Centralized Workflow Tutorial

Release 2.8.0dev1

Bazaar Developers

August 11, 2021

Contents

1 Overview

This document describes a possible workflow for using Bazaar. That of using Bazaar, the distributed version control system, in a centralized manner. Bazaar is designed to be very flexible and allows several different workflows, from fully decentralized to mostly centralized. The workflow used here is meant to ease a new user into more advanced usage of Bazaar, and allow them to work in a mix of centralized and decentralized operations.

In general, this document is meant for users coming from a background of centralized version control systems such as CVS or subversion. It is common in work settings to have a single central server hosting the codebase, with several people working on this codebase, keeping their work in sync. This workflow is also applicable to a single developer working on several different machines.

2 Initial Setup

These are some reasonably simple steps to setup Bazaar so that it works well for you.

2.1 Setting User Email

Your user identity is stored with each commit. While this doesn't have to be accurate or unique, it will be used in log messages and annotations, so it is better to have something real.

```
% bzr whoami "John Doe <jdoe@organization.com>"
```

2.2 Setting up a local Repository

Bazaar branches generally copy the history information around with them, which is part of how you can work in a fully decentralized manner. As an optimization, it is possible for related branches to combine their storage needs so that you do not need to copy around all of this history information whenever you create a new branch.

The best way to do this is to create a Shared Repository. In general, branches will share their storage if they exist in a subdirectory of a Shared Repository. So let's set up a Shared Repository in our home directory, thus all branches we create underneath will share their history storage.

```
% bzr init-repo --trees ~
```

2.3 Setting up a remote Repository

Many times you want a location where data is stored separately from where you do your work. This workflow is required by centralized systems (CVS/SVN). Usually they are on separate machines, but not always. This is actually a pretty good setup, especially in a work environment. Since it ensures a central location where data can be backed up, and means that if something happens to a developer's machine, no committed work has to be lost.

So let's set up a shared location for our project on a remote machine called `centralhost`. Again, we will use a Shared Repository to optimize disk usage.

```
% bzr init-repo --no-trees bzr+ssh://centralhost/srv/bzr/
```

You can think of this step as similar to setting up a new cvsroot, or subversion repository. The `--no-trees` option tells bzr to not populate the directory with a working tree. This is appropriate, since no one will be making changes directly in the branches within the central repository.

Here we're using a `bzr+ssh` URL, which means to use Bazaar's own protocol on top of the SSH secure shell. See the Administrator Guide for information about setting up a bzr+ssh server.

3 Migrating an existing project to Bazaar

Now that we have a repository, let's create a versioned project. Most of the time, you will already have some code that you are working with, that you now want to version using Bazaar. If the code was originally in source control, there

are many ways to convert the project to Bazaar without losing any history. However, this is outside the scope of this document. See Tracking Upstream for some possibilities (section "Converting and keeping history").

3.1 Developer 1: Creating the first revision

So first, we want to create a branch in our remote Repository, where we want to host the project. Let's assume we have a project named "sigil" that we want to put under version control.

```
% bzr init bzr+ssh://centralhost/srv/bzr/sigil
```

This can be thought of as the "HEAD" branch in CVS terms, or as the "trunk" in Subversion terms. We will call this the `dev` branch.

I prefer working in a subdirectory of my home directory to avoid collisions with all the other files that end up there. Also, we will want a project directory where we can hold all of the different branches we end up working on.

```
% cd ~
% mkdir work
% cd work
% mkdir sigil
% cd sigil
% bzr checkout bzr+ssh://centralhost/srv/bzr/sigil dev
% cd dev
% cp -ar ~/sigil/* .
% bzr add
% bzr commit -m "Initial import of Sigil"
```

In the previous section, we created an empty branch (the `/sigil` branch) on `centralhost`, and then checkout out this empty branch onto our workstation to add files from our existing project. There are many ways to set up your working directory, but the steps above make it easy to handle working with feature/bugfix branches. And one of the strong points of Bazaar is how well it works with branches.

At this point, because you have a 'checkout' of the remote branch, any commits you make in `~/work/sigil/dev/` will automatically be saved both locally, and on `centralhost`.

3.2 Developer N: Getting a working copy of the project

Since the first developer did all of the work of creating the project, all other developers would just checkout that branch. **They should still follow** Setting User Email and Setting up a local Repository.

To get a copy of the current development tree:

```
% cd ~/work/sigil
% bzr checkout bzr+ssh://centralhost/srv/bzr/sigil dev
```

Now that two people both have a checkout of `bzr+ssh://centralhost/srv/bzr/sigil`, there will be times when one of the checkouts will be out of date with the current version. At commit time, Bazaar will inform the user of this and prevent them from committing. To get up to date, use `bzr update` to update the tree with the remote changes. This may require resolving conflicts if the same files have been modified.

4 Developing on separate branches

So far everyone is working and committing their changes into the same branch. This means that everyone needs to update fairly regularly and deal with other people's changes. Also, if one person commits something that breaks the codebase, then upon syncing, everyone will get the problem.

Usually, it is better to do development on different branches, and then integrate those back into the main branch, once they are stable. This is one of the biggest changes from working with CVS/SVN. They both allow you to work on separate branches, but their merging algorithms are fairly weak, so it is difficult to keep things synchronized. Bazaar tracks what has already been merged, and can even apply changes to files that have been renamed.

4.1 Creating and working on a new branch

We want to keep our changes available for other people, even if they aren't quite complete yet. So we will create a new public branch on `centralhost`, and track it locally.

```
% cd ~/work/sigil
% bzr branch bzr+ssh://centralhost/srv/bzr/sigil \
             bzr+ssh://centralhost/srv/bzr/sigil/doodle-fixes
% bzr checkout bzr+ssh://centralhost/srv/bzr/sigil/doodle-fixes doodle-fixes
% cd doodle-fixes
```

We now have a place to make any fixes we need to `doodle`. And we would not interrupt people who are working on other parts of the code. Because we have a checkout, any commits made in the `~/work/sigil/doodle-fixes/` will also show up on `centralhost`. [1] It is also possible to have two developers collaborate on one of these branches, just like they would have collaborated on the `dev` branch. [2]

```
% bzr cbranch dev my-feature-branch
```

4.2 Merging changes back

When it is decided that some of the changes in `doodle-fixes` are ready to be merged into the main branch, simply do:

```
% cd ~/work/sigil/dev
% bzr merge ../doodle-fixes
```

Now the changes are available in the `dev` branch, but they have not been committed yet. This is the time when you want to review the final changes, and double check the code to make sure it compiles cleanly and passes the test suite. The commands `bzr status` and `bzr diff` are good tools to use here. Also, this is the time to resolve any conflicts. Bazaar will prevent you from committing until you have resolved these conflicts. That way you don't accidentally commit the conflict markers. The command `bzr status` will show the conflicts along with the other changes, or you can use `bzr conflicts` to just list conflicts. Use `bzr resolve file/name` or `bzr resolve --all` once conflicts have been handled. [3] If you have a conflict that is particularly difficult to solve you may want to use the `bzr remerge` command. It will let you try different merge algorithms, as well as let you see the original source lines (`--show-base`).

4.3 Recommended Branching

One very common way to handle all of these branches is to give each developer their own branch, and their own place to work in the central location. This can be done with:

[1] It may look odd to have a branch in a subdirectory of another branch. This is just fine, and you can think of it as a hierarchical namespace where the nested branch is derived from the outer branch.

[2] When using lots of independent branches, having to retype the full URL all the time takes a lot of typing. We are looking into various methods to help with this, such as branch aliases, etc. For now, though, the bzrtools plugin provides the `bzr cbranch` command. Which is designed to take a base branch, create a new public branch, and create a checkout of that branch, all with much less typing. Configuring `cbranch` is outside the scope of this document, but the final commands are similar to:

[3] Some systems make you resolve conflicts as part of the merge process. We have found that it is usually easier to resolve conflicts when you have the view of the entire tree, rather than just a single file. It gives you much more context, and also lets you run tests as you resolve the problems.

```
% bzr branch bzr+ssh://centralhost/srv/bzr/sigil \
             bzr+ssh://centralhost/srv/bzr/sigil/user-a
% bzr branch bzr+ssh://centralhost/srv/bzr/sigil \
             bzr+ssh://centralhost/srv/bzr/sigil/user-b
```

This gives each developer their own branch to work on. And, they can easily create a new feature branch for themselves with just [2]

```
% bzr branch bzr+ssh://centralhost/srv/bzr/sigil/user-a \
             bzr+ssh://centralhost/srv/bzr/sigil/user-a/feature
% cd ~/work/sigil
% bzr checkout bzr+ssh://centralhost/srv/bzr/sigil/user-a/feature myfeature
```

5 Glossary

5.1 Shared Repository

Bazaar has the concept of a "Shared Repository". This is similar to the traditional concept of a repository in other VCSs like CVS and Subversion. For example, in Subversion you have a remote repository, which is where all of the history is stored, and locally you don't have any history information, only a checkout of the working tree files. Note that "Shared" in this context means shared between branches. It *may* be shared between people, but standalone branches can also be shared between people.

In Bazaar terms, a "Shared Repository" is a location where multiple branches can **share** their revision history information. In order to support decentralized workflows, it is possible for every branch to store its own revision history information. But this is often inefficient, since related branches share history, and they might as well share the storage as well.