Git Quickstart User Tutorial

Revision 1.0, July 30th 2010.

Development Repositories

All developers have a local Git repository in their ~*username*/public_html/ directory. Your personal repository can be viewed with a web browser at http://username.dev.fishhound.com/ where "username" is replaced with your username. Refer to the Git Quickstart Gatekeeper Tutorial for details on how these per-developer repositories are set up.

By default, your repository will track the master 'development', 'staging' and 'production' branches maintained by the Gatekeeper at */srv/git* on the development server. You will create branches in your own source code repository to do development, testing, and bug fixing, then you will merge your changes into your local 'development' branch where they will then be reviewed and pulled into the master repository by the Gatekeeper. The following tutorial gives you the basic background into how git works, allowing you to efficiently use the tool.

Initial Repository Configuration

The first time you set up a git repository, you need to do some initial git configuration. This configuration only has to happen once, and will be remembered for all subsequent uses. (NOTE: The Gatekeeper who sets up your repository will also set your user.name and user.email for you, as well as the auto-coloring logic.) The following two commands are necessary so that your source code changes are properly credited to you:

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

```
$ git config --global user.email you@yourdomain.com
```

It can also be helpful to add a little color to your git output. The color that git employes is non-distracting, and helpful for highlighting things like code differences and the active branch. It is recommended you run the following commands to enable the most usable automatic coloring of output:

```
$ git config --global color.diff auto
$ git config --global color.status auto
$ git config --global color.branch auto
```

Learn more by typing 'git help config'.

Branches

Branching is incredibly fast and lightweight with Git, and there is no limit to the number of branches you can create. It is standard practice to create “throw-away” branches in git simply to quickly test changes, merging what you like and deleting the rest. With git, branch early, branch often! For example, to create a new branch from the main development branch and immediately switch to it:

```
$ git checkout development -b featureX
Switched to a new branch 'featureX'
```

In the above example, we made a copy of the main development branch to create a new branch called 'featureX'. Any changes you make at this point affect the 'featureX' branch. In this example, git only stores the differences between 'development' and 'featureX', so the creation of the new branch is essentially instant and does not use a significant amount of disk space. You may find it helpful to name your branches created from development with "development-" at the beginning of their name, so example you may prefer naming this new branch 'development-featureX'.

If you forget what changes are in any given branch, you can always list all the differences from your current branch to another, for example:

```
$ git diff development
```

The following example lists all existing branches in your development repository, putting an asterisk before the currently active branch (add the -v flag if you'd like to see more information about each branch):

```
$ git branch
development
* featureX
production
staging
```

See the "merging" section of this document for tips on how to merge your changes from one branch into another.

If you need to delete a branch, you switch out of the branch you want to delete, then delete it as follows:

```
$ git checkout development
$ git branch -d development-featureX
Deleted branch development-featureX (was cb29c01).
```

Later in this document, you'll learn how to track remote branches, allowing you to test or merge in changes from another developer.

Whichever branch you currently have checked out is what is visible with your web-browser at http://username.dev.fishhound.com/ allowing you to easily preview different development efforts.

To list all branches, including both local and remote branches, add the '-a' flag, for example, '`git branch -a`'.

Learn more by typing '`git help checkout`', '`git help branch`', and '`git help diff`'.

Committing

A functional difference between git and many other source control systems is that files and file changes must be "added" before they can be committed. The need to add and then commit files and changes is because git has several layers, including your working files, and index (added but not committed files and changes), your local branches, and remotely tracked branches. The added index layer can be confusing at first, and can essentially be ignored while you're getting more comfortable with git, but once mastered adds many more powerful abilities, such as what is explained on this webpage. If you're looking to commit all your changes together essentially ignoring the index, you can combine the "add" step into the "commit" step as follows:

```
$ git commit -a
```

Before committing changes like this, however, you should take a look at the current status of your repository with the following command:

```
$ git status
# On branch development-featureX
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# four
# one
# three
# two
```

If you need to commit changes to specific files, you can add them individually and then commit them as follows, which will then bring up and editor telling you which files are being committed and prompting you for a commit message that will be associated with your changes:

```
$ git add one
$ git add two
$ git commit
```

It's possible to add the changes to multiple files using a single command, and to set your commit message directly from the command line without opening an editor as follows:

```
$ git add three four
$ git commit -m "Put a useful and informational commit message
here."
```

To fully understand why this is a two-step process, it helps to know that git maintains an internal index to allow you to do partial merges, etc.

Learn more by typing '`git help add`', '`git help commit`' and '`git help status`'.

Tagging

Every commit has a unique commit ID, a hexadecimal string that uniquely identifies the entire source tree at the time of that commit. You can use the 'history' command to find recent commit IDs, for example:

```
$ git log --pretty=oneline
51d44994842ced3368209fef0cd3b9e52b2019ea Third commit, more useful
info here.
9b331c60c65d29b854256562b85b2d5c4c9368e2 Second commit, useful info
here.
cb29c01dee978857279250e43e1120b49dc5f782 Initial commit.
```

When referencing a specific commit, you only need to enter enough characters from the commit ID to ensure that git knows which you are referring to. In our above example, we can see what changed between the third commit and the second commit as follows:

```
$ git diff 9b331 51d4
```

As your merge history grows, trying to remember what commit you want to reference by ID alone can be a challenge. Instead, you can quickly add tags for your own local reference. While tags can be shared with other developers, by default they are private to your own repository. For example:

```
$ git tag featureB
```

That tag will apply to the current revision. Thus, the following is now identical to our previous diff example:

```
$ git diff 9b33 featureB
```

It is also simple to create tags against older revisions, for example:

```
$ git tag featureA 9b33
$ git diff featureA featureB
```

Merging

The following example demonstrate how to create a new branch, add a new file in that branch, and then merge this change into your personal bugfix development branch:

```
$ git checkout development -b bugfix123
Switched to a new branch 'bugfix123'
$ echo "new file" > newfile.txt
$ git status
# On branch bugfix123
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
#
# newfile.txt
nothing added to commit but untracked files present (use "git add"
to track)
$ git add newfile.txt
$ git commit -m “Added a new file. Useful explanation of new file
goes here.”
[example 9b331c6] Added a new file. Useful explanation of new file
goes here.
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 newfile.txt
$ git checkout development
Switched to branch 'development'
$ git merge bugfix123
Updating cb29c01..9b331c6
Fast-forward
newfile.txt | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 newfile.txt
```

The above example shows how to merge a trivial change. Git makes it easy to merge all kinds of changes, including adding new files, deleting files, renaming files, and editing the contents of files. It will attempt to automatically merge all changes, but will also help you to manually merge if two branches edit the same place in the same file.

You can merge in multiple directions, for example from 'development' to 'featureX' to pull in upstream changes made by other developers, and then later from 'featureX' to 'development' to make your changes available to the Gatekeeper. Git is very intelligent about merging, and will remember if a change has already been merged regardless of what path the merge follows to find its way into your active branch.

Interactively Adding Changes

If you've accidentally made multiple unrelated changes to a single file, you can tell git to only add and commit some of the changes without adding and committing all of them. When

adding your changes, set the "-i" flag putting git into interactive mode. This will give you a menu of options which allow you to view patches or diffs to manually select which modifications you wish to commit:

```
*** Commands ***
1: [s]tatus 2: [u]pdate 3: [r]evert 4: [a]dd untracked
5: [p]atch 6: [d]iff 7: [q]uit 8: [h]elp
```

For more information, type '`git help add`' and read about these options.

Cherry Picking

When merging changes from one branch to another, by default all commits will be merged at once. If necessary, however, you can instead simply cherry-pick an individual commit when merging from one branch to another, only merging a single change from another branch rather than merging everything. For example, in this example we only merge the "change 2" from the "bugfix234" branch, not "change 1" or "change 3" (and git is smart enough to know that '51d4' refers to the much longer ID as it doesn't match the beginning of any other ID, so it doesn't require you to enter the entire ID string):

```
$ git branch
  development
* bugfix234
production
staging
$ git log --pretty=oneline
ac40b05a14382cf76a39d8f52a10266941fe6ca4 bugfix234 branch, change
3.
51d44994842ced3368209fef0cd3b9e52b2019ea bugfix234 branch, change
2.
9b331c60c65d29b854256562b85b2d5c4c9368e2 bugfix234 branch, change
1.
cb29c01dee978857279250e43e1120b49dc5f782 Initial commit.
$ git checkout development
Switched to branch 'development'
$ git cherry-pick 51d4
Finished one cherry-pick.
```

For more information, type '`git help cherry-pick`'.

Stashing Changes

Sometimes you need to switch to another branch while you're actively working on editing files, and before you're ready to commit your changes. If you have edited files in your current branch, and you switch to another branch, your changes will carry over to the branch you just switched to. This might be confusing if your goal is to just swap over to another branch for a quick look-see and or a quick bug squash. To avoid this, before switching branches with uncommitted changes, you can "stash" these changes temporarily while you go off and work in another branch. Once you complete your work in the other branch, you can

return to the branch you were working in, grab your stashed changes and pick up where you left off.

For example let's say you're working in a branch called 'feature345'. After working on feature 345 for a while, you're suddenly assigned a high priority bug fix on the production branch (for example, a typo that somehow slipped by QA). You could commit what you've done so far in the 'feature345' branch, but this could lead you to write an inaccurate commit message, or to committing untested or unworking code. As an alternative, you can use stash as follows:

```
$ git checkout development -b feature345

... start doing development work of feature 345, then learn need to
quickly fix a production bug ...

$ git stash save "interrupted to work on bug 101"
Saved working directory and index state On feature345: interrupted to work on bug 101
HEAD is now at ...
$ git checkout production -b "bugfix101"

... fix quick typo and commit changes...

$ git checkout feature345
$ git stash pop
```

This example switches you back to your feature branch, and pops the latest stash into it. You can continue working away and commit your new changes fully when ready.

Note the use of '`pop`'. Stashes are arranged in a stack, and you can have multiple stashes in your repo. The '`pop`' command grabs the last changes you stashed, but you can also '`list`' stashes, '`apply`' changes not on the top of the stack, '`drop`' changes if you wish to remove them, and even '`clear`' all of them.

For more information, type '`git help stash`'.

Undoing Changes

How you undo a change with git depends on whether the change has been committed or not. Branches and commits are cheap and fast with git, so if you're thinking of committing a change with the intention of undoing it later, instead consider creating a temporary branch, doing your work there, then deleting or merging that branch when you're done. "Throw-away branches" are perfectly valid when using git.

To revert an uncommitted file back to its last-committed state, you simply check it out again:

```
$ git checkout changed-file.php
```

The above example would overwrite your current file 'changed-file.php' with the latest version stored in your repository. It will *not* warn you before it removes any changes you may have made, so be careful!

If you have made changes to several files that you don't wish to keep, and just want to reset your whole repo to the last committed state, you can use:

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

This will throw away all changes to any of your files since the last commit.

You can also use tags or commit IDs to checkout previous versions of one or more files. For example, to 'rewind' a specific file to a previous commit, you do:

```
$ git checkout <commit hash> <filename>
```

The commit IDs are lengthy hexadecimal strings, but you only need to enter enough characters to be unique. Typically entering the first 4 to 6 characters of any commit ID should suffice.

Tracking Remote Changes By Other Developers

Git is a *distributed* source control tool, and is designed to easily allow you to track changes made by any number of other developers. A human Gatekeeper is responsible for maintaining a master repository, reviewing all code changes, committing completed features and bug fixes, and preparing the code for staging and production. Tracking remote changes made both other developers will allow you to coordinate your changes, helping to test and review the code changes before they are pushed upstream to the Gatekeeper. You can track as many remote repositories as you like. For example, Bdragon's repositories can be tracked in your local repository as follows:

```
$ git remote add bdragon ~bdragon/public_html
$ git fetch bdragon
From /home/bdragon/public_html
 * [new branch]      development -> bdragon/development
 * [new branch]      development-gmap -> bdragon/development-gmap
 * [new branch]      production -> bdragon/production
 * [new branch]      staging    -> bdragon/staging
```

At this point, a complete copy of Bdragon's development repository and all his development branches are contained withing your repository. This allows you to review his changes, pull in his fixes, run diffs between your code and his, etc. It is a very powerful way to share code between multiple developers. Note that if Bdragon commits additional changes to any of these branches, your copy does not get automatically updated. You can update it manually by running '`git fetch bdragon`' again -- or if tracking changes from multiple developers you can simply run '`get fetch --all`' to fetch updates from all remote repositories at once.

By default, the Gatekeeper has set up your repository to automatically track all other developer's branches, visible with the command '`git remote`', and with more detail by running '`git branch -a`'.

In the above example, code in Bdragon's 'development' branch is what he feels is ready to be pulled by the gatekeeper. To see the differences between your 'development' branch, and Bdragon's 'development' branch, you would execute the following commands:

```
$ git fetch bdragon
$ git diff development refs/remotes/bdragon/development
```

Alternatively, you can check out any of Bdragon's branches and look at the git log and individual commits. This is useful if you need to cherry pick commits, or simply need to review his individual changes. For example, git allows you to check out a temporary "detached" branch without the need to create an actual branch as follows:

```
$ git checkout refs/remotes/bdragon/development
Note: checking out 'refs/remotes/bdragon/development'.

You are in 'detached HEAD' state. You can look around, make
experimental
changes and commit them, and you can discard any commits you make
in this
state without impacting any branches by performing another
checkout.

If you want to create a new branch to retain commits you create,
you may
do so (now or later) by using -b with the checkout command again.
Example:

  git checkout -b new_branch_name
```

At this point you can poke around all you need and then switch back to one of your own branches. You can even edit files, merging your changes or throwing them away when you're done. Of course, there's nothing wrong with setting up an actual local branch for this, either. In the following example, we set up a branch, look at the log, then delete the branch to demonstrate how lightweight branching is in git, and to illustrate how easy it is to create and delete branches:

```
$ git checkout refs/remotes/bdragon/development -b bdragon-
development
Branch bdragon-development set up to track remote branch
development from bdragon.
Switched to a new branch 'bdragon-development'
$ git log
$ git checkout development
Switched to branch 'development'
$ git branch -d bdragon-development
Deleted branch bdragon-development (was 8a59b08).
```

As noted before, you can continue pulling the latest changes from Bdragon by running "`git fetch bdragon`". If there was a commit in Bdragon's tree that you wanted to merge into any of your own branches, you could cherry-pick the individual change, or even merge in the entire branch into one of your own. Or, if another developer calls in sick, you could merge his unfinished changes into one of your own branches, review and finish them, and then push them onto the Gatekeeper for staging and production.

Note that these "remote" branches are actually local copies of remote branches. That is to say, if you edit them, you are only editing your local copy, you are not making copies in the remote repositories.

For more information, run '`git help remote`'. Remember, as noted earlier in this section, remote repositories for all developers have already been auto-configured for you.

Pushing Changes To The Gatekeeper

The term Gatekeeper is a title that identifies one or more individuals that have commit permissions to the master git repository that lives in */srv/git*, and is the person or persons in charge of overseeing the flow of code changes. Any code that you would like the Gatekeeper to merge into the main development branch should first be merged into your own development branch. With your development branch fully up to date ("`git checkout development; git pull`"), and your code changes merged in, you simply need to notify the gatekeeper that you are ready for him to pull your development branch. The gatekeeper will then manually review your code, and either pull your changes or make suggestions on further improvements necessary before he can pull your code. In some instances, he may simply cherry-pick specific commits.

If the Gatekeeper takes a long time to pull your changes, you may need to update your development branch. This could result in your having to manually merge changes from other developers if other developers have committed changes to the same files you are changing. By keeping your development branch up to date, you ensure that your active development is against the latest code, and you make it easier for the Gatekeeper to pull in your changes to the master repository, though in general you shouldn't have to worry about this unless so instructed by the Gatekeeper.

Additional Resources

There are many excellent git resource available online. Here are a few places you may wish to start for additional reading:

1. The git community book: http://book.git-scm.com/
2. A git cheat sheet: http://zrusin.blogspot.com/2007/09/git-cheat-sheet.html
3. SVN crash course: http://git-scm.com/course/svn.html
4. The official git tutorial: http://www.kernel.org/pub/software/scm/git/docs/gittutorial.html
5. The git wiki: https://git.wiki.kernel.org/index.php/GitDocumentation
6. The git man pages: http://www.kernel.org/pub/software/scm/git/docs/