10. Saving changes

git add

The git add command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit. However, git add doesn't really affect the repository in any significant way—changes are not actually recorded until you run git commit.

In conjunction with these commands, you'll also need git status to view the state of the working directory and the staging area.

Usage

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

Stage all changes in <file> for the next commit.

```
$ git add <directory>
```

Stage all changes in <directory> for the next commit.

```
$ git add -p
```

Begin an interactive staging session that lets you choose portions of a file to add to the next commit. This will present you with a chunk of changes and prompt you for a command. Use y to stage the chunk, n to ignore the chunk, s to split it into smaller chunks, e to manually edit the chunk, and q to exit.

Discussion

The git add and git commit commands compose the fundamental Git workflow. These are the two commands that every Git user needs to understand, regardless of their team's collaboration model. They are the means to record versions of a project into the repository's history.

Developing a project revolves around the basic edit/stage/commit pattern. First, you edit your files in the working directory. When you're ready to save a copy of the current state of the project, you stage changes with git add. After you're happy with the staged snapshot, you commit it to the project history with git commit.

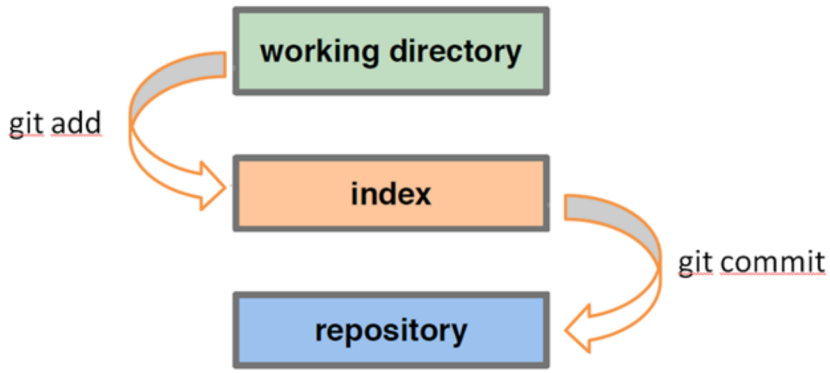
The git add command should not be confused with svn add, which adds a file to the repository. Instead, git add works on the more abstract level of changes. This means that git add needs to be called every time you alter a file, whereas svn add only needs to be

called once for each file. It may sound redundant, but this workflow makes it much easier to keep a project organized.

11. The Staging Area

The staging area is one of Git's more unique features, and it can take some time to wrap your head around it if you're coming from an SVN (or even a Mercurial) background. It helps to think of it as a buffer between the working directory and the project history.

Instead of committing all the changes you've made since the last commit, the stage lets you group related changes into highly focused snapshots before committing it to the project history. This means you can make all sorts of edits to unrelated files, then go back and split them up into logical commits by adding related changes to the stage and commit them piece-by-piece. As in any revision control system, it's important to create atomic commits so that it's easy to track down bugs and revert changes with minimal impact on the rest of the project.



Example

When you're starting a new project, git add serves the same function as svn import. To create an initial commit of the current directory, use the following two commands:

```
$ git add .
$ git commit
```

Once you've got your project up-and-running, new files can be added by passing the path to git add:

```
$ git add hello.py
$ git commit
```

The above commands can also be used to record changes to existing files. Again, Git doesn't differentiate between staging changes in new files vs. changes in files that have already been added to the repository.

git commit

The git commit command commits the staged snapshot to the project history. Committed snapshots can be thought of as "safe" versions of a project—Git will never change them unless you explicitly ask it to. Along with git add, this is one of the most important Git commands.

While they share the same name, this command is nothing like svn commit. Snapshots are committed to the local repository, and this requires absolutely no interaction with other Git repositories.

Usage

```
$ git commit
```

Commit the staged snapshot. This will launch a text editor prompting you for a commit message. After you've entered a message, save the file and close the editor to create the actual commit. git commit -m "<message>"

Commit the staged snapshot, but instead of launching a text editor, use <message> as the commit message.

```
$ git commit -a
```

Commit a snapshot of all changes in the working directory. This only includes modifications to tracked files (those that have been added with git add at some point in their history).

Example

The following example assumes you've edited some content in a file called hello.py and are ready to commit it to the project history. First, you need to stage the file with git add, then you can commit the staged snapshot.

```
$ git add hello.py
$ git commit
```

This will open a text editor (customizable via git config) asking for a commit message, along with a list of what's being committed:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
```

```
#
#modified: samplecode.py
```

```
Change the message displayed by samplecode.py

- Update the sayHello() function to output the user's name
- Change the sayGoodbye() function to a friendlier message
```

12. Syncing

SVN uses a single central repository to serve as the communication hub for developers, and collaboration takes place by passing changesets between the developers' working copies and the central repository. This is different from Git's collaboration model, which gives every developer their own copy of the repository, complete with its own local history and branch structure. Users typically need to share a series of commits rather than a single changeset. Instead of committing a changeset from a working copy to the central repository, Git lets you share entire branches between repositories.

The commands presented below let you manage connections with other repositories, publish local history by "pushing" branches to other repositories, and see what others have contributed by "pulling" branches into your local repository.

git remote

The git remote command lets you create, view, and delete connections to other repositories. Remote connections are more like bookmarks rather than direct links into other repositories. Instead of providing real-time access to another repository, they serve as convenient names that can be used to reference a not-so-convenient URL.

Usage

```
$ git remote
```

List the remote connections you must other repositories.

```
$ git remote -v
```

Same as the above command, but include the URL of each connection.

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

Create a new connection to a remote repository. After adding a remote, you'll be able to use <name> as a convenient shortcut for<url> in other Git commands.

```
$ git remote rm <name>
```

Remove the connection to the remote repository called <name>.

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

Rename a remote connection from <old-name> to <new-name>.

Discussion

Git is designed to give each developer an entirely isolated development environment. This means that information is not automatically passed back and forth between repositories. Instead, developers need to manually pull upstream commits into their local repository or manually push their local commits back up to the central repository. The git remote command is just an easier way to pass URLs to these "sharing" commands.

The origin Remote

When you clone a repository with git clone, it automatically creates a remote connection called origin pointing back to the cloned repository. This is useful for developers creating a local copy of a central repository, since it provides an easy way to pull upstream changes or publish local commits. This behaviour is also why most Git-based projects call their central repository origin.

13. Repository URLs

Git supports many ways to reference a remote repository. Two of the easiest ways to access a remote repo are via the HTTP and the SSH protocols. HTTP is an easy way to allow anonymous, read-only access to a repository. For example:

```
http://host/path/to/repo.git
```

But, it’s generally not possible to push commits to an HTTP address (you wouldn’t want to allow anonymous pushes anyways). For read-write access, you should use SSH instead:

```
ssh://user@host/path/to/repo.git
```

You’ll need a valid SSH account on the host machine, but other than that, Git supports authenticated access via SSH out of the box.

Examples

In addition to origin, it’s often convenient to have a connection to your teammates’ repositories. For example, if your co-worker, John, maintained a publicly accessible repository on dev.example.com/john.git, you could add a connection as follows:

```
$ git remote add john http://dev.example.com/john.git
```

Having this kind of access to individual developers’ repositories makes it possible to collaborate outside of the central repository. This can be very useful for small teams working on a large project.

14. git fetch

The git fetch command imports commits from a remote repository into your local repo. The resulting commits are stored as remote branches instead of the normal local branches that we've been working with. This gives you a chance to review changes before integrating them into your copy of the project.

Usage

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

Fetch all of the branches from the repository. This also downloads all of the required commits and files from the other repository.

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

Same as the above command, but only fetch the specified branch.

Discussion

Fetching is what you do when you want to see what everybody else has been working on. Since fetched content is represented as a remote branch, it has absolutely no effect on your local development work. This makes fetching a safe way to review commits before integrating them with your local repository. It's similar to svn update in that it lets you see how the central history has progressed, but it doesn't force you to actually merge the changes into your repository.

Remote Branches

Remote branches are just like local branches, except they represent commits from somebody else's repository. You can check out a remote branch just like a local one, but this puts you in a detached HEAD state (just like checking out an old commit). You can think of them as read-only branches. To view your remote branches, simply pass the -r flag to the git branch command. Remote branches are prefixed by the remote they belong to so that you don't mix them up with local branches. For example, the next code snippet shows the branches you might see after fetching from the origin remote:

```
git branch -r
# origin/master
# origin/develop
# origin/some-feature
```

Again, you can inspect these branches with the usual git checkout and git log commands. If you approve the changes a remote branch contains, you can merge it into a local branch with a normal git merge. So, unlike SVN, synchronizing your local repository with a remote repository is actually a two-step process: fetch, then merge. The git pull command is a convenient shortcut for this process.

Examples

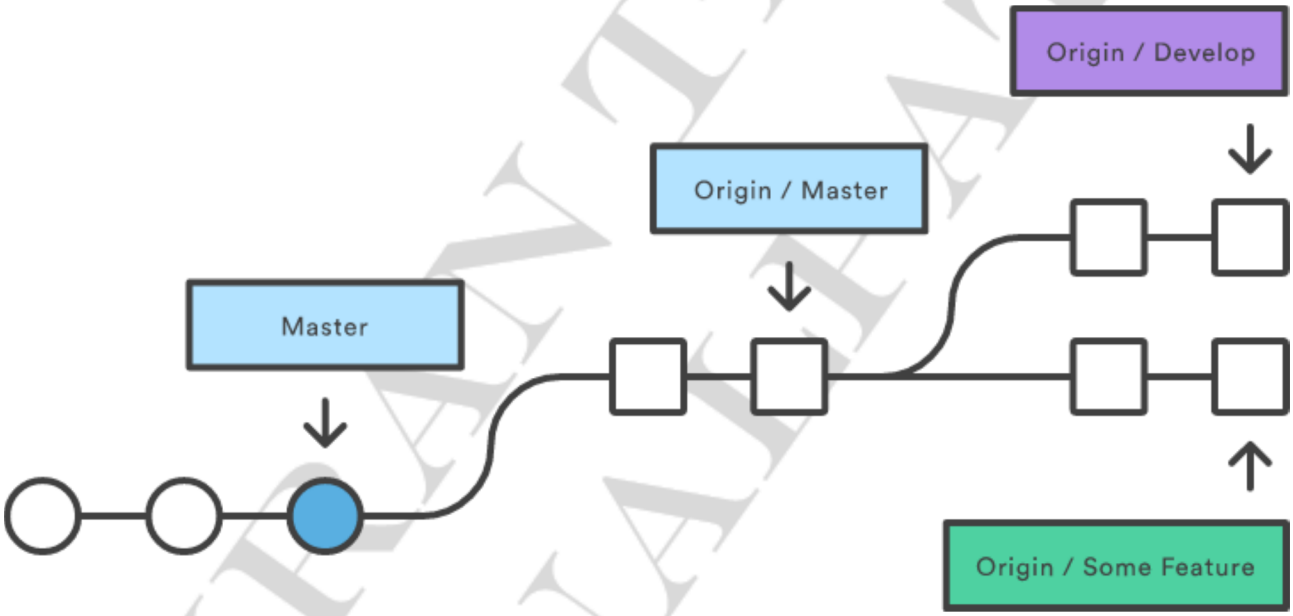
This example walks through the typical workflow for synchronizing your local repository with the central repository's master branch.

```
$ git fetch origin
```

This will display the branches that were downloaded:

```
a1e8fb5..45e66a4 master -> origin/master
a1e8fb5..9e8ab1c develop -> origin/develop
* [new branch] some-feature -> origin/some-feature
```

The commits from these new remote branches are shown as squares instead of circles in the diagram below. As you can see, git fetch gives you access to the entire branch structure of another repository.



To see what commits have been added to the upstream master, you can run a git log using origin/master as a filter

```
$ git log --oneline master..origin/master
```

To approve the changes and merge them into your local master branch with the following commands:

```
$ git checkout master
$ git log origin/master
```

Then we can use git merge origin/master

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

The origin/master and master branches now point to the same commit, and you are synchronized with the upstream developments.

15. git pull

Merging upstream changes into your local repository is a common task in Git-based collaboration workflows. We already know how to do this with git fetch followed by git merge, but git pull rolls this into a single command.

Usage

```
$ git pull <remote>
```

Fetch the specified remote's copy of the current branch and immediately merge it into the local copy. This is the same as git fetch <remote> followed by git merge origin/<current-branch>.

```
$ git pull --rebase <remote>
```

Same as the above command, but instead of using git merge to integrate the remote branch with the local one, use git rebase.

Discussion

You can think of git pull as Git's version of svn update. It's an easy way to synchronize your local repository with upstream changes.

You start out thinking your repository is synchronized, but then git fetch reveals that origin's version of master has progressed since you last checked it. Then git merge immediately integrates the remote master into the local one:

16. Pulling via Rebase

The --rebase option can be used to ensure a linear history by preventing unnecessary merge commits. Many developers prefer rebasing over merging, since it's like saying, "I

want to put my changes on top of what everybody else has done." In this sense, using git pull with the --rebase flag is even more like svn update than a plain git pull.

In fact, pulling with --rebase is such a common workflow that there is a dedicated configuration option for it:

```
$ git config --global branch.autosetuprebase always
```

After running that command, all git pull commands will integrate via git rebase instead of git merge.

Examples

The following example demonstrates how to synchronize with the central repository's master branch:

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

This simply moves your local changes onto the top of what everybody else has already contributed.

17. git push

Pushing is how you transfer commits from your local repository to a remote repo. It's the counterpart to git fetch, but whereas fetching imports commits to local branches, pushing exports commits to remote branches. This has the potential to overwrite changes, so you need to be careful how you use it. These issues are discussed below.

Usage

```
$ git push <remote> <branch>
```

Push the specified branch to <remote>, along with all of the necessary commits and internal objects. This creates a local branch in the destination repository. To prevent you from overwriting commits, Git won’t let you push when it results in a non-fast-forward merge in the destination repository.

```
$ git push <remote> --force
```

Same as the above command, but force the push even if it results in a non-fast-forward merge. Do not use the --force flag unless you’re absolutely sure you know what you’re doing.

```
$ git push <remote> --all
```

Push all of your local branches to the specified remote.

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

Tags are not automatically pushed when you push a branch or use the --all option. The --tags flag sends all your local tags to the remote repository.

Discussion

The most common use case for git push is to publish your local changes to a central repository. After you've accumulated several local commits and are ready to share them with the rest of the team, you (optionally) clean them up with an interactive rebase, then push them to the central repository.

18. Github SSH login

Generating SSH keys for github

1. Open Terminal.

2. Paste the command below, substituting in your GitHub email address.

 # ssh-keygen -t rsa -b 4096 -C "your_email@example.com"

3. When you're prompted to "Enter a file in which to save the key,"
Give below mentioned path

/home/<USERNAME>/.ssh/id_rsa_github

Note: USERNAME in above path is you Linux system user with which you have logged in.

Generating public/private rsa key pair.
Enter a file in which to save the key
(/home/<USERNAME>/.ssh/id_rsa):/home/<USERNAME>/.ssh/id_rsa_DO_github

4. Hit enter when it asks to enter passphrase

Enter passphrase (empty for no passphrase): [Type a passphrase]
Enter same passphrase again: [Type passphrase again]