Git
&
Gitlab
For Engineers

Tejas Roysam

JSC/EV3

Agenda

Part 1: Git

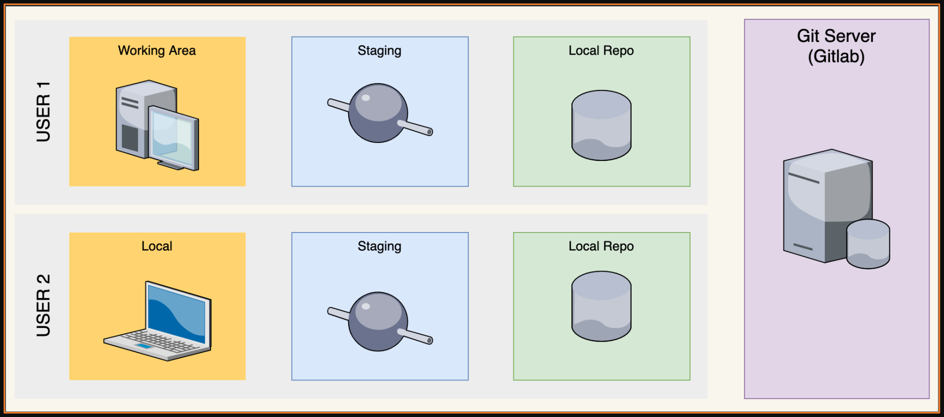
- Intro/Goals
- Git Objects
- Git Operations
    - Committing
    - Branching
    - Pushing
    - Fetching/Merging/Pulling
    - Checking Out
    - Rebasing
    - Stashing
    - Resetting/Reverting
    - Submodules
- Summary
- Tools & Resources

Goals

- To understand Git from an object-driven perspective -> “Under the hood”
- To understand the primary Git operations, their use cases, and general best practices.
- To be aware of basic tools that improve ease of use.
- To be aware of additional resources available to help with Git usage and understanding.

What is Git?

- Revision Control – retain history of all changes
- Access Control – set whitelisted and/or blacklisted access
- Coordination and concurrence
- Distributed environment
  - As opposed to SVN, which is centralized.
  - Allows each user to have the entire history of the repository locally
  - Allows offline work
  - Complex dependency trees can be maintained simply (see submodules/subtrees)



Why Git?

Git benefits organizations.

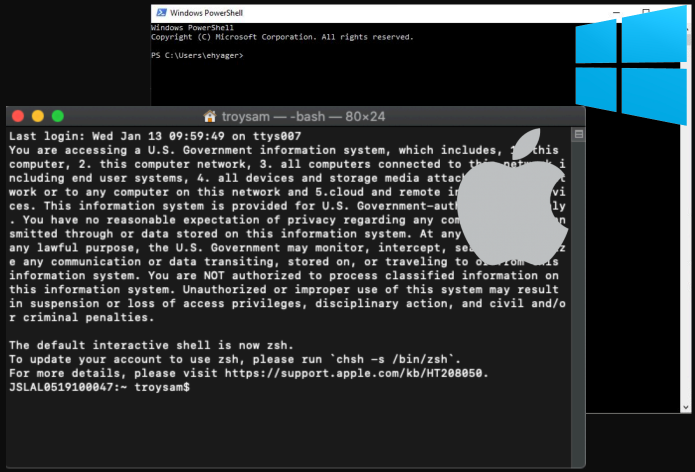
- For developers and engineers, it encourages an agile workflow - where incremental changes can be frequent and decentralized. Faster inter- and intra-team feedback. Faster onboarding/training. Better quality control.
- For project managers, Git provides constant awareness of status. The Git workflow allows priorities to be redirected.
- For UX designers, rapid-prototype-focused engineering, safety oversight, and test engineers, Git allows unlimited sandboxing with real products.
- Etc.

Git is an efficiency enabler, a collaboration tool, and a safety net. Independent of domain. And its open source.

Every tool needs learning though, and Git is no exception. Thankfully, the learning curve is very shallow.

The Terminal

- The standard method of using Git is via a terminal.
  - On Mac, you can use the terminal application.
  - On Windows, you can use Powershell.
  - On a Linux distro, use the shell of your choice
- Terminal basics to help you navigate directories, etc.:
  - Windows Powershell aliases some Unix commands, which means for all platforms you can use:
    - `cd` – "change directory": move to a specified location
      - ex: `cd C:/users/Myname/Documents`
    - `ls` – list items in current directory location
    - `pwd` – get current location (full path)
  - That should be enough to get by! Everything else is accomplished through Git commands, which we will go through in depth. Git commands are entered in the terminal the same way.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\ehyager>
```

troysam — -bash — 80×24

```
Last login: Wed Jan 13 09:59:49 on ttys007
You are accessing a U.S. Government information system, which includes, 1 this
computer, 2. this computer network, 3. all computers connected to thi[illegible] i
ncluding end user systems, 4. all devices and storage media attac[illegible]t
work or to any computer on this network and 5.cloud and remote i[illegible]vi
ces. This information system is provided for U.S. Government-aut[illegible]ly
. You have no reasonable expectation of privacy regarding any co[illegible]n
smitted through or data stored on this information system. At any[illegible]
any lawful purpose, the U.S. Government may monitor, intercept, se[illegible]z
e any communication or data transiting, stored on, or traveling to o[illegible]is
information system. You are NOT authorized to process classified information on
this information system. Unauthorized or improper use of this system may result
in suspension or loss of access privileges, disciplinary action, and civil and/o
r criminal penalties.

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
JSLAL0519100047:~ troysam$
```

There is no need to be intimidated by the terminal!

- Terminal allows full functionality
- There are tools that can let you mostly avoid the terminal for day-to-day use.

(Stick around for the Gitlab topic!)

Commands

- A command is a way of running a program/application.
- To run Git in different ways, we invoke the Git program and pass it some more information about what we want it to do.
- Below is a sample command, don’t worry about what it does yet, just take note of its components:

$$\underbrace{\texttt{git}}_{\text{Application}}\ \underbrace{\texttt{push}}_{\text{Command Specifier}}\ \underbrace{\texttt{-u}}_{\text{Flag(s)}}\ \underbrace{\texttt{origin branchname}}_{\text{Arguments(s)}}$$

Basic Entities

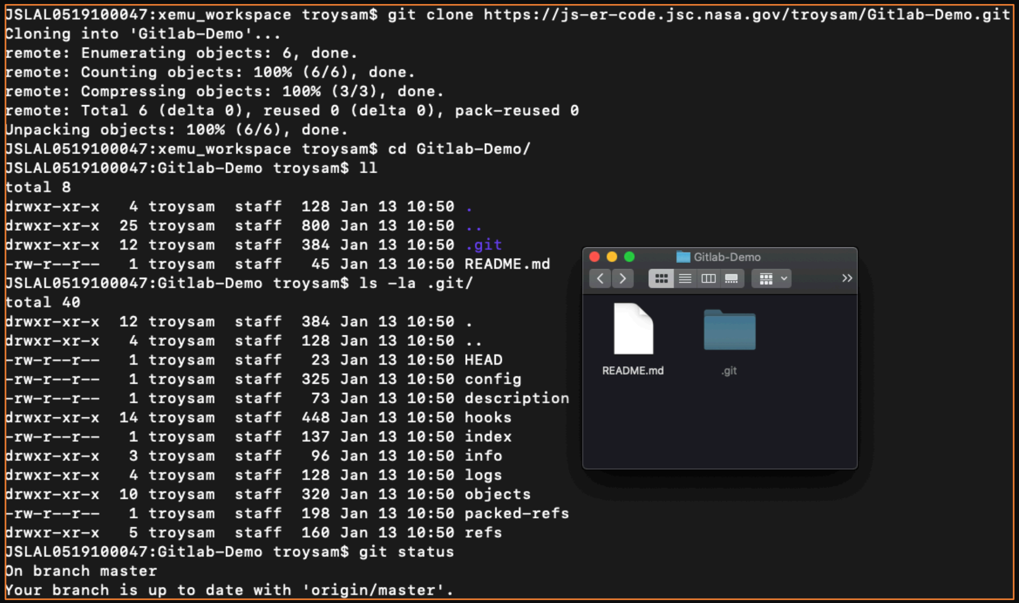
- **Repository** – directory containing everything Git is aware of. Denoted by a `.git` folder at the top level.
  - Local – the Git repository as it lives on your device or user space.
  - Remote – the Git repository as it resides on another device, e.g. a server.
- **Commit** – a “snapshot” of a repository. Associated with a time, author, and description.
  - Each commit has a unique **hash** to identify it.
    - **Hash** - a unique randomly-generated hexadecimal identifier, also sometimes called a SHA1 or Commit ID.
  - Each commit has a **diff** – representing changes to the repository from its parent(s).
- **Branch** – a moving ”bookmark” or “pointer” that tracks commits.

Together, all commits on all branches create a **tree**, which encompasses the repository’s entire contents and history.

Let’s start by looking at the `.git/` folder to get an idea of what Git is under the hood!

The Repository

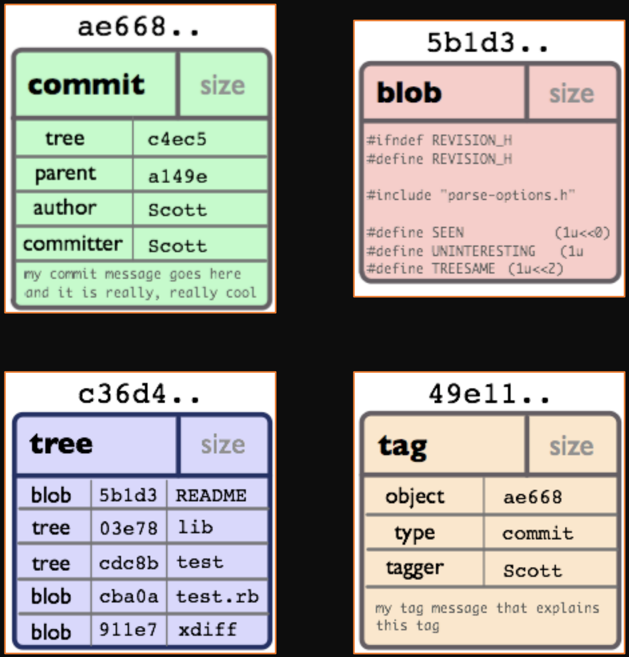
- `.git` folder is the container for all project information required for revision control
- To understand how Git is structured and how it functions, let us examine the most important items therein:
  - `.git/objects/`
  - `.git/refs/`
  - `.git/HEAD`
  - `.git/index`



```
JSLAL0519100047:xemu_workspace troysam$ git clone https://js-er-code.jsc.nasa.gov/troysam/Gitlab-Demo.git
Cloning into 'Gitlab-Demo'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.
JSLAL0519100047:xemu_workspace troysam$ cd Gitlab-Demo/
JSLAL0519100047:Gitlab-Demo troysam$ ll
total 8
drwxr-xr-x   4 troysam  staff  128 Jan 13 10:50 .
drwxr-xr-x  25 troysam  staff  800 Jan 13 10:50 ..
drwxr-xr-x  12 troysam  staff  384 Jan 13 10:50 .git
-rw-r--r--   1 troysam  staff   45 Jan 13 10:50 README.md
JSLAL0519100047:Gitlab-Demo troysam$ ls -la .git/
total 40
drwxr-xr-x  12 troysam  staff  384 Jan 13 10:50 .
drwxr-xr-x   4 troysam  staff  128 Jan 13 10:50 ..
-rw-r--r--   1 troysam  staff   23 Jan 13 10:50 HEAD
-rw-r--r--   1 troysam  staff  325 Jan 13 10:50 config
-rw-r--r--   1 troysam  staff   73 Jan 13 10:50 description
drwxr-xr-x  14 troysam  staff  448 Jan 13 10:50 hooks
-rw-r--r--   1 troysam  staff  137 Jan 13 10:50 index
drwxr-xr-x   3 troysam  staff   96 Jan 13 10:50 info
drwxr-xr-x   4 troysam  staff  128 Jan 13 10:50 logs
drwxr-xr-x  10 troysam  staff  320 Jan 13 10:50 objects
-rw-r--r--   1 troysam  staff  198 Jan 13 10:50 packed-refs
drwxr-xr-x   5 troysam  staff  160 Jan 13 10:50 refs
JSLAL0519100047:Gitlab-Demo troysam$ git status
On branch master
Your branch is up to date with 'origin/master'.
```

The Repository

- `.git/objects/` – database
  - Every **object** is saved as a hash in this database
  - Objects:
    - Commits
    - Trees (**Directory** or **Subdirectory**)
    - Blobs (**B**inary **L**arge **Ob**jects, i.e. **Files**)
    - Tags (commit aliases)



ae668..

commit	size
tree	c4ec5
parent	a149e
author	Scott
committer	Scott

my commit message goes here
and it is really, really cool

5b1d3..

blob	size

```
#ifndef REVISION_H
#define REVISION_H

#include "parse-options.h"

#define SEEN          (1u<<0)
#define UNINTERESTING   (1u
#define TREESAME  (1u<<2)
```

c36d4..

tree	size	
blob	5b1d3	README
tree	03e78	lib
tree	cdc8b	test
blob	cba0a	test.rb
blob	911e7	xdiff

49e11..

tag	size
object	ae668
type	commit
tagger	Scott

my tag message that explains
this tag

Images: http://shafiul.github.io/gitbook/1_the_git_object_model.html

The Repository

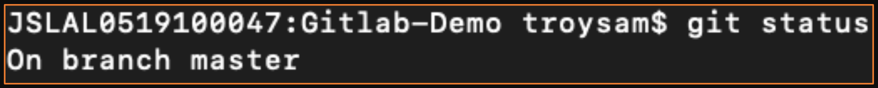
- `.git/refs/` – “phonebook” - references, i.e. stored hashes of commits
  - `.git/refs/heads/` – local branches
  - `.git/refs/remotes/` – branches from remote
  - `.git/refs/tags/` – local tags

- **References = refs = ”pointers” or “bookmarks”**

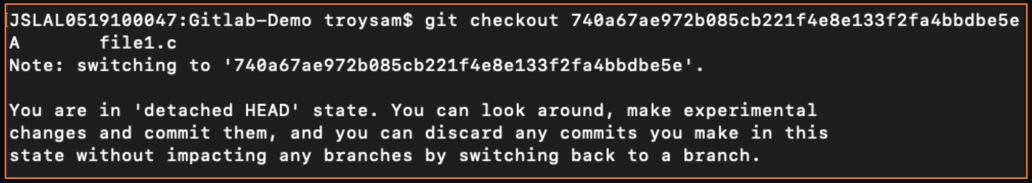
- **Whenever we say “refs” we are talking about commit hashes.**

The Repository

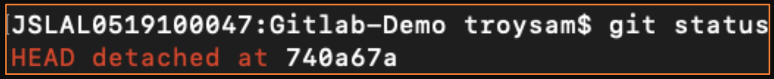
- `.git/HEAD`
  - Pointer to the current branch.
  - This is a special Git reference.
  - This pointer can exist in a “detached” state, pointing to a commit rather than a branch.
    - This is not inherently bad. We will examine it in detail later.



```
JSLAL0519100047:Gitlab-Demo troysam$ git status
On branch master
```



```
JSLAL0519100047:Gitlab-Demo troysam$ git checkout 740a67ae972b085cb221f4e8e133f2fa4bbdbe5e
A       file1.c
Note: switching to '740a67ae972b085cb221f4e8e133f2fa4bbdbe5e'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.
```

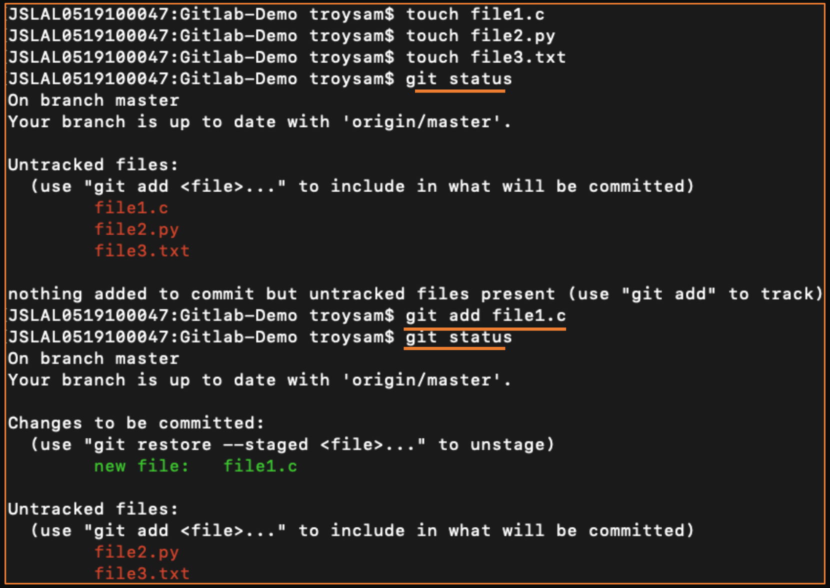


```
JSLAL0519100047:Gitlab-Demo troysam$ git status
HEAD detached at 740a67a
```

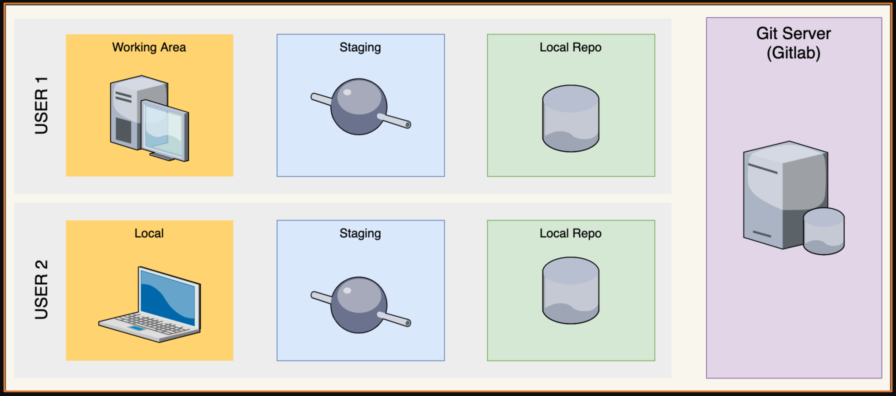
The Repository

- `.git/index` – staging area
  - "virtual working tree"
  - Tells Git what to commit.

- On the right:
  - Untracked files – local files that are not tracked by the **current git commit**
  - Changes to be committed
    - Modifications to files present in the current git commit
    - New files that have been added to the `.git/index`, which will be added to the repository in the next commit



```
JSLAL0519100047:Gitlab-Demo troysam$ touch file1.c
JSLAL0519100047:Gitlab-Demo troysam$ touch file2.py
JSLAL0519100047:Gitlab-Demo troysam$ touch file3.txt
JSLAL0519100047:Gitlab-Demo troysam$ 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)
        file1.c
        file2.py
        file3.txt

nothing added to commit but untracked files present (use "git add" to track)
JSLAL0519100047:Gitlab-Demo troysam$ git add file1.c
JSLAL0519100047:Gitlab-Demo troysam$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   file1.c

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file2.py
        file3.txt
```



The Repository

- **Summary**:
  - Git has objects and references.
    - Commits are objects.
    - Branches and tags are references (to commit objects, via a hash).
    - References tie different types of objects together.
  - The repository is made up of two data structures:
    - `.git/objects/` – database of objects
    - `.git/index` – staging area
  - and a ”phone book” to look up objects:
    - `.git/refs/`

Thus, the Git repository will contain an **indexable complete history of all the project’s content**.

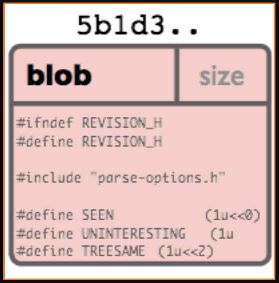
This is true wherever the repository resides: every local and remote instance -> **decentralization.**

Moving on…

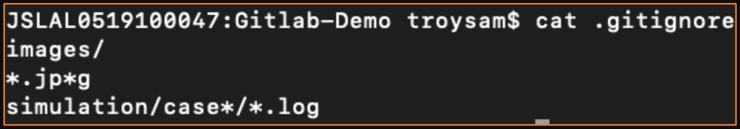
Now that we know the basic Git objects and how they relate, let’s examine each type of object in more detail.

Starting with Files (Blobs)!

Ignored Files



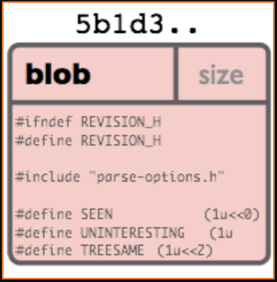
- You may not want to track all files in your repository directory
  - Example: files generated by a simulation, images transferred from a device, etc.
- Luckily, we can add a file to our Git repository that lets us ignore some files selectively.
  - `.gitignore`
  - On each line, we can specify one of the following:
    - A complete file path
    - A complete file name
    - A pattern (regular expression)



```
JSLAL0519100047:Gitlab-Demo troysam$ cat .gitignore
images/
*.jp*g
simulation/case*/*.log
```

- Git will automatically assume all files matching patterns in the `.gitignore` should not be tracked, but you can manually override this if needed.

Ignored vs Untracked Files

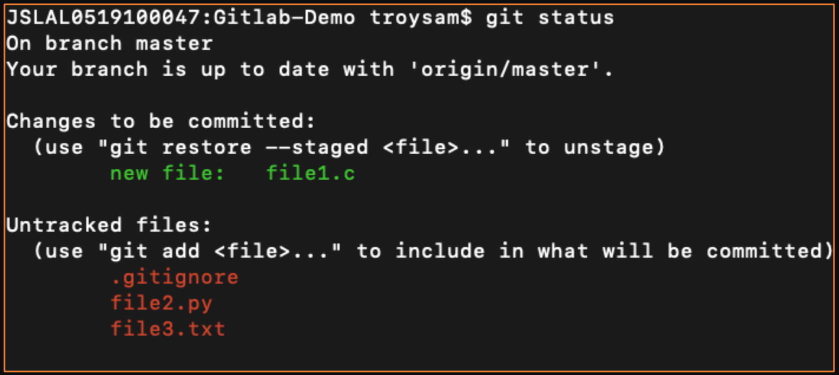


```
#ifndef REVISION_H
#define REVISION_H

#include "parse-options.h"

#define SEEN              (1u<<0)
#define UNINTERESTING   (1u
#define TREESAME (1u<<2)
```

- Reminder: an untracked file is one that is not tracked in the **current state of the repository**.

- Check on repository state with the `git status` command
- Here we see two categories:
  - Changes to be committed
    - These are new or modified files that we have placed in the staging area with `git add`.
  - Untracked files
    - These are files that are inside the Git repository directory, but which are **not known to Git**.

- Ignored files will not appear at all, Git knows to disregard them.

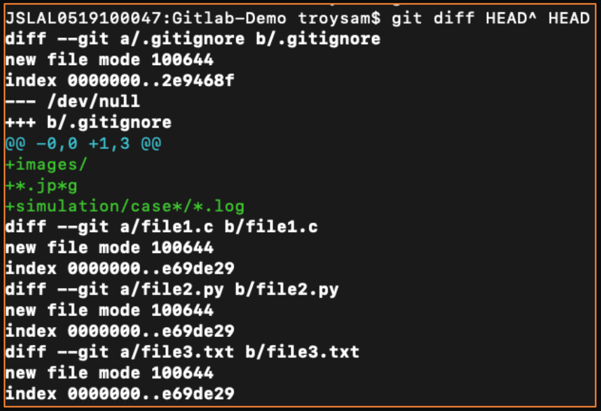


```
JSLAL0519100047:Gitlab-Demo troysam$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   file1.c

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        file2.py
        file3.txt
```

Commits

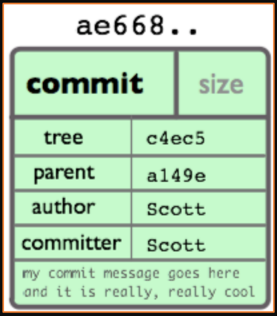
- Commits are a:
  - Child of at least one other commit*
  - Project state
  - diff from its parent(s)

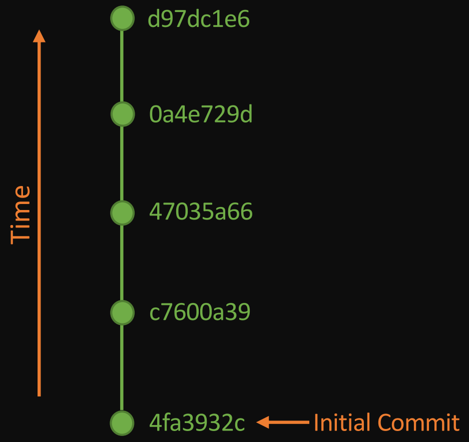
- When do commits have no parents?
  - The first (initial) commit
- When do commits have two parents?
  - **Merge commits**: created when one branch is merged with another
- All other commits have exactly one parent.

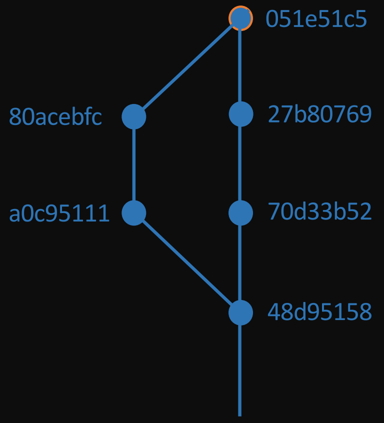


```
JSLAL0519100047:Gitlab-Demo troysam$ git diff HEAD^ HEAD
diff --git a/.gitignore b/.gitignore
new file mode 100644
index 0000000..2e9468f
--- /dev/null
+++ b/.gitignore
@@ -0,0 +1,3 @@
+images/
+*.jp*g
+simulation/case*/*.log
diff --git a/file1.c b/file1.c
new file mode 100644
index 0000000..e69de29
diff --git a/file2.py b/file2.py
new file mode 100644
index 0000000..e69de29
diff --git a/file3.txt b/file3.txt
new file mode 100644
index 0000000..e69de29
```

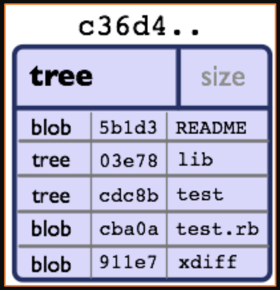
A Git diff between the current commit and the prior commit







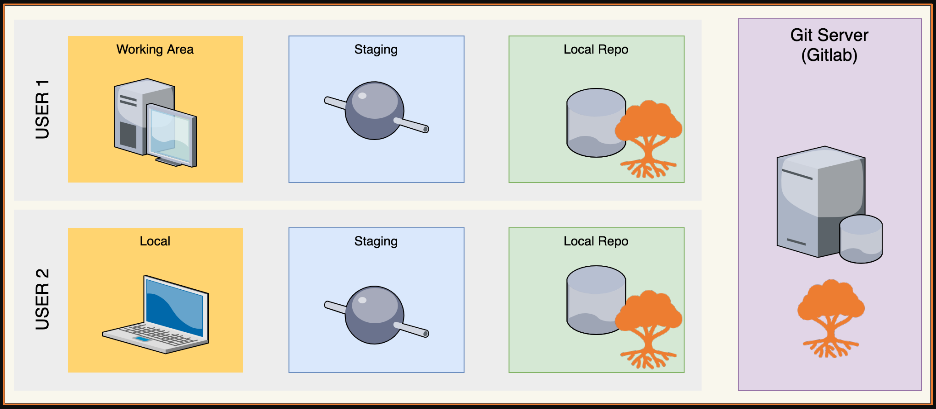
Trees

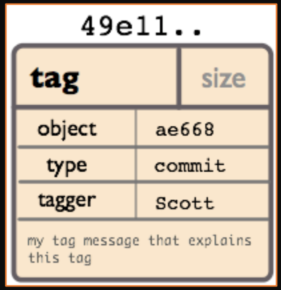


c36d4..

tree		size
blob	5b1d3	README
tree	03e78	lib
tree	cdc8b	test
blob	cba0a	test.rb
blob	911e7	xdiff

- Trees represent your file structure, and groupings of files.
- Trees therefore mirror a Unix filesystem, and can contain other tree objects representing subsequent subdirectory levels.
- A tree object contains the references to its component blobs and trees.
- The top level tree is known as the root tree. Therefore, “tree” is often use synonymously with repository.



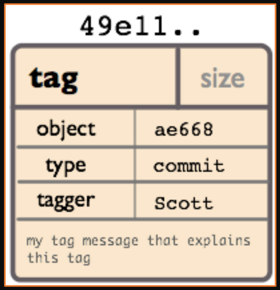


Tags

- Tags are either annotated or lightweight.
  - **Annotated tags** are objects, with fields as shown.
  - **Lightweight tags** are tags that are not objects, but rather references that reside in `.git/refs/tags`
  - Practically the use case is:
    - Normal (annotated) tags should be used to add additional context and ownership.
    - Lightweight tags should be used if you need to create a human-readable alias for a commit.

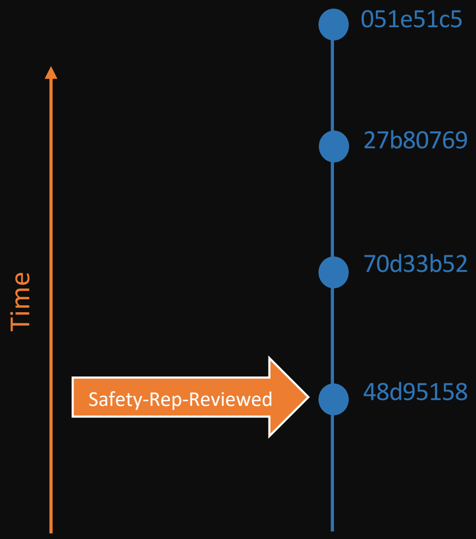
- **Tags are not movable**. They are a permanent pointer to one commit hash unless they are removed.
  - If you remove a tag, you can reuse the name.

Tags



- Tags are useful to mark some commits in a more human-readable way.
- A visual example is on the right. Say we have a product (document, code, drawing, etc.) tracked in a repository.
  - Commit 48d95158 is tagged as Safety-Rep-Reviewed to indicate that safety has signed off on the product, in the form it existed in at that point.
  - Now it is easy to see what has changed since safety reviewed this product. If safety reviewed it again at 051e51c5, we could either:
    - Delete Safety-Rep-Reviewed tag and reassign it to 051e51c5.
    - Create a new unique tag name to describe 051e51c5.

  - What type of tag do we want here?
    - If Safety-Rep-Reviewed just needs to be an alias with no further information, a lightweight tag will suffice.
    - If, for example, the safety representative's name and any non-compliances they found need to be noted with this tag, an annotated tag is desirable.



Note

We have now covered Git objects in detail.

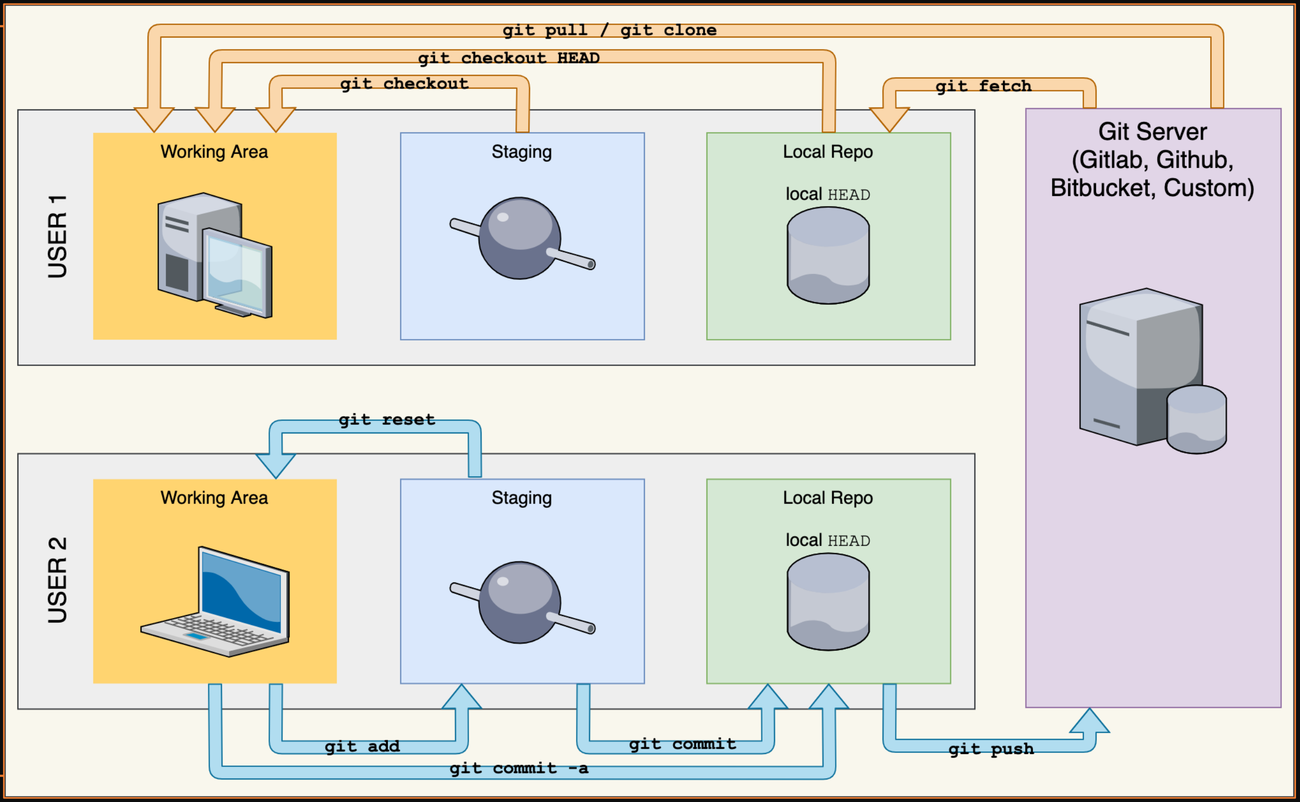
Time for a note:

Q: We have not explicitly called out “code” yet. Why?

A: Git is for **revision control**. NOT just for code. We can and do use Git to control documents, webpages, org charts, drawings, and much more.

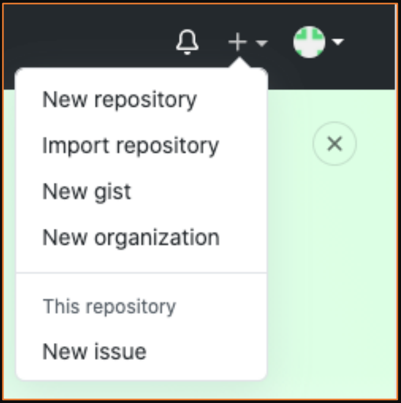
Let’s now examine more practical usage!

Git Operations

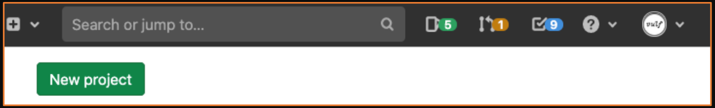


Creating a Repository

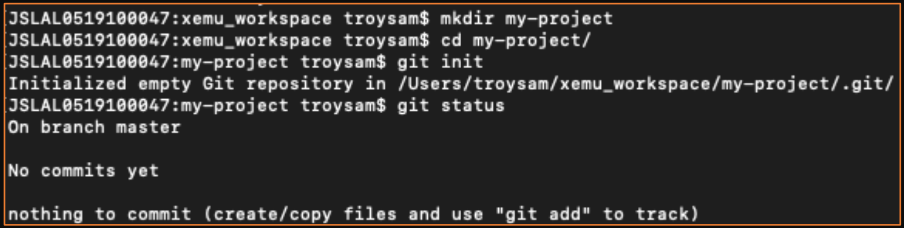
- First things first: how do we get a Git repository started?
  - Most of the time, we're using Git with some service like Github, Bitbucket, or Gitlab.
  - These services will host the remote repositories.
  - It is possible to run our own Git server, but we will not cover that in this class.
  - Using one of the services above, we will have the option to create a repository once signed in with an account.
- We can also create a new repository locally (we will have to manually link it to a remote later on).
  - Simply create a new directory with the name for your project, or use an existing directory that you want to start Git tracking.
  - Run `git init` in the directory. You're done!



Github repo creation



Gitlab repo creation



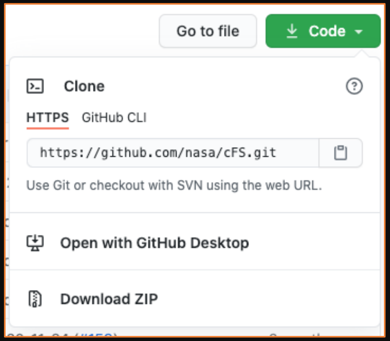
```
JSLAL0519100047:xemu_workspace troysam$ mkdir my-project
JSLAL0519100047:xemu_workspace troysam$ cd my-project/
JSLAL0519100047:my-project troysam$ git init
Initialized empty Git repository in /Users/troysam/xemu_workspace/my-project/.git/
JSLAL0519100047:my-project troysam$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Cloning

- If we did create our repository using a remote service like Github, Bitbucket, or Gitlab, we still need to create a local copy to work on it.
- Likewise, if we want to contribute to or use an existing repository, we also need to create a local repository.

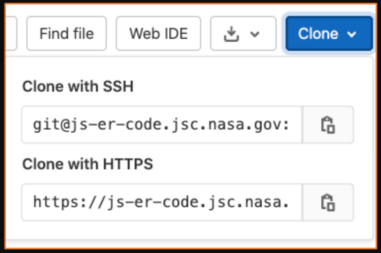
- Enter `git clone`:
  - A repository should provide a SSH or HTTPS URL.
  - Perform `git clone <url>` to download the repository and initialize a local copy.

Now we can get straight into the meat and potatoes of Git operations!

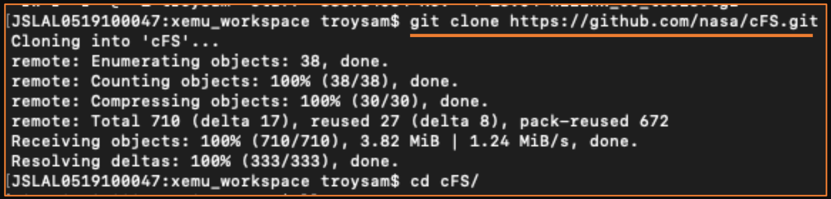


Go to file
Code
Clone
HTTPS
GitHub CLI
https://github.com/nasa/cFS.git
Use Git or checkout with SVN using the web URL.
Open with GitHub Desktop
Download ZIP

Github cloning



Gitlab cloning



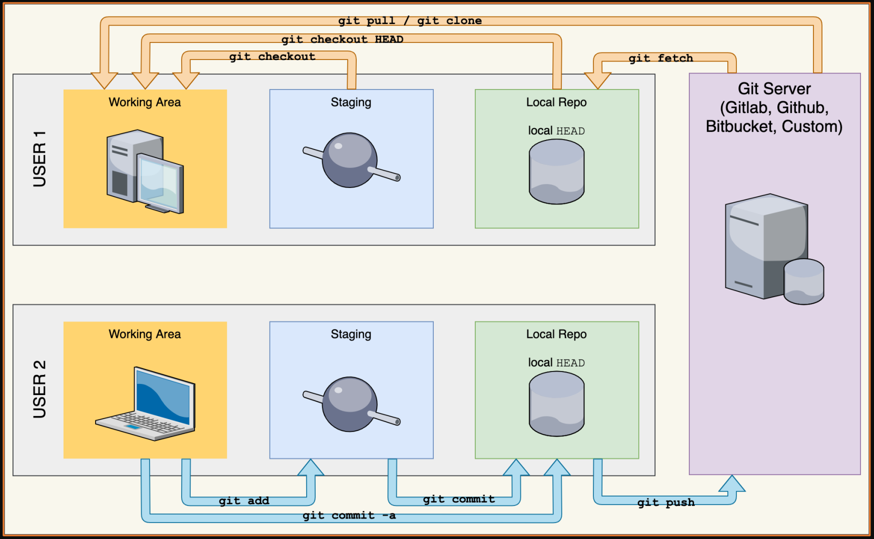
```
JSLAL0519100047:xemu_workspace troysam$ git clone https://github.com/nasa/cFS.git
Cloning into 'cFS'...
remote: Enumerating objects: 38, done.
remote: Counting objects: 100% (38/38), done.
remote: Compressing objects: 100% (30/30), done.
remote: Total 710 (delta 17), reused 27 (delta 8), pack-reused 672
Receiving objects: 100% (710/710), 3.82 MiB | 1.24 MiB/s, done.
Resolving deltas: 100% (333/333), done.
JSLAL0519100047:xemu_workspace troysam$ cd cFS/
```

Creating Commits

- Create a commit with the `git commit` command.
  - `git commit` will take all files with changes that are staged, and create a new commit object that encompasses and describes those staged changes.
    - Changes can be staged using:

      ```
      git add <dir/file1> <dir/file2> ...
      ```
    - Changes can also be staged and committed at the same time by passing the files when you run git commit.
  - Git commit requires* a descriptive message to accompany a commit. When you run `git commit`, you will be prompted to type a commit message.
    - Alternatively, you can directly specify the message when running `git commit` using the `–m` flag:

      ```
      git commit –m ”My commit message.”
      ```

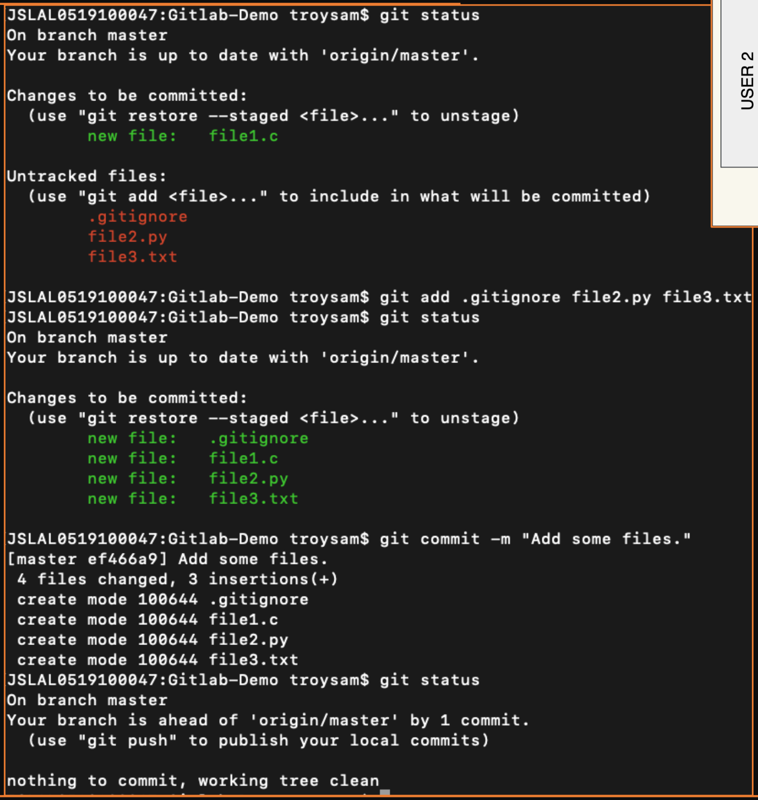


* You can override this, but descriptive messages are good so you shouldn't.

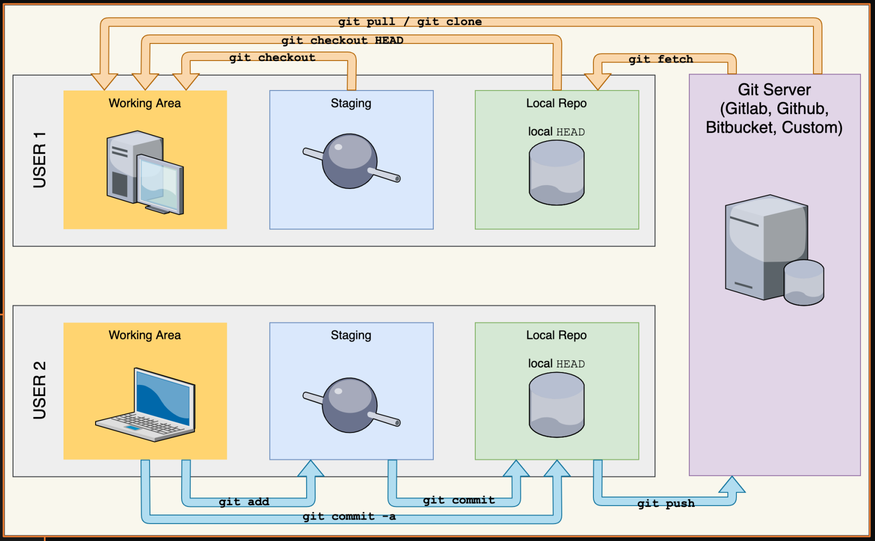
Commits

Example:

Here we start off with `file1.c` in the staging area, and 3 files that are untracked but which we want to add and commit.

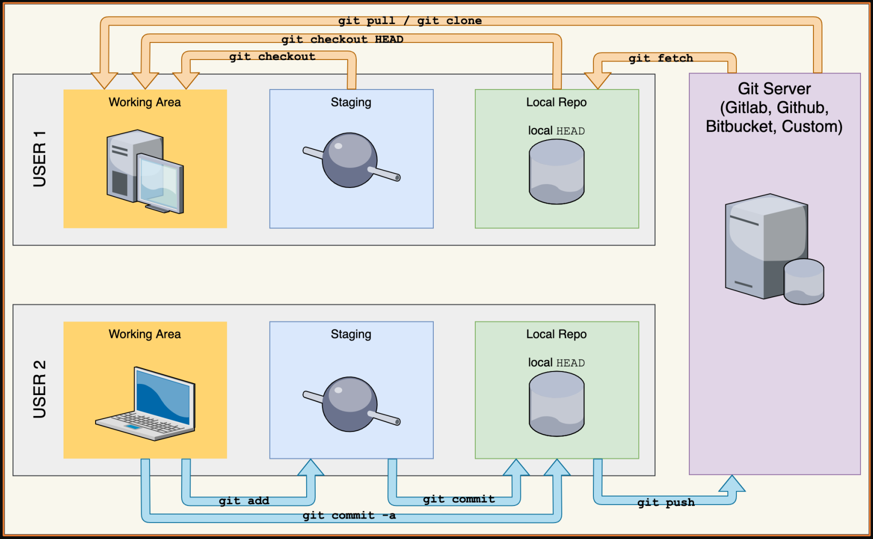


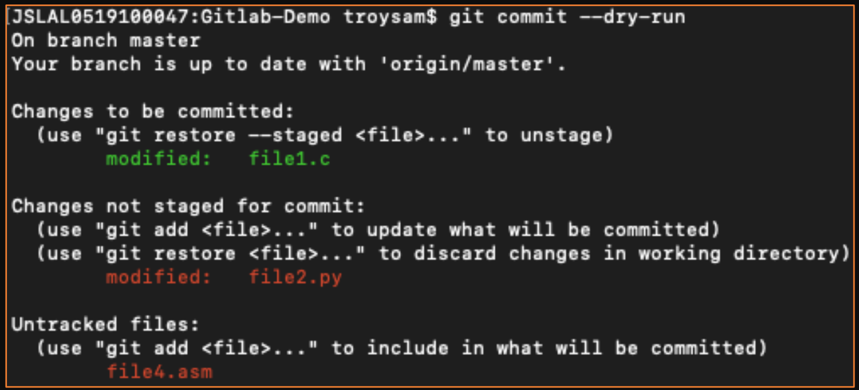
```
JSLAL0519100047:Gitlab-Demo troysam$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   file1.c

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        file2.py
        file3.txt

JSLAL0519100047:Gitlab-Demo troysam$ git add .gitignore file2.py file3.txt
JSLAL0519100047:Gitlab-Demo troysam$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   .gitignore
        new file:   file1.c
        new file:   file2.py
        new file:   file3.txt

JSLAL0519100047:Gitlab-Demo troysam$ git commit -m "Add some files."
[master ef466a9] Add some files.
 4 files changed, 3 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 file1.c
 create mode 100644 file2.py
 create mode 100644 file3.txt
JSLAL0519100047:Gitlab-Demo troysam$ 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
```



Commit Commands

- More commit-related commands:
  - `git commit —a` (or `git commit --all`)
    - Automatically stages all files known to Git and commits them.
    - Be careful with this! Always check `git status` to make sure nothing undesired was added.
    - This will not stage any new files that Git doesn't yet know of.
  - `git commit --dry-run`
    - Perform a "dry-run" commit, without actually creating a new commit.
    - This will output a list of what will be committed, what will not be committed, and what is untracked.

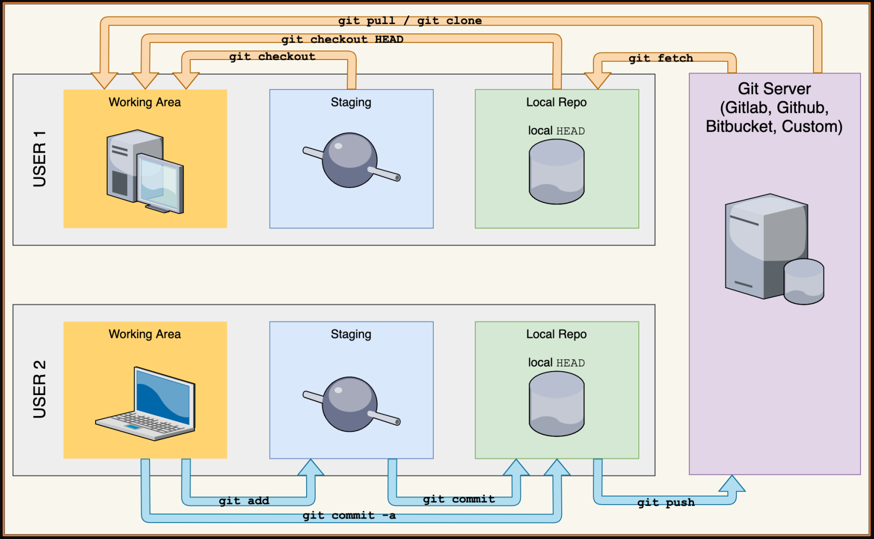




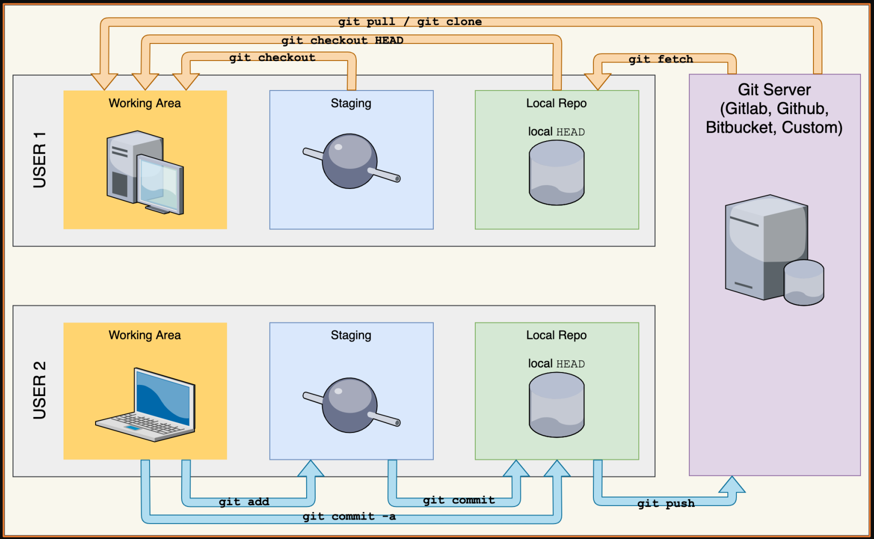
```
JSLAL0519100047:Gitlab-Demo troysam$ git commit --dry-run
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file1.c

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file2.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file4.asm
```

Tracking and Staging

- More commit-related commands:
  - `git add --all <dir/file1> ...`
    - Stage not only modified files, but also:
      - Newly added and deleted files in any subdirectories.
      - Newly added or deleted subdirectories themselves.
    - `git add` on its own will only stage new files and modified files explicitly called out as arguments.
    - `git add -u` stages modified files and deletions, but not newly added files.



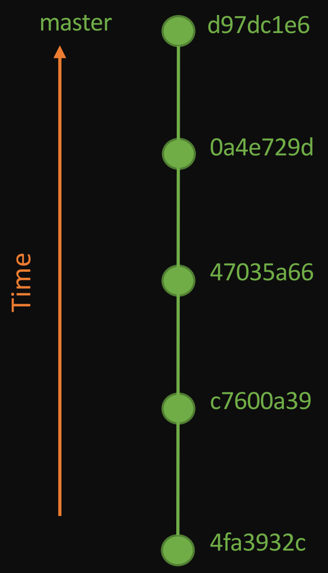
Un-Tracking



- We know that git add lets us add files to Git tracking.
- There is a companion command that lets us do the inverse: remove files from Git tracking.
- `git rm <file>`
  - Note this will also delete the file itself.

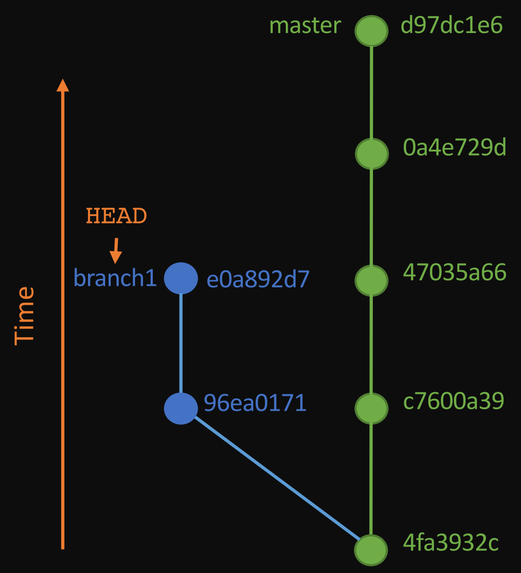
Branches

- Every Git repository has at least one branch: master
- What is a branch? **A moving lightweight tag**.
- A branch is **not** a sequence of commits.
  - Recall that lightweight tags are simply aliases (with no additional information) to a commit.
  - This is what makes Git so lightweight, fast, and versatile.
- Ex: On the right, the branch master currently exists as a tag aliasing `d97dc1e6`.
- The branch tag is moved with a `git commit`.



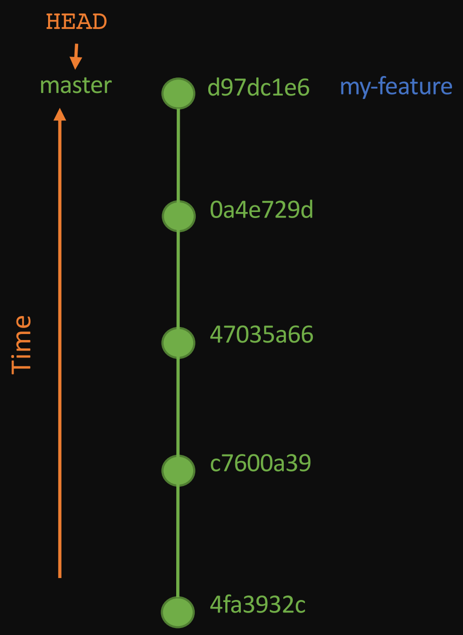
Branches - HEAD

- Branches are pointers, commits are objects, where am I???
  - There is another pointer that represents the temporal location of our local repository: `HEAD`
  - On the right, our `HEAD` pointer is at branch1, signifying that the current state of our local repository is the snapshot represented by e0a892d7.
- You can change your `HEAD` to different commits or tags using `git checkout`.
  - Ex: `git checkout master` will change our `HEAD` to point to the commit with the master branch tag.
  - More on this coming up.



Creating a Branch

- How do I create a branch?
  - Say we are currently on master (d97dc1e6) and we want to create a new branch on which we will implement a new feature.
  - `git branch my-feature`
  - New branch pointer my-feature points to the same commit as master.
  - However, HEAD is still pointing at master, so if we want our future commits to advance the my-feature branch tag, we need to make HEAD point to my-feature instead of master.
    - `git checkout my-feature`
    - Now, we can check `git status` to verify that HEAD is now pointing to the my-feature branch.

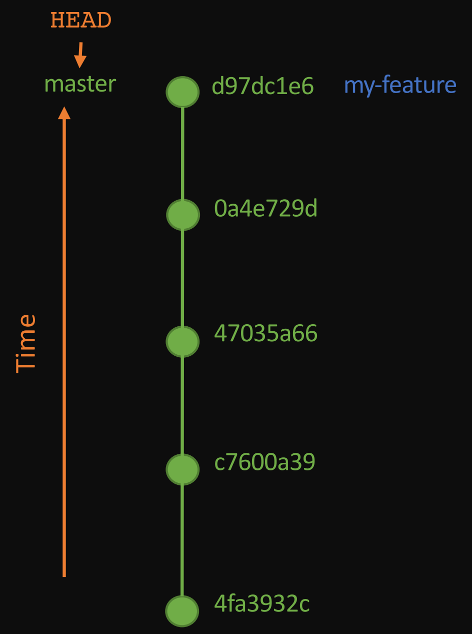


Easier Branch Creation

- How do I create a branch?
  - Shortcut: in the same scenario, we can simultaneously create a new branch and swap our HEAD to track it by simply performing `git checkout -b my-feature`.
  - The `–b` tells git that this is a branch operation, and git interprets this single line as:
    - `git branch my-feature`
    - `git checkout my-feature`

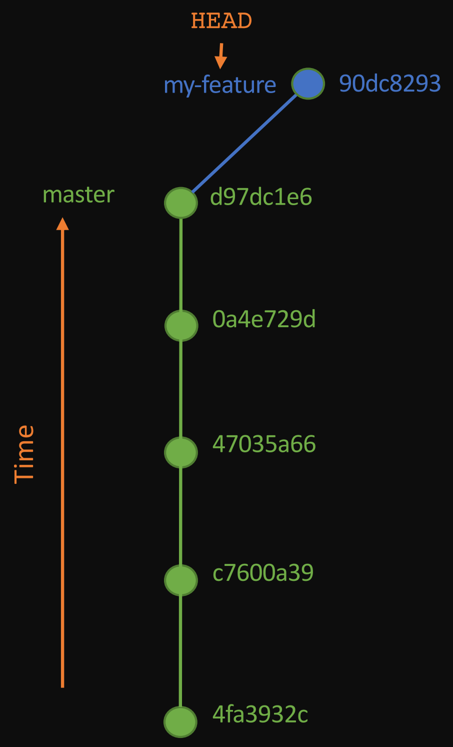
    Which is exactly what we did earlier!

- This shortcut applies to creating a new branch locally, if the branch already exists, all we need to do is `git checkout` the branch.



Using a Branch

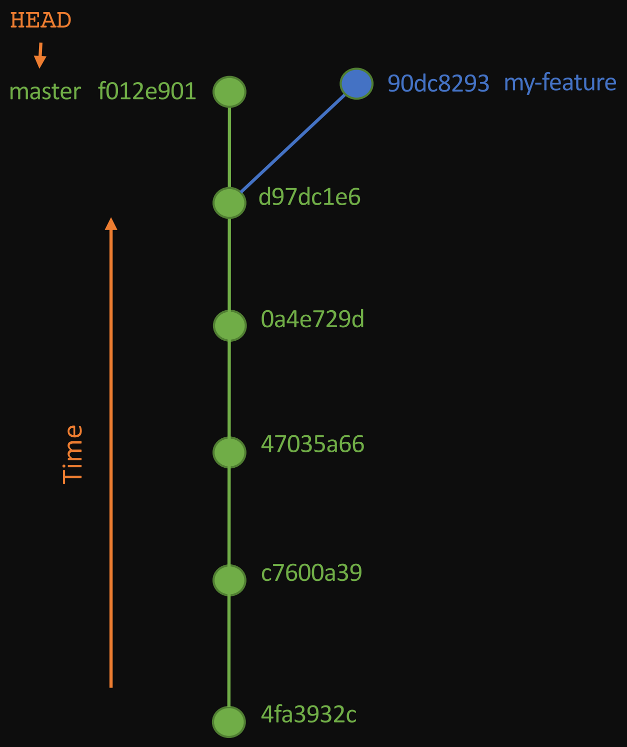
- What happens when I start using a branch?
  - Let’s say we created the my-feature branch as described in the previous slide.
  - When we make some modification to the repo and commit it, the `HEAD` will always advance to the new commit.
    - This is because the commit is the current state of your repository, and the `HEAD` is always pointing at the current local state.
  - Since the commit was made with the my-feature branch checked out, the my-feature tag is also advanced to the new commit.
  - We now have the structure shown on the right, where master still points to the parent of 90dc8293.



Changing Branches

- Now let’s say we are not sure whether we like my-feature. But we still have some updates that need to be made.
- We decide to update master directly.
  - To switch our `HEAD` back to master, we execute `git checkout master`.
  - We make some changes and commit once more.
  - Our tree now looks like this (a little more tree-like) ->
- `HEAD` is again pointing to the commit we just made, and since we had master checked out, the master branch tag is pointing at the commit as well.

- Now we’re comfortable with creating and swapping branches. But what if we want to swap to a commit that is not “at the tip of a branch”?

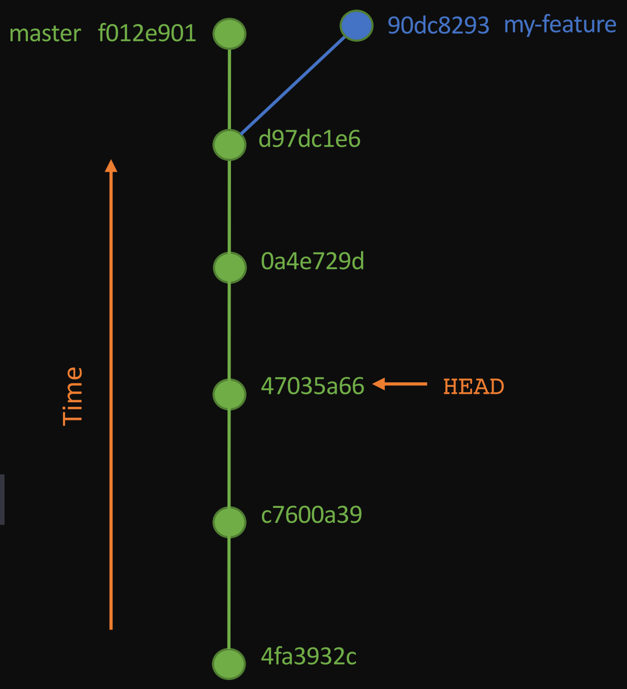


Detached HEAD

- If we change our head to a commit that is not aliased with a tag, then we are in a **“detached `HEAD`”** state.
  - In this state, we generally don’t want to make changes, since we do not have a tag to track the most recent commit.
- In our current example, let’s say we execute `git checkout 47035a66`
  - This commit is not associated with a branch tag, but performing a checkout means that our `HEAD` now points to 47035a66.
  - When we checkout, Git will inform us

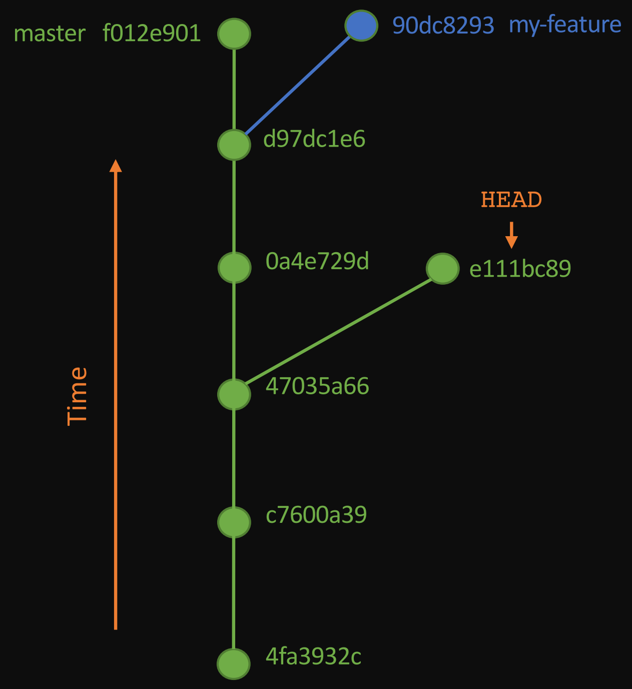
```
You are in 'detached HEAD' state...
```

- Why don’t we want to make changes in this state?



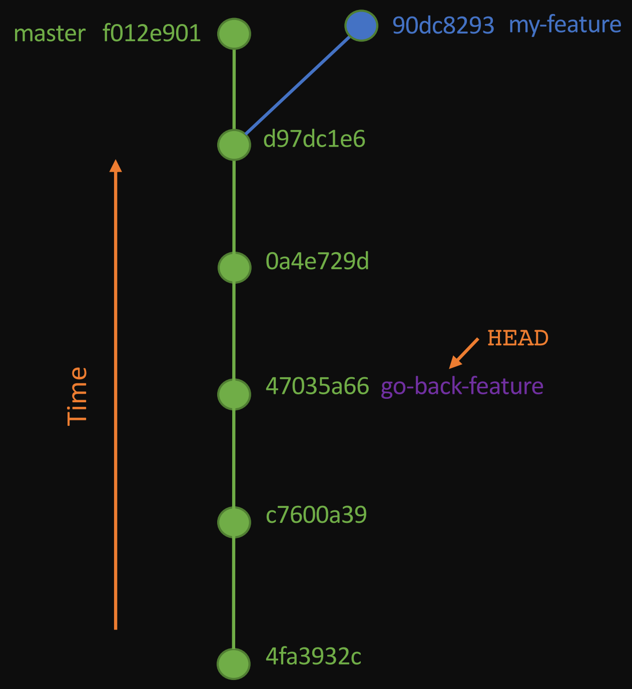
Detached HEAD

- Why don't we want to make changes in this state?
  - Answer: No branch context.
  - Example: Say we make changes and commit them while 47035a66 is checked out.
    - Our `HEAD` is, of course, pointing to the current repository state which is the commit we just made (e111bc89).
    - If we look at the tree, however, we can see that there is no branch tag corresponding to this physical "branch in the tree".
    - This means that when you check out a different commit or branch, **the only way to get back to this commit is by finding the commit hash again**.
      - This makes it likely for this commit to get lost in the haystack and/or forgotten.



Detached HEAD - Solution

- Solution to detached HEAD state?
  - Just make a branch!
- If you find yourself in a position where you want to make changes starting from a commit that does not have a branch tag, create a new branch tag at that commit.
  - In our working example, let's pretend we didn't commit in a detached `HEAD` state and let's instead execute:
  - `git checkout –b go-back-feature 47035a66`
  - Now we have a new branch tag to track our changes, and we have the new branch checked out.
  - Our tree looks like the right, and executing `git status` will confirm we have the go-back-feature branch checked out.



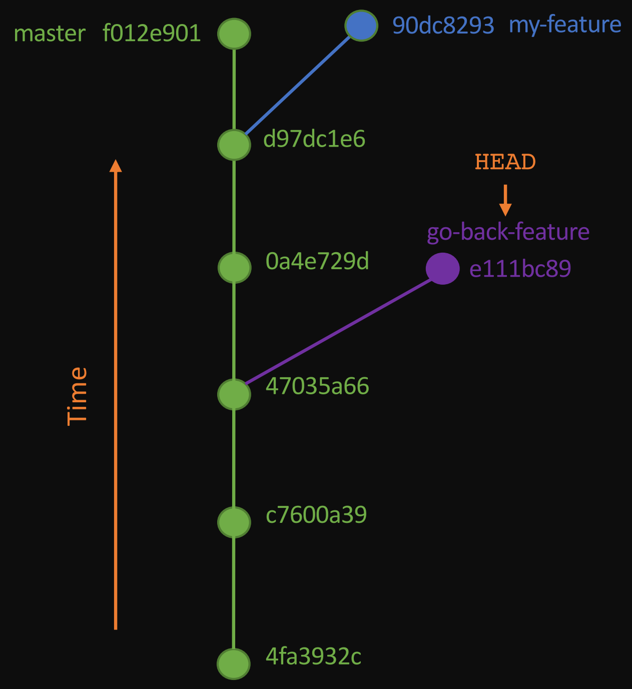
Detached HEAD - Tips

- If we make a commit from this point, we see that go-back-feature advances with HEAD.
- We can now checkout other commits and branches with the knowledge that we can easily return to the go-back-feature branch whenever we want.

- **Useful**: We can always list all the branch tags known to our local repository by simply executing `git branch`
  - In our current local repo, it would produce the following list:

```
   master
   my-feature
 * go-back-feature
```

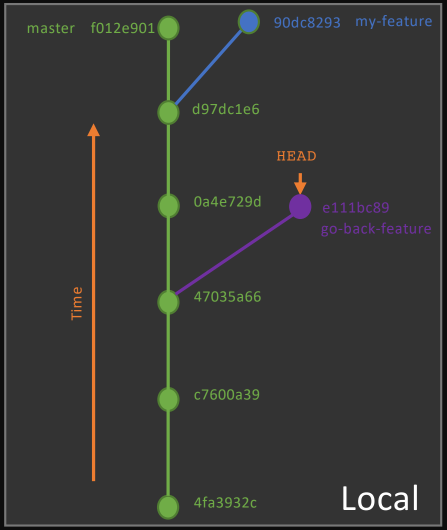
  - Note that the currently checked out branch will be marked with an asterisk.

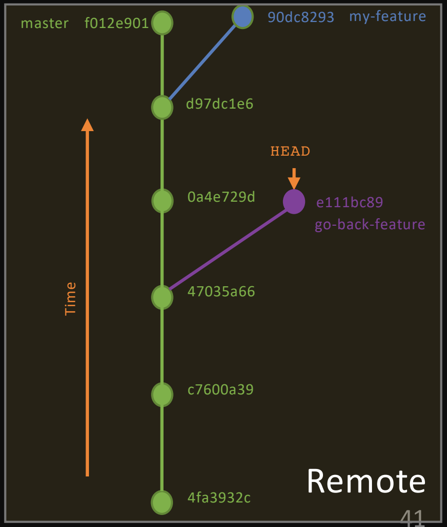


Branching Tips

- We can also see Git's list of all known branches both locally and remotely by appending the `—a` flag:
  - Example: Let's assume that we have `git push`-ed all our local changes to the remote repository, and the remote tree is now identical to the local tree (shown on the right).
  - `git branch —a` has the following output:

```
  master
  my-feature
* go-back-feature
  remotes/origin/master
  remotes/origin/my-feature
  remotes/origin/go-back-feature
```





Recap - Local Operations

So far, we have discussed how to create and work in your repository locally.

We should now understand:

- How to create or clone a repository.
- The distinction between our working area, staging area, and local repository.
- How to track files, untrack files, and commit changes to files.
- What a branch is, and how to use branches.

If any of the above is unclear, let's take this opportunity to clarify!

Otherwise, let us now discuss working with the remote repository.

Pushing

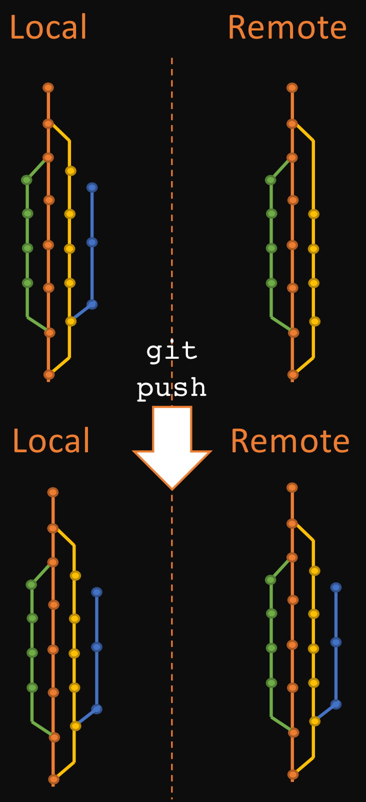
- When we `git commit`, the commit is created and stored in our local repository.
- In order to update the remote repository (and allow external users to view/access your changes) we need to `git push`.
  - `git push <dest-repo> <refs>` will take the refs and objects modified in our local repository and send them to the remote repository.
  - If we just want to update one branch, for example, we can `git push origin my-branch`.

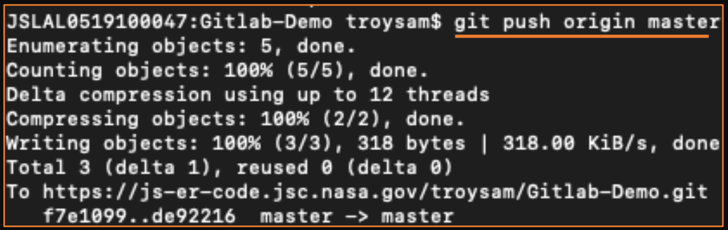
- If the local name and the remote name differ, you will need to specify both:

```
git push origin <local branch name>:<remote branch name>
```

- If we have created a branch locally that is not present on the remote, we can create the remote ref and link it to our local ref while pushing with the `-u` (`--set-upstream`) flag:

```
git push -u origin my-branch-name
```

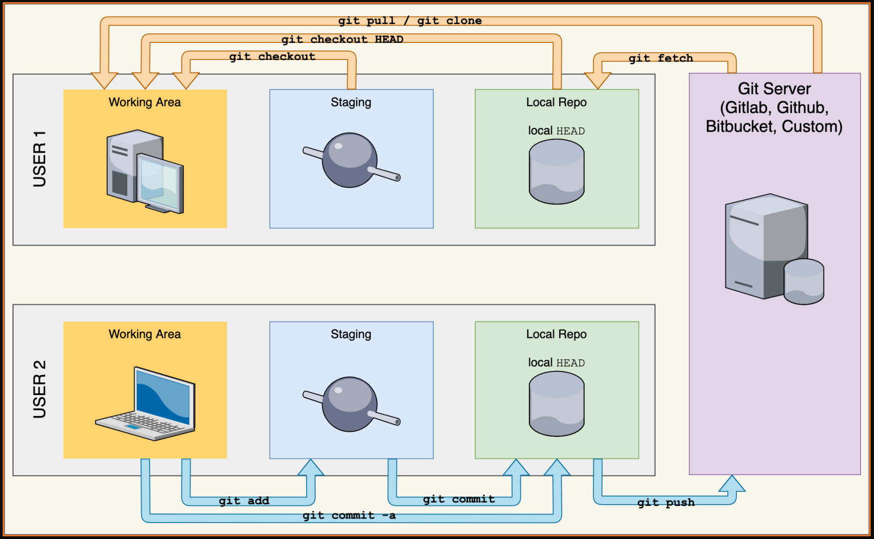




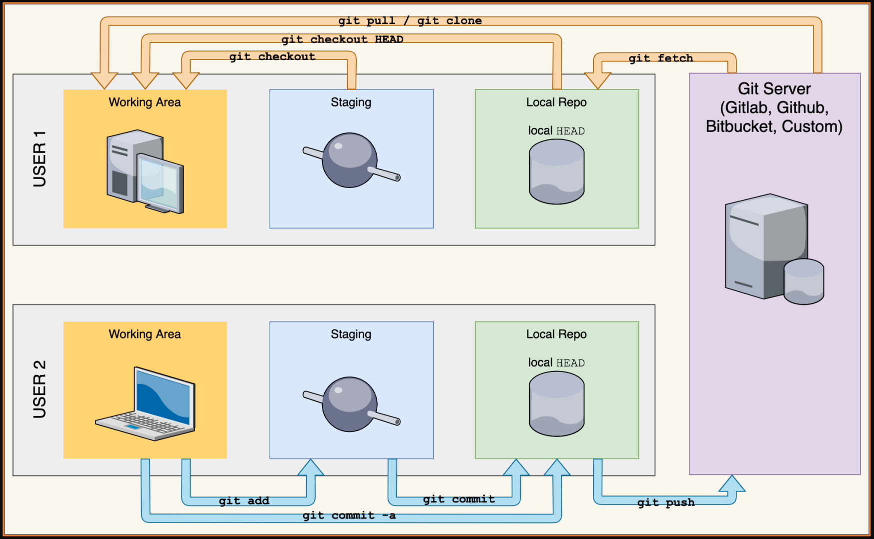
```
JSLAL0519100047:Gitlab-Demo troysam$ git push origin master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 318 bytes | 318.00 KiB/s, done
Total 3 (delta 1), reused 0 (delta 0)
To https://js-er-code.jsc.nasa.gov/troysam/Gitlab-Demo.git
   f7e1099..de92216  master -> master
```

Fetching

- Conversely to pushing, we need a mechanism to grab changes to the tree from the remote and "**fetch**" them into our local repository.
  - Remember that our local repository is not the same as our local working area!
  - When we `git fetch`, we augment our local tree with the changes present in the remote tree.
    - Branches and tags (refs)
    - Objects (commits, blobs, trees)
  - Fetching does not incorporate any changes into the working area (current state of the local repository).
  - After we fetch, `git status` will inform us if/when there are remote changes to the currently checked-out branch.
    - If there are changes, we will not be able to push to that branch until we have `git merge`-d the changes into our working area (to prevent overwriting).

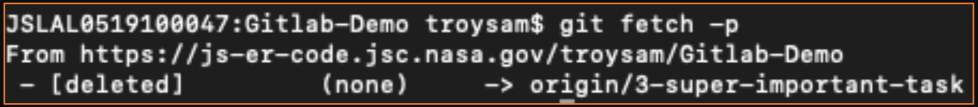


Fetching



- `git fetch -p` (`git fetch --prune`) will fetch and remove any remote-tracking refs that no longer exist on the remote.

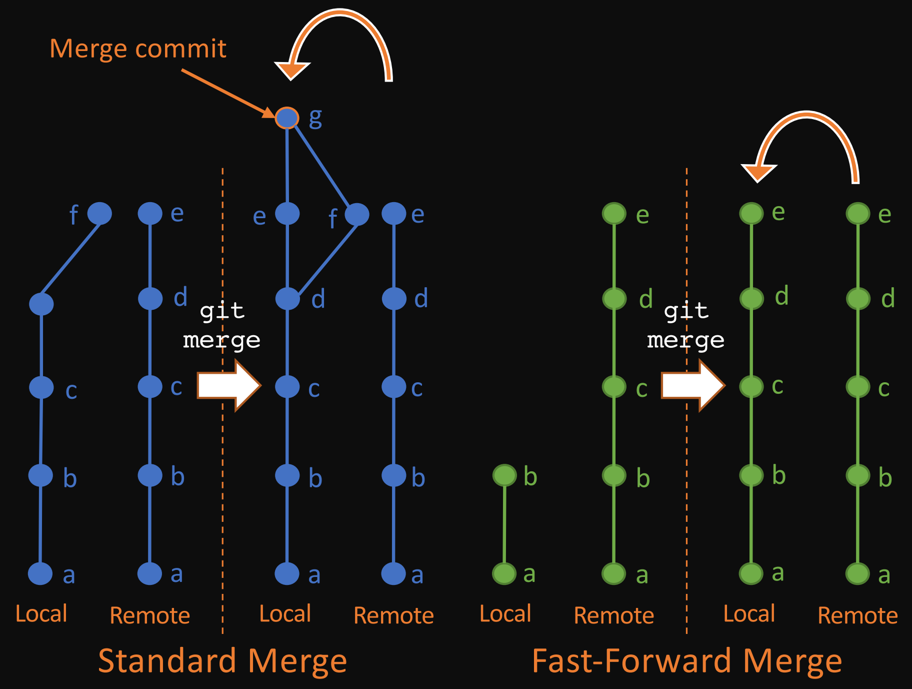
  Example (right):
  - We have deleted a branch called "`3-super-important-task`" from the remote repository.
  - When we `git fetch -p`, the local ref that tracked the remote branch "`3-super-important-task`" is deleted.
  - We are notified of changes to refs when `git fetch` is run.



```
JSLAL0519100047:Gitlab-Demo troysam$ git fetch -p
From https://js-er-code.jsc.nasa.gov/troysam/Gitlab-Demo
 - [deleted]         (none)     -> origin/3-super-important-task
```

Merging

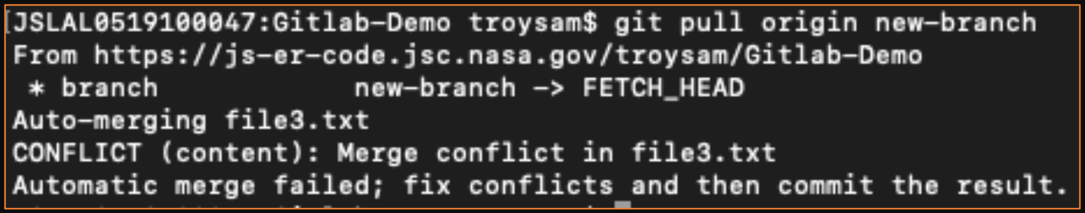
- Once we have fetched, our local repository is up-to-date with the remote. We can now "**merge**" these remote changes with our local changes.
  - Recall: Merge commits - commits with two parents.
- `git merge <ref>` - local tag/branch
- `git merge origin/<ref>` - remote tag/branch
  - If there is a linear path from the HEAD to the target branch, then we are performing a fast-forward merge.
  - Otherwise, we perform a standard merge.



Merging

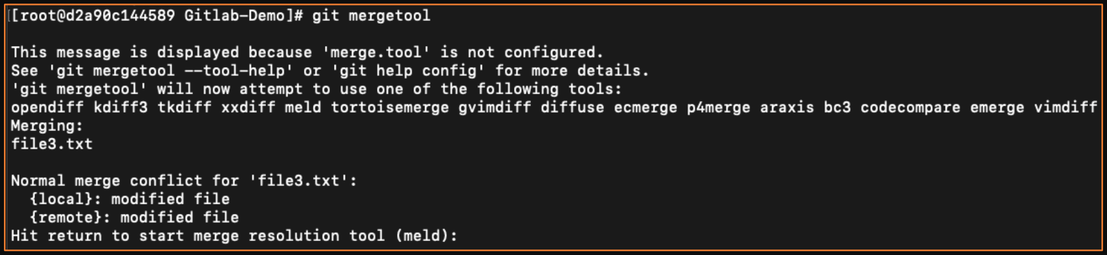
- Example: Say we are updating a document called `webpage.html` to add a search bar, using a branch called "`cool-features`".
  - When we have finished creating the search bar locally (and committed our changes), we `git fetch` and find that our colleague has updated the color palette.
  - The new color palette is very pretty, so we perform a `git merge origin/cool-features` to grab the new/changed refs and objects from the `cool-features` branch and incorporate them into our working area.
  - There are no conflicts (yay!) so Git can perform the merge and leaves it to us to create the merge commit.
    - We could also have told Git to create the merge commit for us: `git merge origin/cool-features --commit`.

Merge Conflicts



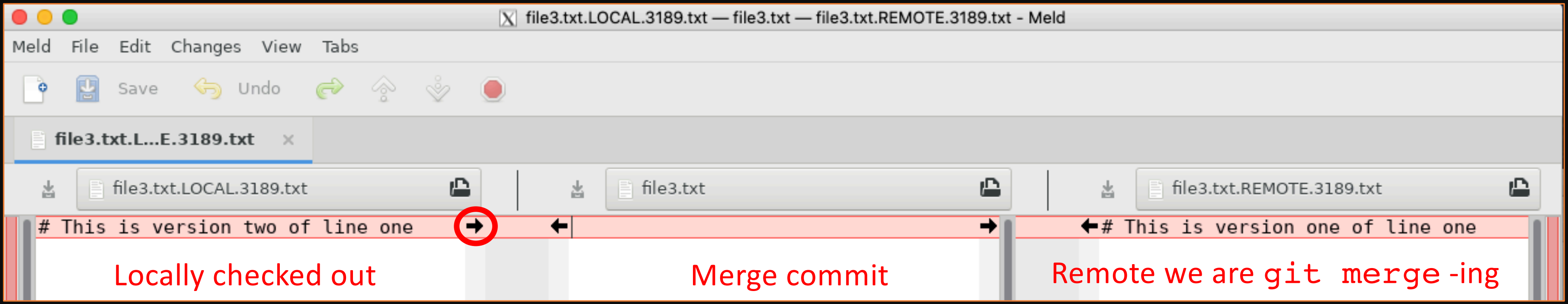
```
[JSLAL0519100047:Gitlab-Demo troysam$ git pull origin new-branch
From https://js-er-code.jsc.nasa.gov/troysam/Gitlab-Demo
 * branch            new-branch -> FETCH_HEAD
Auto-merging file3.txt
CONFLICT (content): Merge conflict in file3.txt
Automatic merge failed; fix conflicts and then commit the result.
```

- What if there are merge conflicts?
  - This is common when multiple people are working on the same files.
  - Merge conflicts require human intervention.
  - Git does its best to auto-merge, but ultimately we must examine the content of a file from each **parent commit**, and determine what content we want in the **merge commit**.
  - Rather than doing this manually, it is easiest to use a **merge tool**. There are many available, and the appendix to these slides contains a quick reference for one of them.
  - Using `git mergetool` to call up our merge tool of choice, we can resolve the merge conflict file-by-file and when complete, Git will generate the merge commit.



```
[[root@d2a90c144589 Gitlab-Demo]# git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse ecmerge p4merge araxis bc3 codecompare emerge vimdiff
Merging:
file3.txt

Normal merge conflict for 'file3.txt':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (meld):
```

Merge Conflicts


file3.txt.LOCAL.3189.txt — file3.txt — file3.txt.REMOTE.3189.txt - Meld
Meld File Edit Changes View Tabs
Save
Undo
file3.txt.L...E.3189.txt
file3.txt.LOCAL.3189.txt
file3.txt
file3.txt.REMOTE.3189.txt

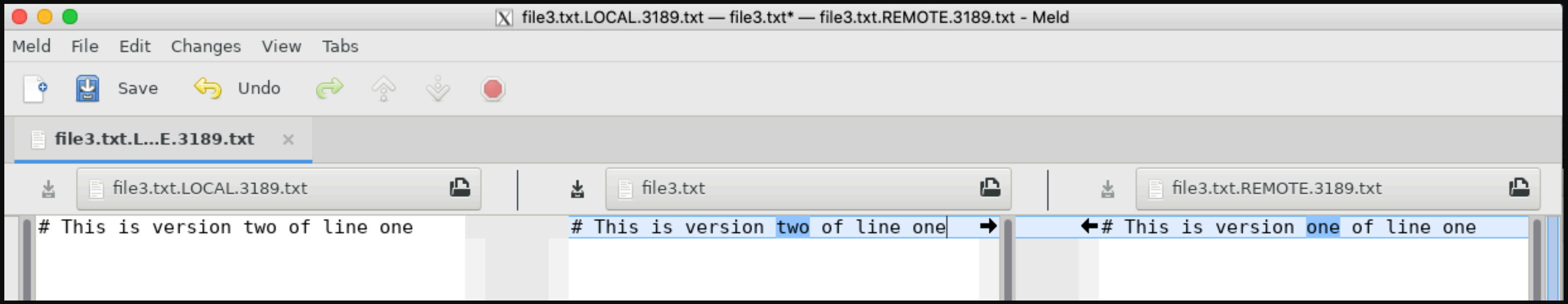
```
# This is version two of line one
```

```
# This is version one of line one
```

Locally checked out

Merge commit

Remote we are `git merge` -ing


file3.txt.LOCAL.3189.txt — file3.txt* — file3.txt.REMOTE.3189.txt - Meld
Meld File Edit Changes View Tabs
Save
Undo
file3.txt.L...E.3189.txt
file3.txt.LOCAL.3189.txt
file3.txt
file3.txt.REMOTE.3189.txt

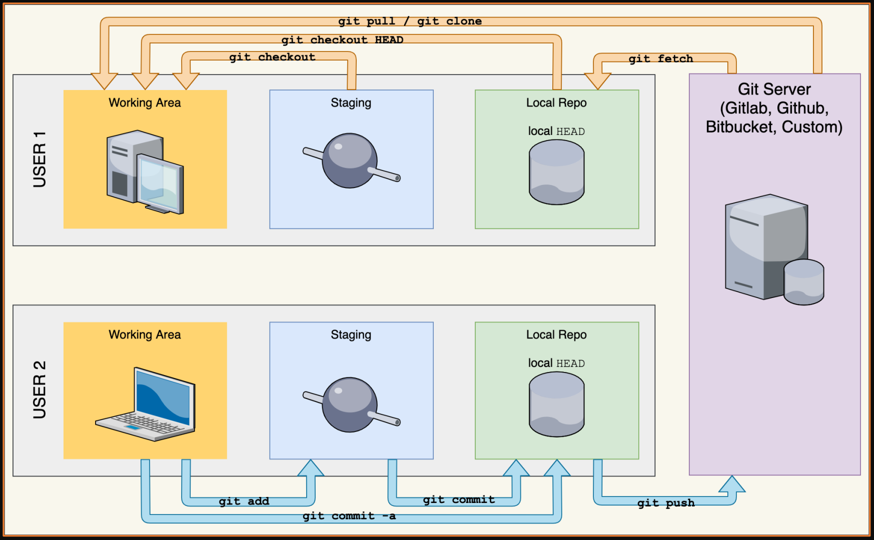
```
# This is version two of line one
```

```
# This is version two of line one
```

```
# This is version one of line one
```

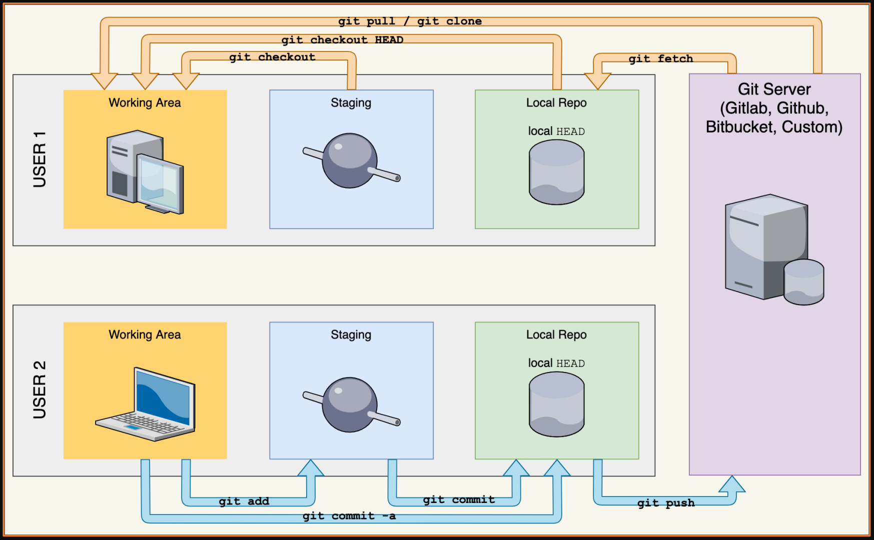
Pulling

- Git also incorporates a feature that lets us both fetch and merge with one command: `git pull <remote> <ref>`
  - `git pull` is a `git fetch` followed by a `git merge`.
    - By default, `git pull` with no arguments is equivalent to `git pull origin`.
      - This will **fetch** all refs from the remote not present locally. It will then **merge** the remote branch corresponding to your current checked out branch into your working area.
  - It is generally a good idea to be precise with `git pull`.
    - If we want to pull the remote changes from branch "`feature-1`", specify `git pull origin feature-1`.
  - In practice, we will most often use `git pull` rather than fetching and merging.



Pulling

- Frequent use case:
  - Before creating a new branch, we want to make sure that it's based upon the most recent changes (i.e. the latest `origin/master`).
  - We `git pull origin master` to **fetch** and **fast-forward merge** our local `master` branch so that it is identical to the remote `master` branch.
  - Then we can `git checkout -b my-new-branch`.



Recap - Less than it Seems!

We have just covered:

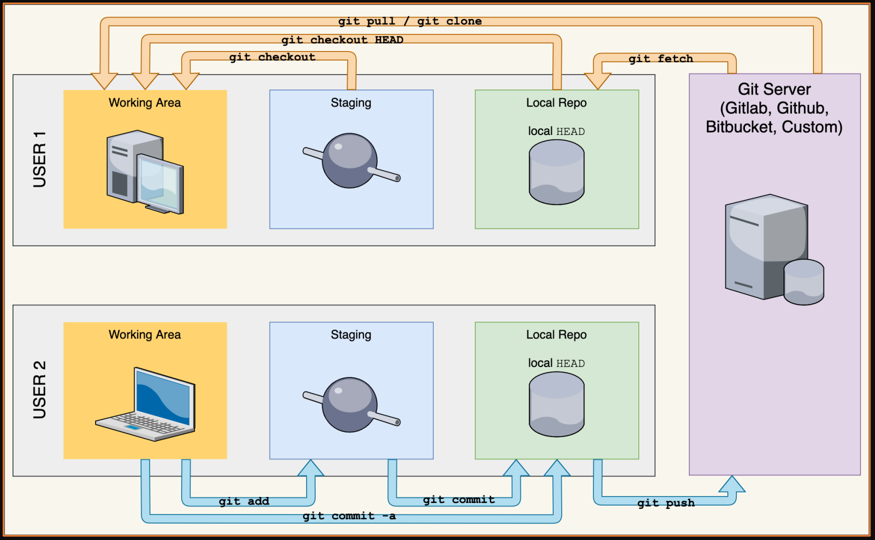
• `git init`, `git clone`	to set up our repository.
• `git status`	to inform us of our repository state.
• `git add`, `git rm`, `git commit`	to make local changes.
• `git branch`, `git checkout`	to organize work.
• `git push`	to update our remote repository.
• `git fetch`, `git merge`, `git pull`	to update our local repository.

We have also shown that day-to-day, you can get by with:

```
git add      git commit   git checkout
git push     git pull     git status
```

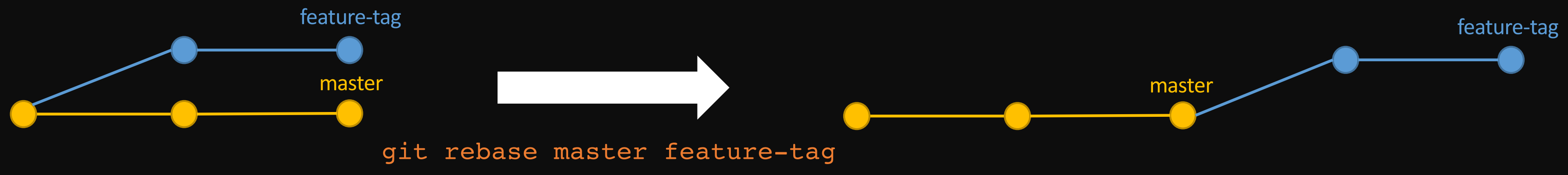
Checkout

- We've discussed how to use `git checkout` to select which branch is active in our working area.
- We can also use git checkout on a file level, to check out single files from another ref (branch, commit hash, tag):
  - `git checkout <ref> <file1> <file2> ...`
- Example:
  - `git checkout f0abc1e1 diagram.pdf config.ini`
    - Restore the revisions of `diagram.pdf` and `config.ini` from the commit `f0abc1e1`.
    - Only these two files are reverted, all other changes in the current working area are unaffected.



Rebasing

- Rebasing is the process of moving a commit or sequence of commits from one base commit to another.



```
git rebase master feature-tag
```

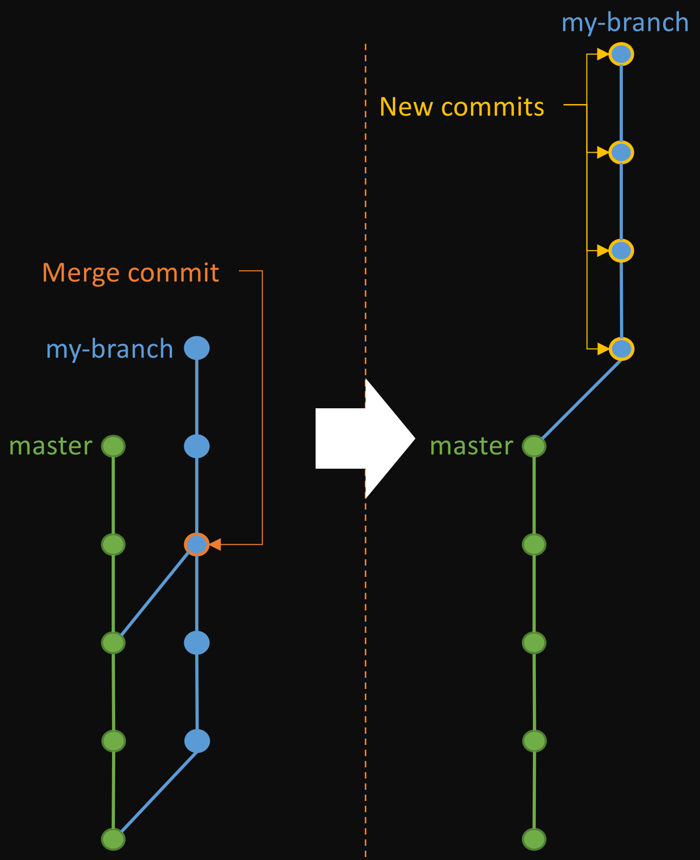
- Despite looking like a normal merge at first glance, the crucial caveat here is that **the result is composed entirely of new commits**.
- Rebasing is a tradeoff.
  + We gain linearity in your tree, which makes the history more human-readable.
  - We lose the project history that existed on that branch/tag prior to rebasing.
- Rebasing is often used as a method of “cleaning up” a messy commit history after the fact. (ex. WIPs)

Rebasing

- Rebasing is best thought of as an alternative to merging: in both cases we are bringing two refs together.
- Benefit: when you rebase, any intermediate merge commits are discarded.

- The general rule to rebasing:
  - Only rebase local commits that have not been pushed.
    - Never rebase commits that someone else may have.
- If you keep that in mind, you can safely use rebasing in your workflow as an alternative to merging.

- Note: rebasing can become very complex, we have covered only the basic usage here.



```
git rebase master my-branch
```

Stashing

- Often, we will be working on multiple branches concurrently, and may want to switch between them without creating a commit just yet.
- Once again, Git provides a handy solution for us: `git stash`.
- `git stash` will take all of the changes in our working area and save it to a local stack. We can then pull those changes back off the stack and into the working area whenever we are ready.
  - Stashing is branch-independent, meaning we can stash from one branch, swap to another, and apply the stashed changes there.
- The basic stash operations are:
  - `git stash` (equivalent to `git stash push`) - save all changes since the last commit to the stack.
  - `git stash pop` - restore the last stash from the stack, and remove it from the stack.
  - `git stash apply` - restore the last stash from the stack, but don't remove it from the stack.
  - `git stash clear` - delete all stashes from the stack.

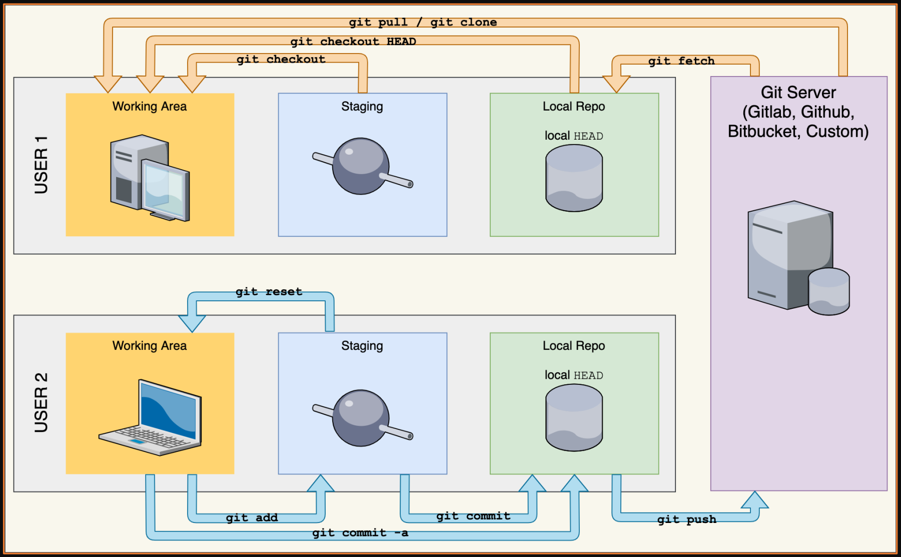
Stashing

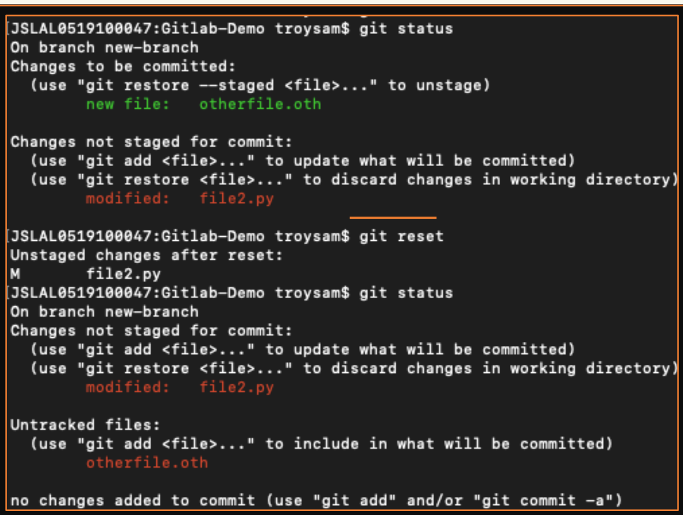
- Other handy stashing operations:
  - `git stash list` - show the entire stack of saved stashes.
  - `git stash apply <selected-stash>` / `git stash pop <selected stash>` - apply/pop a specific stash from the stack, rather than the most recent stash.
  - `git stash --patch` - pick and choose which changes to stash, rather than stashing everything.

- We can also create a new branch directly from a saved stash:
  - `git stash branch my-new-branchname <stash>`
    - Creates and checks out a new branch called `my-new-branchname`, starting from the commit at which `<stash>` was created, and applies `<stash>` to the working area.

Resetting

- There are times when we may `git commit` in error and wish we could just click “undo”.
  - The same goes for `git add`, and un-staging changes.
- Thankfully, we can ”undo” in both cases using another Git feature: `git reset`.
- Examples:
  - Undoing `git add`:
    - `git reset`: un-stage files staged by `git add`, but keep the changes in the working tree.
    - `git reset -- <file>`: un-stage a single file, but keep its changes in the working tree.

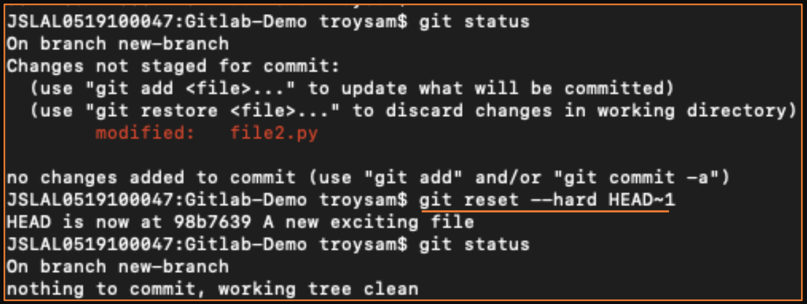




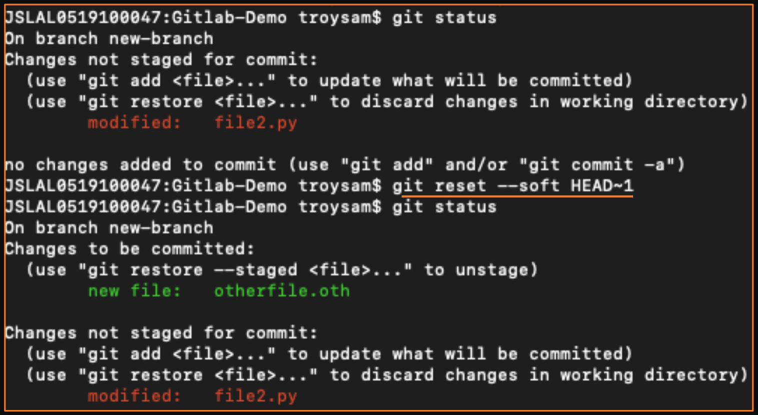
```
JSLAL0519100047:Gitlab-Demo troysam$ git status
On branch new-branch
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   otherfile.oth

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file2.py

JSLAL0519100047:Gitlab-Demo troysam$ git reset
Unstaged changes after reset:
M       file2.py
JSLAL0519100047:Gitlab-Demo troysam$ git status
On branch new-branch
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file2.py

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

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

Resetting

- Undoing `git commit`:
  - `git reset --hard HEAD`: rolls back your working area to the last commit (`HEAD`), deleting all local changes.
  - `git reset --hard HEAD~1`: rolls back your working area to the commit before last (i.e. HEAD - 1), deleting all local changes in addition to the last commit.
  - `git reset --soft HEAD~1`: rolls back your working area to the commit before last, but preserves local changes and keeps the rolled back commit's content in staging.



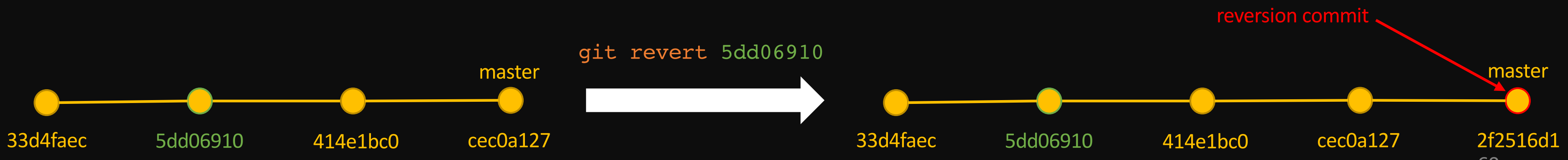
```
JSLAL0519100047:Gitlab-Demo troysam$ git status
On branch new-branch
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file2.py

no changes added to commit (use "git add" and/or "git commit -a")
JSLAL0519100047:Gitlab-Demo troysam$ git reset --hard HEAD~1
HEAD is now at 98b7639 A new exciting file
JSLAL0519100047:Gitlab-Demo troysam$ git status
On branch new-branch
nothing to commit, working tree clean
```



```
JSLAL0519100047:Gitlab-Demo troysam$ git status
On branch new-branch
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file2.py

no changes added to commit (use "git add" and/or "git commit -a")
JSLAL0519100047:Gitlab-Demo troysam$ git reset --soft HEAD~1
JSLAL0519100047:Gitlab-Demo troysam$ git status
On branch new-branch
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   otherfile.oth

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file2.py
```

Reverting

- There is another option for undoing a commit, which has its own use case: `git revert`.
- What if we want to undo a commit that we have already pushed to the remote?
  - It is dangerous to change the project history once an object or ref makes it to the remote.
  - Remote project history = shared history.
  - For this reason, we need a “safe” operation to undo on the remote.
- `git revert <my-commit>` will simply create a new commit which is the inverse of `my-commit`.
  - This is powerful not only because it preserves the history, but it also allows us to safely undo a commit **anywhere in the history**.



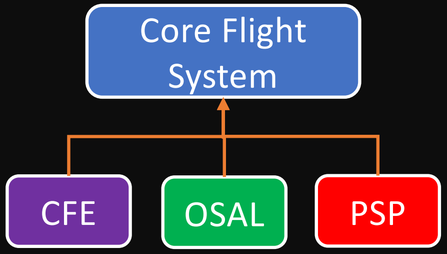
Submodules

- Git submodules are one method of using one Git repository inside another.
  - Example: A system-level document that incorporates multiple subsystem-level documents.
  - Example: Finer-grained access restrictions to a large repository.
- A submodule is incorporated as a **reference** to a specific commit in a repository. **They do not track branches or tags**.
  - This means that we control when to update the state of the outside repository we are tracking.
- The benefit: We can use another repository inside our own without needing to copy its content (and thus lose its Git history).

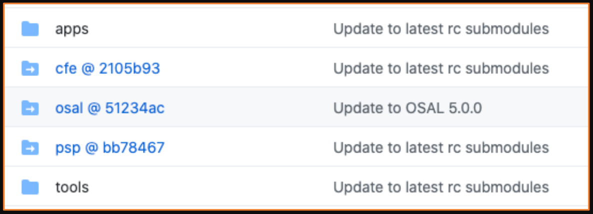
- To visualize this, lets look at a real-world example from NASA's open source software:
  - Core Flight System (CFS)

ref: https://github.com/nasa/cFS

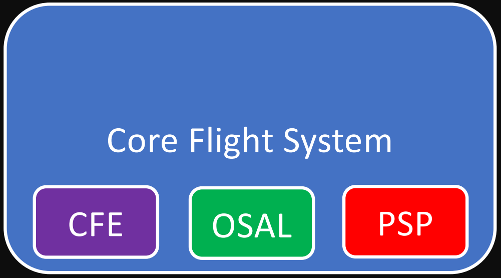
Submodules



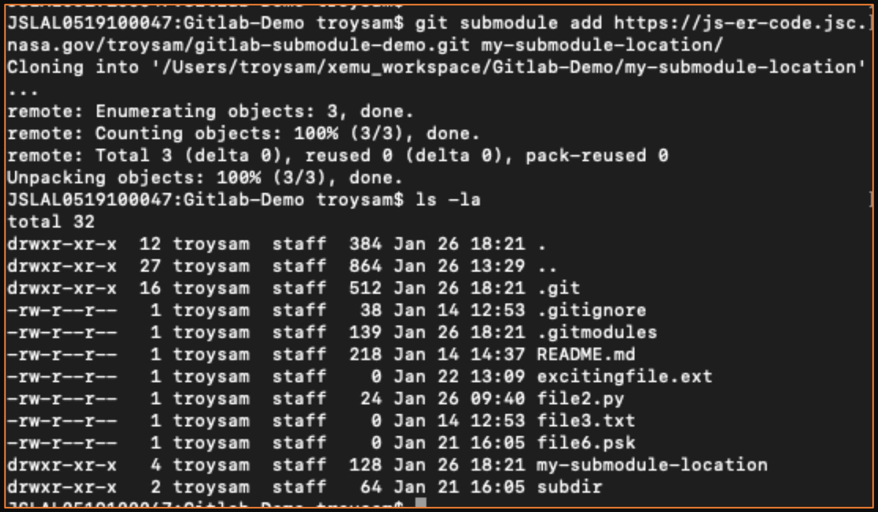
- The Core Flight System framework is composed of three components, plus some apps and tools that utilize those components.
  - Each of these components (CFE, OSAL, PSP) is tracked as a submodule.
  - The components are developed independently by other groups, and have no inter-relation.
- 7 months ago, the OSAL component was updated to release 5.0.0.
  - Thanks to the submodule relation, this was not automatically updated in the Core Flight System repository.
  - The Core Flight System developers were able to test and fix issues introduced by this version, and then ”step up” the submodule commits of OSAL and the other components when they were ready to do so.
  - This demonstrates the basic submodule workflow!



apps	Update to latest rc submodules
cfe @ 2105b93	Update to latest rc submodules
osal @ 51234ac	Update to OSAL 5.0.0
psp @ bb78467	Update to latest rc submodules
tools	Update to latest rc submodules

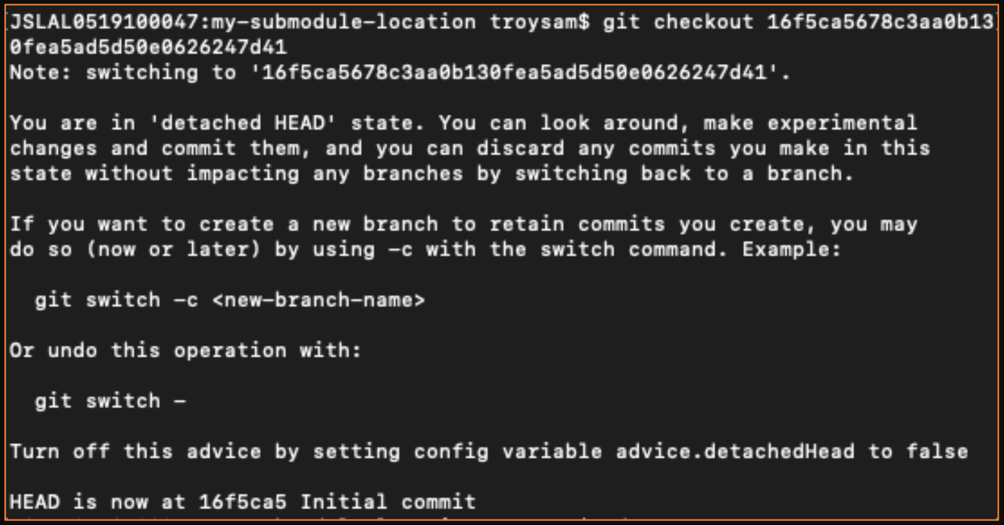


Submodules



```
JSLAL0519100047:Gitlab-Demo troysam$ git submodule add https://js-er-code.jsc.
nasa.gov/troysam/gitlab-submodule-demo.git my-submodule-location/
Cloning into '/Users/troysam/xemu_workspace/Gitlab-Demo/my-submodule-location'
...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
JSLAL0519100047:Gitlab-Demo troysam$ ls -la
total 32
drwxr-xr-x  12 troysam  staff  384 Jan 26 18:21 .
drwxr-xr-x  27 troysam  staff  864 Jan 26 13:29 ..
drwxr-xr-x  16 troysam  staff  512 Jan 26 18:21 .git
-rw-r--r--   1 troysam  staff   38 Jan 14 12:53 .gitignore
-rw-r--r--   1 troysam  staff  139 Jan 26 18:21 .gitmodules
-rw-r--r--   1 troysam  staff  218 Jan 14 14:37 README.md
-rw-r--r--   1 troysam  staff    0 Jan 22 13:09 excitingfile.ext
-rw-r--r--   1 troysam  staff   24 Jan 26 09:40 file2.py
-rw-r--r--   1 troysam  staff    0 Jan 14 12:53 file3.txt
-rw-r--r--   1 troysam  staff    0 Jan 21 16:05 file6.psk
drwxr-xr-x   4 troysam  staff  128 Jan 26 18:21 my-submodule-location
drwxr-xr-x   2 troysam  staff   64 Jan 21 16:05 subdir
```

- Basic Git submodule commands:
  - All submodule commands must be run from the top level directory (i.e. location of `.git/` folder).
  - Add a submodule:
    - `git submodule add <submodule url> <path in repository>`
    - This will create a new file called `.gitmodules` if it is your first submodule of the repository. `.gitmodules` is human-readable and shows each submodule's name, path, and source URL.
    - It will also `clone` the submodule into `<path in repository>`.
    - To finish adding the submodule to your project, `add` and `commit` the new `.gitmodules` file, as well as the folder that the submodule was created in.

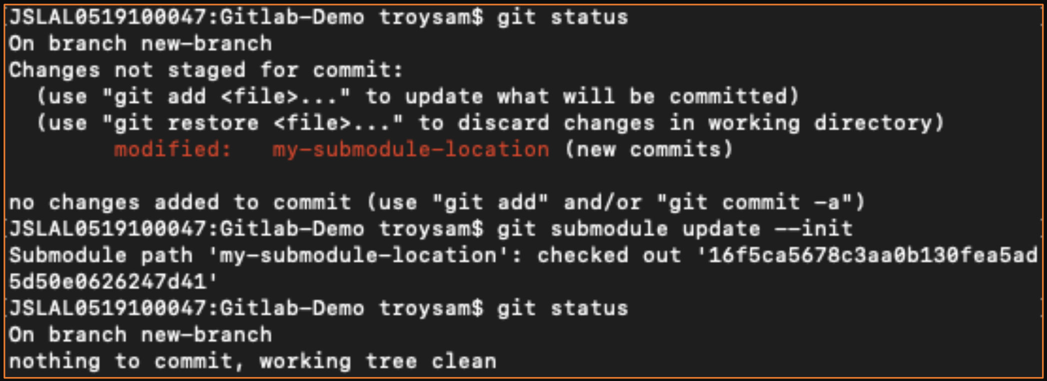


```
JSLAL0519100047:my-submodule-location troysam$ git checkout 16f5ca5678c3aa0b13
0fea5ad5d50e0626247d41
Note: switching to '16f5ca5678c3aa0b130fea5ad5d50e0626247d41'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 16f5ca5 Initial commit
```

Submodules

- Basic Git submodule commands:
  - Choosing a commit to track within the submodule:
    - Since the submodule **is** a Git repository too, when we navigate into it we are functionally now working inside that submodule.
    - **From within a submodule, any Git commands you run are now taking effect inside the submodule repository and NOT your parent repository**.
    - The parent repository is aware of the location of the submodule's `HEAD` pointer though, which is what tells the parent what commit it is tracking.
    - Thus, if we want to "step up" or "step down" a submodule to a newer or older ref, we need only `git checkout <commit/tag/branch>`

Submodules



```
JSLAL0519100047:Gitlab-Demo troysam$ git status
On branch new-branch
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   my-submodule-location (new commits)

no changes added to commit (use "git add" and/or "git commit -a")
JSLAL0519100047:Gitlab-Demo troysam$ git submodule update --init
Submodule path 'my-submodule-location': checked out '16f5ca5678c3aa0b130fea5ad
5d50e0626247d41'
JSLAL0519100047:Gitlab-Demo troysam$ git status
On branch new-branch
nothing to commit, working tree clean
```

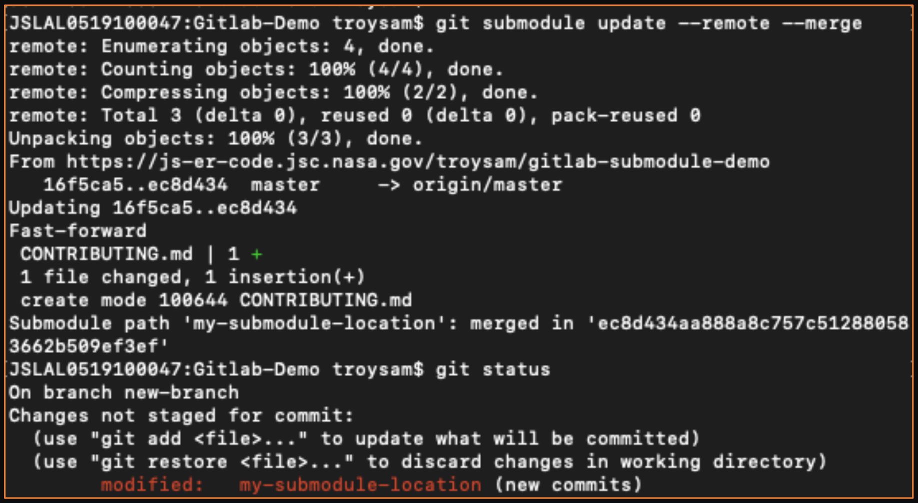
- Basic Git submodule commands:
  - Cloning a project with a submodule(s) (equivalent to `git clone` for each submodule):
    - After doing the standard git clone, the locations where the submodules reside within the repository will be empty folders.
    - We need to explicitly grab them using `git submodule update --init --recursive`
      - This command will recursively initialize, clone, and checkout all submodules in the project.
      - i.e. if any submodule repositories have submodules themselves, those will also be initialized, cloned, and checked out.
    - To initialize, clone, and checkout a single submodule, simply specify its path:
      - `git submodule update <path to specific submodule>`
        - You can also append the `--recursive` flag here if needed.
  - We can also skip all of that and just use `git clone --recurse-submodules <url>`.

Submodules

- Basic Git submodule commands:
  - A submodule can track a branch (which means when the submodule changes) or it can track a specific commit.
  - By default, `git submodule add` will make your repository start tracking the default branch of the submodule repo (usually `master`).
  - When you are tracking a branch or tag (i.e. a movable ref), you can pull the latest commit from a submodule at any time using:

```
git submodule update --remote --merge
```

  - This performs a git pull for each of our submodules on their checked-out branches, bringing them up-to-date with their remotes.



```
JSLAL0519100047:Gitlab-Demo troysam$ git submodule update --remote --merge
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://js-er-code.jsc.nasa.gov/troysam/gitlab-submodule-demo
   16f5ca5..ec8d434  master     -> origin/master
Updating 16f5ca5..ec8d434
Fast-forward
 CONTRIBUTING.md | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 CONTRIBUTING.md
Submodule path 'my-submodule-location': merged in 'ec8d434aa888a8c757c51288058
3662b509ef3ef'
JSLAL0519100047:Gitlab-Demo troysam$ git status
On branch new-branch
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   my-submodule-location (new commits)
```

Submodules

- Basic Git submodule commands:
  - Updating existing submodules (equivalent to `git pull`/fast-forward for submodules):
    - Use case: we switch branches, fetch someone else's changes from the remote, etc.
    - Someone decided to change which commit in the submodule we are pointing to.
    - We can always ensure that our local working area contains the correct submodule commits using:
      - `git submodule update --init`
      - This will set your submodule(s)'s HEAD to the commit that it is set to track.
      - This is always safe to run, and will bring us back to a safe state if we mess up our submodule(s) state.
      - We can also run this for just a single submodule:
        - `git submodule update --init <path to specific submodule>`

Submodules

- Basic Git submodule commands:
  - Removing a submodule is simple. If we no longer need it in the repository, we perform:
    - `git submodule deinit <path to submodule>`
    - `git rm <path to submodule>`
  - If that was the only submodule in the repository, we can also remove the `.gitmodules` file:
    - `git rm .gitmodules`

  - After that, we just `git add` and `git commit` those changes and we’re done.

We have now covered all the fundamentals of Git!

If that was overwhelming - don’t worry.

In practice, we won’t be using everything we just went over all the time.

Let’s summarize in two slides…

Git Summary

```
git status
git init
git clone
git add
git rm
git checkout
git branch
git commit
git push
git fetch
git merge
git rebase
git pull
git stash
git submodule add
git submodule init
git submodule update
git submodule deinit
```

Essentials

```
git status
git add
git checkout
git commit
git push
git pull
```

Supplements

```
git rm
git branch
git fetch
git stash
```

Infrequents

```
git init
git clone
git merge
git rebase
```

Submodules

```
git submodule add
git submodule init
git submodule update
git submodule deinit
```

Git Tips

Remember when Git-ing:

- When in doubt, `git status`. And when not in doubt. All the time really.
- Use tools and visual aids to make life easier!
  - The appendix slides list the two tools I think are most useful.
  - Gitlab, Github, Bitbucket, etc.
- Remember that no matter what you do, there is usually a way to undo it.
- If you're ever unsure, one Google search is all you need.
- Revision control can benefit anything you work on!

Resources

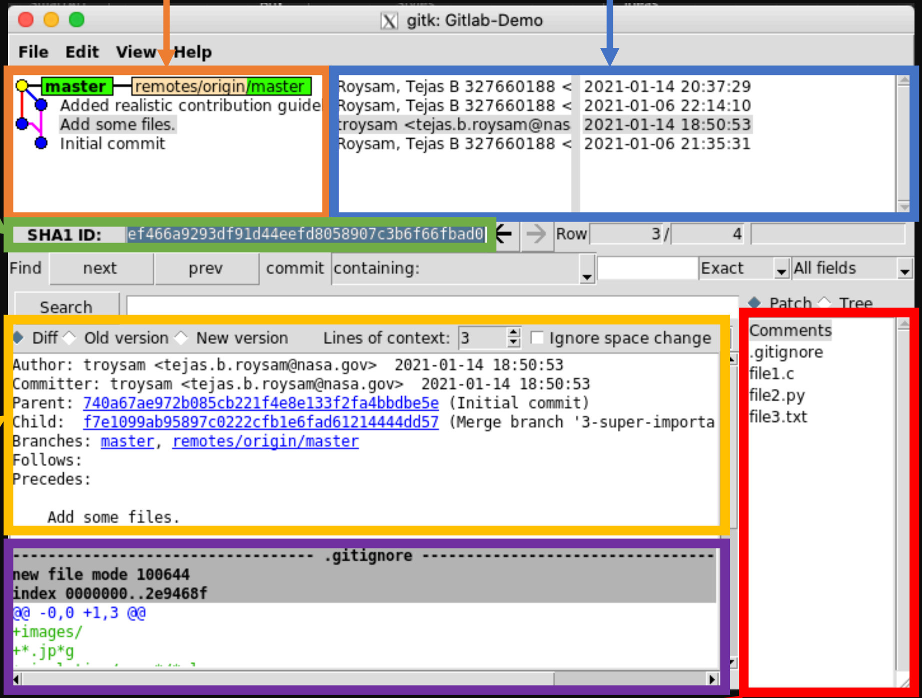
- Atlassian Git Tutorials - https://www.atlassian.com/git/tutorials
  - Excellent visuals and walkthroughs
- Pro Git (open license Git book in webpage-form): https://git-scm.com/book/en/v2/
  - Fundamentals, Beginner usage, all the way up to very advanced usage and administration.
- Git manpages: https://git-scm.com/docs
  - Official documentation of all Git commands.
- Git Command Cheat Sheet: https://training.github.com/downloads/github-git-cheat-sheet.pdf
  - Make your own! The exercise itself will commit most of it to memory.
- Command Line (Mac/Linux): https://courses.cs.washington.edu/courses/cse391/16sp/bash.html
- Powershell (Windows): https://devblogs.microsoft.com/scripting/table-of-basic-powershell-commands/

#1 Resource for everything: Google

Tools

- Visualize Git operations
  - `gitk`
    - Graphical history viewer
    - GUI for `git log`
    - Manual page: https://git-scm.com/docs/gitk
    - As your repository grows, the commit graph will grow larger and more complex.
    - Gitk can help visualize both your local tree and the remote tree.
    - You can also perform many operations (branching, etc.).

Commit object information



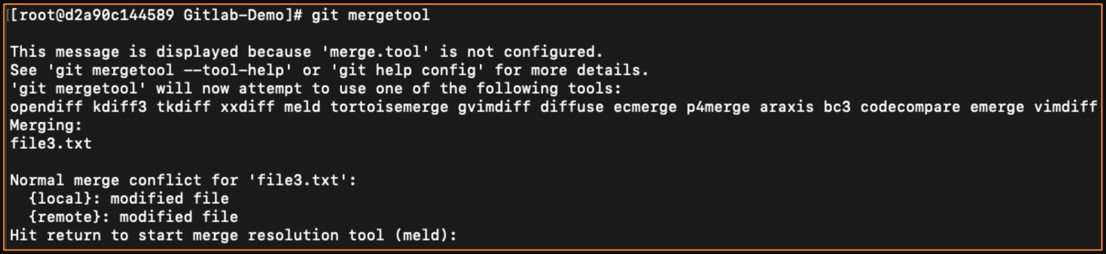
```
Added realistic contribution guide   Roysam, Tejas B 327660188 <   2021-01-14 20:37:29
Add some files.                      Roysam, Tejas B 327660188 <   2021-01-06 22:14:10
Initial commit                       troysam <tejas.b.roysam@nas   2021-01-14 18:50:53
                                     Roysam, Tejas B 327660188 <   2021-01-06 21:35:31

SHA1 ID: ef466a9293df91d44eefd8058907c3b6f66fbad0   Row 3 / 4
```

```
Author: troysam <tejas.b.roysam@nasa.gov>  2021-01-14 18:50:53
Committer: troysam <tejas.b.roysam@nasa.gov>  2021-01-14 18:50:53
Parent: 740a67ae972b085cb221f4e8e133f2fa4bbdbe5e (Initial commit)
Child:  f7e1099ab95897c0222cfb1e6fad61214444dd57 (Merge branch '3-super-importa
Branches: master, remotes/origin/master
Follows:
Precedes:

    Add some files.

---------------------------------- .gitignore ----------------------------------
new file mode 100644
index 0000000..2e9468f
@@ -0,0 +1,3 @@
+images/
+*.jp*g
```

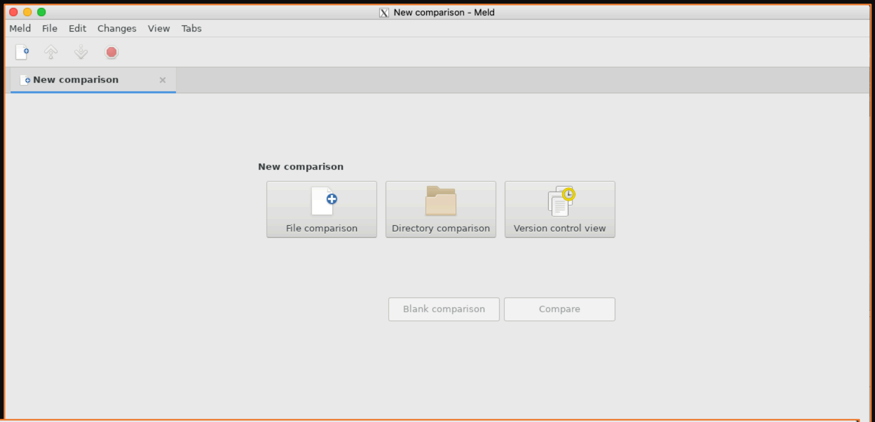
```
.gitignore
file1.c
file2.py
file3.txt
```

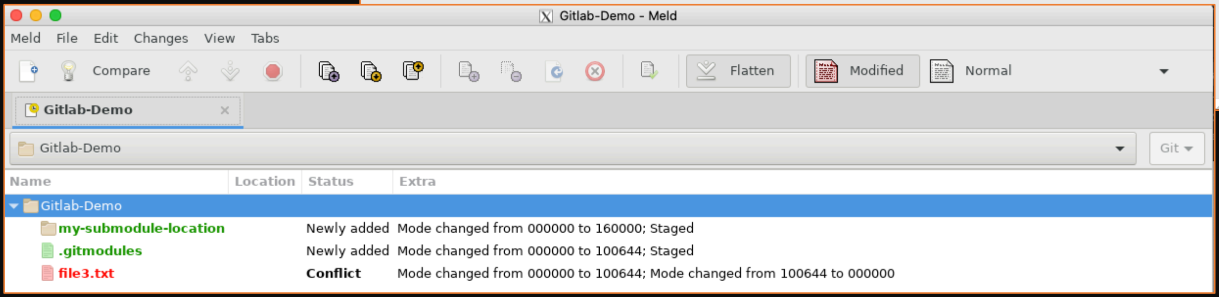
Tools

- Meld: A Git merge tool
  - Compare anything to anything else.
    - 2-way and 3-way comparisons.
    - Files and directories.
  - Can be run with `git mergetool` in order to resolve merge conflicts.
  - Version control view: can be run anytime to check the status of any part of any repository(ies).
  - Manual:
    - https://meldmerge.org/help/



```
[root@d2a90c144589 Gitlab-Demo]# git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse ecmerge p4merge araxis bc3 codecompare emerge vimdiff
Merging:
file3.txt

Normal merge conflict for 'file3.txt':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (meld):
```





Name	Location	Status	Extra
Gitlab-Demo			
my-submodule-location		Newly added	Mode changed from 000000 to 160000; Staged
.gitmodules		Newly added	Mode changed from 000000 to 100644; Staged
file3.txt		Conflict	Mode changed from 000000 to 100644; Mode changed from 100644 to 000000

END OF PART 1