Bazaar Tutorial

Release 2.8.0dev1

Bazaar Developers

March 22, 2019

Contents

1 Introduction

If you are already familiar with decentralized version control, then please feel free to skip ahead to "Introducing Yourself to Bazaar". If, on the other hand, you are familiar with version control but not decentralized version control, then please start at "How DVCS is different." Otherwise, get some coffee or tea, get comfortable and get ready to catch up.

2 The purpose of version control

Odds are that you have worked on some sort of textual data – the sources to a program, web sites or the config files that Unix system administrators have to deal with in /etc. The chances are also good that you have made some sort of mistake that you deeply regretted. Perhaps you deleted the configuration file for your mailserver or perhaps mauled the source code for a pet project. Whatever happened, you have just deleted important information that you would desperately like to get back. If this has ever happened to you, then you are probably ready for Bazaar.

Version control systems (which I'll henceforth call VCS) such as Bazaar give you the ability to track changes for a directory by turning it into something slightly more complicated than a directory that we call a **branch**. The branch not only stores how the directory looks right now, but also how it looked at various points in the past. Then, when you do something you wish you hadn't, you can restore the directory to the way it looked at some point in the past.

Version control systems give users the ability to save changes to a branch by "committing a **revision**". The revision created is essentially a summary of the changes that were made since the last time the tree was saved.

These revisions have other uses as well. For example, one can comment revisions to record what the recent set of changes meant by providing an optional log message. Real life log messages include things like "Fixed the web template to close the table" and "Added SFTP suppport. Fixes #595"

We keep these logs so that if later there is some sort of problem with SFTP, we can figure out when the problem probably happened.

3 How DVCS is different

Many Version Control Systems (VCS) are stored on servers. If one wants to work on the code stored within a VCS, then one needs to connect to the server and "checkout" the code. Doing so gives one a directory in which a person can make changes and then commit. The VCS client then connects to the VCS server and stores the changes. This method is known as the centralized model.

The centralized model can have some drawbacks. A centralized VCS requires that one is able to connect to the server whenever one wants to do version control work. This can be a bit of a problem if your server is on some other machine on the internet and you are not. Or, worse yet, you **are** on the internet but the server is missing!

Decentralized Version Control Systems (which I'll call DVCS after this point) deal with this problem by keeping branches on the same machine as the client. In Bazaar's case, the branch is kept in the same place as the code that is being version controlled. This allows the user to save his changes (**commit**) whenever he wants – even if he is offline. The user only needs internet access when he wants to access the changes in someone else's branch that are somewhere else.

A common requirement that many people have is the need to keep track of the changes for a directory such as file and subdirectory changes. Performing this tracking by hand is a awkward process that over time becomes unwieldy. That is, until one considers version control tools such as Bazaar. These tools automate the process of storing data by creating a **revision** of the directory tree whenever the user asks.

Version control software such as Bazaar can do much more than just storage and performing undo. For example, with Bazaar a developer can take the modifications in one branch of software and apply them to a related branch – even if those changes exist in a branch owned by somebody else. This allows developers to cooperate without giving write access to the repository.

Bazaar remembers the ''ancestry" of a revision: the previous revisions that it is based upon. A single revision may have more than one direct descendant, each with different changes, representing a divergence in the evolution of the tree. By branching, Bazaar allows multiple people to cooperate on the evolution of a project, without all needing to work in strict lock-step. Branching can be useful even for a single developer.

4 Introducing yourself to Bazaar

Bazaar installs a single new command, **bzr**. Everything else is a subcommand of this. You can get some help with `bzr help`. Some arguments are grouped in topics: `bzr help topics` to see which topics are available.

One function of a version control system is to keep track of who changed what. In a decentralized system, that requires an identifier for each author that is globally unique. Most people already have one of these: an email address. Bazaar is smart enough to automatically generate an email address by looking up your username and hostname. If you don't like the guess that Bazaar makes, then three options exist:

1. Set an email address via `bzr whoami`. This is the simplest way.

   To set a global identity, use:

   ```
   % bzr whoami "Your Name <email@example.com>"
   ```

   If you'd like to use a different address for a specific branch, enter the branch folder and use:

   ```
   % bzr whoami --branch "Your Name <email@example.com>"
   ```

2. Setting the email address in the `~/.bazaar/bazaar.conf` [1] by adding the following lines. Please note that `[DEFAULT]` is case sensitive:

[1] On Windows, the users configuration files can be found in the application data directory. So instead of `~/.bazaar/branch.conf` the configuration file can be found as: `C:\Documents and Settings\<username>\Application Data\Bazaar\2.0\branch.conf`. The same is true for `locations.conf`, `ignore`, and the `plugins` directory.

```
[DEFAULT]
email=Your Name <email@isp.com>
```

As above, you can override this settings on a branch by branch basis by creating a branch section in `~/.bazaar/locations.conf` and adding the following lines:

```
[/the/path/to/the/branch]
email=Your Name <email@isp.com>
```

3. Overriding the two previous options by setting the global environment variable `$BZR_EMAIL` or `$EMAIL` (`$BZR_EMAIL` will take precedence) to your full email address.

5 Creating a branch

History is by default stored in the .bzr directory of the branch. In a future version of Bazaar, there will be a facility to store it in a separate repository, which may be remote.

We create a new branch by running `bzr init` in an existing directory:

```
% mkdir tutorial
% cd tutorial
% ls -a
./  ../
% pwd
/home/mbp/work/bzr.test/tutorial
%
% bzr init
% ls -aF
./  ../  .bzr/
%
```

As with CVS, there are three classes of file: unknown, ignored, and versioned. The **add** command makes a file versioned: that is, changes to it will be recorded by the system:

```
% echo 'hello world' > hello.txt
% bzr status
unknown:
  hello.txt
% bzr add hello.txt
added hello.txt
% bzr status
added:
  hello.txt
```

If you add the wrong file, simply use `bzr remove` to make it unversioned again. This does not delete the working copy in this case, though it may in others [2].

6 Branch locations

All history is stored in a branch, which is just an on-disk directory containing control files. By default there is no separate repository or database as used in svn or svk. You can choose to create a repository if you want to (see the `bzr`

[2] `bzr remove` will remove the working copy if it is currently versioned, but has no changes from the last committed version. You can force the file to always be kept with the `--keep` option to `bzr remove`, or force it to always be deleted with `--force`.

`init-repo` command). You may wish to do this if you have very large branches, or many branches of a moderately sized project.

You'll usually refer to branches on your computer's filesystem just by giving the name of the directory containing the branch. bzr also supports accessing branches over SSH, HTTP and SFTP, amongst other things:

```
% bzr log bzr+ssh://bazaar.launchpad.net/~bzr-pqm/bzr/bzr.dev/
% bzr log http://bazaar.launchpad.net/~bzr-pqm/bzr/bzr.dev/
% bzr log sftp://bazaar.launchpad.net/~bzr-pqm/bzr/bzr.dev/
```

By installing bzr plugins you can also access branches using the rsync protocol.

See the Publishing your branch section for more about how to put your branch at a given location.

7 Reviewing changes

Once you have completed some work, you will want to **commit** it to the version history. It is good to commit fairly often: whenever you get a new feature working, fix a bug, or improve some code or documentation. It's also a good practice to make sure that the code compiles and passes its test suite before committing, to make sure that every revision is a known-good state. You can also review your changes, to make sure you're committing what you intend to, and as a chance to rethink your work before you permanently record it.

Two bzr commands are particularly useful here: **status** and **diff**.

7.1 bzr status

The **status** command tells you what changes have been made to the working directory since the last revision:

```
% bzr status
modified:
   foo
```

`bzr status` hides "boring" files that are either unchanged or ignored. The status command can optionally be given the name of some files or directories to check.

7.2 bzr diff

The **diff** command shows the full text of changes to all files as a standard unified diff. This can be piped through many programs such as ''patch'', ''diffstat'', ''filterdiff" and ''colordiff'':

```
% bzr diff
=== added file 'hello.txt'
--- hello.txt   1970-01-01 00:00:00 +0000
+++ hello.txt   2005-10-18 14:23:29 +0000
@@ -0,0 +1,1 @@
+hello world
```

With the `-r` option, the tree is compared to an earlier revision, or the differences between two versions are shown:

```
% bzr diff -r 1000..          # everything since r1000
% bzr diff -r 1000..1100      # changes from 1000 to 1100
```

The `--diff-options` option causes bzr to run the external diff program, passing options. For example:

```
% bzr diff --diff-options --side-by-side foo
```

Some projects prefer patches to show a prefix at the start of the path for old and new files. The `--prefix` option can be used to provide such a prefix. As a shortcut, `bzr diff -p1` produces a form that works with the command `patch -p1`.

8 Committing changes

When the working tree state is satisfactory, it can be **committed** to the branch, creating a new revision holding a snapshot of that state.

8.1 bzr commit

The **commit** command takes a message describing the changes in the revision. It also records your userid, the current time and timezone, and the inventory and contents of the tree. The commit message is specified by the `-m` or `--message` option. You can enter a multi-line commit message; in most shells you can enter this just by leaving the quotes open at the end of the line.

```
% bzr commit -m "added my first file"
```

You can also use the `-F` option to take the message from a file. Some people like to make notes for a commit message while they work, then review the diff to make sure they did what they said they did. (This file can also be useful when you pick up your work after a break.)

8.2 Message from an editor

If you use neither the `-m` nor the `-F` option then bzr will open an editor for you to enter a message. The editor to run is controlled by your `$VISUAL` or `$EDITOR` environment variable, which can be overridden by the `editor` setting in `~/.bazaar/bazaar.conf`; `$BZR_EDITOR` will override either of the above mentioned editor options. If you quit the editor without making any changes, the commit will be cancelled.

The file that is opened in the editor contains a horizontal line. The part of the file below this line is included for information only, and will not form part of the commit message. Below the separator is shown the list of files that are changed in the commit. You should write your message above the line, and then save the file and exit.

If you would like to see the diff that will be committed as you edit the message you can use the `--show-diff` option to `commit`. This will include the diff in the editor when it is opened, below the separator and the information about the files that will be committed. This means that you can read it as you write the message, but the diff itself wont be seen in the commit message when you have finished. If you would like parts to be included in the message you can copy and paste them above the separator.

8.3 Marking bugs as fixed

Many changes to a project are as a result of fixing bugs. Bazaar can keep metadata about bugs you fixed when you commit them. To do this you use the `--fixes` option. This option takes an argument that looks like this:

```
% bzr commit --fixes <tracker>:<id>
```

Where `<tracker>` is an identifier for a bug tracker and `<id>` is an identifier for a bug that is tracked in that bug tracker. `<id>` is usually a number. Bazaar already knows about a few popular bug trackers. They are bugs.launchpad.net, bugs.debian.org, and bugzilla.gnome.org. These trackers have their own identifiers: lp, deb, and gnome respectively. For example, if you made a change to fix the bug #1234 on bugs.launchpad.net, you would use the following command to commit your fix:

```
% bzr commit -m "fixed my first bug" --fixes lp:1234
```

For more information on this topic or for information on how to configure other bug trackers please read Bug Tracker Settings.

8.4 Selective commit

If you give file or directory names on the commit command line then only the changes to those files will be committed. For example:

```
% bzr commit -m "documentation fix" commit.py
```

By default bzr always commits all changes to the tree, even if run from a subdirectory. To commit from only the current directory down, use:

```
% bzr commit .
```

9 Removing uncommitted changes

If you've made some changes and don't want to keep them, use the **revert** command to go back to the previous head version. It's a good idea to use `bzr diff` first to see what will be removed. By default the revert command reverts the whole tree; if file or directory names are given then only those ones will be affected. `bzr revert` also clears the list of pending merges revisions.

10 Ignoring files

10.1 The .bzrignore file

Many source trees contain some files that do not need to be versioned, such as editor backups, object or bytecode files, and built programs. You can simply not add them, but then they'll always crop up as unknown files. You can also tell bzr to ignore these files by adding them to a file called `.bzrignore` at the top of the tree.

This file contains a list of file wildcards (or "globs"), one per line. Typical contents are like this:

```
*.o
*~
*.tmp
*.py[co]
```

If a glob contains a slash, it is matched against the whole path from the top of the tree; otherwise it is matched against only the filename. So the previous example ignores files with extension `.o` in all subdirectories, but this example ignores only `config.h` at the top level and HTML files in `doc/`:

```
./config.h
doc/*.html
```

To get a list of which files are ignored and what pattern they matched, use `bzr ignored`:

```
% bzr ignored
config.h                ./config.h
configure.in~           *~
```

It is OK to have either an ignore pattern match a versioned file, or to add an ignored file. Ignore patterns have no effect on versioned files; they only determine whether unversioned files are reported as unknown or ignored.

The `.bzrignore` file should normally be versioned, so that new copies of the branch see the same patterns:

```
% bzr add .bzrignore
% bzr commit -m "Add ignore patterns"
```

10.2 bzr ignore

As an alternative to editing the `.bzrignore` file, you can use the `bzr ignore` command. The `bzr ignore` command takes filenames and/or patterns as arguments and then adds them to the `.bzrignore` file. If a `.bzrignore` file does not exist the `bzr ignore` command will automatically create one for you, and implicitly add it to be versioned:

```
% bzr ignore tags
% bzr status
added:
  .bzrignore
```

Just like when editing the `.bzrignore` file on your own, you should commit the automatically created `.bzrignore` file:

```
% bzr commit -m "Added tags to ignore file"
```

10.3 Global ignores

There are some ignored files which are not project specific, but more user specific. Things like editor temporary files, or personal temporary files. Rather than add these ignores to every project, bzr supports a global ignore file in `~/.bazaar/ignore` [1]. It has the same syntax as the per-project ignore file.

11 Examining history

11.1 bzr log

The `bzr log` command shows a list of previous revisions. The `bzr log --forward` command does the same in chronological order to get most recent revisions printed at last.

As with `bzr diff`, `bzr log` supports the `-r` argument:

```
% bzr log -r 1000..          # Revision 1000 and everything after it
% bzr log -r ..1000          # Everything up to and including r1000
% bzr log -r 1000..1100      # changes from 1000 to 1100
% bzr log -r 1000            # The changes in only revision 1000
```

12 Branch statistics

The `bzr info` command shows some summary information about the working tree and the branch history.

13 Versioning directories

bzr versions files and directories in a way that can keep track of renames and intelligently merge them:

```
% mkdir src
% echo 'int main() {}' > src/simple.c
% bzr add src
added src
added src/simple.c
% bzr status
added:
  src/
  src/simple.c
```

14 Deleting and removing files

You can delete files or directories by just deleting them from the working directory. This is a bit different to CVS, which requires that you also do `cvs remove`.

`bzr remove` makes the file un-versioned, but may or may not delete the working copy [2]. This is useful when you add the wrong file, or decide that a file should actually not be versioned.

```
% rm -r src
% bzr remove -v hello.txt
?       hello.txt
% bzr status
removed:
  hello.txt
  src/
  src/simple.c
unknown:
  hello.txt
```

If you remove the wrong file by accident, you can use `bzr revert` to restore it.

15 Branching

Often rather than starting your own project, you will want to submit a change to an existing project. To do this, you'll need to get a copy of the existing branch. Because this new copy is potentially a new branch, the command is called **branch**:

```
% bzr branch lp:bzr bzr.dev
% cd bzr.dev
```

This copies down the complete history of this branch, so we can do all operations on it locally: log, annotate, making and merging branches. There will be an option to get only part of the history if you wish.

You can also get a copy of an existing branch by copying its directory, expanding a tarball, or by a remote copy using something like rsync.

16 Following upstream changes

You can stay up-to-date with the parent branch by "pulling" in their changes:

```
% bzr pull
```

After this change, the local directory will be a mirror of the source. This includes the ‘’revision-history” - which is a list of the commits done in this branch, rather than merged from other branches.

This command only works if your local (destination) branch is either an older copy of the parent branch with no new commits of its own, or if the most recent commit in your local branch has been merged into the parent branch.

17 Merging from related branches

If two branches have diverged (both have unique changes) then `bzr merge` is the appropriate command to use. Merge will automatically calculate the changes that exist in the branch you’re merging from that are not in your branch and attempt to apply them in your branch.

```
% bzr merge URL
```

If there is a conflict during a merge, 3 files with the same basename are created. The filename of the common base is appended with ”.BASE”, the filename of the file containing your changes is appended with ”.THIS” and the filename with the changes from the other tree is appended with ”.OTHER”. Using a program such as kdiff3, you can now comfortably merge them into one file. In order to commit you have to rename the merged file (”.THIS”) to the original file name. To complete the conflict resolution you must use the resolve command, which will remove the ”.OTHER” and ”.BASE” files. As long as there exist files with .BASE, .THIS or .OTHER the commit command will report an error.

```
% kdiff3 file.BASE file.OTHER file.THIS
% mv file.THIS file
% bzr resolve file
```

[**TODO**: explain conflict markers within files]

18 Publishing your branch

You don’t need a special server to publish a bzr branch, just a normal web server. Just mirror the files to your server, including the .bzr directory. One can push a branch (or the changes for a branch) by one of the following three methods:

- The best method is to use bzr itself to do it.

  ```
  % bzr push bzr+ssh://servername.com/path/to/directory
  ```

  (The destination directory must already exist unless the `--create-prefix` option is used.)

- Another option is the `rspush` plugin that comes with BzrTools, which uses rsync to push the changes to the revision history and the working tree.
- You can also copy the files around manually, by sending a tarball, or using rsync, or other related file transfer methods. This is usually less safe than using `push`, but may be faster or easier in some situations.

19 Moving changes between trees

It happens to the best of us: sometimes you’ll make changes in the wrong tree. Maybe because you’ve accidentally started work in the wrong directory, maybe because as you’re working, the change turns out to be bigger than you expected, so you start a new branch for it.

To move your changes from one tree to another, use

```
% cd NEWDIR
% bzr merge --uncommitted OLDDIR
```

This will apply all of the uncommitted changes you made in OLDDIR to NEWDIR. It will not apply committed changes, even if they could be applied to NEWDIR with a regular merge. The changes will remain in OLDDIR, but you can use `bzr revert OLDDIR` to remove them, once you're satisfied with NEWDIR.

NEWDIR does not have to be a copy of OLDDIR, but they should be related. The more different they are, the greater the chance of conflicts.