Git

Git Workshop for Learning Basic Git Usage

by Matt Runion

Git -- An Overview

What Git Is

- Version Control -- and not just for source code
- Decentralized -- there is not always one source of truth
- Standard -- it is the *defacto* standard in many areas and growing
- Built for speed, data integrity and non-linear workflow

What Git Is Not

- Project Management -- no tasks, assignments, planning tools
- Centralized -- no single source of truth
- Massive -- git does one thing and does it well
- Bloated -- it doesn't need many (any?) external dependencies

Interesting Git History

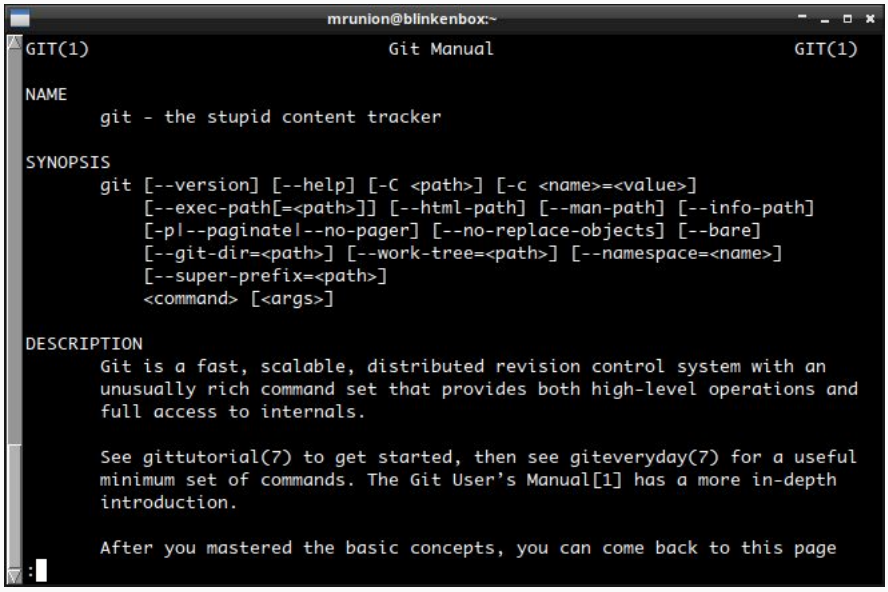
- Development started on April 3, 2005
- Project announced on April 6, 2005
- Git started self-hosting it's own development on April 7, 2005
- First multi-branch merge occurred on April 18, 2005
- Performance goals were met on April 29, 2005

Content is from Wikipedia's Git article: https://en.wikipedia.org/wiki/Git

Interesting Git Anecdotes

- Linus detested CVS and touted it as the example of what ***not*** to do in a version control system
- When making design decisions in developing git, when doubt about an approach arose the decision was to ***do the exact opposite*** of what CVS had done
- As of version 4.6.5 there were over ***17 million lines of code*** in the Linux Kernel -- all managed by git
- Google Trends indicates that git is the most widely used source code management tool

Some content is from Wikipedia's Git article: https://en.wikipedia.org/wiki/Git
Some content is from a Google Tech Talk:
https://www.youtube.com/watch?v=4XpnKHJAok8
Kernel statistical information is taken from:
https://www.linuxcounter.net/statistics/kernel
Usage trends taken from Google Trends:
https://trends.google.com/trends/explore?date=all&q=%2Fm%2F05vqwg,%2Fm%2F012ct9,%2Fm%2F08441_,%2Fm%2F08w6d6,%2Fm%2F09d6g&hl=en-US



Linus Torvalds describes git as “the stupid content tracker”

Screenshot generated by deepin-screenshot from a Dell laptop running Arch Linux and showing the git man page

Git -- Basic Usage

Attendees are encouraged to follow along

Follow Along

Attendees are encouraged to follow along with the presentation using their own computer equipment. Repeating the actions covered in the presentation re-enforces the material, and can encourage confidence as well as raise questions.

Requirements for following along:

- Windows, Mac or Linux machine
- Git installed (only command line tools are required)

Talk attendees through process

Setup

- Create a folder on the local machine named “gitworkshop” specifically for this training
- Windows users are recommended to install git from https://git-scm.com/download/win, as it will make sure the path is correct, etc. *Ensure you select “**Use Git from the Windows Command Prompt**” when configuring the PATH, and select “**Checkout Windows-style, commit Unix-style line endings**” when configuring line ending conversions.*

Have attendees do all workshop work in the workshop folder

Creating a Git Repository

Create and navigate into a folder called “gittraining” that will contain the repository, and use the ***git init*** command:

```
git init
Initialized empty Git repository in ...
```

To make things easier for the demo, go ahead and set the repository’s ***user.name*** and ***user.email*** values. This does not always need to be done if you clone a repository, but we will set this information here to make the rest of the workshop easier.

```
git config user.name = “<Your Name>”
git config user.email = “<your@company.com>”
```

Talk attendees through process. Make sure they are using a folder INSIDE “gitworkshop” and it is called “gittraining”. This will be important later on in the workshop when assumtions are made about folder names. Explain that there is now a hidden folder called “.git” in the gittraining folder. Explain a bit about the user.name and user.email properties. Note that git config --global user.xxxx sets these values globally.

Status of a Git Repository

Querying the current status of a Git repository is done with the ***git status*** command:

```
git status
On branch master

Initial commit

Nothing to commit (create/copy files and use “git add” to track)
```

Talk attendees through process

What Are We Seeing So Far?

- How easy it is to create a new repository using ***git init***
- Git defaults to calling the initial branch “master”
- The ***git status*** command tells us what branch we are on, what commit we are on and how to add files to the repository

Call attention to the master branch being named “master” by default

Adding a File to a Git Repository

Create a simple text file:

```
echo “My first file” > firstfile.txt
```

Run the ***git status*** command and notice we are given instructions on how to add files, and information on any untracked files:

```
git status
On branch master

Initial commit

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

    firstfile.txt

Nothing added to commit but untracked files present (use “git add” to track)
```

Talk attendees through process. Call attention to the file color coding.

Adding a File to a Git Repository

Add the file to the repository using the ***git add <file>*** command:

```
git add firstfile.txt
```

Run the ***git status*** command and notice we are shown what is cached (staged) and will be committed, as well as instructions on how to remove cached (staged) files:

```
git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    New file:   firstfile.txt
```

At this point the file is cached. It will be committed when the ***git commit*** command is ran.

Talk attendees through process. Call attention to the file color coding. Call attention to the fact that there is more than one way to add/stage the files for committing -- for example, git add -A (adds all files that are new, modified or removed -- basically “syncs” with the working tree), git add ., git commit -a (only handles changed files, not new files)

Adding a File to a Git Repository

Committing the changes to the repository moves the cached file(s) to the local repository. Use the ***git commit*** command:

```
git commit -m “My first commit”
[master (root-commit) 513ac29] My first commit
 1 file changed, 1 insertion(+)
 create mode 10644 firstfile.txt
```

Talk attendees through process. Clarify that the file is no longer cached, but is actually committed to the local repository. Clarify that in this demo scenario, there is no remote repository, but that in the “real world” there most likely would be. Simply committing a file DOES NOT put the file in any remote repositories.

What Are We Seeing So Far?

- There is a difference between caching changes and committing changes
- All history is stored locally
- Questions so far?

Maintaining Files in a Git Repository

Create a new text file:

```
echo “My second file” > secondfile.txt
```

Modify the contents of the existing firstfile.txt by adding some new content:

```
// On *nix
echo >> firstfile.txt
echo “Another line!” >> firstfile.txt
```

```
// On Windows
echo. >> firstfile.txt
echo “Another line!” >> firstfile.txt
```

Talk attendees through process. Make sure the Windows users and *nix users follow the proper example.

Maintaining Files in a Git Repository

Examine the repository status:

```
git status
On branch 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:   firstfile.txt

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

    secondfile.txt

No changes added to commit (use “git add” and/or “git commit -a”)
```

Talk attendees through process. Call attention to the fact that git is offering help on various options the user may want to take.

Maintaining Files in a Git Repository

We can see what has changed by using the ***git diff*** command:

```
git diff
diff --git a/firstfile.txt b/firstfile.txt
index 363d8b7..7577626 100644
--- a/firstfile.txt
+++ b/firstfile.txt
@@ -1 +1,3 @@
 My first file
+
+Another line!
```

Talk attendees through process. Why is there no output about the secondfile.txt? Because it isn't being tracked yet.

Maintaining Files in a Git Repository

Add firstfile.txt to the cache:

```
git add firstfile.txt
```

Run the ***git status*** command to see the results of adding the file to the cache:

```
git status
On branch master
Changes to be committed:
  (use "git reset HEAD" to unstage)

    modified:   firstfile.txt

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

    secondfile.txt
```

Talk attendees through process.

Maintaining Files in a Git Repository

Run the ***git diff*** command again:

```
git diff
```

There was no output on the previous command. The ***git diff*** command shows only the working tree versus the repository. Since we added firstfile.txt to the cache, it is not being shown. To compare the cache to the repository use the ***git diff --cached*** command:

```
git diff --cached
diff --git a/firstfile.txt b/firstfile.txt
index 363d8b7..7577626 100644
--- a/firstfile.txt
+++ b/firstfile.txt
@@ -1 +1,3 @@
 My first file
+
+Another line!
```

Talk attendees through process. Why is there no output about the secondfile.txt? Because it isn't being tracked. Call specific attention to the difference between working tree, cache and the repository itself.

Maintaining Files in a Git Repository

Add the rest of the changes to the cache:

```
// Adding ALL changes from the working copy to the cache
git add -A

// Adding only specific file changes to the cache
git add secondfile.txt
```

Run the ***git status*** command to see the results of adding the files to the cache (Note that running the ***git diff --cached*** command will now show both files):

```
git status
On branch master
Changes to be committed:
  (use "git reset HEAD" to unstage)

    modified:   firstfile.txt
    new file:   secondfile.txt
```

Talk attendees through process. Call attention to the fact that git is offering help on how to unstage files. Briefly cover git add -i (interactive mode)

Maintaining Files in a Git Repository

Use the ***git commit*** command to commit all these changes to the repository:

```
git commit -m “More changes”
[master 944fc0f] More changes
 2 files changed, 3 insertions(+)
 create mode 10644 secondfile.txt
```

Talk attendees through process.

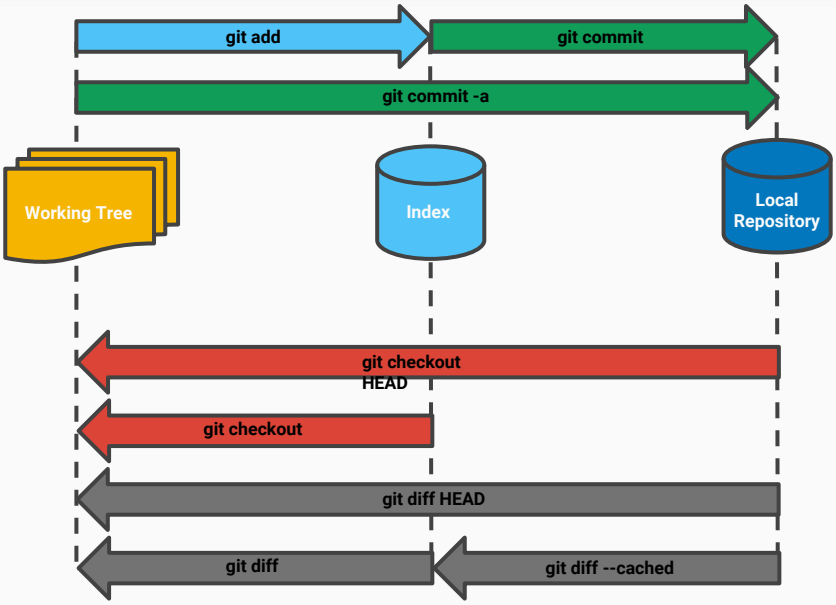
What Are We Seeing So Far?

- Git knows if a file is in the working tree, cache or repository
- Isn’t this too complex?
- Questions so far?

Local Git Repository Diagram

- Git has three “divisions” of local files -- the Working Tree, the Index (or cache) and the Local Repository
- Git handles files in these three areas
- Git commands operate on files in these three areas, or are even specific to files in certain areas

Git Local Repository Diagram



Take time to explain what command affects what area of the repository. Draw attention to the color coding.

Git -- Branching

Branching in Git

- Think of branching like “Save As…”
- Branching allows multiple development paths at the same time
- Many branching strategies exist, no single strategy is “right”, though some may be “wrong”
- No matter which branching strategy is used, know which branch to use for new branches, and which to merge back to when finished working

A (the most?) popular branching strategy is Git Flow. Another known (I wouldn’t call it popular) is “Cactus” branching.

A Note About “Checkout”

Developers generally use the word/command “checkout” when indicating one is about to acquire and modify code under version control. In **git** nomenclature the word/command “clone” is more in line with that action. In **git**, the word “checkout” means something slightly different. As long as the meaning is clear from the context it is being used in, it is assumed when a developer says “checkout” they are implying a *concept* and not a specific command.

Simple Branching in Git

Continuing the example from previously, we should be on the only branch in our repository, called “master”. To check that, list the branches in the repository using the ***git branch*** command:

```
git branch
* master
```

Create a new branch called “myfirstbranch” using the ***git branch*** command, and switch to that branch:

```
git branch myfirstbranch
git checkout myfirstbranch
Switched to branch ‘myfirstbranch’
```

Talk attendees through process. Call attention to the fact that a branch can be created and switched to in one command by using git checkout -b myfirstbranch (which is just sugar for the two commands above)

Simple Branching in Git

Add a new file on this branch:

```
echo “My third file” > thirdfile.txt
```

Modify the contents of the secondfile.txt by adding some new content:

```
// On *nix
echo >> secondfile.txt
echo “Another line!” >> secondfile.txt
```

```
// On Windows
echo. >> secondfile.txt
echo “Another line!” >> secondfile.txt
```

Talk attendees through process.

Simple Branching in Git

Look at the repository status using git status:

```
git status
On branch myfirstbranch
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:   secondfile.txt

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

    thirdfile.txt

No changes added to commit (use “git add” and/or “git commit -a”)
```

Talk attendees through process. Call attention to the current branch being displayed in the git status output.

Simple Branching in Git

Now switch to the “master” branch using the ***git checkout*** and run **git status** again:

```
git checkout master
M       secondfile.txt
Switched to branch ‘master’

git status
On branch 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:   secondfile.txt

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

    thirdfile.txt

No changes added to commit (use “git add” and/or “git commit -a”)
```

Talk attendees through process. Call attention to the current branch being displayed in the git status output. Call attention to the fact that any un-staged files are NOT affected by changing branches.

Simple Branching in Git

Once again switch back to the branch named “myfirstbranch” using the ***git checkout*** command:

```
git checkout myfirstbranch
```

List all the branches using the ***git branch*** command:

```
git branch
  master
* myfirstbranch
```

Stage all currently unstaged changes using the ***git add*** command:

```
git add -A
```

Talk attendees through process. Call attention to the fact that the current branch is listed with both an asterisk and color indicator. Call attention to the fact that switching branches STILL doesn’t affect the staged files -- they are staged but not committed to a specific branch yet

What Are We Seeing So Far?

- Changing branches with uncommitted changes **does not** keep those changes tied to a specific branch
- A new branch can also be created and switched to with one command: ***git checkout -b <new_branch_name>***
- Questions so far?

If attendees want to, they can switch branches and see that the files themselves are only staged and will “follow” what branch they are on.

Simple Merging in Git

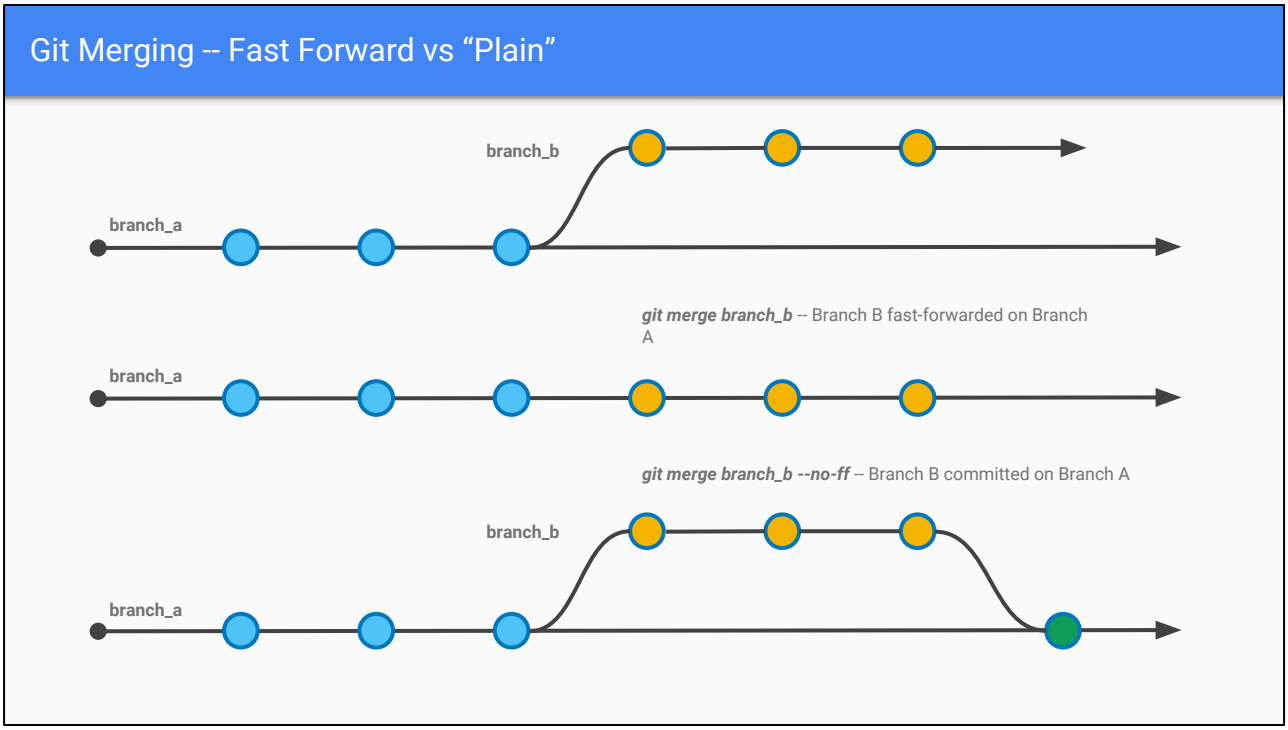
Commit the changes to the repository using the ***git commit*** command:

```
git commit -m “More changes to my files”
[myfirstbranch 20ecfd0] More changes to my files
 2 files changed, 3 insertions(+)
 create mode 10644 thirdfile.txt
```

Now let’s merge the “myfirstbranch” branch back into the master branch using the ***git merge*** command:

```
git checkout master
Switched to branch ‘master’
git merge myfirstbranch
Updating 944fc0f..20ecfd0
Fast-forward
 secondfile.txt | 2 ++
 thirdfile.txt  | 1 +
 2 files changed, 3 insertions(+)
 Create mode 100644 thirdfile.txt
```

Talk attendees through process. Call attention to the output of the git merge command -- notice that it says “Fast-forward”.



The top image shows two branches with one branched from the other. The second image shows the result of a git merge command doing a fast forward merge, which results in NO NEW COMMIT. (The git merge command uses -ff as the default, so using git merge is synonymous with git merge -ff.)The third image is a merge/commit (not a fast-forward) that shows the branch being merged and creating a new commit. (Note that git merge --ff-only can me used to FORCE a fast forward merge, or fail if it can’t.)

Simple Merging in Git

Finally we will delete the "myfirstbranch" branch using the ***git branch*** command:

```
git branch -d myfirstbranch
Deleted branch myfirstbranch (was 20ecfd0).
```

Talk attendees through process. Deleting the branch does not remove the committed/merged changes.

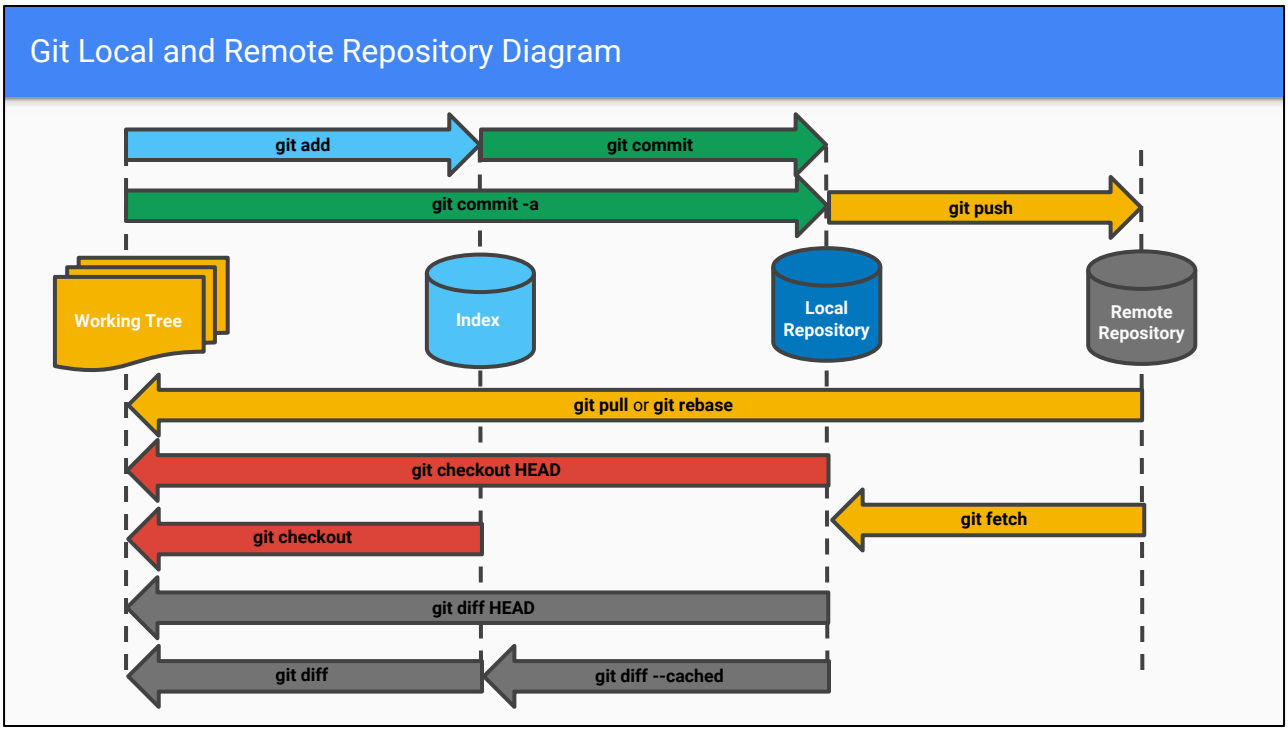
What Are We Seeing So Far?

- Merging branches can result in a different look with respect to version history depending on how the merge is performed
- Deleting a branch does not remove the changes that branch provided if it was merged into another branch
- Questions so far?

Git -- Remote Repositories

Remote Repositories

- Remote repositories are simply “different” repositories from the one that is being used “locally”
- Though it is logical to consider one repository as the “Master” repository, git itself has no concept of a “Master” repository
- More than one remote repository can be specified for the local repository



Take time to explain what command affects what area of the repository. Draw attention to the color coding. Encourage attendees to acquire a copy of this slide and keep it handy as a visual reference.

Cloning a Remote Repository

- This means acquiring a copy of a specific repository from another location to a local “working” location where work will be performed
- Use the **clone** command when no local repository exists
- Note that *remote* doesn’t have to mean another machine, it just means another repository

Quickly hit that though we sometimes say “pull down a copy of….”, that may translate to the clone command, or a pull command or a fetch command. Again, in the context of “pull down a copy of…..” it may not mean LITERALLY use PULL.

Creating a Clone

Navigate to the “gitworkshop” folder and use the ***git clone*** command to create a clone of the “gittraining” repository called “gittrainingclone”:

```
git clone gittraining gitworkshopclone
Cloning into ‘gittrainingclone’...
done.
```

Run **git status** and notice some additional detail mentioning the origin:

```
git status
On branch master
Your branch is up-to-date with ‘origin/master’.
nothing to commit, working tree clean
```

Talk attendees through process. Make sure the users run the git config commands to update user.name and user.email!

Creating a Clone

Run the **git remote** command to see any remote repositories this local repository is aware of:

```
git remote
origin
```

More info about remote repositories can also be obtained by using the **git remote** command:

```
git remote -v
origin  /home/mrunion/gitworkshop/gittraining (fetch)
origin  /home/mrunion/gitworkshop/gittraining (push)
```

Talk attendees through process. The fetch and push identifiers show which URL is used for each of those actions.

What Are We Seeing So Far?

- We have cloned a remote repository
- The clone operation has automatically added a remote called “origin” and specified the branch tracking options
- Questions so far?

Setup

- Previously covered commands will be assumed common knowledge at this point, and less emphasis will be placed on them and their output
- Open a second terminal or command prompt (or use tmux or screen, as desired) and navigate to the ".../gitworkshop/gittraining" repository
- **For the next slides it will be important to pay attention which terminal is opened in the "gittrainingclone" and "gittraining" folders**

Stress that knowing which folder is which is important for following along and understanding what is happening.

Routine Work Flow

- Routine work on data managed by a **git** repository is not different than most other version control systems
- Routine process: branch, change, commit
- Pushing and Merging also plays a regular part in routine use, but circumstances will dictate when these are done

Explain when one may want to push or merge, and try to answer why.

What Are We Doing Next?

- Create a branch called “mywork” off the “gittrainingclone” repository
- Make local changes and commit those changes
- Push those changes back to the “gittraining” origin repository on the new “mywork” branch
- Merge the “mywork” branch in the “gittrainingclone” repository and push that change back to the “gittraining” origin repository

Cover briefly what we are going to do. Allow attendees to work on the branching and committing as individually as they can. Regroup them before pushing and merging.

Working with a Remote Repository

Create a branch off the “master” branch in the “gittrainingclone” repository:

```
// REPOSITORY: gittrainingclone
git checkout -b mywork
origin
```

Make some file changes and additions:

```
// On *nix
echo >> thirdfile.txt
echo “More changes” >> thirdfile.txt
echo “Fourth file” >> fourthfile.txt
```

```
// On Windows
echo. >> thirdfile.txt
echo “More changes” >> thirdfile.txt
echo “Fourth file” >> fourthfile.txt
```

Talk attendees through process.

Working with a Remote Repository

Add the changes to the cache, then commit the changes to the local repository:

```
// REPOSITORY: gittrainingclone
git add -A
git commit -m “Made some changes on my branch”
```

Verify that the branch “mywork” only exists locally on the “gittrainingclone” repository and not on the “gittraining” origin repository:

```
// REPOSITORY: gittrainingclone
git branch
  master
* mywork
```

```
// REPOSITORY: gittraining
git branch
* master
```

Talk attendees through process. Remind attendees that all changes are now committed, but only to the local repository. The new branch we created called “mywork” only exists locally. Nothing has went to the remote repository.

Working with a Remote Repository

Use the **git push** command to push the local changes up to the remote repository:

```
// REPOSITORY: gittrainingclone
git push -u origin mywork
Counting objects: 4, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 402 bytes | 0 bytes/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To /home/mrunion/gitworkshop/gittraining
 * [new branch]      mywork -> mywork
Branch mywork set up to track remote branch mywork from origin.
```

Talk attendees through process. Call attention to the remote repository URL being listed. The -u switch tells git to set up the local mywork branch to track the remote mywork branch (notice the “Branch mywork set up to track…..).

Working with a Remote Repository

Verify that the branch “mywork” now exists in both repositories:

```
// REPOSITORY: gittrainingclone
git branch
  master
* mywork
```

```
// REPOSITORY: gittraining
git branch
* master
  mywork
```

Talk attendees through process.

Working with a Remote Repository

Use the **git remote** command to show what local branches track remote branches:

```
// REPOSITORY: gittrainingclone
git remote show origin
* remote origin
  Fetch URL: /home/mrunion/gitworkshop/gittraining
  Push  URL: /home/mrunion/gitworkshop/gittraining
  HEAD branch: master
  Remote branches:
    master tracked
    mywork tracked
  Local branches configured for 'git pull':
    master merges with remote master
    mywork merges with remote mywork
  Local refs configured for 'git push':
    master pushes to master (up to date)
    mywork pushes to mywork (up to date)
```

Talk attendees through process. Call attention to the branches lining up with each other and the status of those branches

Working with a Remote Repository

Make some file changes:

```
// On *nix
echo >> fourthfile.txt
echo “Something else changed” >> fourthfile.txt
```

```
// On Windows
echo. >> fourthfile.txt
echo “Something else changed” >> fourthfile.txt
```

Commit the changes to the local repository:

```
// REPOSITORY: gittrainingclone
git add -A
git commit -m “More changes on this branch”
```

Use the **git remote** command to show a branch is now ahead of the remote:

```
// REPOSITORY: gittrainingclone
git remote show origin
  :
    mywork pushes to mywork (fast-forwardable)
```

Talk attendees through process. Call attention to the “fast-forwardable” text

Working with a Remote Repository

Switch to the “master” branch, merge the “mywork” branch, then push that code to the “gittraining” repository:

```
// REPOSITORY: gittrainingclone
git checkout master
git merge mywork
git push
```

Talk attendees through process. Note: users will most likely get an error message about being “unable to update the current branch in a non-bare repository….”. This is because in this workshop we did not create the “gittraining” repository as a bare repository and probably have the master branch checked out. To fix this for the workshop, switch to the terminal with the gittraining repository and git checkout mywork so we are not on the master branch. Then git push from the gittrainingclone repository again and it should work. After the push, make sure to got back to the “gittraining” repository and switch back to the master branch before mving on to the next part of the workshop.

What Are We Seeing So Far?

- We have made changes to our local repository and pushed them to a remote repository
- We have inadvertently made assumptions about the state of the remote repository in a few circumstances (not checking for updates before we push or merge, etc.)
- Questions so far?

Updating a Local Repository from the Remote

- Updating a local repository from a remote repository is accomplished by fetching and pulling
- Pulling from a remote updates all the way to the working tree (and involves an implicit fetch and git merge)
- Fetching from a remote only updated the local repository, not the working tree

Call attention to the fact that some developers recommend never using git pull, but always using git fetch and git merge explicitly. This is sound advice, but not following it will not result in catastrophic problems, but could sometimes result in extra work. Just know this if you use git pull.

What Are We Doing Next?

- Make changes to the “gittraining” origin repository on the master branch
- Fetch those changes into the “gittrainingclone” local repository
- Merge those changes into the “gittrainingclone” local repository’s working tree

Cover briefly what we are going to do.

Updating from a Remote Repository

Make sure we are on the master branch in the “gittraining” repository:

```
// REPOSITORY: gittraining
git checkout master
```

Make some file changes and additions:

```
// On *nix
echo >> fourthfile.txt
echo “Changes on the remote” >> fourthfile.txt
echo “A fifth new file appears...” >> fifthfile.txt
```

```
// On Windows
echo. >> fourthfile.txt
echo “Changes on the remote” >> fourthfile.txt
echo “A fifth new file appears...” >> fifthfile.txt
```

Add and commit the changes to the local (“gittraining”) repository:

```
// REPOSITORY: gittraining
git add -A
git commit -m “Changes to be distributed”
```

Talk attendees through process. Make sure attendees are in the “gittraining” repository!

Updating from a Remote Repository

Make sure we are on the master branch in the “gittrainingclone” repository:

```
// REPOSITORY: gittrainingclone
git checkout master
```

Run the **git status** command and look at the message:

```
git status
On branch master
Your branch is up-to-date with ‘origin/master’.
nothing to commit, working tree clean
```

Talk attendees through process. Why are we told we are up to date with origin/master when we have clearly made changes? Stress that git is distributed and does diff-ing of versions LOCALLY. We have not updated our LOCAL COPY of the remote.

Updating from a Remote Repository

Use the **git fetch** command to retrieve any changes from the remote:

```
// REPOSITORY: gittrainingclone
git fetch origin master
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From /home/mrunion/gitworkshop/gittraining
   fd5714c..f7fdbbd  master     -> origin/master
```

Now run the **git status** command again and look at the message:

```
git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be
fast-forwarded.
  (use "git pull" to update your local branch)
nothing to commit, working tree clean
```

Talk attendees through process. You can simply use git fetch with no parameters and have basically the same results if you are on the branch that you want to fetch changes from and remote tracking is set up (it is for us in our example because we cloned from a remote, which sets this up for us). We specified the remote and branch here for clarity only.

Updating from a Remote Repository

Use the **git diff** command to examine the differences in what has been fetched:

```
// REPOSITORY: gittrainingclone
git diff master origin/master
diff --git a/fifthfile.txt b/fifthfile.txt
new file mode 100644
index 0000000..09e0eb8
--- /dev/null
+++ b/fifthfile.txt
@@ -0,0 +1 @@
+A fifth new file appears...
diff --git a/fourthfile.txt b/fourthfile.txt
new file mode 100644
index 06c6be9..662f6f4 100644
--- a/fourthfile.txt
+++ b/fourthfile.txt
@@ -1,3 +1,5 @@
 Fourth file

 Something else changed
+
+Changes on the remote
```

Talk attendees through process. First, the command could have also be ran as git diff master origin (leaving off the “/master”) because as stated previously, we are already set to track the remote branch “master” from this branch. (Now may be a good time to run git remote show origin and show where this information can bee seen.) Call attention to the fact that if the command was ran as git diff origin master, the diff results would have been reversed. All the changes should still be seen, but it would be seen from the “opposite” perspective. Depending on the attendee audience, a very very breif shpill about the diff output being in a “patch” syntax might be a nice sidebar.

Updating from a Remote Repository

Use the **git merge** command merge the newly fetched data into the local repository:

```
// REPOSITORY: gittrainingclone
git merge origin/master
Updating fd5714c..f7fdbbd
Fast-forward
 fifthfile.txt  | 1 +
 fourthfile.txt | 2 ++
 2 files changed, 3 insertions(+)
 create mode 100644 fifthfile.txt
```

Talk attendees through process. Again, the “/master” part after origin is not explicitly needed in this case because tracking is set up already. See previous slide comments for the low-down on that. Call attention that the “Updating…” line shows what range of commits were merged. This will match what was output when the get fetch was performed. Call attention to the merge type -- in this case a Fast-Forward merge.

What Are We Seeing So Far?

- We have retrieved changes that have been made to a remote repository and merged them into our own clone of that repository, but only for branches that already exist
- We see there is such a thing as tracked remote branches, and we see that we can control what branches are fetch/merge with our own
- Questions so far?

What Are We Doing Next?

- Make changes to the “gittraining” origin repository on a newly created branch
- Fetch those changes into the “gittrainingclone” local repository
- Checkout the new branch and see that it sets up tracking for us

Cover briefly what we are going to do.

Getting a new Branch from a Remote Repository

Make sure we are on the master branch in the “gittraining” repository, then create a new repository named “otherwork” and check it out:

```
// REPOSITORY: gittraining
git checkout master
git checkout -b otherwork
```

Make some file additions:

```
echo “Sixth file” >> sixthfile.txt
```

Add and commit the changes to the local (“gittraining”) repository:

```
// REPOSITORY: gittraining
git add -A
git commit -m “A new branch on the remote”
```

Talk attendees through process. Make sure attendees are in the “gittraining” repository! We make the attendees switch to the master branch before creating the new “otherwork” branch to esnure the master branch is the one being branched. We could also have done git checkout -b otherwork master and accomplished the same thing. We could have also have done it the long way as well -- git branch otherwork | git checkout otherwork, etc.

Getting a new Branch from a Remote Repository

Get a list of all remote branches by using the **git remote** command:

```
// REPOSITORY: gittrainingclone
git remote show origin
* remote origin
  Fetch URL: /home/mrunion/gitworkshop/gittraining
  Push  URL: /home/mrunion/gitworkshop/gittraining
  HEAD branch: master
  Remote branches:
    master tracked
    mywork tracked
    otherwork new (next fetch will store in remotes/origin)
  Local branches configured for 'git pull':
    master merges with remote master
    mywork merges with remote mywork
  Local refs configured for 'git push':
    master pushes to master (up to date)
    mywork pushes to mywork (up to date)
```

Talk attendees through process. Notice we have switched back to the "gittrainingclone" repository!

Getting a new Branch from a Remote Repository

List the remote branches our local repository currently knows about with the **git branch** command:

```
// REPOSITORY: gittrainingclone
git branch -r
  origin/HEAD -> origin/master
  origin/master
  origin/mywork
```

Now run the **git fetch** command:

```
// REPOSITORY: gittrainingclone
git fetch origin
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /home/mrunion/gitworkshop/gittraining
 * [new branch]      otherwork  -> origin/otherwork
```

Talk attendees through process. Why doesn't git branch -r show that new remote branch? Git does things LOCALLY, and that command operates on the LOCAL info about the remote. Call attention to the fact that the remote branches area now lists "otherwork" as a new branch, and even tells us it will be pulled down to our remote copy on the next git fetch. Call attention to the fact that we could have just run git fetch and there would be no difference. If we had of had more than one remote defined though, git fetch alone would fetch from ALL defined remotes.

Getting a new Branch from a Remote Repository

Rerun the **git branch** command:

```
// REPOSITORY: gittrainingclone
git branch -r
  origin/HEAD -> origin/master
  origin/master
  origin/mywork
  origin/otherwork
```

Check out the new branch using the **git checkout** command:

```
// REPOSITORY: gittrainingclone
git checkout otherwork
Branch otherwork set up to track remote branch otherwork from origin.
Switched to a new branch 'otherwork'
```

Talk attendees through process. Now we can see the remote branch when running git branch -r. When we check out the “otherwork” branch we also see that git went ahead and set up tracking for this branch. Rerun git remote show origin and confirm this.

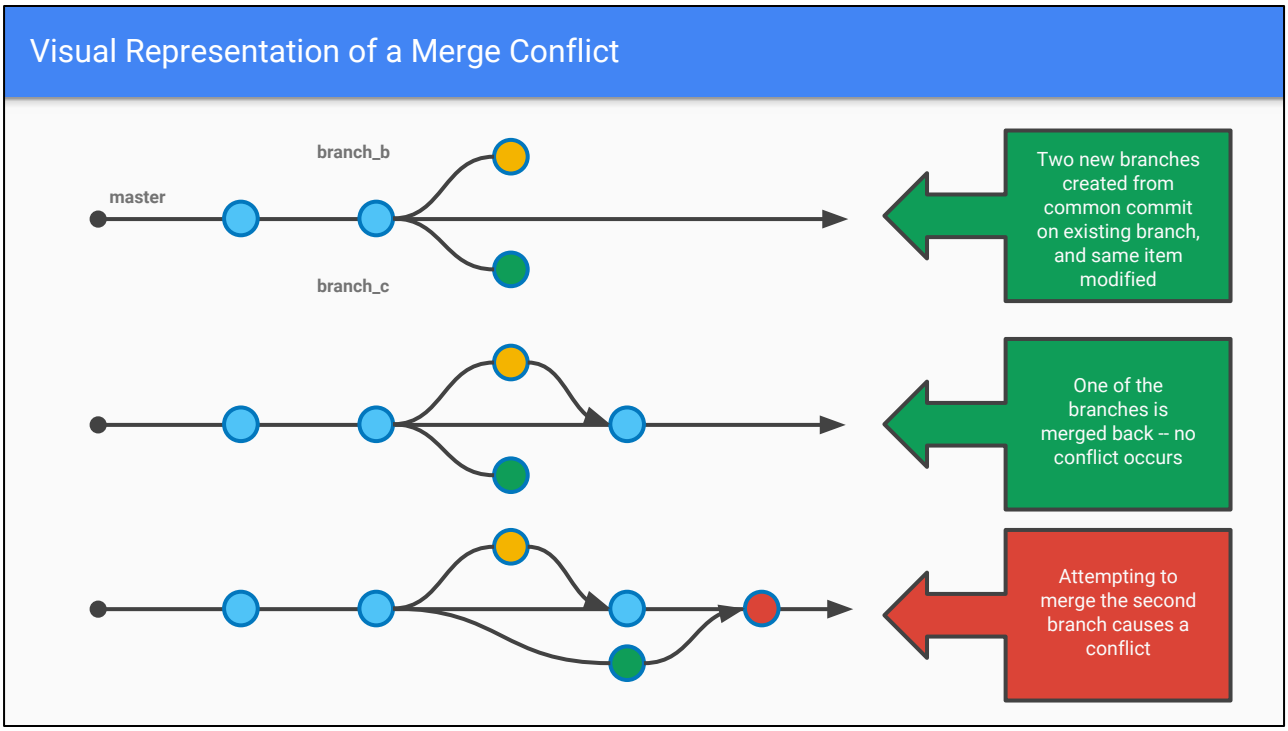
What Are We Seeing So Far?

- We have queried the remote repository to see its branches
- We have fetched and checked out a new branch that was added to the remote repository and confirmed we are tracking that branch
- Questions so far?

Git -- Merge Conflicts

Merge Conflicts

- Happen when data changes in two different branches or repositories and git cannot resolve the commits
- Sometimes resolution is simple and sometimes it can be very complex
- Merge conflicts have to be evaluated on a case-by-case basis to find the proper solution



The image shows two branches (B, C) branched from a common commit on A, and each B and C have had a different change made to the same item.

What Are We Doing Next?

- Make changes to the “gittrainingclone” that will cause a conflict
- Resolve that conflict
- Understand that this is a simple case and a contrived example

Cover briefly what we are going to do.

Handling a Merge Conflict

Make sure we are on the master branch in the “gittrainingclone” repository:

```
// REPOSITORY: gittrainingclone
git checkout master
```

Using the **git branch** command, create two new branches – “branch_b” and “branch_c”:

```
git branch branch_b
git branch branch_c
```

Talk attendees through process. This might be a good time to mention branches can’t have spaces in their names.

Handling a Merge Conflict

Use the **git checkout** command and checkout “branch_b”:

```
git checkout branch_b
```

Make some changes to a file:

```
echo >> firstfile.txt // On Windows use echo. >> firstfile.txt
echo “Check out my awesome bug fix!” >> firstfile.txt
```

Stage and commit that change:

```
git add -A
git commit -m “Another bug bites the dust”
```

Talk attendees through process.

Handling a Merge Conflict

Use the **git checkout** command and checkout “branch_c”:

```
git checkout branch_c
```

Make some changes to the same file that was previously changed:

```
echo >> firstfile.txt // On Windows use echo. >> firstfile.txt
echo “I am adding a new feature” >> firstfile.txt
```

Stage and commit that change:

```
git add -A
git commit -m “New feature added”
```

Talk attendees through process. Make double sure that attendees are on the proper branch (git status) and changing the SAME file previously changed.

Handling a Merge Conflict

Now **git checkout** the master branch:

```
git checkout master
```

Use the git merge command to merge "branch_b" into master:

```
git merge branch_b
```

Now merge "branch_c":

```
git merge branch_c
Auto-merging firstfile.txt
CONFLICT (content): Merge conflict in firstfile.txt
Automatic merge failed; fix conflicts and then commit the result
```

Talk attendees through process. The second merge attempt fails.

What Are We Seeing So Far?

- We have a merge conflict created by changing the same file on two different branches
- We now have to look at the file and determine what to do to resolve the conflict
- Questions so far?

Handling a Merge Conflict

Look at the file content to see what git has told us about the conflict:

```
less firstfile.txt

// Or

type firstfile.txt
```

Look at the file's content to see what git has told us about the conflict:

```
My first file

Another line!

<<<<<<< HEAD
Check out my awesome bug fix
=======
I am adding a new feature
>>>>>>> branch_c
```

Talk attendees through process. Git has marked the HEAD revision and the branch_c version inside some special formatting.

Handling a Merge Conflict

Using the tool of choice (vi, Notepad, Sublime), correct the output of the file to be as shown and save the file:

```
My first file

Another line!

Check out my awesome bug fix
I am adding a new feature
```

Now add and commit the file to the repository:

```
git add firstfile.txt
git commit -m “Fixed merge conflict”
```

Talk attendees through process.

What Are We Seeing So Far?

- We saw the process for solving a merge conflict
- Merge conflicts can be simple to solve, or very difficult -- each case is unique
- Questions so far?

Conclusion of Basic Git

Where to Next?

- Discuss strategies for organizing a repository
- Discuss working with outside developers, forking, etc.
- Deep-diving into specific questions, or role-playing “what if” scenarios
- **Thank you for your time and attention!**