WELCOME TO THE WORLD OF GIT

Devops 20 hours course

JOHN BRYCE
Leading in IT Education
matrix company

- GIT OPERATION -

•••

THE TRUTH IS OUT THERE....

LINK FOR OUR SLIDE DECK
https://bit.ly/git-2020-devopshift

ABOUT ME

Ex CEO , CTO and SVP Engineering for multiple Israeli startups with 25 Years experience in the world of Software architecture , IT and Management.

Founder & CEO of :
DevopShift Israel - Devops solutions & consulting
Founder & CEO of of DevopShift Vienna

Im also Acting as Head of Of Devops for JohnBryce - Israeli biggest computer academy.
Last but not least a Devops and Microservices design lecturer around the world (6200 Students in just the last 2.5 Years).

Linkedin profile:
http://bit.ly/yaniv_linkedin
EMAIL: yaniv@ynot.work

Questions and notes

- What Do we Know About source control?
- What scenarios do we know that source control is being used?
- Do we know what GITOPS is?
- Do we use GIT?
    - Code Development only ?

© All rights reserved to devopShift

- PREFACE -

© All rights reserved to devopShift

- WHAT IS GIT -

WHAT IT'S ALL ABOUT

© All rights reserved to devopShift

PART 1

In Part 1 we are going to cover Version Control Systems (VCSs) and Git basics — no technical stuff, just what Git is, why it came about in a land full of VCSs, what sets it apart, and why so many people are using it.

Once we complete this part we will learn how to Install , Configure it and use it (the right way)

ABOUT VERSION CONTROL

Manage our code (version)

- “What is “version control”, and why should we care?
  - Version control is a system that records changes to a file or set of files over time so that we can recall specific versions later. For the examples over our DEVOPS course, we will use software source code as the files being version controlled, though in reality we can do this with nearly any **type of file on a computer**.

ABOUT VERSION CONTROL

GIT IS SUPER POPULAR

- Distributed Version Control
- Open source completely
- Compatible with all OS
- Faster than other SCM’s (almost 100x in some cases)

GIT IS A DISTRIBUTED VERSION CONTROL

What it **does not** mean?

Many people’s version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they’re clever).

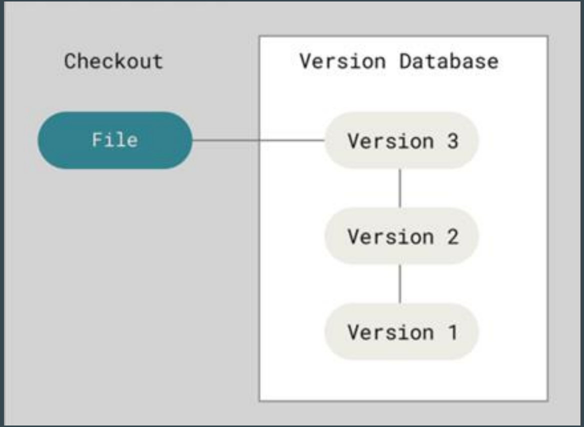
This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory we’re in and accidentally write to the wrong file or copy over files we don’t mean to…

Question:
Who did just that?

GIT IS A DISTRIBUTED VERSION CONTROL

How was it solve back in the days?

Programmers long ago developed local **Version Control Systems** (VCSs) that had a simple database that kept all the changes to files under **revision control**”



GIT IS A DISTRIBUTED VERSION CONTROL

MAJOR PROBLEM WITH LOCAL VCS

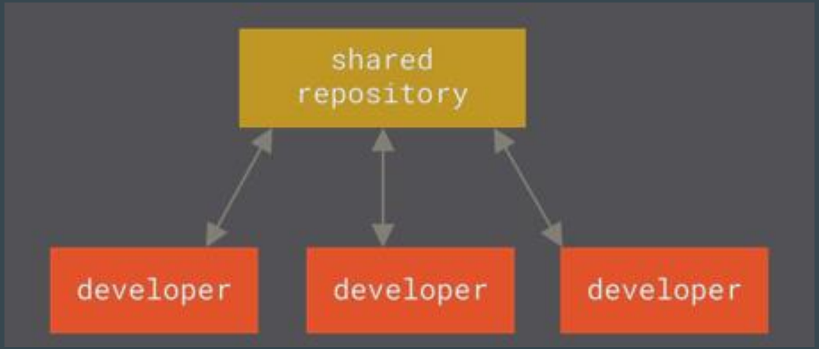
A major issue encountered by all whs a simple need of **Collaboration** with other Developers , Editor or systems.
To deal with the problem, Centralized Version were developed - CVCSs.

Systems such as:

- CVS
- Subversion
- Perforce

GIT IS A DISTRIBUTED VERSION CONTROL

CENTRALIZED VERSION CONTROL



Centralized server that hosts all files revisions and other clients (ex. Developers) can login,
Push and checkout files from the central server

GIT IS A DISTRIBUTED VERSION CONTROL

CENTRALIZED VERSION CONTROL - CONS

But this setup also had some serious downsides.

- Single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on.
- If the hard disk of the central database is on becomes corrupted, and proper backups haven't been kept, we lose absolutely everything — the entire history of the project except whatever people happen to have on their local machines.

** Local VCS systems suffer from this same problem — whenever we have the entire history of the project in a single place,
we risk losing everything.

- GIT -

DISTRIBUTED VERSION CONTROL SYSTEM

© All rights reserved to devopShift

GIT IS A DISTRIBUTED VERSION CONTROL

To solve all of the previous problems and much more,
A DCVSs systems stepped in (Such as Git, AWS CodeCommit, Microsoft Team Foundation Server etc…).

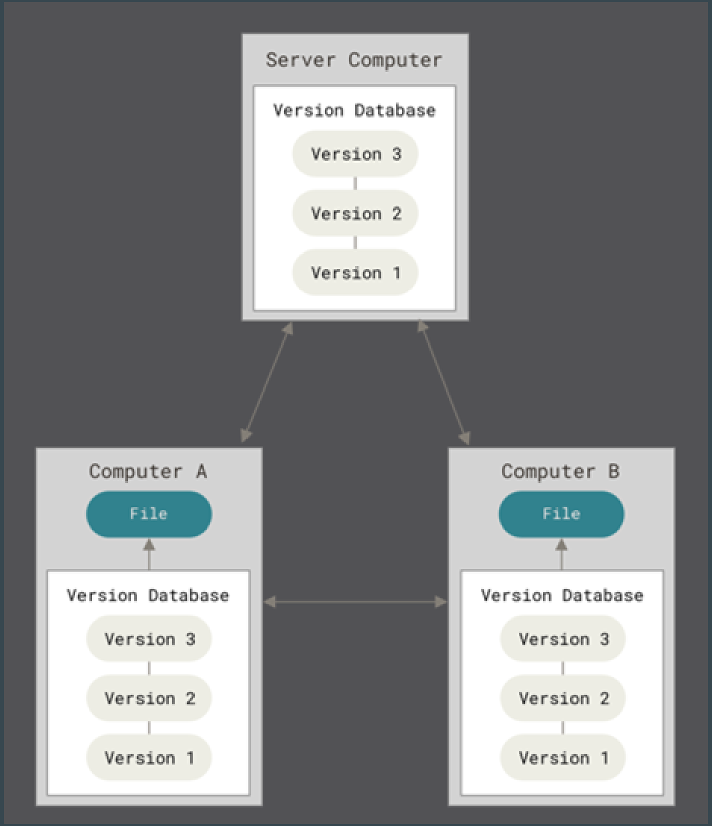
GIT IS A DISTRIBUTED VERSION CONTROL

What **does it** mean?

It means that clients don’t just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history. Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it.

Every clone is really a full backup of all the data.

GIT IS A DISTRIBUTED VERSION CONTROL



GIT IS A DISTRIBUTED VERSION CONTROL

PROS!

No Need to communicate with central server

- Faster
- No network access required
- No single failure point
- Encourages participation and “forking[tbd]” or projects
- Developers can work independently
- Submit changes sets for inclusion or rejections!

- GIT HISTORY? -

GO FETCH: https://en.wikipedia.org/wiki/Git

This is not a history class :)

© All rights reserved to devopShift

- GIT IN A NUTSHELL -

© All rights reserved to devopShift

GIT IN A NUTSHELL

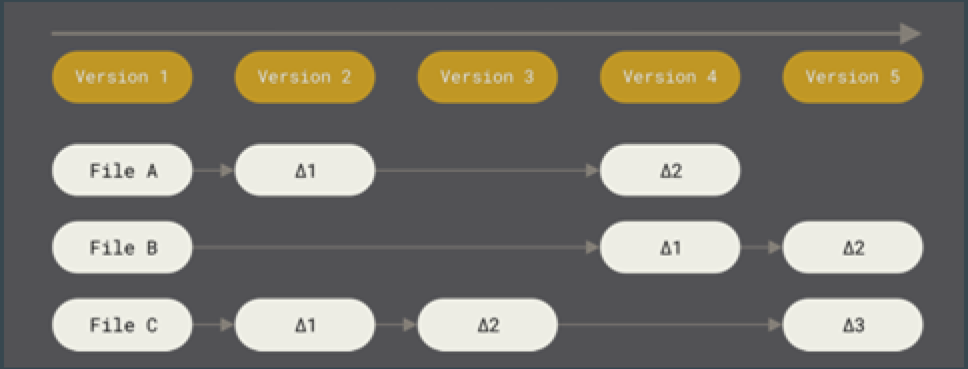
This is an important section to absorb, because if we understand what Git is and the fundamentals of how it works, then using Git effectively will probably be much easier for we

GIT IN A NUTSHELL

Snapshots NOT Differences

The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Most systems build a list of file-based changes. They think of the information they store as a set of files and the changes made to each over time (Commonly described as **Delta-Based** Version Control)

Checking over time >



GIT IN A NUTSHELL

Snapshots NOT Differences

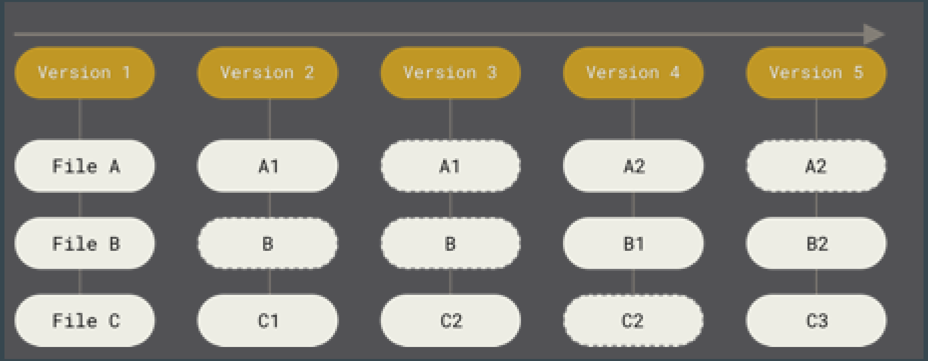
Git on the other way doesn't think of or store it’s data this way.
Instead git thinks of its data more like a series of snapshots of a miniature filesystem

With Git, every time we commit, or save the state of our project, Git basically takes a picture of what all our files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn’t store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a stream of snapshots.

GIT IN A NUTSHELL

Snapshots NOT Differences

Checking over time >



We'll explore some of the benefits we gain by thinking of our data this way when we cover Git branching in Git Branching.

- GIT CAPABILITIES -

© All rights reserved to devopShift

GIT CAPABILITIES

Nearly Every Operation Is Local

Most operations in Git need only local files and resources to operate — generally no information is needed from another computer on our network. This makes git seems light speed fast! As most operations are done locally as everything we required stored locally

For example:

- To browse the history of the project, Git doesn’t need to go out to the server to get the history and display it for we — it simply reads it directly from our local database.
- If we want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation, instead of having to either ask a remote server to do it or pull an older version of the file from the remote server to do it locally
- Remote work ? no problem at all

GIT CAPABILITIES

GIT HAS INTEGRITY…

Everything in Git is checksummed before it is stored and is then referred to by that checksum. This means it’s impossible to change the contents of any file or directory without Git knowing about it

The philosophy behind it is the core objective of git.
“we can’t lose information in transit or get file corruption without Git being able to detect it.”

The mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this:

24b9da6552252987aa493b52f8696cd6d3b00373 - “we will see these hash values all over the place in Git because it uses them so much as git stores everything in it’s db not by file name but by the hash value of it’s content

GIT CAPABILITIES

Git Generally Only Adds Data

When we do actions in Git, nearly all of them only add data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way. Not like VCS where we can lose or mess up changes one have not committed

- THE THREE STATES -

© All rights reserved to devopShift

THE THREE STATES

PAY ATTENTION...

Git has three main states that our files can reside in: **modified**, **staged**, and **committed**:

- **Modified** means that we have changed the file but have not committed it to our database yet.
- **Staged** means that we have marked a modified file in its current version to go into our next commit snapshot
- **Committed** means that the data is safely stored in our **local** database.

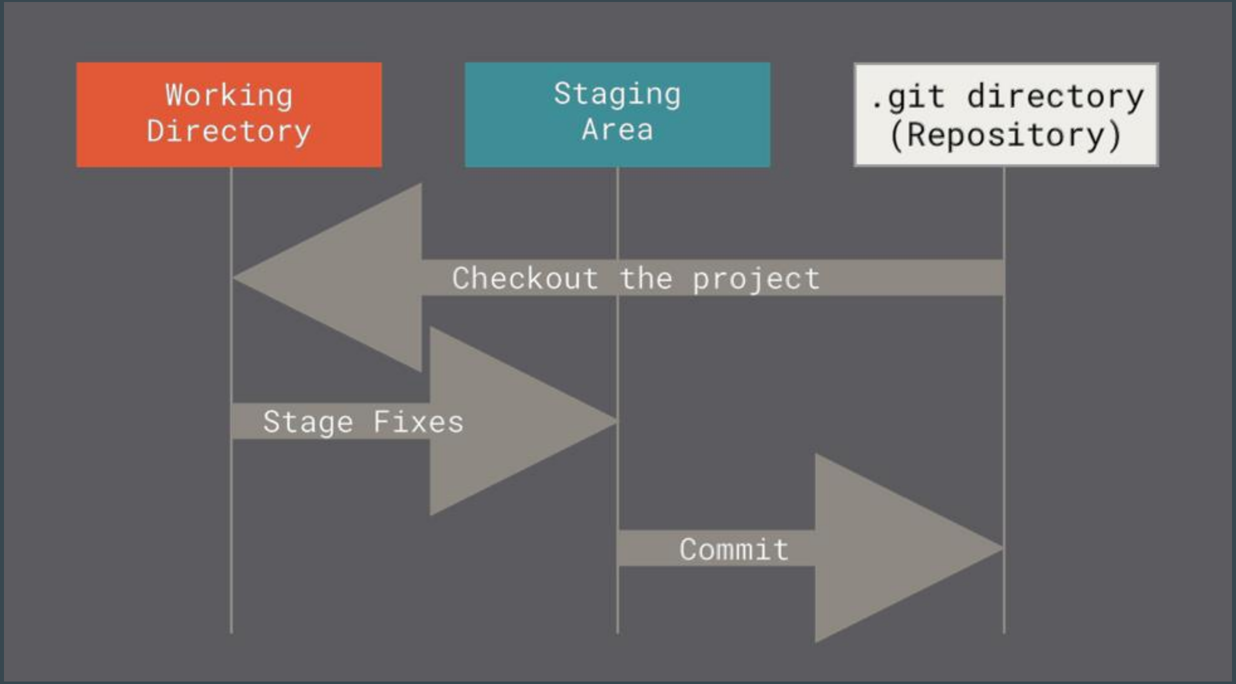
THE THREE STATES

This leads us to the GIT project 3 main sections.
The working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for we to use or modify.

The **staging area** is a file, generally contained in our Git directory, that stores information about what will go into our next commit. Its technical name in Git parlance is the “**index**”, but the phrase “staging area” works just as well.

The .**git directory** is where Git stores the metadata and object database for our project. This is the most important part of Git, and it is what is copied when we clone a repository from another computer.

THE THREE STATES



- COMMAND LINE -

© All rights reserved to devopShift

COMMAND LINE

There are a lot of different ways to use Git.

But to become really familiar with GIT we will be using Git on the command line.

Why?

For one, the command line is the only place we can run all Git commands — most of the GUIs implement only a partial subset of Git functionality for simplicity. If we know how to run the command-line version, we can probably also figure out how to run the GUI version, while the opposite is not necessarily true.

Secondly while our choice of graphical client is a matter of personal taste, all users will have the command-line tools installed and available.

COMMAND LINE

INSTALLING GIT

Easy -

Just go and download the package base on our OS here: https://git-scm.com/downloads

Once installed let’s verify functionality by running in our Terminal (Open GIT BASH)

```
#> git version
git version 2.26.2
```

COMMAND LINE

FIRST-TIME GIT SETUP

customize our Git environment. we should have to do these things only once on any given computer; they'll stick around between upgrades. we can also change them at any time by running through the commands again.

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

# View our configuration
$ git config --list --show-origin
(click q to exit)
```

COMMAND LINE

SETTING UP OUR EDITOR

configure the default text editor that will be used when Git needs we to type in a message. If not configured, Git uses our system's default editor.

1. Let's download and install Notepad++ https://notepad-plus-plus.org/downloads/v7.8.6/

```
# Setting our git to use NOTEPAD++ ON WINDOWS
$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin”
# Validate it's working by running
$ git config --global -e
# Above will open git configuration inside we notepad++
# Close notepad++ once verified.
```

- GIT BASIC -

© All rights reserved to devopShift

GIT BASIC

GETTING A GIT REPO

we typically obtain a Git repository in one of two ways:

1. we can take a local directory that is currently not under version control, and turn it into a Git repository, or
2. we can clone an existing Git repository from elsewhere.

In either case, we end up with a Git repository on our local machine, ready for work.

Initializing a Repository in an Existing Directory

For the following walkthrough please open **GIT BASH**.

```
# CREATE A NEW FOLDER
$ mkdir project1
$ cd project1
# ADD SOME FILES
$ touch project-spec.txt
$ touch hello-world.bash
# INIT A NEW GIT REPO
$ git init
Initialized empty Git repository in /Users/yanivos/work/repos/project1/.git/
```

This creates a new subdirectory named .git that contains all of our necessary repository files — a Git repository skeleton.
At this point, nothing in our project is tracked yet.

GIT BASIC

Initializing a Repository in an Existing Directory

To start version-controlling existing files we need to run the following (we will discuss the commands in details shortly)

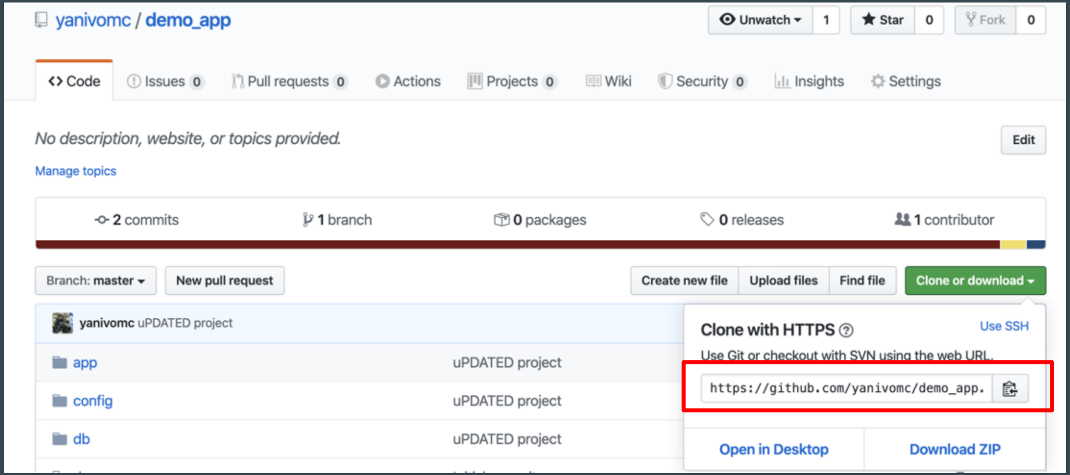
```
# START TRACKING OUR FILES
$ git add *.bash
$ git add project-spec.txt
$ git commit -m "Initial project1 files"
[master (root-commit) e16b829] Initial project1 files
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 hello-world.bash
 create mode 100644 project-spec.txt
```

GIT BASIC

Cloning an Existing Repository

Get a copy of an existing Git repository — for example, a project we'd like to contribute to — the command we need is git clone
we clone a repository with git clone <url>. For example, let's clone a small Git repo:

git clone https://github.com/yanivomc/demo_app.git



```
# CLONE A PROJECT
$ cd ~
$ git clone https://github.com/yanivomc/demo_app.git
Cloning into 'demo_app'...
remote: Enumerating objects: 116, done.
remote: Total 116 (delta 0), reused 0 (delta 0), pack-reused 116
Receiving objects: 100% (116/116), 33.65 KiB | 522.00 KiB/s, done.
Resolving deltas: 100% (13/13), done.
```

"That creates a directory named demo_app, initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version to folder named "demo_app"

GIT BASIC

RECORDING CHANGES TO THE REPOSITORY

Recording is when our project reaches a state that we want GIT to record.

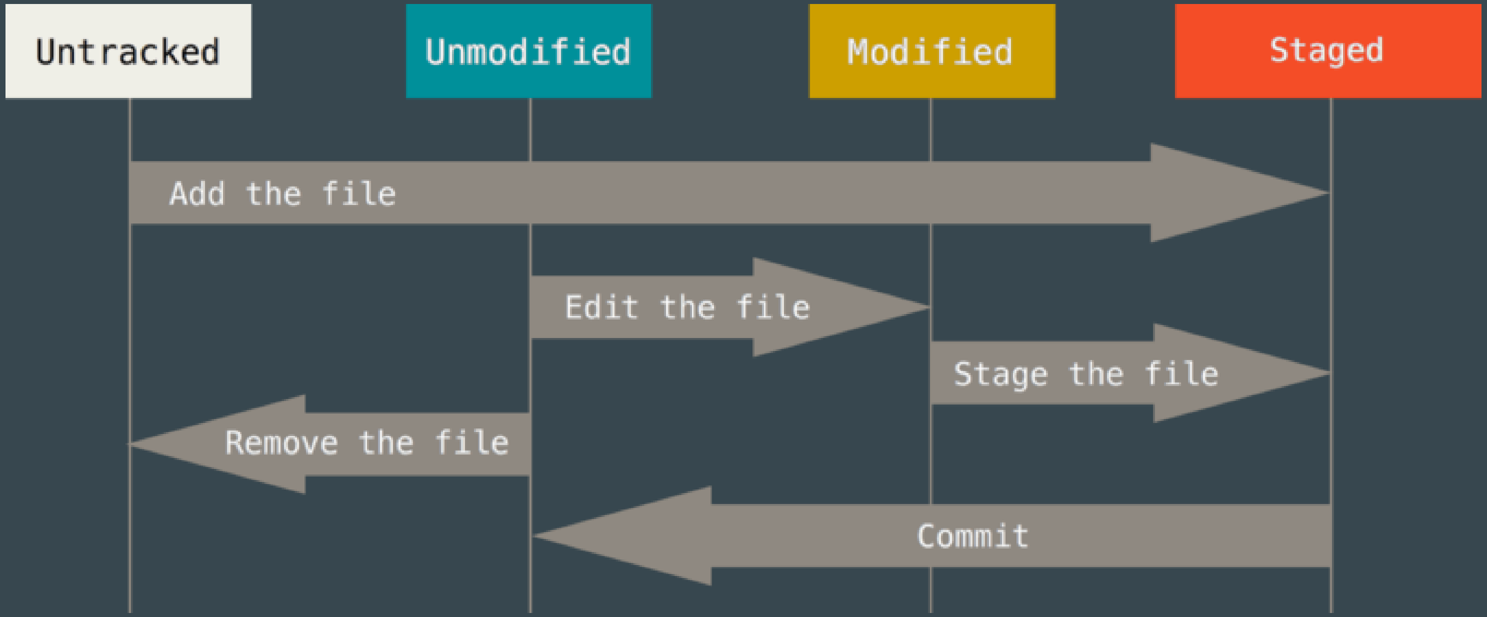
Remember that each file in our working directory can be in one of two states:

- Tracked - Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about
- Untracked- Untracked files are everything else — any files in our working directory that were not in our last snapshot and are not in our staging area.

When we first clone a repository, all of our files will be **tracked** and **unmodified** because Git just checked them out and we haven’t edited anything.

GIT BASIC

As we edit files, Git sees them as modified, because we've changed them since our last commit. As we work, we selectively stage these modified files and then commit all those staged changes, and the cycle repeats.



GIT BASIC

CHECKING THE STATUS OF OUR FILES

The main tool you use to determine which files are in which state is the “**git status**” command
If we run this command directly after a clone, we should see something like this:

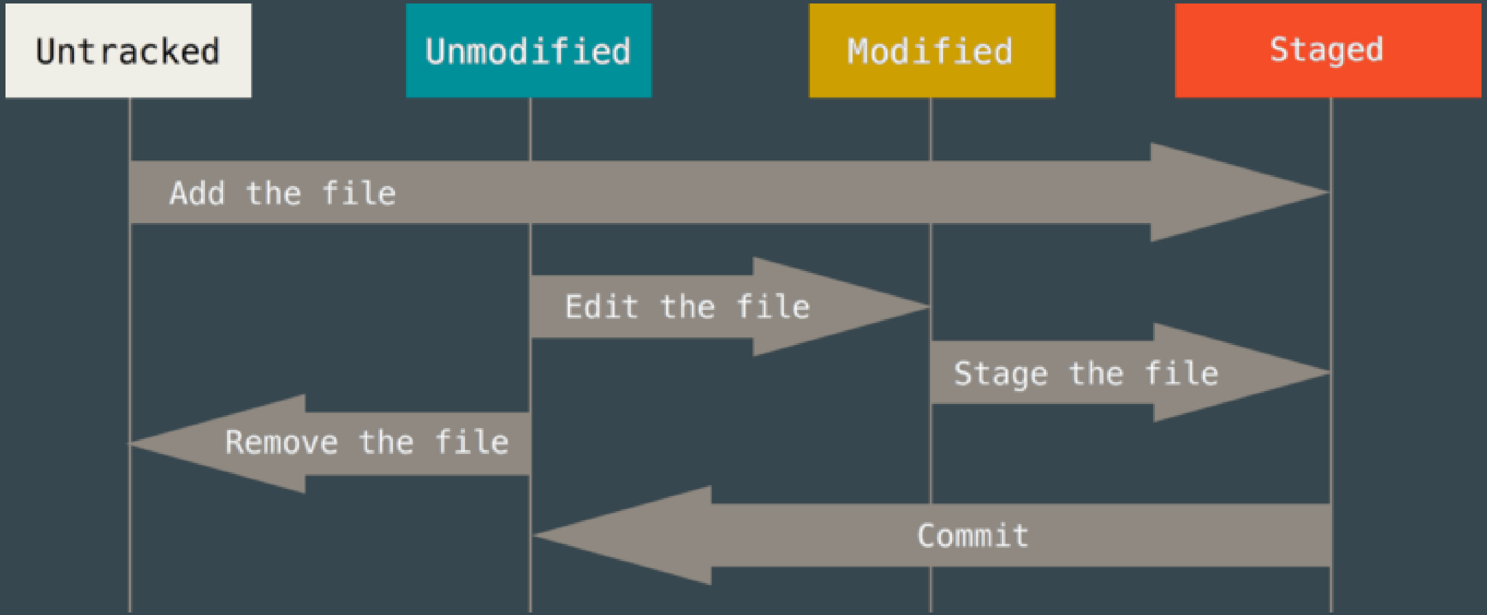
```
# CHECK FILE STATUS
$ git clone https://github.com/yanivomc/demo_app.git
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

This means we have a clean working directory; none of our tracked files are modified.
Git also doesn’t see any untracked files, or they would be listed here. Finally, the command tells us which branch we’re on and informs you that it has not diverged from the same **branch (TBD)** on the server.

- GIT BASIC -
RECAP

© All rights reserved to devopShift

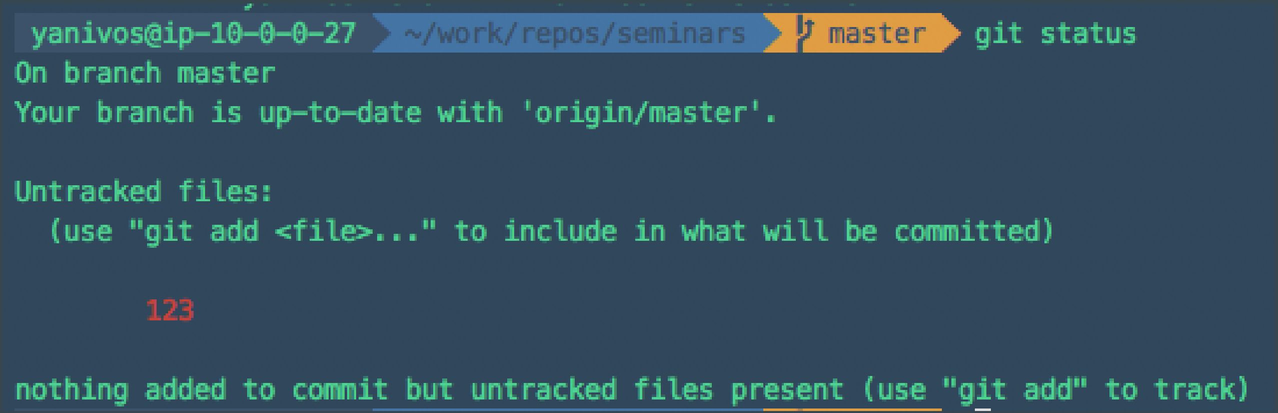
GIT BASIC - RECAP



GIT BASIC - RECAP

Untracked

Are any files in your working directory that were not in your last snapshot and are not in your staging area and were not committed

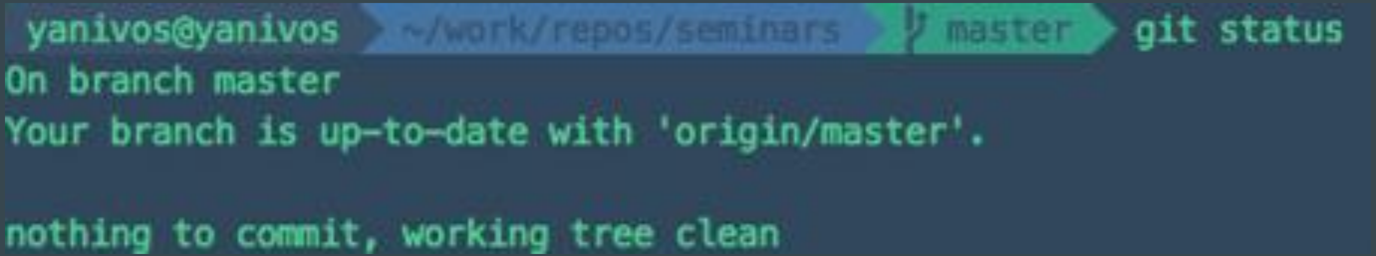


```
yanivos@ip-10-0-0-27  ~/work/repos/seminars   master  git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        123

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

GIT BASIC - RECAP

Unmodified

Are files that you just checked out or cloned from a remote repository and you haven't edited anything yet.

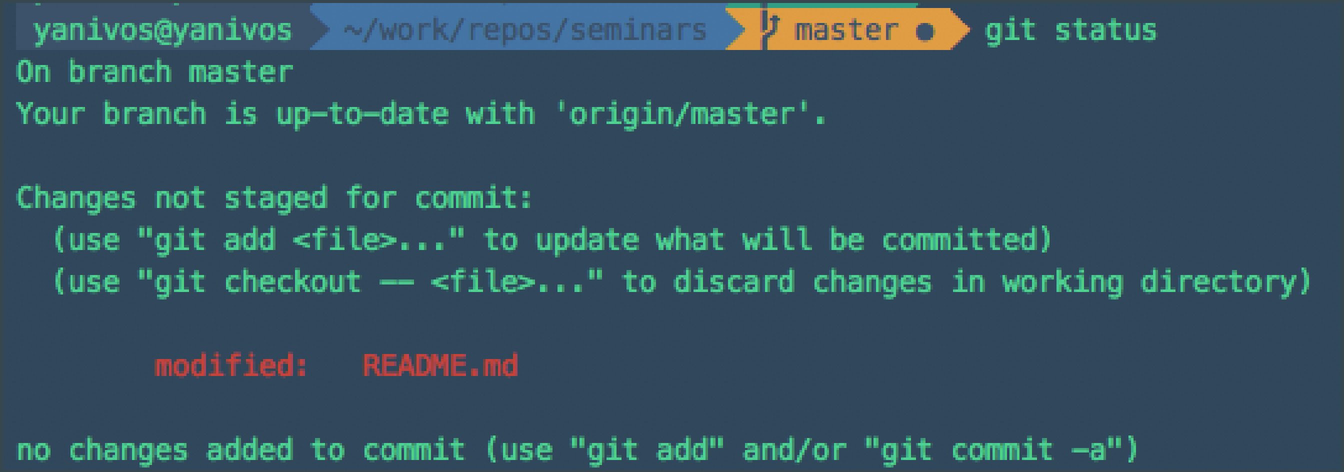


```
yanivos@yanivos  ~/work/repos/seminars  master  git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean
```

GIT BASIC - RECAP

Modified

Are any files in your working directory that were modified since your last snapshot / commit and are not staged

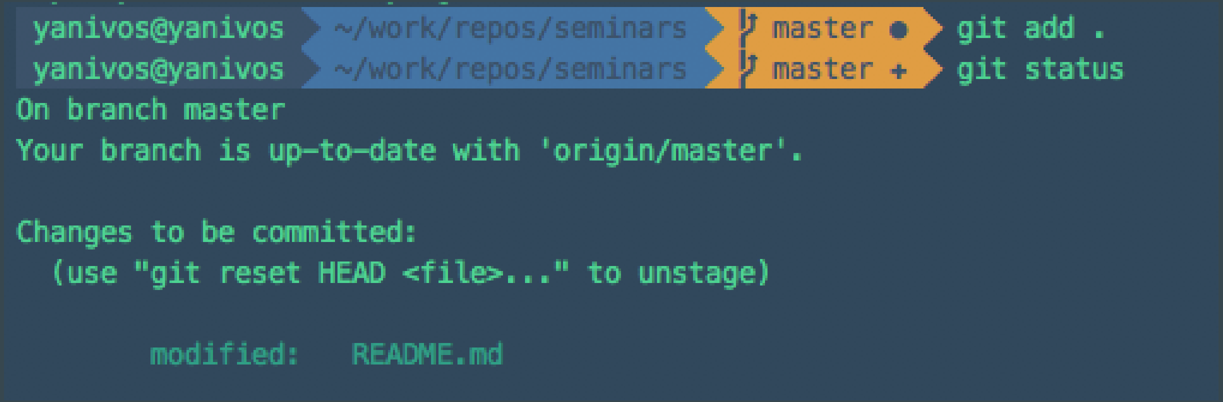


```
 yanivos@yanivos  ~/work/repos/seminars   master ●  git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README.md

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

GIT BASIC - RECAP

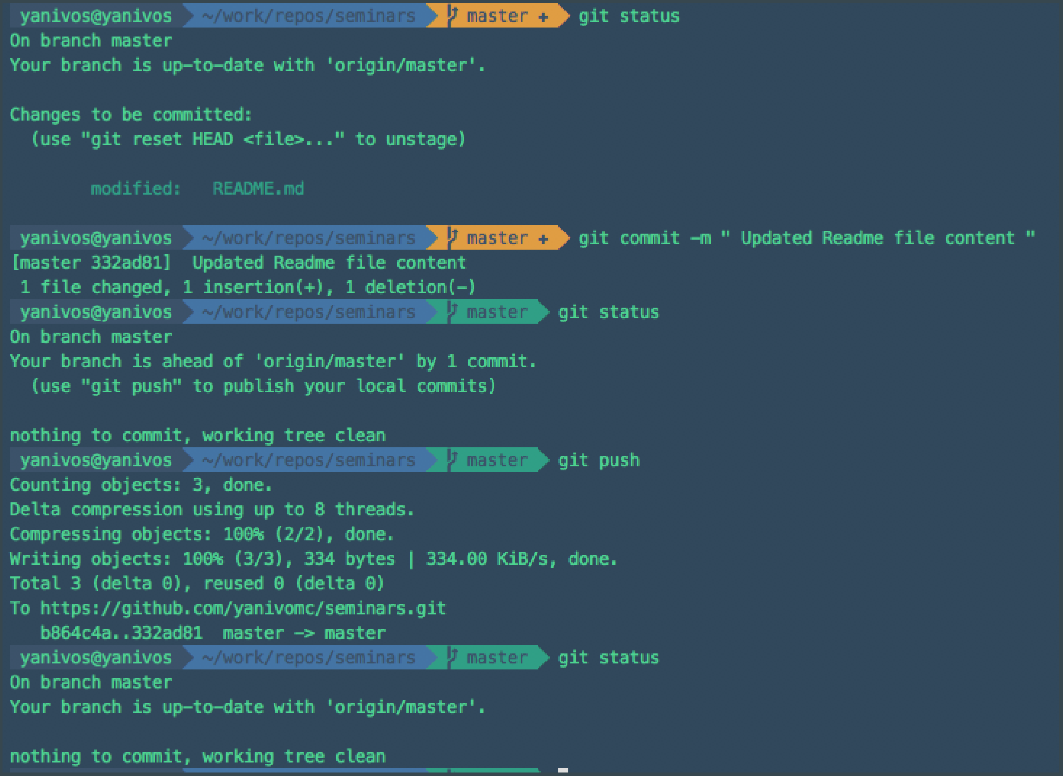
Staged

Are any files in your working directory that were modified and added since your last snapshot / commit and are NOW staged and ready to be committed



```
yanivos@yanivos  ~/work/repos/seminars  master ●  git add .
yanivos@yanivos  ~/work/repos/seminars  master +  git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.md
```

GIT BASIC - RECAP



```
yanivos@yanivos  ~/work/repos/seminars  master +  git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.md

yanivos@yanivos  ~/work/repos/seminars  master +  git commit -m " Updated Readme file content "
[master 332ad81]  Updated Readme file content
 1 file changed, 1 insertion(+), 1 deletion(-)
yanivos@yanivos  ~/work/repos/seminars  master  git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
yanivos@yanivos  ~/work/repos/seminars  master  git push
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 334 bytes | 334.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/yanivomc/seminars.git
   b864c4a..332ad81  master -> master
yanivos@yanivos  ~/work/repos/seminars  master  git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean
```

Once we added the file[s], we can commit them to our local repo and then push the changes to the remote repository

COMMIT

PUSH

UNCHANGED

- GIT KEY CONCEPTS -

BEFORE WE CONTINUE

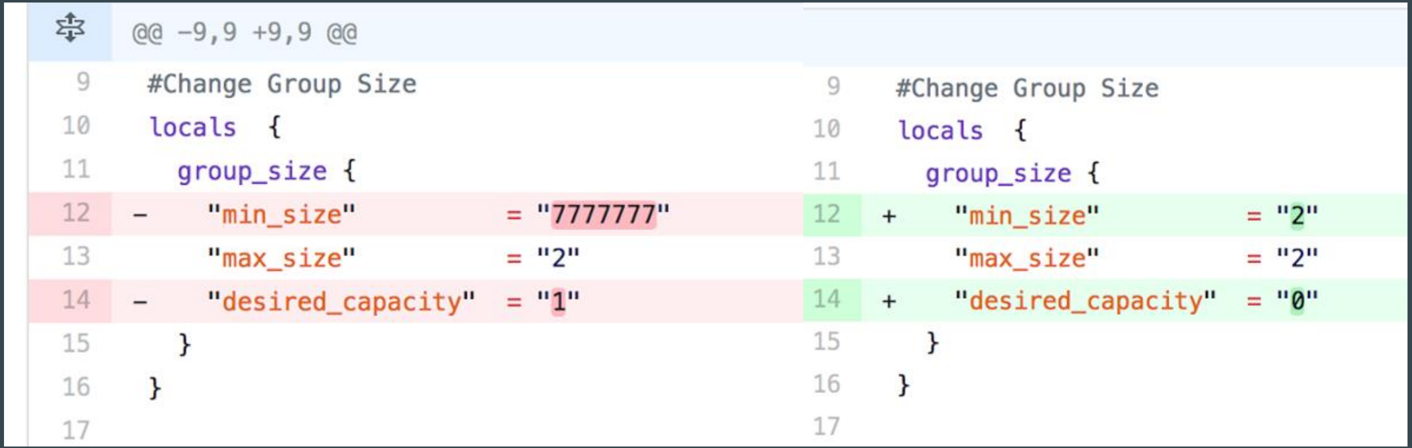
© All rights reserved to devopShift

GIT KEY CONCEPTS

Commit

Is the act itself of creating a snapshot(NEXT SLIDE)
Eventually your project will be made up of a bunch of commits you perform along the lifetime of your project.

Commits contains three layers of information:
Information about how the files changed from previous commit[s]



```
@@ -9,9 +9,9 @@
 #Change Group Size                              #Change Group Size
 locals  {                                       locals  {
   group_size {                                    group_size {
-    "min_size"          = "7777777"         +      "min_size"          = "2"
     "max_size"          = "2"                     "max_size"          = "2"
-    "desired_capacity"  = "1"               +      "desired_capacity"  = "0"
   }                                               }
 }                                               }
```

GIT KEY CONCEPTS

Commit

- A reference to the commit that came before it - Called the “parent commit”
- As Git is a content-addressable filesystem. meaning that at the core of Git is a simple key-value data store. Git Stores everything as a file and uses Hashcode name as the KEY and your changes as the VALUE.
- Once we COMMIT - Git will hand us back a unique key we can use later to retrieve that content. Git Hash looks something like this: fb2d2ec5069fc6776c80b3ad6b7cbde3cade4e

GIT KEY CONCEPTS

SnapShots

Are the way git keeps track of your code history.
Essentially what git does is to record what all your files look like at a given point in time. You as the user have the ability decide when to take a snapshot by committing your changes , and of what files.

Once done, you’ll have the ability to go back to visit any snapshot,
Review, delete or rollback to them

GIT KEY CONCEPTS

Repositories aka Repo

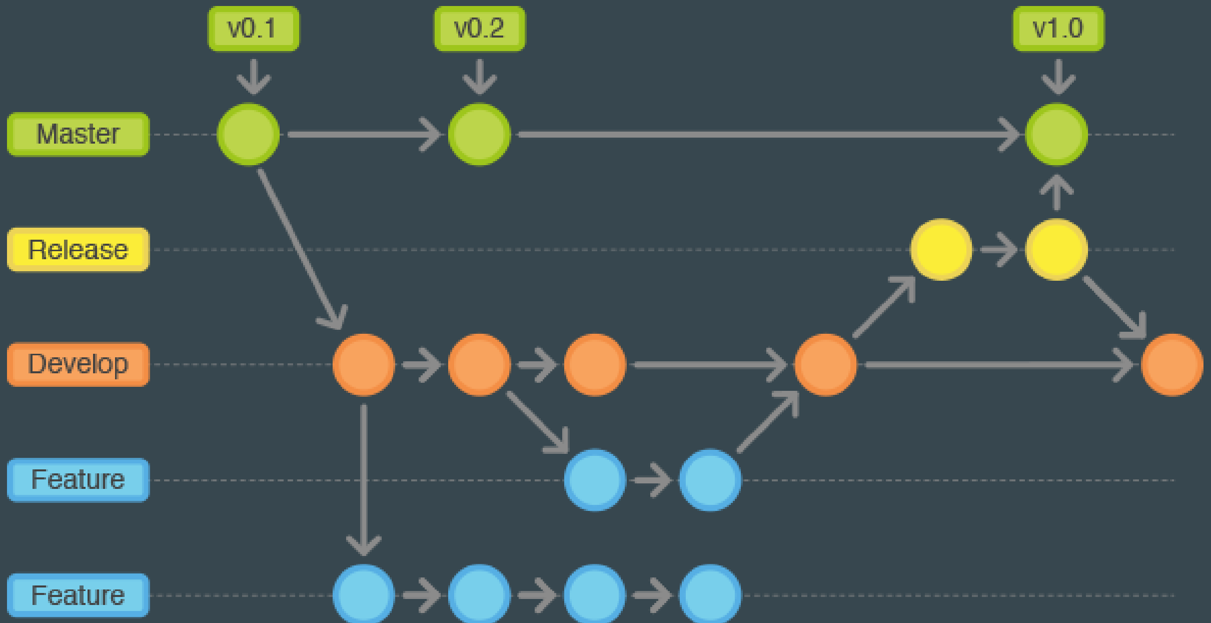
Are a collection of all the files and the history of those files you committed. The Repo consists with all of your versions, branches and history of your code / files.
Git Repo can live on a local (on premises)machine or as a SaaS on a remote server (such as GitHub!).

Common Tasks on a Repo:

- **Copying** a repository from a remote server/repo is called cloning. Cloning allow teams to work together in their local system.
- **Pulling** Changes is The process of downloading commits that don’t exist on your local machine from a remote repository.
- **Pusing** is the process of adding your local changes to the remote repository.

GIT KEY CONCEPTS

Branching (TBD)



- GIT BASIC -

HANDSON

© All rights reserved to devopShift

GIT BASIC - HANDSON

FOLLOW THROUGH 01

In the next follow through we will add a new files , run git status , modify files and check how their status is being changed.

```
# FOLLOW THROUGH 01 - Use our project1 folder
$ cd project1
$ echo '#Devops Seminar information' > README.md
$ git status
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
                    README.md
nothing added to commit but untracked files present (use "git add" to track)
```

GIT BASIC - HANDSON

LAB01

1. Add one more file called information.txt
   a. Run git status and see the output
   b. Add both files to git local repo using “git add .”
   c. Check status - outcome should be:

```
new file:   README.md
new file:   information.txt
```

   a. Edit One of the files and check the status now
   b. Commit changes with a message: “testing git basic commands”
   c. Check status once again
      i. Why the information file is still marked as modified and not staged ?
      ii. Fix it
      iii. Use git short status command to see changes in a more clean style - **“git status -s**”

GIT BASIC - HANDSON

LAB02

1. Lets study our git history (changes)
   a. Run: git log --stat
   b. See the changes that you have made to the files
   c. See the last commit of a file that you choose using
      i. git show [commitID]
   d. View that changes made between commit 1 & 2 using
      i. git diff [1stCommitID] [2ndCommitID]
   e. Revert to previous commit state
      i. Git checkout [commitID]
   f. Go back to the latest state
      i. git checkout master

- GIT OPERATION -

•••

END OF PART I