

Git for Developers

Getting the Most out of Git

October 1, 2018

Peter J. Jones

✉ pjones@devalot.com

🐦 @devalot

<http://devalot.com>



What's In Store

Day One	Day Two
Introduction / History	Merging
Git Internals	Rebasing
Daily Workflow	Remotes
Branching	Resetting
Tagging	Branch Management

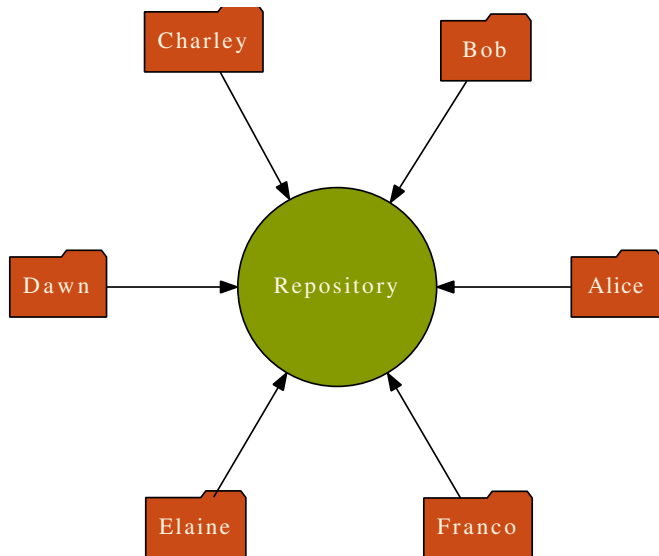
Git Resources

- [Git Command Reference](#)
- [Pro Git Book Online Version](#)

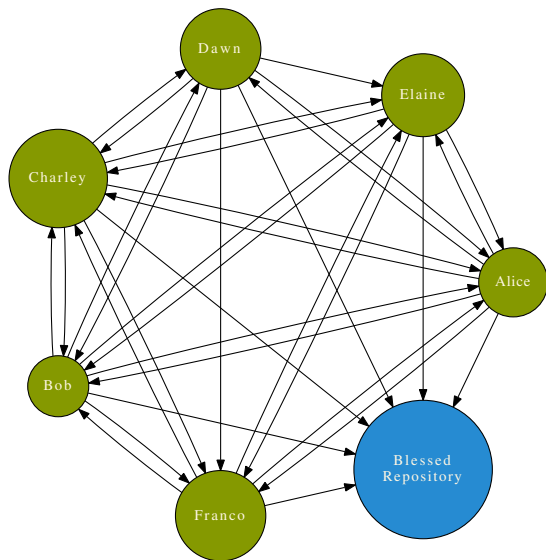
Why Use a Source Code Control System?

- It's a time machine for code and assets
- Coordinate work among project contributors
- Track what changes went into which releases
- Record *why* something changed and who made the change

Centralized Systems: Subversion, Perforce, etc.



Distributed Systems: Git, Mercurial, etc.



Centralized vs. Distributed

Feature	Centralized	Distributed
Checkout	Partial	Full Clone
Committing	Done on Remote	Local Only
Push/Pull	Automatic Push	Manual

Patches vs. Snapshots

- Most version control systems use a set of patches (difference between a file before and after it was changed) and each set of patches is assigned an incremented ID.
- When a commit is made in Git it creates a new snapshot of the repository and assigns it a SHA1 hash
- The SHA1 hash is cryptographically secure and is used to verify the integrity of the repository
- Since the commit IDs do not follow one another numerically you have to use the `git log` command to see the commit order

Working Directory, Index, and Repository

The three “trees”:

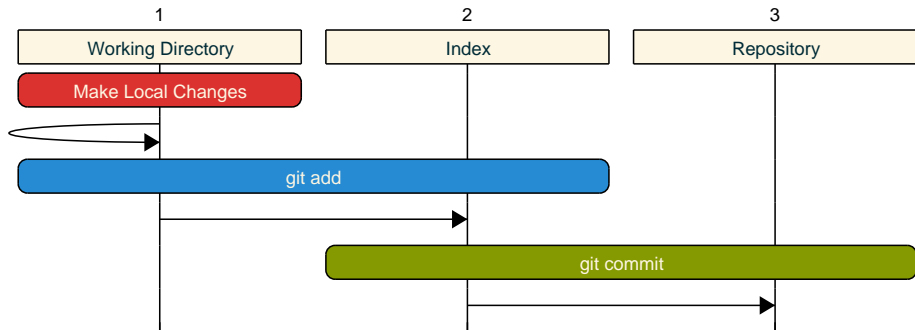
Working Directory The project's files as you and your tools see them.

The Index Changes that will be included in the next commit.

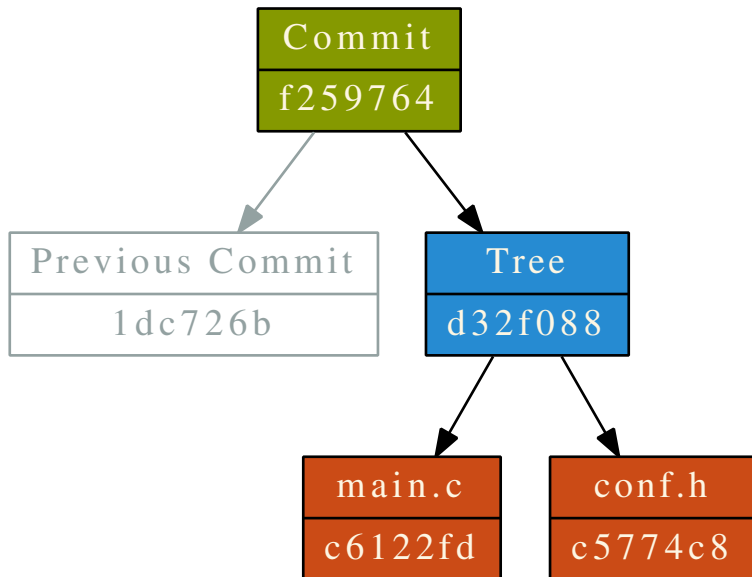
The Git Database (Repository) All commits, files, and history information.

(Note: The index is sometimes referred to as the *staging area*.)

Basic Git Workflow



What is a Commit in Git?



Starting Out: Cloning a Repository

To get a copy of an existing repository you *clone* it:

```
$ git clone <url>
```

Exercise: Cloning a Repository

1. Change to the following directory:
`repos`
2. Clone the `basic.git` repository:
`$ git clone basic.git`
3. This should have created a `basic` directory

Exercise: Viewing the History Log

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Use the following command to see the commit history:
`$ git log --oneline`
3. Pick a commit and run:
`$ git cat-file -p <commit>`
4. Also cat-file the listed tree, then the listed blob

The `git config` Tool

All Git configuration information is stored in simple text files using the INI format. If you don't want to edit files by hand you can use the `git config` tool to do it for you.

Setting Your Name and Email Address

Before you do anything at all with Git you should tell it your name and email address. This information is recorded with each commit you make.

```
$ git config --global user.name "Your Name Here"
```

```
$ git config --global user.email "Your Email Address Here"
```


Exercise: Telling Git Who You Are

If you haven't done so previously:

1. Set your name and email address using `git config`
2. Verify your settings:

```
$ git config --list
```

Exercise: Telling Git Which Editor to Use

If you haven't done so previously:

1. Tell Git which text editor to use:

```
$ git config --global core.editor <program>
```

Here are some examples for <program>:

- ▶ emacs
- ▶ vi
- ▶ "'C:/Program Files/Notepad++/notepad++.exe'"
- ▶ "code --wait"

Or search the web for instructions for your preferred editor.

Adding Untracked Files

- When you create a new file in the working directory Git will show the file as *untracked*
- You can see what Git thinks about a file using `git status`:

```
$ git status <file>
```

- To tell Git to start tracking the file use `git add`:

```
$ git add <file>
```

- This creates a commit object to hold the file
- If you change the file you need to use `git add` again
- To record your change to the repository use `git commit`:

```
$ git commit
```

Exercise: Adding a New File

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create a new file called `config.h`
3. Add some content to the file (any text will do)
4. Review the output of `git status`
5. Add the file to the index
6. Check the output of `git status` again
7. Commit the change

Modifying Existing Files

Git knows when you have changed a file that it is tracking:

- The file will show as *modified* in a `git status`
- You can use `git diff` to see what has changed
- Stage the change with `git add`
- Commit the change with `git commit`

Exercise: Editing an Existing File

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Edit the `main.c` file
3. Add some content to the file (any text will do)
4. Review the output of `git status`
5. Review the output of `git diff`
6. Add and commit the change

Renaming Files

Git allows you to rename files in two different ways:

1. Rename the file any way you want and let Git figure it out.
2. Use Git to rename the file:

```
$ git mv <old> <new>
```

When you rename files Git will track the rename in the repository.

(Note: If possible, using the `git mv` command is the safest way to rename a file.)

Exercise: Renaming a File

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Rename the `config.h` file from the previous exercise to `init.h`
3. Review the output of `git status`
4. Notice that Git already staged the rename
5. Commit your change

Removing Files

When you no longer need a file you can remove it by:

1. Remove the file with any tool, then run `git rm`.
In this case `git status` will report that the file was deleted and `git rm` will stage the removal.
2. Use `git rm` directly.
The `git rm` command can delete the file and then stage the removal in one step.

Exercise: Removing a File

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Remove the `init.h` file
3. Review the output of `git status`
4. Notice that the file deletion was staged for you
5. Commit your changes
6. Restore the file

Restoring a Removed File

You can always restore a file from a previous commit using the `git checkout` command:

```
git checkout HEAD <file> Restore <file> from the last commit  
git checkout HEAD^ <file> Restore <file> from two commits ago  
git checkout 325a910 <file> Restore <file> from commit 325a910
```

Accessing the Commit Log

The easiest way to review the commit history is by viewing the output from the `git log` command. To get a compact commit listing use the following command:

```
$ git log --oneline
```

Try it out!

Getting More Detail from the Log

The Log command supports many options for controlling its output. The three most common ways of calling it are:

1. Basic details with full commit message:

```
$ git log
```

2. Single line output with commit subject:

```
$ git log --oneline
```

3. Basic details with diff output:

```
$ git log --patch
```

Customizing the Commit Log

We can also fully customize the output of the log command:

```
$ git log \  
  --pretty=format:'%Cgreen%h%Creset %Cred%cd%Creset %Cblue%ae'
```

(Note: We'll see later how we can create aliases for these complicated commands.)

Searching for a Commit

The `log` command can be used to search the commit history for changes involving a string:

- Find commits that added or removed lines containing `printf`:

```
$ git log -G printf
```

- The same, but with regular expressions:

```
$ git log -G 's?printf'
```

- Must change the number of occurrences:

```
$ git log -S printf
```

The Reference Log

Git also keeps a temporary history file which tracks every change you make to HEAD. It's call the *reflog*:

```
$ git reflog
```

To see the history of all branches you've worked on:

```
$ git reflog --all
```

Reference log with dates:

```
$ git log -g --pretty=format:"%h %cd %gd %gs"
```

(Entries in the reflog are automatically removed after 90 days.)

Specifying Revisions

Examples of absolute revisions `f259764`, `feature`, `HEAD`, `@`

First parent of a commit `f259764^` or `f259764~1`

Grandparent of a commit `f259764~2`

Entries from the reflog `master@{yesterday}`, `HEAD@{5}`

Ranges (useful for generating diffs) `1cfd75c..0ee2723`

Exercise: Using git rev-parse

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Use the following command to see the commit history:
`$ git log --oneline`
3. Experiment with `git rev-parse`, for example:
`$ git rev-parse HEAD`

Fixing the Last Commit

You finish creating a commit and then realize:

- Forgot to add a file to the index
- You have typos in your commit message
- etc.

Before pushing the commit you can edit it:

```
$ git commit --amend
```

The Process of Amending a Commit

1. Add any missing files to the index
2. Run `git commit --amend`
3. Edit the commit message if necessary
4. Exit your text editor

(Note: If you want to remove a file from a commit you will need to perform a soft reset and create a new commit.)

Exercise: Amending the Last Commit

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create a branch that starts at the commit named `merge-start`
`$ git checkout -b NAME merge-start`
3. Take note of the last commit hash
4. Amend the last commit, changing the commit message
5. Notice that the most recent commit was rewritten

Unstaging a Modification or File

Have you ever staged a file by accident and then wanted to *unstage* it?

```
$ git reset HEAD <file>
```

Or:

```
$ git reset -- <file>
```

(Or, create an *unstage* alias. We'll do this later.)

Exercise: Unstaging a File

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create a branch that starts at the commit named `merge-start`
`$ git checkout -b NAME merge-start`
3. Edit the `main.c` file, making a simple change to it
4. Stage (`git add`) and unstage (`git reset`) the file

Restoring a Modified File

Have you ever changed a file and wanted to restore it back to how it was in the last commit?

```
$ git checkout -- <file>
```

What about its state two commits ago?

```
$ git checkout HEAD~2 <file>
```


Exercise: Restoring a File

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. In the last exercise we changed the `main.c` file, restore it back to its previous state

Exercise: Restoring a File

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Look through the commit history and find the commit that added a version number to `main.c`
3. Restore `main.c` to its content before the version number was added

Reverting a Commit

Sometimes an entire commit needs to be undone.

```
$ git revert <commit>
```

...will create a new commit that reverses the changes in <commit>

Exercise: Reverting a Commit

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create a branch that starts at the commit named `merge-start`
`$ git checkout -b NAME merge-start`
3. Edit the `main.c` file, making a simple change to it
4. Commit the change
5. Revert the commit you just created

What are Aliases

- Just like with your favorite shell, Git supports aliases.
- When you run a Git command, it will first be looked up as an alias
- For example:

```
$ git unstage main.c
```

Could be an alias for:

```
$ git reset HEAD -- main.c
```

Creating New Aliases

The easiest way to create an alias is with the `git config` command:

```
$ git config --global alias.unstage "reset HEAD --"
```

The `git config --global` command simply edits `~/.gitconfig`:

```
[alias]
```

```
unstage = reset head --
```

Some Useful Aliases

[alias]

b = branch -vv

s = status --short

ci = commit

co = checkout

ds = describe --long --tags --dirty --always

lg = log --pretty=format:'%Cgreen%h%Creset %Cred%cd%Creset %'

sb = submodule

sbu = submodule update --init --recursive

sbp = submodule update --remote --checkout

unstage = reset head --

(Taken from the examples/aliases.ini file.)

Exercise: Create Some Aliases

1. Take a moment and create some helpful aliases
2. Pick a repository and test your new aliases

Why You May Want to Ignore Certain Files

There are several types of files you may want Git to ignore:

- Build artifact files (e.g., object files)
- Temporary files or directories
- Log files generated during testing

Telling Git to Ignore Files

- When you tell Git to ignore a file it will no longer show up in the `git status` listing as modified or untracked.
- Ignore files are listed in the `.gitignore` file as a list of file patterns

Ignoring Files Using Glob Patterns

Examples of various glob patterns you can use:

- Using a file extension glob (these match files at any directory depth):
 - ▶ `*.o`
 - ▶ `*.a`
- Anchoring a pattern to a specific directory (start with `/`):
 - ▶ `/dist/*.o`
 - ▶ `/log/*.log`
- Ignoring entire directories (end in `/`):
 - ▶ `log/`
 - ▶ `/tmp/`
- Remove a pattern from the ignore list (start with `!`):
 - ▶ `!*.*`

What is Stashing?

Stashing is for those times when you are in the middle of working but need to switch tasks. With stashing you can:

1. Save modifications to tracked files, new files, and staged changes
2. Safely clean your working directory
3. Come back later and restore the changes from step 1

Saving Work In Progress without Committing

The fastest way to save your work in progress and restore the three trees to HEAD:

```
$ git stash
```

This will leave you with a clean index and working directory.

(Note: You can use the `--message` option to name the new stash object. You can also use the `--all` option to capture ignored files.)

Listing and Recovering Stashes

List stashes:

```
$ git stash list
```

Restore the first stash (`stash@{0}`) and then delete it:

```
$ git stash pop
```

Restore the first stash without deleting it:

```
$ git stash apply
```

Exercise: Pushing and Popping the Stash

1. Change to the directory holding a clone of the following repository:
`repos/conflicts.git`
2. Create a branch that starts off the `origin/feature` branch
3. Add a comment to the bottom of `main.c`
4. Create a new stash
5. Review the index and working directory state
6. Restore the stash that was previously created

What is Interactive Staging?

When you give the `--interactive` option to the `git add` command you enter an interactive shell where you can:

- See the differences between the index and the working directory
- Select files to unstage
- Stage individual patch hunks

Starting an Interactive Staging Session

Anytime you would normally use `git add` or `git rm` to update the index you can use:

```
$ git add --interactive
```

or:

```
$ git add -i
```

instead.

Staging Individual Patch Hunks

There are two ways to stage individual patch hunks:

1. Enter the interactive staging tool:

```
$ git add -i
```

2. Select the patch tool:

```
What now> patch
```

Alternatively, you can jump right into the patch tool with:

```
$ git add --patch main.c
```

Unstaging Individual Patch Hunks

If you have staged more changes than you want to commit you can unstage individual patch hunks:

```
$ git reset --patch main.c
```

This will drop you into the interactive patch tool for unstaging.

Exercise: Staging Patch Hunks

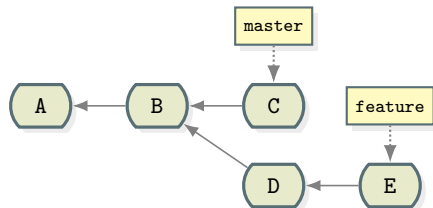
1. Change to the directory holding a clone of the following repository:
`repos/conflicts.git`
2. Create a branch that starts off the `origin/feature` branch
3. Edit `main.c`; add comments to the top and bottom of the file
4. Review the difference between the working directory and the index:

```
$ git diff
```
5. Interactively stage one of the patch hunks by splitting the first hunk:

```
$ git add -p main.c
...
Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]? s
```
6. Review the updated index:

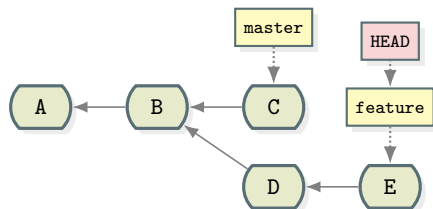
```
$ git status
$ git diff --cached
```

Separate Lines of Development



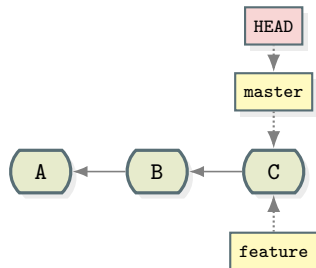
- Branches allow you to work independently of others
- The default branch in Git is called *master*
- Branches can be joined back together via *merging*

What Branch Are You Using?



- Git uses a pointer called HEAD to track the current branch
- The `.git/HEAD` file references a branch's head
- A branch's head is its latest commit ID

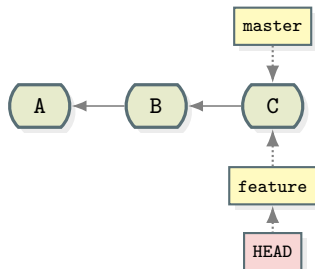
How Do You Create a Branch?



```
$ git branch feature
```

- Creates a branch called feature
- HEAD still points at master

How Do You Switch Branches?



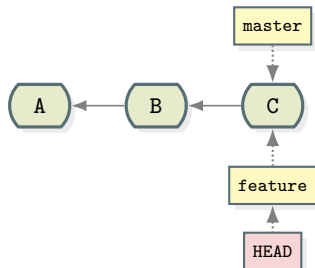
```
$ git checkout feature
```

- Changes where HEAD points
- (HEAD now points to the feature branch)

Exercise: Moving HEAD Around

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. List the names of all branches
3. For each branch:
 1. Check out the branch
 2. Use `git log --oneline` to find the latest commit hash

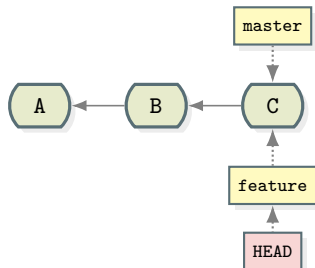
Creating and Switching Branches



```
$ git checkout -b feature
```

- Create a feature branch
- And point HEAD at feature

Creating a Branch from a Revision



```
$ git checkout -b feature master
```

- Create a feature branch which starts at master
- And point HEAD at feature

Exercise: Creating Branches

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Switch to the `master` branch
3. Create a new branch called `m2`
4. Use `git log` to see the latest commit:

`$ git log --oneline -1`
5. Create a branch called `f2` that starts at `merge-start`
6. Compare the latest commits on `m2` and `f2`

Viewing Files from Other Branches

Sometimes you just want to view a complete file the way it is on another branch (or revision). You can do that with `git show`:

```
$ git show master:main.c  
# [file contents]
```

```
$ git show HEAD^:main.c  
# [file contents]
```

Note: You can use the `--no-pager` option to `git` to dump the entire file to the terminal without using a pager.

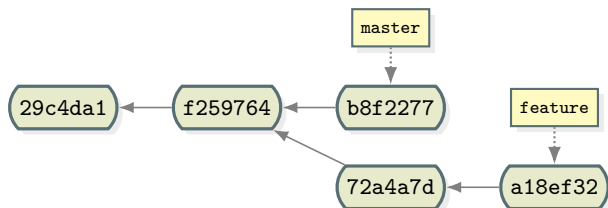
Using `git diff` with Branch Names

See how files changed in `feature` compared to `master`:

```
$ git diff master..feature
```

- Compares the tip of `master` with the tip of `feature`
- Note: uses two dots ("`..`")
- If `master` has diverged you need to compare specific revisions instead

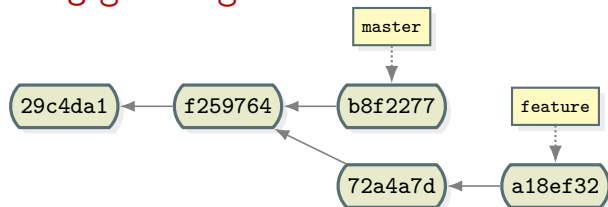
Using git diff with a Merge Base



```
$ git merge-base feature master  
f259764e2f5a16eae7b33a96a8fb5105df99cbfb  
$ git diff f259764..feature
```

- Used when the base branch has diverged
- The merge-base subcommand locates the common ancestor
- Shortcut: `git diff master...feature` (Note: three dots)

Using git log to Visualize Branches



```
$ git log --oneline --abbrev-commit \  
    --all --graph --decorate --color
```

```
* a18ef32 (feature) Add a missing newline character  
* 72a4a7d Add a version number  
| * b8f2277 (master) Add a proper greeting message  
|/  
* f259764 Indicate success by returning zero  
* 29c4da1 Initial import of main.c
```


Using gitk to Visualize Branches

Sometimes it's easier to use a GUI:

```
$ gitk --all
```

The screenshot shows the gitk GUI with the following components:

- Menu Bar:** File, Edit, View, Help
- Commit Graph:** A graph on the left showing the relationship between commits. The 'feature' branch is highlighted in green, and the 'master' branch is highlighted in yellow. The graph shows a sequence of commits: 'Initial import of main.c', 'Indicate success by returning zero', 'Add a proper greeting message', 'Add a version number', and 'Add a missing newline character'.
- Commit List:** A table of commit details with three columns: Author, Commit Message, and Date/Time.

Author	Commit Message	Date/Time
Samantha Bee <	Add a missing newline character	2018-04-17 16:58:50
Samantha Bee <	Add a version number	2018-04-17 16:58:50
Jane Doe <jdoe@	Add a proper greeting message	2018-04-17 16:58:50
Jane Doe <jdoe@	Indicate success by returning zero	2018-04-17 16:58:50
Jane Doe <jdoe@	Initial import of main.c	2018-04-17 16:58:50
- Status Bar:** At the bottom, it shows the SHA1 ID: a18ef32212ae8b283c11acbadc729645e70d370c, navigation arrows, and a row indicator: Row 1 / 5.

Safely Deleting Branches

If a branch has been pushed or merged it can be safely deleted:

```
$ git branch --delete feature
```

or, the shortcut...

```
$ git branch -d feature
```

Deleting Branches with Extreme Prejudice

You can delete a branch without any safety checks:

```
$ git branch --delete --force feature
```

beware of the shortcut:

```
$ git branch -D feature
```

Recreating a Deleted Branch

```
$ git branch -D feature  
Deleted branch feature (was a18ef32).
```

```
$ git branch feature a18ef32  
# branch is now restored
```

- As long as the deleted branch's commits have not been pruned by the garbage collector you can recover the branch.

Deleting Remote Branches

When you are completely done with a branch you may want to delete your local copy and any remote copies. Remote copies can be deleted with `git push`:

```
$ git push --delete origin feature
```

What is a Tag?

- In Git a tag is a pointer to a specific commit object.
- Typically they are used for recording a software release and include details about who made the release.

Creating Tags

To create a tag that points to HEAD:

```
$ git tag <name>
```

For release you probably want to create an annotated tag:

```
$ git tag -a <name>
```

You can also:

- Sign a tag using your GPG private key
- Create a tag that points at any commit

Listing Tags

Once you have tags in your repository you can list them:

```
$ git tag --list
```

To list tags from a remote you can:

```
$ git ls-remote --tags <remote>
```


Deleting or Recreating Tags

Tags can be safely deleted when no longer needed:

```
$ git tag --delete <name>
```

If you absolutely must recreate an existing tag, and you understand the problems it might cause, you can:

```
$ git tag --force -a <name>
```

Branching from a Tag

Once a tag is created you can start a new branch off it:

```
$ git checkout -b <branch-name> <tag-name>
```

This is useful for creating a branch to fix a bug in a past release.

Pushing Tags

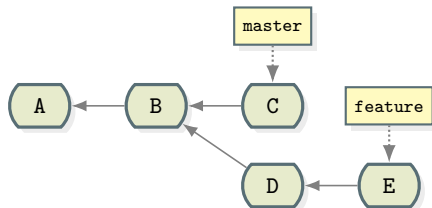
Which pushing branches to a remote Git will *not* push tags automatically.
You must do it manually:

```
$ git push --tags
```

Exercise: Tagging the Current Commit

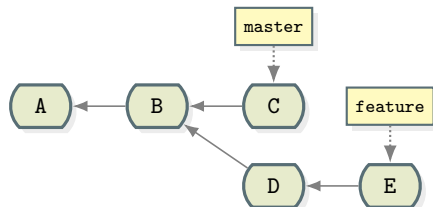
1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create a tag called `v1.1`
3. Use `git tag --list` to list your tags
4. Try using `git rev-parse` to see how the tag name resolves
5. Create a branch starting from the tag
6. Push the tag back to the `origin` remote
7. Use `git ls-remote` to confirm the tags were pushed

Merging Branches



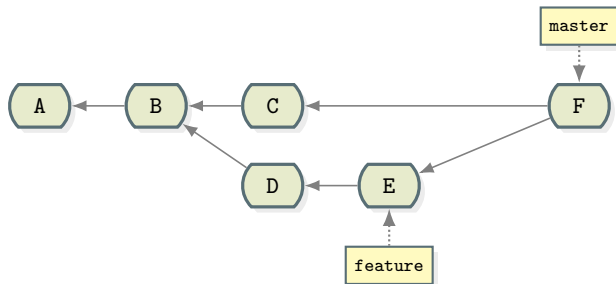
- Work on the feature branch is complete
- We would like commits D and E to move to the `master` branch
- We will do this through merging

Revisiting the Merge Base



- The feature branch and master branch share a common ancestor
- That ancestor is commit B
- Git will perform a 3-way merge with B, C, and E
- The result is a commit with two parents

Merges Cause Multiple Parents



- A new commit is created: F
- This is called a *merge commit* and has two parents

Performing a Merge

1. Switch to the destination branch:

```
$ git checkout master
```

2. Merge in the other branch:

```
$ git merge feature
```

3. Write a commit message in your text editor

Exercise: Merging a Branch

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create and switch to a new branch that starts at v1.0:
`$ git checkout -b NAME v1.0`
3. Merge the `origin/feature` branch into your new branch

What is a Merge Conflict?

- You are merging two branches that changed the same thing
- Git doesn't assume one change is more important than the other
- The merge process is paused and you must resolve the issue
- After resolving issues, you resume the merge process

How Git Reports a Merge Conflict

```
$ git merge feature
Auto-merging main.c
CONFLICT (content): Merge conflict in main.c
Automatic merge failed; fix conflicts and then commit the result
```

What git status Shows During a Conflict

On branch master

You have unmerged paths.

(fix conflicts and run "git commit")

(use "git merge --abort" to abort the merge)

Unmerged paths:

(use "git add <file>..." to mark resolution)

both modified: main.c

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

Merge Conflict Markers

```
int main (int argc, char** argv) {  
<<<<<< HEAD  
    printf("%s\n", "Hello Everyone");  
=====  
    print_version();  
    printf("%s\n", "Hello World!");  
>>>>>> feature  
    return 0;  
}
```

Resolving Merge Conflicts

1. Edit all files that are *unmerged*
2. Use `git add` to add them to the index
3. Finish the merge by using `git commit`

Exercise: Merging with Conflicts

1. Change to the directory holding a clone of the following repository:
`repos/conflicts.git`
2. Create and switch to a new branch that starts at v1.0:
`$ git checkout -b NAME v1.0`
3. Merge the `origin/feature` branch into your new branch
4. Resolve conflicts and finish the merge

Merging a Remote Branch into HEAD

Let's say you want to merge `origin/master` into `master`:

```
$ git pull origin master
```

Or more simply:

```
$ git pull
```

Which is a shortcut for:

```
$ git fetch origin
```

```
$ git merge origin/master
```

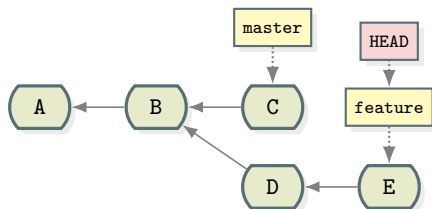

What is Rebasing?

When rebasing a topic branch (i.e. `feature`) onto an ancestor branch (i.e. `master`):

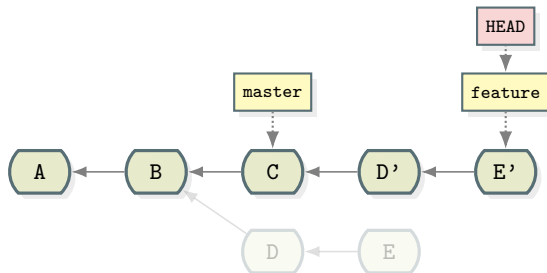
- A new commit history is created for `feature` so that it comes *after* all work on `master`
- New commits are created, old commits are abandoned
- Commits are created by *replaying* diffs on top of `master`

Visualizing a Rebase

Before the rebase:



After the rebase:



A Word of Caution

While rebasing is completely safe, it can make things difficult if you are not careful.

- Rebasing published branches can lead duplicated commits for other team members
- Best practice: Rebase before pushing commits to a remote branch
- Each team member should work on their own branches
- We'll talk about ways to fix things if you break these rules

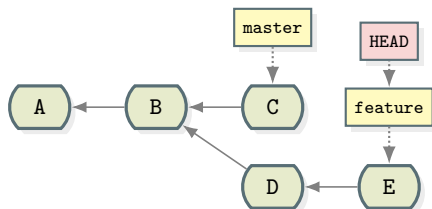
What is Rebasing?

When rebasing a topic branch (i.e. `feature`) onto an ancestor branch (i.e. `master`):

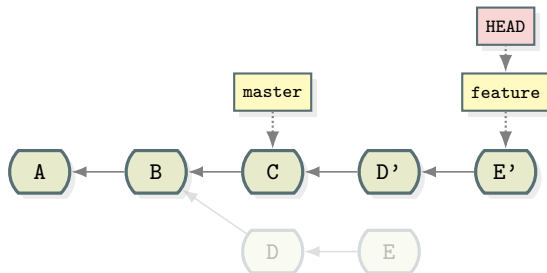
- A new commit history is created for `feature` so that it comes *after* all work on `master`
- New commits are created, old commits are abandoned
- Commits are created by *replaying* diffs on top of `master`

Visualizing a Rebase

Before the rebase:



After the rebase:



A Word of Caution

While rebasing is completely safe, it can make things difficult if you are not careful.

- Rebasing published branches can lead duplicated commits for other team members
- Best practice: Rebase before pushing commits to a remote branch
- Each team member should work on their own branches
- We'll talk about ways to fix things if you break these rules

How to Rebase a Branch

1. Use `git checkout` to move HEAD to the branch you want to rebase. Typically this will be a topic branch.
2. The index and working directory should be clean. Commit or stash changes as necessary.
3. Run `git rebase` listing the ancestor branch you want to rebase onto.
4. If there are merge conflicts, resolve them and continue with `git rebase --continue`.

Example Rebase

Rebasing feature onto master:

```
$ git checkout feature
```

```
Switched to branch 'feature'
```

```
$ git rebase master
```

```
First, rewinding head to replay your work on top of it...
```

```
Applying: ...
```

```
...
```


Exercise: Rebasing a Branch

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create a branch that starts at the commit named `merge-start`
`$ git checkout -b NAME merge-start`
3. Use `git log --oneline` to review the commit history
4. Rebase onto of the `master` branch
5. Review the commit history again and identify the rewritten commits and the merged commits from `master`

Exercise: Rebasing with Conflicts

1. Change to the directory holding a clone of the following repository:
`repos/conflicts.git`
2. Create a branch that starts at the commit named `merge-start`
`$ git checkout -b NAME conflict-start`
3. Use `git log --oneline` to review the commit history
4. Rebase onto of the `master` branch
5. Fix the conflict and use `git rebase --continue` to resume
6. Review the commit history again and identify the rewritten commits and the merged commits from `master`

Pulling and Rebasing

When rebasing onto a remote branch you have two options:

1. Use `git fetch` to fetch the remote branch, followed by `git rebase` to start the rebase process:

```
$ git fetch origin master  
$ git rebase origin/master
```

2. Use `git pull --rebase`

```
$ git pull --rebase origin master
```

When `git pull` is given the `--rebase` flag it will do a rebase after the fetch instead of the usual merge.

Using the Rebase Command Interactively

When you want to have full control of the rebase process you can perform an interactive rebase instead. An interactive rebase allows you to:

- Select a range of commits to rewrite (start with the *parent* of the commit you want to start with)
- Optionally stop at each commit so you can change the commit message or add more files
- Squash multiple commits into a single commit (we'll do this later)
- Run a command after stopping on specified commits

Starting an Interactive Rebase

After selecting a range of commits, start the rebase with either the `--interactive` flag or the corresponding short version: `-i`. Here's an example:

```
$ git rebase -i HEAD~3
```

The rebase command will start your text editor with a rebase script where you can specify what you want to do with each commit in the given range.

(Note: Remember, you are rewriting the commit history.)

Rebase Order and Commands

The rebase script allows you to perform several actions:

- `pick`: Keep the commit as is
- `edit`: Stop and amend the commit
- `squash`: join commit to preceding commit
- Reorder script lines: reorder commits
- Delete script lines: delete commits

Overall Flow of an Interactive Rebase

1. Pick a range of commits you want to rewrite
2. Edit the rebase script and exit the text editor
3. Do necessary work on each commit, then continue the rebase

Exercise: Interactive Rebasing

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create a branch that starts at the commit named `merge-start`
3. Interactively edit the commit which has the message "Add a version number"
4. Amend the commit when the rebase stops on it:

```
$ git commit --amend
```
5. Resume the rebase:

```
$ git rebase --continue
```


What Does It Mean to Squash Commits?

Have you ever made several small commits while debugging something?
Ever wanted to join all those commits into a final, good commit?
That's what squashing is.

Squashing Commits with a Rebase

To squash commits:

1. Start an interactive rebase for all commits to squash
2. Leave the first listed commit as “pick”
3. Change all remaining commits to “squash”
4. Exit your text editor

(Note: If you want to rebase all commits you will need to give the `--root` flag to the rebase command.)

Exercise: Squashing Commits

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create a branch that starts at the commit named `merge-start`
`$ git checkout -b NAME merge-start`
3. Take note of the commit history
4. Start an interactive rebase with the first commit in the history
5. Squash all commits into the first listed commit
6. Review the commit history

Rebasing Published Branches

First, avoid rebasing commits that have been published.

However, if you must, then keep these items in mind:

- Rebasing creates *new* commits and *abandons* existing commits
- Your teammates will see duplicates of all rebased commits
- Anyone who has based work on rebased commits will be confused and will probably end up having a very complicated commit history
- Those people should run: `git pull --rebase`

Everyone Must Pull

When you rebase published commits *everyone else* should immediately perform a `git pull --rebase` to update their branches.

This type of pull is smart enough to detect rebased commits and fix any local commits that relied on them.

You may even want to make this the default kind of pull:

```
$ git config --global pull.rebase true
```

Listing Remotes

- When you clone a repository it will have one remote by default: *origin*
- The origin remote points back to the original clone URL
- You can list remotes with the following command:

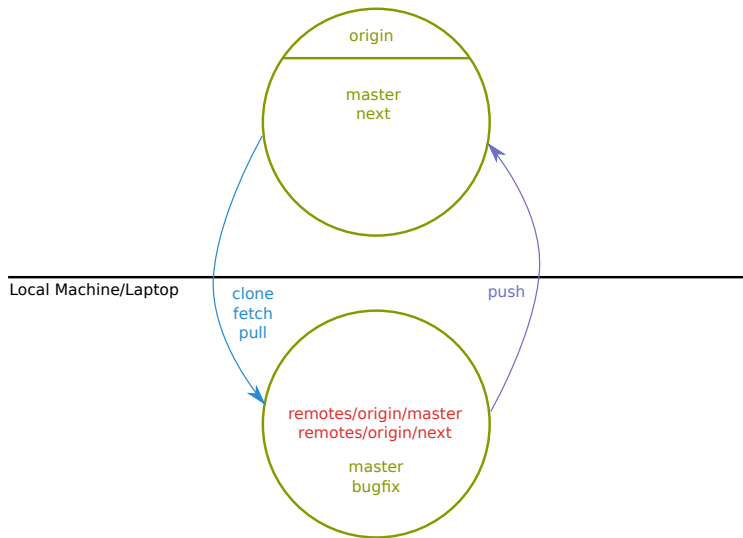
```
$ git remote -v show
```

Exercise: Listing Remotes

1. Go into any repository that we've used today
2. List the remotes for that repository
3. What does the `-v` flag do?

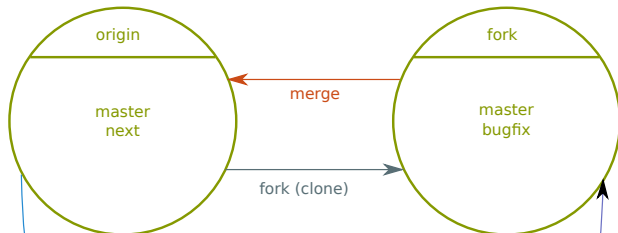
Typical Remote Setup

Remote Server

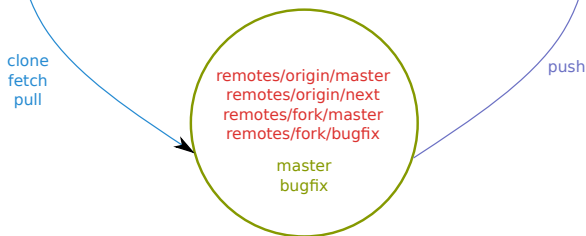


Dual Remote Setup (Fork)

Remote Server



Local Machine/Laptop



Adding Remotes

It's common to have more than one remote. To add another remote you:

```
$ git remote add <name> <url>
```

Reasons to have additional remotes:

- Pull from a coworker's repo
- Backup your repository without pushing to the central one
- Creating a pull request on an open source project

Renaming and Removing Remotes

You can rename a remote:

```
$ git remote rename <old> <new>
```

or completely remove a remote from your repository:

```
$ git remote remove <name>
```

Fetching Objects from a Remote

You can fetch all objects from a remote by:

```
$ git fetch <remote>
```

(Note: this fetches objects and places them in the repository but does not change any branches.)

Fetching and Updating Branches

If you want to fetch objects from a remote and then update your branch to match upstream you have two choices:

1. Fetch and then merge:

```
$ git fetch <remote>  
$ git merge origin/master
```

2. Use git pull:

```
$ git pull origin master
```

(Note: if you are on a tracking branch you can use `git pull` without any arguments.)

Pushing Objects to Remotes

To publish your changes to a remote you can:

```
$ git push origin master
```

This pushes your changes to the origin server's master branch.

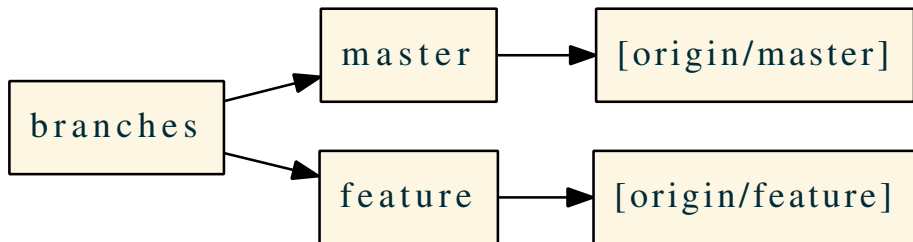
(Note: if you are on a tracking branch you can use `git push` without any arguments.)

Pushing Tags to Remotes

Using `git push` won't send your tags to the remote by default. You have to use the `--tags` flag:

```
$ git push --tags origin
```

What is a Tracking Branch?



A *tracking branch* is branch that is linked to a specific remote repository.

Creating Tracking Branches

If you `git checkout` a branch that has the same name as a remote branch it will automatically track the remote branch:

```
$ git branch --all
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/feature
  remotes/origin/master
$ git checkout feature
Branch 'feature' set up to track remote branch 'feature' from
```

Manually Creating Tracking Branches

You can manually create a branch and set its tracking information:

```
$ git checkout -b ft origin/feature
```

Branch 'ft' set up to track remote branch 'feature' from 'origin'

Getting Tracking Information

To see tracking information for local branches:

```
$ git branch -vv
```

```
* master 7f86f19 [origin/master] Add a proper greeting message
```

Setting Tracking Information

To change the tracking information for the current branch:

```
$ git branch -u origin/master
```

```
Branch 'master' set up to track remote branch 'master' from 'origin'
```

What a Reset Can Affect

Before figuring out what a `git reset` is, let's first talk about what it can change:

- The working directory: the project's files
- The index: what changes will be in your next commit
- HEAD: the tip of the current branch

What is a Reset?

A `git reset` changes the state of one or more trees. It's meant as a permanent change unlike `git checkout`.

There are three types of a reset:

- Soft (only change the tip of the current branch)
- Mixed (soft + change the index)
- Hard (mixed + update the working directory)

(Note: Like with rebase, a reset will change your Git history. You should only do this for unpublished commits.)

Performing a Soft Reset

A *soft* reset will:

- Move HEAD back to the specified commit
- Keep the index at the current commit
- Leave the working directory alone

(This is very similar to `git commit --amend.`)

Exercise: Soft Reset

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create a branch that starts at the commit named `merge-start`
`$ git checkout -b NAME merge-start`
3. Use `git log --oneline` to review the commit history
4. Use `git reset --soft` to back up 2 commits:
`$ git reset --soft HEAD~2`
5. Use `git log --oneline` to see how the history changed
6. Use `git status` to review the index and working directory
7. Use `git commit` to create a new commit

Performing a Mixed Reset

A *mixed* reset will:

- Move HEAD back to the specified commit
- Move the index back to the specified commit
- Leave the working directory alone

(This is the default type of reset if you don't use any flags.)

Exercise: Mixed Reset

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create a branch that starts at the commit named `merge-start`
`$ git checkout -b NAME merge-start`
3. Use `git log --oneline` to review the commit history
4. Use `git reset --mixed` to back up 2 commits:
`$ git reset --mixed HEAD~2`
5. Use `git log --oneline` to see how the history changed
6. Use `git status` to review the index and working directory
7. Use `git commit` to create a new commit

Performing a Hard Reset

A *hard* reset will:

- Move HEAD back to the specified commit
- Move the index back to the specified commit
- Update the working directory to the specified commit

(Warning: A hard reset **can delete files** from your working directory. It's a good idea to commit or stash changes before doing a reset.)

Exercise: Hard Rest

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create a branch that starts at the commit named `merge-start`
`$ git checkout -b NAME merge-start`
3. Use `git log --oneline` to review the commit history
4. Use `git reset --hard` to back up 2 commits:
`$ git reset --hard HEAD~2`
5. Use `git log --oneline` to see how the history changed
6. Use `git status` to review the index and working directory
7. Notice that the working directory was also changed

Unstaging Changes

A reset is most often used to *unstage* a file. Since the default type of reset is a mixed reset, and the default commit is HEAD, you can unstage a file using:

```
$ git reset -- file.c
```

Which is the same as

```
$ git reset --mixed HEAD file.c
```

Introduction to Branch Management

If you don't have a plan it's easy to create a mess of confusing branches in your repository. Branch management is a way for all team members to agree on a plan for:

- Keeping the master branch in a releasable state
- Creating branches for new features and bug fixes
- Where to merge these feature branches
- How to make and record software releases

Mainline Branches

It's generally advisable to have two main branches in your repository:

- `master` represents an always releasable stable branch
- `next` or `develop` is where work for the next release goes

Typically you never directly commit to these branches.

Topic Branches

When you need to create a new feature or fix a bug you:

1. Create a new *topic* branch off of the `next` branch
2. Work on your branch until it is ready for release
3. Merge your topic branch back into `next`

(Note: In a lot of cases your topic branch should be pushed back to a private clone and only show up in the main repository after it is merged.)

Release Branches

When the time comes to create a release you:

1. Create a *release* branch from the `next` branch (at this point `next` can resume being used for a future release)
2. Finalize the release on the release branch (i.e. bump the version number)
3. Merge the release branch into `master` and tag the release
4. Merge the release branch back into `next`

Tools to Automate Branch Management

If you are interested in using this model of branch management you may be interested in a way to automate it.

That's precisely what gitflow does.