Basic project management with git

1. Starting a new project

1.1. Initializing the database

Creating a new repository is done only one time for every project, so it is easy to forget it. The command is simply `git init`:

```
$ mkdir newproject
$ cd newproject
$ git init
Initialized empty Git repository in /home/dbindner/newproject/.git/
```

1.2. Adding new files to a repository

When a git repository is created, it is initially empty. For files to be tracked, they must first be marked for inclusion using the `git add` command. Here we create a simple `hello.c` program and mark it for tracking.

```
$ cat > hello.c <<END
void main( void ) {
  printf( "Hello." )
};
END
$ git add hello.c
```

It is possible to add many files with one `git add` command, and it is very common to simply add all of the files in the current directory structure with

```
$ git add .
```

Either way, `hello.c` has been marked for addition to the repository, but it has not actually been committed (saved). There is still time to make edits or change your mind before you record this version for posterity. If you decide that you do not want a file to be tracked, `git rm --cache file` can achieve this.

When you are ready, use `git commit` to freeze the current state of the project and save a snapshot.

```
$ git commit -a
```

Every commit has a comment attached to it. Short comments can be included directly on the command line (if you remember). Otherwise, an editor is started for your comments. Since we didn't specify a comment above, you'll have to type one. Type "Initial commit." on the first line and save.

A note on commit messages: Though not required, it's a good idea to begin the commit message with a single short (less than 50 character) line summarizing the change, followed by a blank line and then a more thorough description. Tools that turn commits into email, for example, use the first line on the Subject line and the rest of the commit in the body.

Git will respond that it has committed the initial tree and will print a long string of letters and numbers that serves to uniquely identify this commit. Later on this change can be examined or even undone by referring to this identifier (more on this later).

2. Git for the lone developer

Although it is often overlooked, revision control can be very useful even for a lone developer without collaborators. Revision control still allows an author to gracefully back out of mistakes and to try experimental changes safely.

2.1. Simple edit cycle

For a lone developer, the current working tree is always up to date, so the edit cycle is very simple:

Edit cycle

1. perform edits
2. use `git add` when new files are created that should be tracked
3. `git commit -a` periodically to commit snapshots
4. goto step 1

2.2. Remembering where you are

One of the advantages of revision control comes when you've done a series of edits and lose track of the changes you've made. Git makes it easy to compare the current state of your files against the last commit. Two commands for this purpose are `git status` and `git diff`.

The command `git status` reports a summary of the current state of your tree. It reports files that are part of your project but not marked for tracking, and it reports files that have been changed. For example, after a small edit to our `hello.c` this is the state:

```
$ 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:   hello.c

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

        hello.c.bak

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

To see the actual changes, i.e. that two lines were added to the top of the file:

```
$ git diff
diff --git a/hello.c b/hello.c
index e5f9ec2..159ef94 100644
--- a/hello.c
+++ b/hello.c
@@ -1,3 +1,5 @@
+#include <stdio.h>
+
 void main( void ) {
   printf( "Hello." )
 };
```

If you need to see changes from a particular point in history, you can specify a specific commit to diff against using the unique identifier string reported by `git commit` or `git log`. You can also generate a diff between two different points in the history by giving two unique identifiers.

2.3. A little bit of history

Sometimes you will forget the history of changes that have been committed so far, or you will forget which commit included some particular change. The `git log` and `git whatchanged` commands can help. So far in our project, the only history we have is the initial commit, and there have been no further changes.

```
$ git log
commit 6c473a4fdafae1ab94aea1d47dbdf98c542a654f
Author: Donald J. Bindner <dbindner@emma.bindner.home>
Date:   Thu Apr 20 19:24:45 2006 -0500

    Initial commit.
```

So see something useful from `git whatchanged` first we need to commit some kind of change. Let's commit the changes to `hello.c` that we created above. We'll include the comment for our commit directly on the command line.

```
$ git commit -a -m "Remembered to include header files"
```

Now in addition to the initial commit, we have a commit that creates a change and `git whatchanged` will show it:

```
$ git whatchanged
diff-tree 814b141... (from 6c473a4...)
Author: Donald J. Bindner <dbindner@emma.bindner.home>
Date:   Thu Apr 20 20:32:43 2006 -0500

    Remembered to include header files

:100644 100644 e5f9ec2... 159ef94... M  hello.c
```

The format is similar to the output of `git log` but includes a relatively cryptic line at the end. Of particular note is the capital "M" noting that the file `hello.c` has been modified.

Note: `git whatchanged` also accepts a `-p` argument that shows complete diffs at each step.

3. Recovering from mistakes

If mistakes never occurred, the need for revision control would be much diminished. It is a fact of life that authors make errors of all kinds, from simple typos to grand complicated edits that run amok.

3.1. Undoing uncommitted changes

The simplest kind of mistake is making an unexpected changed to a file in your project then wishing you could go back to the last committed state. Perhaps your cat walked across the computer keyboard or your editor exploded. This kind of error is simple to correct. You can take the file back to the last committed version. Simply check it out again.

```
$ git checkout -- hello.c
$ git status
nothing to commit
```

Another approach is to use `git reset`. This can be especially handy if multiple files are involved (for example when you've accidently deleted a whole handful of files). To make all of the working files in the project identical to the last commit in the repository (this is called the HEAD state), do a hard reset.

> **Warning**
>
> Be careful with this, because any real work done since the last commit will also be lost. A quick `git status` before using `git reset` would not be out of line.

```
$ git reset --hard HEAD
```

3.2. Resetting a regretted commit

It will happen from time to time that you get all of your files together, make a commit, and then immediately regret it. Sometimes you'll realize that you included more in your commit than you intended. Other times you'll realize you left one thing out. Sometimes you just botch the comment and would like to redo it. If you notice your mistake before you've gone on to do other work, `git reset` can fix it.

```
$ touch Makefile
$ git add Makefile
$ git commit -a -m 'Added Makefile'
(realized our mistake)
$ git reset --soft HEAD^
```

Here we've used a *soft* reset. That means that the reset only applies to the repository and not to the working files. A *hard* reset in this instance would also have had the unintended effect of removing our newly-created Makefile. The meaning of `HEAD^` is the state just above (leading to) our current state. This is also referred to as the parent state of HEAD, and you should think of the caret as an arrow pointing up.

Resetting the HEAD to its parent effectively destroys the last commit. It is then possible to correct things and make the commit that was actually intended.

```
$ cat > Makefile<<END
hello: hello.c
       gcc -o hello hello.c
END
$ git commit -a -m 'Added Makefile'
```

In versions of git starting from 1.3.0, `git commit` has a new option, `--amend`, which essentially performs an automated version of this procedure. Without the `git reset --soft HEAD^` step above, you can fix the working files and then:

```
$ cat > Makefile<<END
hello: hello.c
```

```
        gcc -o hello hello.c
END
$ git commit -a --amend
```

A quick application of `git log` can verify that everything is now exactly as intended.

3.3. Reverting an old change

Not every error in a project is detected before it is committed to the repository, and sometimes the commit will be shared with other users or followed by later correct work. In a situation like this, it is not appropriate to use `git reset` to turn back the clock. Rather, what is needed is a mechanism for undoing an individual commit.

The `git revert` command reverts the change that a previous commit introduced. It applies the "reverse patch" to your working tree and records the changes into your repository.

> **Note:** Because `git revert` changes your working tree, before using it you must have committed all of your changes so that your working tree and HEAD agree.

It is common to make use of `git log` and `git diff` first to isolate the change to be undone.

```
$ git log
commit 5635b6fce1aec37900496957fefb1d5a4e374970
Author: Donald J. Bindner <dbindner@emma.bindner.home>
Date:   Sun Apr 23 20:57:10 2006 -0500

    Added Makefile

commit 814b141776c51b1622f1913393c4eb7a95714a33
Author: Donald J. Bindner <dbindner@emma.bindner.home>
Date:   Thu Apr 20 20:32:43 2006 -0500

    Remembered to include header files

commit 6c473a4fdafae1ab94aea1d47dbdf98c542a654f
Author: Donald J. Bindner <dbindner@emma.bindner.home>
Date:   Thu Apr 20 19:24:45 2006 -0500

    Initial commit.
$ git diff 6c473a4fdaf..814b141776
diff --git a/hello.c b/hello.c
index e5f9ec2..159ef94 100644
--- a/hello.c
+++ b/hello.c
@@ -1,3 +1,5 @@
+#include <stdio.h>
```

```
+
 void main( void ) {
   printf( "Hello." )
 };
$ git revert 814b141776
First trying simple merge strategy to revert.
Finished one revert.
```

4. Managing branches

A single git repository can maintain multiple branches of development. To create a new branch named "experimental", use

```
$ git branch experimental
```

If you now run `git branch` you'll get a list of all existing branches:

```
$ git branch
  experimental
* master
```

The "experimental" branch is the one you just created, and the "master" branch is a default branch that was created for you automatically. The asterisk marks the branch you are currently on. To switch to the experimental branch, type

```
$ git checkout experimental
```

Now edit a file, commit the change, and switch back to the master branch.

```
(edit file)
$ git commit -a
$ git checkout master
```

Check that the change you made is no longer visible, since it was made on the experimental branch and you're back on the master branch. You are free to edit files on the master branch now, even making edits that are different or incompatible with the experimental branch.

```
(edit file)
$ git commit -a
```

At this point, the two branches have diverged, with different changes made in each. To merge the changes from experimental into the currently checked-out master branch, run

```
$ git pull . experimental
```

If the changes don't conflict, you're done.

If there are conflicts, markers will be left in the problematic files showing the conflict, and `git diff` will show this. Once you've edited the files to resolve the conflicts, `git commit -a` will commit the result of the merge.

Finally,

```
$ gitk
```

will show a nice graphical representation of the resulting history.

4.1. Deleting branches

If you develop on a branch crazy-idea, then regret it, you can always delete the branch permanently with

```
$ git branch -D crazy-idea
```

Branches are cheap and easy, so this is a good way to try something out.

5. Using git for collaboration

5.1. The everyone-pulls model

Suppose that Alice has started a new project with a git repository in /home/alice/project, and that Bob, who has a home directory on the same machine, wants to contribute.

Bob begins with:

```
$ git clone /home/alice/project myrepo
```

This creates a new directory "myrepo" containing a clone of Alice's repository. The clone is on an equal footing with the original project, possessing its own copy of the original project's history.

Bob then makes some changes and commits them:

```
(edit files)
$ git commit -a
(repeat as necessary)
```

When he's ready, he tells Alice to pull changes from the repository at /home/bob/myrepo. She does this with:

```
$ cd /home/alice/project
$ git pull /home/bob/myrepo
```

This actually pulls changes from the branch in Bob's repository named "master". Alice could request a different branch by adding the name of the branch to the end of the git pull command line.

This merges Bob's changes into her repository; "git whatchanged" will now show the new commits. If Alice has made her own changes in the meantime, then Bob's changes will be merged in, and she will need to manually fix any conflicts.

A more cautious Alice might wish to examine Bob's changes before pulling them. She can do this by creating a temporary branch just for the purpose of studying Bob's changes:

```
$ git fetch /home/bob/myrepo master:bob-incoming
```

which fetches the changes from Bob's master branch into a new branch named bob-incoming. (Unlike `git pull`, `git fetch` just fetches a copy of Bob's line of development without doing any merging). Then

```
$ git whatchanged -p master..bob-incoming
```

shows a list of all the changes that Bob made since he branched from Alice's master branch.

After examining those changes, and possibly fixing things, Alice can pull the changes into her master branch:

```
$ git checkout master
$ git pull . bob-incoming
```

The last command is a pull from the "bob-incoming" branch in Alice's own repository.

Later, Bob can update his repo with Alice's latest changes using

```
$ git pull
```

Note that he doesn't need to give the path to Alice's repository; when Bob cloned Alice's repository, git stored the location of her repository in the file .git/remotes/origin, and that location is used as the default for pulls.

Bob may also notice a branch in his repository that he didn't create:

```
$ git branch
```

```
* master
  origin
```

The "origin" branch, which was created automatically by "git clone", is a pristine copy of Alice's master branch; Bob should never commit to it.

If Bob later decides to work from a different host, he can still perform clones and pulls using the ssh protocol:

```
$ git clone alice.org:/home/alice/project myrepo
```

Alternatively, git has a native protocol, or can use rsync or http; see gitlink:git-pull[1] for details.

5.2. The CVS model

CVS users are accustomed to giving a group of developers commit access to a common repository. Start with an ordinary git working directory containing the project, and remove the checked-out files, keeping just the bare .git directory:

```
$ mv project/.git /pub/repo.git
$ rm -r project/
```

Next, give every team member read/write access to this repository. One easy way to do this is to give all the team members ssh access to the machine where the repository is hosted. If you don't want to give them a full shell on the machine, there is a restricted shell which only allows users to do git pushes and pulls; see gitlink:git-shell[1].

Put all the committers in the same group, and make the repository writable by that group:

```
$ chgrp -R $group repo.git
$ find repo.git -mindepth 1 -type d |xargs chmod ug+rwx,g+s
$ GIT_DIR=repo.git git-repo-config core.sharedrepository true
```

Make sure committers have a umask of at most 027, so that the directories they create are writable and searchable by other group members.

Suppose this repository is now set up in /pub/repo.git on the host foo.com. Then as an individual committer you can clone the shared repository:

```
$ git clone foo.com:/pub/repo.git/ my-project
$ cd my-project
```

and hack away. The equivalent of `cvs update` is

```
$ git pull origin
```

which merges in any work that others might have done since the clone operation.

> **Note:** The first `git clone` places the following in the `my-project/.git/remotes/origin` file, and that's why the previous step and the next step both work.
>
> ```
> URL: foo.com:/pub/project.git/ my-project
> Pull: master:origin
> ```

You can update the shared repository with your changes using:

```
$ git push origin master
```

If someone else has updated the repository more recently, `git push`, like `cvs commit`, will complain, in which case you must pull any changes before attempting the push again.

In the `git push` command above we specify the name of the remote branch to update (`master`). If we leave that out, `git push` tries to update any branches in the remote repository that have the same name as a branch in the local repository. So the last `push` can be done with either of:

```
$ git push origin
$ git push repo.shared.xz:/pub/scm/project.git/
```

as long as the shared repository does not have any branches other than `master`.

> **Note:** Because of this behavior, if the shared repository and the developer's repository both have branches named `origin`, then a push like the above attempts to update the `origin` branch in the shared repository from the developer's `origin` branch. The results may be unexpected, so it's usually best to remove any branch named `origin` from the shared repository.

6. Keeping track of history

Git history is represented as a series of interrelated commits. The most recent commit in the currently checked-out branch can always be referred to as HEAD, and the "parent" of any commit can always be referred to by appending a caret, "^", to the end of the name of the commit. So, for example,

```
git diff HEAD^..HEAD
```

shows the difference between the most-recently checked-in state of the tree and the previous state, and

```
git diff HEAD^^..HEAD^
```

shows the difference between that previous state and the state two commits ago. Also, HEAD~5 can be used as a shorthand for HEAD^^^^^, and more generally HEAD~n can refer to the nth previous commit. Commits representing merges have more than one parent, and you can specify which parent to follow in that case; see gitlink:git-rev-parse[1].

The name of a branch can also be used to refer to the most recent commit on that branch; so you can also say things like

```
git diff HEAD..experimental
```

to see the difference between the most-recently committed tree in the current branch and the most-recently committed tree in the experimental branch.

But you may find it more useful to see the list of commits made in the experimental branch but not in the current branch, and

```
git whatchanged HEAD..experimental
```

will do that, just as

```
git whatchanged experimental..HEAD
```

will show the list of commits made on the HEAD but not included in experimental.

You can also give commits convenient names of your own: after running

```
$ git tag v2.5 HEAD^^
```

you can refer to HEAD^^ by the name "v2.5". If you intend to share this name with other people (for example, to identify a release version), you should create a "tag" object, and perhaps sign it; see gitlink:git-tag[1] for details.

You can revisit the old state of a tree, and make further modifications if you wish, using `git branch`: the commands

```
$ git branch stable-release v2.5
$ git checkout stable-release
```

will create a new branch named "stable-release" starting from the commit which you tagged with the name v2.5 then check it out. This is the safest way to go back and look at earlier states of a project. You can easily delete the new branch when you no longer need it.

You can also reset the state of any branch to an earlier commit at any time with

```
$ git reset --hard v2.5
```

This will remove all later commits from this branch and reset the working tree to the state it had when the given commit was made.

> **Warning**
>
> If this branch is the only branch containing the later commits, those later changes will be lost. Don't use "git reset" on a publicly-visible branch that other developers pull from, as git will be confused by history that disappears in this way.

7. The Git Index

To this point, the discussion has presented git has having essentially two parts, the working tree and the repository. In other documents, the repository may be called the Object Database, because it consists essentially of carefully catalogued blob objects (files), commit objects, tag objects and so on.

There is actually a third part to git, the Index. The Index is sometimes (although infrequently) called the Cache, and it acts as a layer between the working tree and the repository. When files are committed, they are actually copied to the Index and committed from there to the repository.

A hint of this was apparent in the "Remembering where you are" section above. After we had modified `hello.c` and then executed `git status` we were told that `hello.c` was changed but not updated.

```
$ 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:   hello.c

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

        hello.c.bak

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

What that means is that the file had changed in the working tree, but had not yet been copied to the Index. In fact, we were told that we could use `git add` to add the file to the Index by hand and mark it for the next commit.

We avoided this whole issue earlier by running `git commit -a`. The `-a` argument automatically runs `git add` for any files that have had updates (but it will not add new files, an explicit `git add` is still needed for that). For most work, this is the desired behavior.

There is nothing to stop you from explicitly adding files to the Index. For example, you might work on several different files in your working tree and decide you wish break the work up into multiple commits, each containing just the changes to one or two related files. To do this, use `git add` for only the files you wish to be committed, then apply `git commit` without extra flags. Only the files in the Index will change with the new project snapshot. Continue in the same fashion until all of your work has been committed.

7.1. Git-diff and the Index

It is important to know that by default `git diff` shows differences between the working tree and the Index, not the working tree and the repository (although the Index and the Repository will usually be the same if you have not used `git add`). To specify differences between the working tree and the HEAD explicitly, you could specify

```
$ git diff HEAD
```

If you need to see differences between the Index (i.e. the cached version) and the repository, add the `--cached` flag:

```
$ git diff --cached HEAD
```

7.2. The Index as a staging area

The Index is designed to be a place to stage the next commit. As you work on files and get them into shape for a commit, you can add them to the Index via `git add`. When used this way, the Index acts as a kind of revision system Purgatory; a temporary holding place for files to wait while things are purified.

```
$ git add hello.c
```

It is bound to happen that you stage your changes only to have a later mistake occur. To recover a file from the Index (even if it has not been committed) back into the working tree, say

```
$ git checkout hello.c
```

If you notice an error, but not before you've placed the file into the Index, it is still possible to get back to the last committed change with an explicit checkout from HEAD (which will roll back both the Index and the working tree).

```
$ git checkout HEAD hello.c
```

When everything is ready to go (i.e. you have completed your edits), and you have staged all of your files, run

```
$ git commit
```

7.3. Files to ignore

Another thing often reported by `git status` is a collection of untracked files that you would probably prefer to ignore, including the backup files from editing sessions, or object files produced by a compiler. These are never going to be tracked by git, so it would be convenient if it would quit reporting them.

To ignore a file or files, put a glob pattern into `.gitignore` and commit it to the repository.

```
$ echo '*.bak' > .gitignore
$ git add .gitignore
$ git commit -a
```