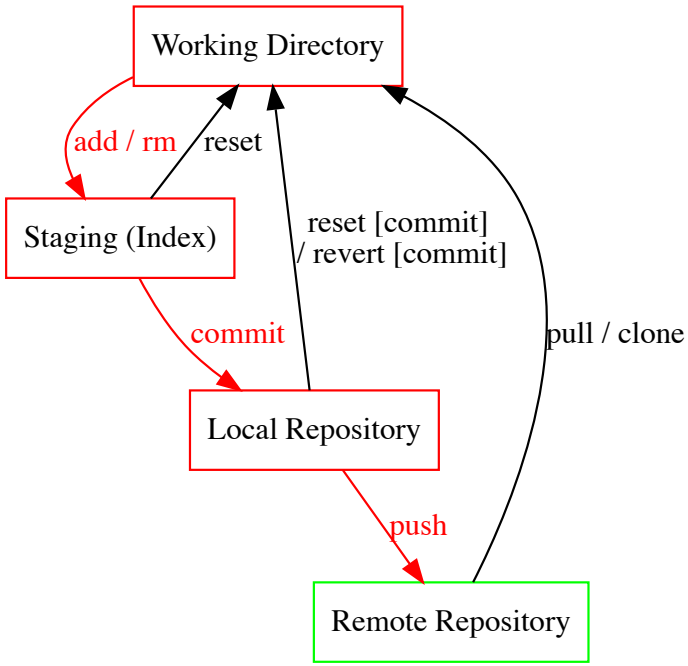
Lab 04 - More Git and Bash Scripts

CS 1XA3

Jan. 29, 2018

Recap On Version Control With Git

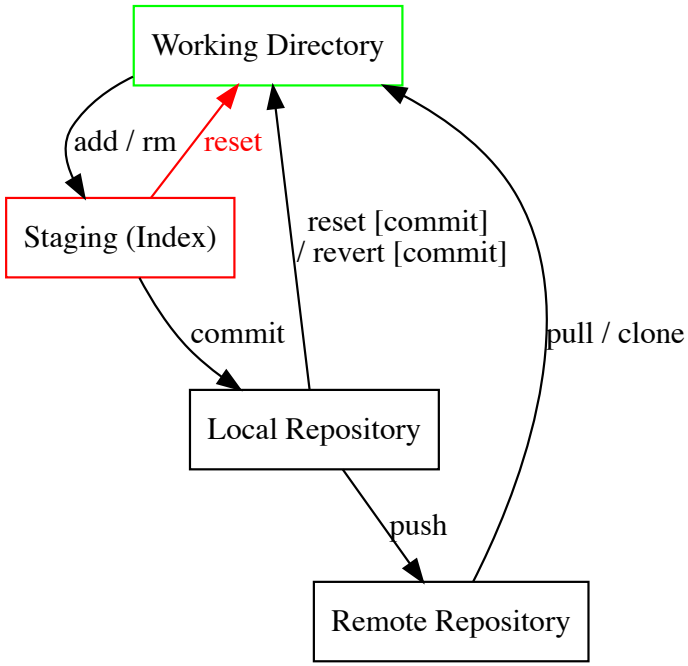


Use ALL of the following steps to add a file to GitHub

- Add files with git add
- Commit changes with git commit
- Push the commit to GitHub with git push

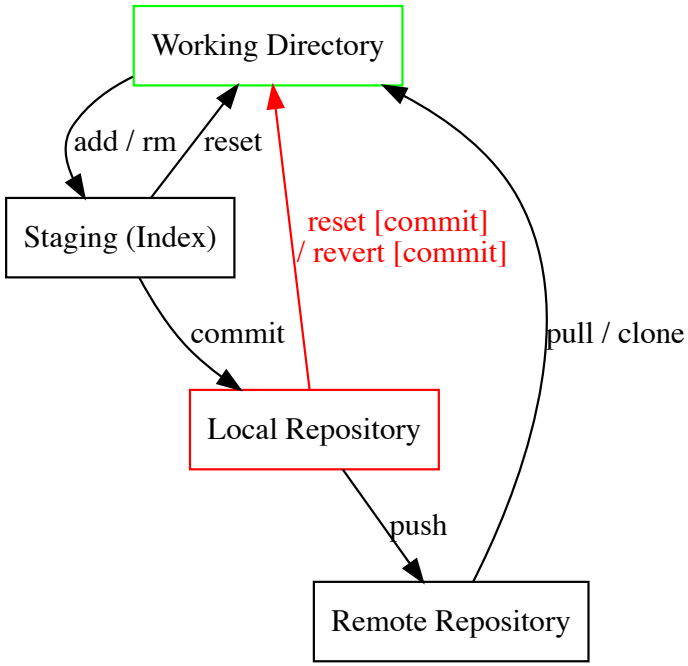
Note: make certain your changes have been pushed to GitHub with git status or by logging into your GitHub account in your browser

Undoing Mistakes With Git Reset



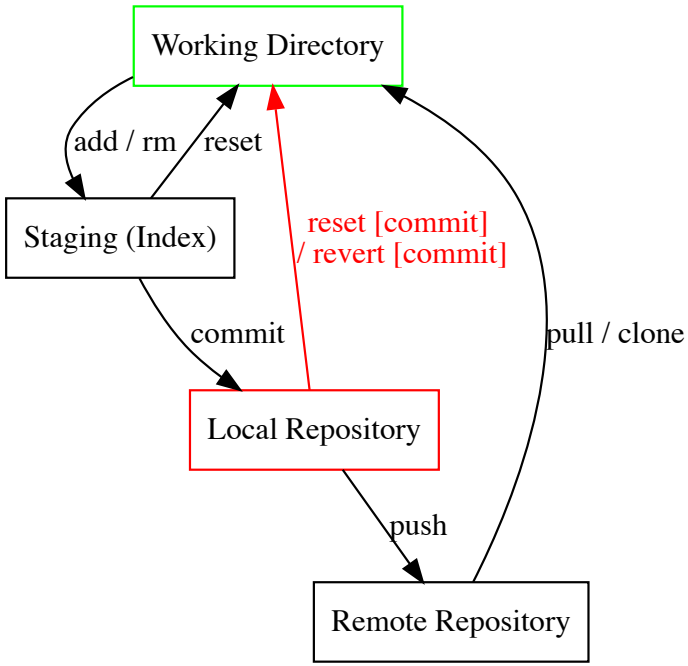
- Scenario: you perform a git add or git rm you didn't mean to
- Use git reset before you commit to undo all add's/rm's

Undoing Mistakes With Git Reset Hard



- Scenario: you make some changes to your files you want to discard
- Use git reset –hard before you commit to undo all changes

Undoing Mistakes With Git Revert



- Scenario: you've made a series of commits and want to revert to a previous commit
- Use git log to find the commit you want to revert to
- Use

  `git revert HEAD~n..HEAD`

  and commit to create a new commit, that undo's the last *n* commits
- Use git push to apply this change to the remote repo

Case Study: Git Merge

Follow these instructions and make careful observations

- Create two directories: CaseA and CaseB and clone a copy of your git repo to each
- Add a file tmp.txt to CaseA with line1,line2,..line10 written on each line respectively, and push to GitHub
- Change to CaseB and pull the file. Change line1 to lineA, commit and push to GitHub
- Switch back to CaseA, change line1 to lineC and line2 to lineD. Then pull from GitHub (Note: should result in a merge conflict)
- Check the file, it should have both versions of the code inserted inside it
- Fix your file, commit and push

Hello World Script

- Create a script Hello.hs and fill it with the following code

```
#!/bin/bash

echo "Hello World!"
```

- Make your script an executable and run it with

```
chmod a+x Hello.sh
./Hello.sh
```

Bash Variables

Assign a variable with = and access it's value with $

```
#!/bin/bash

VAR=Hello
echo "$VAR World!"
```

Note: be careful of spaces, variable assignment should contain no spaces between the = operator

Variable Expansion

- When expanding a variable with $, use quotes to prevent issues with spaces

```
VAR="SOME FILE.txt"
mv $VAR FILE.txt    # incorrect, 3 args given
mv "$VAR" FILE.txt # correct, 2 args
```

- Avoid ambiguities with curly braces

```
VAR="GOOD"
echo $VARBYE    # incorrect, no VARBYE exists
echo ${VAR}BYE # correct
```

Bash Conditionals - If Statements

Syntax

```
if [ cond1 ]
then
    command1
elif            # optional
    command2
else            # optional
    command3
fi
```

Note: the **Square Brackets** are like a command (in fact they're an alias for the **test** command), you **need spaces** between them and their inner expression

Conditionals (Test)

Anything inside of square brackets is actually a reference to the command test. The following operators are proper flags to test

```
!Expression           # Not Expression
-n STRING             # Length STRING > 0
STRING1 = STRING2     # STRING Equality
STRING1 != STRING2    # STRING Inequality
INT -eq INT           # Integer Equality
INT -gt INT           # Integer >
INT -lt INT           # Integer <
    -d FILE           # File is directory
    -e FILE           # File exists
```

Note: for a more complete list, check the manual with man test

For Loops - Over Integers

Two choices of syntax for iterating over Integers

- Curly Brace Syntax

```
for i in {1..5}
do
    echo $i
done
```

- Range Syntax

```
for ((i=1;i<=5;i+=1))
do
    echo $i
done
```

For Loops Curly Braces Expansion

The following script creates 4 new files with Hello inside

```
for c in {a,b,c,d}
do
    echo "Hello" > "file${c}.txt"
done
```

Note: you can put any strings inside curly braces, including glob patterns

Bash Loops - While

Syntax

```
while [ cond ]
do
    command1
    command2
    ...
done
```

Example

```
i=0
while [ $i -lt 5 ]
do
    echo $i
    i=$(($i+1))
done
```

Note: we'll discuss the $((expr)) syntax in a few slides

Loop Control - Continue

You can skip an iteration of a loop with the continue command

```
for ((i=0;i<5;i+=1))
do
    if [ $i -eq 3 ]
    then
        continue # skip 3
    else
        echo $i
    fi
done
```

Note: also works with while loops

Loop Control - Break

You can exit out of a loop explictly with the break command

```
i=0
while [ true ]
do
    if [ $i -gt 5 ]
    then
        echo $i
        i=$(($i+1))
    else
        break
    fi
done
```

Note: also works with for loops

Scripts With Arguments

- Scripts can be given arguments the same way commands are
- Bash provides the following key variables for accessing and working with given arguments

Character	Description
$@	access all args at once
$#	number of args given
$1	1^{st} arg
$2	2^{nd} arg
...	continue pattern for 3^{rd},4^{rth},etc

Table: Bash Args Key Variables

Scripts With Arguments Example

Consider the following script that takes two inputs

```
#!/bin/bash

if [ $# -eq 2 ] # test num args equals 2
then
    mv "$1" "$2" # copy arg1 arg2
else
    echo "my_cp Error - improper #args"
fi
```

Name it rename.sh and execute it with the following syntax

```
./rename.sh file1 file2 # remember to chmod it first
```

Note: quotes around $1, $2 prevent args with spaces in their names from confusing mv

Arithmetic Expansion

Arithmetic Expansion is performed through double parenthesis ((expr)). You need to put any arithmetic expression inside of these

```
i=0
while [ $i -lt 10 ]
do
    if [ $((i % 2)) -eq 0 ]
    then
        echo $i
    fi
    i=$(($i + 1))
done
```

The above example outputs only even numbers between 0 and 9, using the mod operator %

Subshell Expansion

Piping isn't the only way to pass to use the output of a command, subshell expansion can be used with the $(command) syntax
Example check if a file contains a particular word with if

```
echo "word" > test.txt

if [ $(grep word test.txt | wc -l) -gt 0 ]
then
    echo "wrote word to test.txt"
else
    echo "black magic is afoot"
fi
```

Subshell Expansion

Note: Commands executed in a subshell are done so in a different environment than the orignal one, i.e commands that change the state of the current environment are not affected

Example

```
mkdir Hi
$( cd Hi ; echo "Hi" > hi.txt )
cat hi.txt    # error
cd Hi
cat hi.txt
```

Note: the subshell cd did not change the working directory of the script