ... # master, origin/feature, feature pointing to same commit but origin master is one commit behind
git push
git log ... # master and origin/master on same location
# check on github
# feature is done, we want to close this branch
git push -d origin feature/password
git branch
git branch -d feature/change-password
git branch
git branch -r

cd /home/amy
git pull
git branch
git branch -d feature/change pass
git branch -r # we still have branch on remote
git remote prune origin
git branch -r
```

# Pull Request

توی پروژه‌های تیمی قبل از اینکه مرج کنیم معمولاً نیاز هست که روی اون مرج صحبت و بحث کنیم، یا جاب CI/CD مون نیاز باشه اجرا بشه و تست‌ها رو انجام بده که ببینیم قابل مرج هست یا نه و همین طور احتمالاً داخل تیم یک نفر که احتمالاً تیم لید هست نیاز هست که کد ریویو انجام بده و در صورت تایید مرج رو انجام بده و اگر اوکی نبود درخواست تغییرات بده. خود گیت داخل خودش چنین مکانیزمی نداره و برای این مورد طراحی نشده. گیت‌هاب این مورد رو تحت عنوان pull request و گیت لب تحت عنوان request این قابلیت رو اضافه کردند.

```
cd /home/amy
git switch -C feature/login
echo hello > file3.txt
git add .
git commit -m "Write hello to file3"
git push -u origin feature/login # first time needs -u
# go to github pull request tab
```

# Resolving Conflicts

اگر برای یک پول ریکوئست کانفلیکت پیش بیاد؟

```
git switch -C feature/logout
echo hello > file1.txt
git commit -am "write hello to file1"
git push -u origin feature/logout

cd /home/amy
echo world > file1.txt
git commit -am "write world to file1"
git push
# github says cant automatically merge
# you can use CLI (exactly like before or use ad resolve cinfglicts)
```

# Collaborating Open-Source Projects

```
git colne ...
git switch -c bugfix
echo hello > readme.md
git commit -am "bug fixed"
git push -u origin bugfix
# go to github page, done and start pull request
```

# keeping a forked repository up to date

نمایش گرافیکی pull & push که contributer و base, origin می‌کنه.

```
git remote
git remote -v
git remote add upstream URL
git remote
git remote -v
git remote rename rupstream base
git remote rm base
# make a commit on base
git fetch base # defaults to origin
git log...
git switch master
git merge base/master
git log... # base/master and local master on same commit
git push # our fork will be updated

# making bugfix branch updtodate
git switch bugfix
git merge master # conflict
```

# Rewriting History

# Rewriting History

چرا نیاز داریم که تاریخچه رو بازنویسی کنیم؟ --> در تضاد با فلسفه ایجاد گیت

ما نیاز داریم بدونیم که چه چیزی تغییر کرده؟ چرا و چه زمانی؟ همون طور که در قسمت best practice های گیت گفتیم، این موارد آنتی پترن هستند:

- متن کامیت نامناسب
- کامیت بیش از اندازه بزرگ
- کامیت بیش از اندازه کوچک

ما یک تاریخچه تمیز و مرتب نیاز داریم که برای هر تغییری اون سه تا سوال رو بتونیم جواب بدیم توی هر کامیت **چه چیزی تغییر کرده و چرا و چه زمانی**.

اگر این موارد هم رعایت نشده باشه، گیت ابزارهایی رو برای ما قرار داده که اصلاحات لازم رو انجام بدیم.

# Rewriting History (Cont.)

- اگر کامیت‌های ما بیش از اندازه کوچک باشند می‌شوند کامیت‌های مرتبط با هم را squash کنیم روی یک کامیت.
- اگر داخل کامیت بزرگ چندین تغییر همزمان اعمال شده باشند می‌توانیم آن را به چند تا کامیت کوچیک‌تر تقسیم کنیم که هر کدام از کامیت‌ها شامل یک تغییر مشخص و واحد باشند.
- می‌توانیم متن کامیت‌ها را ویرایش کنیم و تغییر بدھیم.
- می‌توانیم کامیت‌های به درد نخور رو کامل حذف کنیم.
- می‌توانیم محتوای کامیت را تغییر بدھیم. در هر موردی غیر این اصلاح‌ها تغییر تاریخچه اگر روی ریموت باشند چیزی هست که باید ازش دوری کنیم تا جایی ممکن چون خودش به خودی خود آنتی‌پترنه. فقط باید زمانی ازش استفاده کنیم که یک دلیل محکم داشته باشیم. مثلاً اگر کامیتی به اشتباه او مده توی تاریخچه یا پسورد رو کسی اشتباهی push کرده روی ریموت و این طور موارد.
- باید هم این موارد با کل تیم هماهنگ بشه که آن‌ها هم لوکالشون رو آپدیت کنن.

# Undoing Commits

اگر که آخرین کامیت رو هنوز پوش نکرده باشیم به ریموت که قبلا توضیح دادیم چطور می‌تونیم کامیت رو حذف کنیم با `reset`. اما اگر روی ریموت پوش کرده باشیم ممکنه یک نفر اوون رو `pull` کرده باشه و بر مبنای اوون بخواهد تغییرات بده و کامیت جدیدی بزنه. تو این شرایط باید `revert` استفاده کنیم.

```
git revert HEAD

# --soft removes the commit only
# --mixed Unstages files
# --hard Discards local changes
git reset --hard HEAD~1
git show HEAD # toc.txt has changed
git reset --soft HEAD~1
git status -s
git diff --cached
git log ...
git reset --mixed HEAD # no HEAD~1 because we want to put last commit in staging area
git status -s
git diff
git reset --hard HEAD # local change is gone
git status -s
```

# Reverting Commits

```
# git revert HEAD we can use HEAD~2 or HEAD~3..HEAD
git revert HEAD~3..HEAD # this reverts one by one
git log --oneline --graph --all # last three commits are added as revert. noise in history

# how to do in one commit
git reset --hard HEAD~3
git revert --no-commit HEAD~3.. # simply add required changes in staging area
git status -s
git revert --abort # unhappy with changes
git revert --continue # replace commit message to show all 3 commits are reverted
git log --oneline --all --graph
```

# Recovering Lost Commits

```
# Accidentally
git reset --hard HEAD~6
git log --oneline --all --graph # What hasppened to our commits? are they lost? no with git we are not gonna lose anything
git reflog
# copy identifier next to HEAD@{1}
git reset --hard HEAD@{1}
git log --oneline --graph --all

# for a branch
git reflog show feature
```

# Amending last commit

no need another commit

```
echo cafes >> map.txt
git commit -am "Render cafes on the map"
code map.txt # blue cafes
got add .
git commit --amend # -m "message" git actually created new commit
git show HEAD

# accidentally included a file in last commit. How can we remove it?
git reset --mixed HEAD~1 # resets staging area, but changes stay in local working directory
git status -s
git clean -fd
git add .
git commit -m "message" # i am not gonna use amend option, because we deleted last commit using reset
```

# Amending an earlier commit

```
git log ... # copy commit id
git rebase -i ID # interactive rebase
# change pick to edit. all commits after that will also be recreated. rebasing rewrites history
echo license > license.txt
git add .
git commit --amend
git log --oneline --all --graph # created a new commit unlinear history
# you can use git rebase --abort at any moment to cancel operation
git rebase --continue
git log ... # linear clean history, commit IDs have changed
git checkout # some commit in between
cat ... # our change is here
```

# Removing Secret Using filter-repo

```
pip install git-filter-repo
git filter-repo --path <file_with_secret> --invert-paths
# Use BFG Repo-Cleaner to remove specific content within a file
```

BFG Repo-Cleaner

# Dropping a Commit

```
git log... # copy commit id  
git rebase -i ID^ # parent  
# change pick to drop, or delete the line  
# conflict  
git status -s  
git mergetool  
# deletion or modification? deletion  
git rebase --continue  
git log --oneline --all --graph
```



# Rewording Commit messages

```
git log .. # copy commit id  
git rebase -i ID^  
# replace picks with reword  
git log ...
```



# Re-Ordering Commits

```
git log ..  
# adding a dependency after using it  
git rebase -i # first commir  
# reorder using ALT + UP Arrow in VSCode and save  
git log ...
```



# Squashing commits together

```
git log ...
git rebase -i COMMIT_BEFORE_HASH
# change picks to squash
git log ...

git reflog
git reset --hard #before squash rebase
git log ...
# copy commif id before commits to be merged
# replace pick with fixup
# git gets the message for previous commit and appl:
```

ChatGPT Answer



# Splitting a Commit

```
git log...
git rebase -i ID^
# change pick to edit
git log ... # head is on that commit co
# git reset --soft HEAD~1 # or HEAD^ so changes will be on staged area
# or two steps back and changes on working directory. so we can stage and commit them seperately.
git reset HEAD^ # defaults to mixed
git add package.txt
git commit -m "Update google map sdk vetsion 1.0 -> 2.0"
git log... # divergance
git add license.txt
git commit -m "Add terms of service"
git log ...
git rebase --continue # git will replay in between commtis
git log ...
```



# GitLab CI/CD

آنچه را می‌شنوم، فراموش می‌کنم.

آنچه را می‌بینم، به خاطر می‌سپارم.

آنچه را انجام می‌دهم، درک می‌کنم.

کنفوسيوس



CI/CD

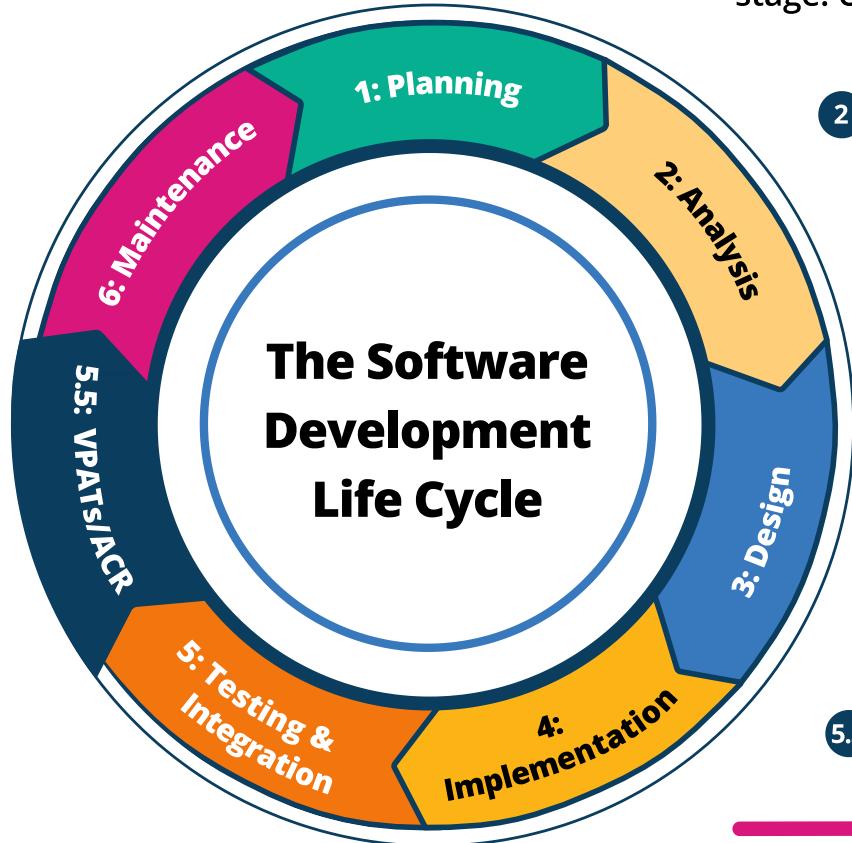
## SDLC یا چرخه عمر توسعه نرم افزار

- توسعه فیچر
- تست
- Build & Package
- Deploy

## محیط‌های پیاده‌سازی

- Local development
- Testing
- Stage (QA Testing/Preview/Pre-Production)
- Production





- 1 Integrating insights into planning stage: User Stories/Code
- 2 Accessibility Audits: Manual and Machine Detectable
- 3 Prioritization and integration of inputs into design phase
- 4 New feature/functionality builds: ARC KnowledgeBase & ARC Tutor, code annotations
- 5 ARC API Automated Testing and policy compliance
- 5.5 VPATs/Accessibility Conformance Reports
- 6 ARC Monitoring Monthly Scanning and User Generated Content

# ایجاد یک CI/CD Pipeline

هر زمان برنامه‌نویس کد را روی ریموت مرج کرد

- تست‌ها اجرا شوند.
- آیمیج داکر ساخته شود.
- به image repository, push شود.

## ساختار تعریف GitLab CI pipeline در

- فایل با نام `gitlab-ci.yml` در root پروژه
- فرمت YAML
- تشکیل شده از تعدادی Job (مثال: (... ,Test, Build, Push, Deploy)



# Gitlab CI Runner Modes

- Runner
  - Shell Executor
  - Docker Executor
  - Virtual Machine Executor
  - Kubernetes Executor
  - Docker Machine Executor
  - SSH Executor

# نصب پیش نیازها

- Docker (+ Docker Compose Plugin)
- GitLab (CE version)
- GitLab Runner (dind method) [Repo Link](#)
- Register Runner on GitLab Home Page >  
Admin > CI/CD > Runners > New Instance  
Runner
- Create New Repository
- Add Pipeline



# فرمت Job

```
run_tests:  
before_script:  
  - echo "Preparing test data"  
script:  
  - echo "Running Tests"  
  - echo "Running more Tests"  
after_script:  
  - echo "Running Cleanup"
```



# استیج

در صورتی که برای جاب استیج تعریف نشود از استیج Test به صورت پیش فرض استفاده می شود.

```
stages:  
- test  
- build  
- deploy  
  
run_unit_tests:  
  stage: test  
  
run_lint_tests:  
  stage: test  
  
build_image:  
  stage: build  
  
push_image:  
  stage: build  
  
deploy:  
  stage: deploy  
  script:  
    - echo "Deploying to server"
```

## سطح تسک های اجرا شده در پایپلاین

Stage .1

Job .2

## زمان بندی اجرای جاب ها در استیج

- جاب های داخل یک استیج به صورت موازی اجرا می شوند.
- استیج ها به صورت تک به تک و سریال اجرا می شوند.

# Job Dependency

با این که داخل یک استیج هستند، اما به صورت سریال اجرا می‌شوند.

به صورت دیفالت پایپلاین و تمام جاب‌ها برای تمام اتفاقات (کامیت، مرج، ...) روی تمام برنج‌ها اجرا می‌شوند. به صورت زیر می‌توانیم اجرای جاب را به یک برنج مشخص محدود کنیم.

```
push_image:  
  needs:  
    - build_image
```

# Limit job to branch

```
build_image:  
  only:  
    - main
```

# Workflow Rules

```
workflow:  
  rules:  
    # https://docs.gitlab.com/ee/ci/variables/predefined_variables.html  
    - if: $CI_COMMIT_BRANCH != "main" # logic, condition branch is not main  
      when: never # logic  
    - when: always
```

## Predefined Variables



# Trigger Pipeline on Merge Requests

```
workflow:
  rules:
    - if: $CI_COMMIT_BRANCH != "main" && $CI_PIPELINE_SOURCE != "merge_request_event"
      when: never
    - when: always

build_image:
  only:
    - main

push_image:
  only:
    - main

deploy_image:
  only:
    - main
```

# Custom Variables

علاوه بر اون variable هایی که گیتلب دیفالت در اختیار ما می‌گذاشت، ما می‌توانیم خودمون هم variable تعریف کنیم.

مثال از مورد استفاده:

پایپلاینی داریم که برای همه مايكروسرويس‌های اپليکيشن‌مون اجرا می‌شه. می‌خواهیم که همون کد پایپلاین رو به صورت مشترک استفاده کنیم فقط نام سرويس عوض بشه.

اين نام رو می‌تونیم پارامتر در نظر بگیریم و پایپلاین که اجرا می‌شه بهش پاس بدیم.

```
run_unit_test:  
  script: echo "Running for $MICRO_SERVICE_NAME"
```

# File Variable

ممکن است variable ما از نوع key/value باشد و یک فایل کانفیگ باشد. مثال: فایل environments.ts یا env file

```
deploy_image:  
  script:  
    - echo "Deploying new docker image to $DEPLOY_ENVIRONMENT using the following configuration file $PROPERTIES_FILE"  
    - cat $PROPERTIES_FILE
```

# Custom Variables (Cont.)

```
build_image:
  script:
    - echo "Tagging the docker image docker.io/my-username/myapp:v1.0"
push_image:
  script:
    - echo "Pushing the docker image docker.io/my-username/myapp:v1.0"
deploy_image:
  script:
    - echo "Deploying the docker image docker.io/my-username/myapp:v1.0 to $DEPLOY_ENVIRONMENT using the following config"
```

## Local in job

```
build_image:
  variables:
    image_repository: docker.io/my-username/myapp
    image_tag: v1.0
```

## Global variable in pipeline

```
variables:
  image_repository: docker.io/my-username/myapp
```



# Custom Docker Image

- تنظیم ایمیج دیفالت در تنظیمات Runner

```
[runners.docker]  
  image = "dind"
```

- Pipeline Global

```
# global  
image: node:17-alpine
```

- Job Level

```
# Job Level  
run_lint  
  image: debian:12
```





# Nexus Install using Docker Compose

---



# GitOps

# چیست؟ GitOps

Infrastructure as Code done right with all the best practices

- GitOps manages the whole stack
  - Cluster and application versioned configuration
  - Security and policy enforcement
  - Monitoring and observability
  - Continuous deployment of workloads



# Infrastructure as Code چیست؟

- Define Infrastructure as Code instead of creating it manuallly.
- Infrastructure can be much easier easily reproduced and replicated.
- IaC evolved as X as code
  - Infrastructure as coed
  - Network as Code
  - Policy as Code
  - Configuration as Code
  - Security as Code

# Infrastructure as Code چیست؟ (.cont)

به جای این که زیرساخت را به صورت دستی و با کلیک کردن روی پنل aws یا کلادپروایدر دیگه ایجاد کنیم، با استفاده از:

- Terraform
- Ansible
- Kubernetes Manifests

اون ها رو از روی کد می سازیم. که یک سری yaml file پلتفرم و تنظیمات ما رو توصیف می کنن.





# پیاده سازی Code

راه اول:

- توسعه روی محیط لوکال
- تست روی محیط لوکال
- تست روی محیط استیچ و اجرا روی محیط اصلی
- `kubectl apply`
- `ansible-playbook`
- `terraform apply`

راه بهتر:

نگهداری کدها روی گیت.



# پیاده سازی Code (cont.)

مزیت‌های استفاده از گیت در :X as Code

- امکان استفاده از IaC Version Control روی
- مشارکت و دسترسی سایر اعضای تیم

اما از Branch و Review ها و ساختار gitflow و pull request خبری نیست و مستقیم روی main branch کامیت می‌زنیم.

# پیاده سازی (cont.)

تا اینجا با اینکه laC را پیاده کردیم، اما هنوز بخش زیادی manual باقی مونده و ناکارآمد هست

معایب این روش:

- No Review/Approval Process
- No pull/merge requests
- No Code Review
- No Collaboration
- No Automated Tests: (Invalid Yaml File, Typos, Breaking Infrastructure, Breaking APP Environment, ...)
- Infrastructure Updates are not automated
- Everyone needs access to infrastructure
- Hard to trace who executed what?

# Git - Single Source of Truth

- Environment is always in sync with that repository
- Increased security --> Only CD has access to Infrastructure
- Everyone can propose changes through pull requests



# GitOps

مجموعه‌ای از:

- ساختارها
- الگوها
- ابزار

برای پیاده سازی بهتر محیط اجرای .DevOps

## اصول GitOps

- با کدهای زیرساخت دقیقاً مانند کدهای اپلیکیشن رفتار شود.
- کدهای توصیف کننده زیرساخت همه declarative باشند. (... ,terraform, ansible, k8s)
- از گیت به عنوان source of truth استفاده شود.
- در واقع ترکیب Infrastructure as Code با CD

# چطور کار می‌کند؟ GitOps

در GitOps ما یک Git Repository مجزا برای انسپیل و کوبرنتیز و ... داریم و این خودش CI/CD کامل و مجازی خودش رو دارد.

- در GitOps ما Desired State سیستم‌ها را داخل گیت نگه‌داریم.
- تغییرات اول روی branch‌ها اعمال می‌شود.
- روی برنج می‌توانیم تغییرات رو review کنیم و بعد merge approve کنیم که با main merge بشه.
- از اینجا به بعد یک اپراتور وضعیت Desired State را بررسی می‌کنه و تغییرات لازم را اعمال می‌کنه.



# GitOps Tools

- Store Code Management / Config (on Git)
  - Must be declarative. Ansible with idempotency, Terraform and kubernetes are good choices
- CD Automation
  - Argo and Flux mostly used
- Runtime Environment (K8s)

# CI & CD جدا کردن

فرآیند Deployment اپلیکیشن باید جدا از CI باشد.

- **Separation of Concerns:**

Developers Release, Operators Deploy -->

Security: Different Attack surface

- **Many deployment environments**

maybe thousands

- **Recreating a deployment should not require  
a new build**



Thers is no CI/CD

There are CI and CD

# چرا GitOps خوبه؟

- تمام مزایای استفاده از SCM را به همراه دارد. (نگهداری تاریخچه، کار تیمی و (... ,branch, pull requests
- مستند بودن فرآیند ستاپ (Document as Code)
- تکرار پذیری ستاپ زیرساخت بدون خطای انسانی در کوتاهترین زمان ممکن.
- ساده‌تر کردن DRP (faster feedback and control loop)
- قابل اطمینان‌تر بودن. وضعیتی که می‌خواهیم با یک روش استاندارد تعیین می‌شود.
- کارآمد بودن. پروسه‌های مربوط به Test & Code Review خیلی راحت‌تر قابل انجام و تکرار هست.
- امکان استفاده از CD و خودکار سازی فرایندهای مربوط به زیرساخت و تنظیمات

# Push Model vs Pull Model

## Push based Deployment

- Jenkins
- GitLab CI/CD

## Pull Deployment

- Argo
- flux

Pull Based:

- Agent installed in the environment, e.g. in K8s cluster
- Agent pulls changes from Git repository Monitors and compares desired state with actual state
- Applies the changes necessary to get to desired state
- Easy rollback to any previous state --> git revert



# GitOps is

- IaC
- Version Control
- Pull/Merge Requests
- CI/CD Pipeline

# Platform Engineering

# Terraform

```
        } terraform
    } required_providers
} = aws
"source  = "hashicorp/aws
"version = "~> 4.16
{
{
"required_version = ">= 1.2.0
{

} "provider "aws
"region  = "us-west-2
{
}

} "resource "aws_instance" "app_server"
"ami          = "ami-0b029b1931b347543
"instance_type = "t3.micro
{
} = tags
"Name = "ExampleAppServerInstance
{
{
}
```



## Bitnami SealedSecrets 12.4

Bitnami offers a lot of solutions to kubernetes and one of them is Sealed secrets that encrypts kuberneted ■  
■ Secrets

This Solution creates a controller that is based on a CRD, and which takes a Kubernetes Secret YAML ■  
■ manifest file as input

It generates a SealedSecret, which can next be used, and safely stored in a Git repository ■  
■ The SealedSecret is encrypted with the private key of the SealedSecret controller, and for that reason will ■  
■ dynamically be decrypted when used in the same cluster

When used, the SealedSecret creates a Kubernetes Secret, which in the Kubernetes cluster still isn't ■  
■ encrypted, but now doesn't need to be stored in a Git repository anymore

kubeseal and Namespaces

By default, the `kubeseal` generated encryptedData in the SealedSecret is valid in the current Namespace ■  
■ only

This adds protection, but in a GitOps environment, limits the usability as it complicates using the ■  
■ SealedSecret in different Namespaces that represent the GitOps environments

