Bazaar Developer Guide

Release 2.8.0dev1

Bazaar Developers

August 11, 2021

CONTENTS

This document describes the Bazaar internals and the development process. It's meant for people interested in developing Bazaar, and some parts will also be useful to people developing Bazaar plugins.

If you have any questions or something seems to be incorrect, unclear or missing, please talk to us in `irc://irc.freenode.net/#bzr`, or write to the Bazaar mailing list. To propose a correction or addition to this document, send a merge request or new text to the mailing list.

The latest developer documentation can be found online at http://doc.bazaar.canonical.com/developers/.

CHAPTER

ONE

GETTING STARTED

1.1 Exploring the Bazaar Platform

Before making changes, it's a good idea to explore the work already done by others. Perhaps the new feature or improvement you're looking for is available in another plug-in already? If you find a bug, perhaps someone else has already fixed it?

To answer these questions and more, take a moment to explore the overall Bazaar Platform. Here are some links to browse:

- The Plugins page on the Wiki - http://wiki.bazaar.canonical.com/BzrPlugins
- The Bazaar product family on Launchpad - https://launchpad.net/bazaar
- Bug Tracker for the core product - https://bugs.launchpad.net/bzr/

If nothing else, perhaps you'll find inspiration in how other developers have solved their challenges.

1.2 Finding Something To Do

Ad-hoc performance work can also be done. One useful tool is the 'evil' debug flag. For instance running `bzr -Devil commit -m "test"` will log a backtrace to the bzr log file for every method call which triggers a slow or non-scalable part of the bzr library. So checking that a given command with `-Devil` has no backtraces logged to the log file is a good way to find problem function calls that might be nested deep in the code base.

1.3 Planning and Discussing Changes

There is a very active community around Bazaar. Mostly we meet on IRC (#bzr on irc.freenode.net) and on the mailing list. To join the Bazaar community, see http://wiki.bazaar.canonical.com/BzrSupport.

If you are planning to make a change, it's a very good idea to mention it on the IRC channel and/or on the mailing list. There are many advantages to involving the community before you spend much time on a change. These include:

- you get to build on the wisdom of others, saving time
- if others can direct you to similar code, it minimises the work to be done
- it assists everyone in coordinating direction, priorities and effort.

In summary, maximising the input from others typically minimises the total effort required to get your changes merged. The community is friendly, helpful and always keen to welcome newcomers.

1.4 Bazaar Development in a Nutshell

One of the fun things about working on a version control system like Bazaar is that the users have a high level of proficiency in contributing back into the tool. Consider the following very brief introduction to contributing back to Bazaar. More detailed instructions are in the following sections.

1.4.1 Making the change

First, get a local copy of the development mainline (See Why make a local copy of bzr.dev?.)

```
$ bzr init-repo ~/bzr
$ cd ~/bzr
$ bzr branch lp:bzr bzr.dev
```

Now make your own branch:

```
$ bzr branch bzr.dev 123456-my-bugfix
```

This will give you a branch called "123456-my-bugfix" that you can work on and commit in. Here, you can study the code, make a fix or a new feature. Feel free to commit early and often (after all, it's your branch!).

Documentation improvements are an easy place to get started giving back to the Bazaar project. The documentation is in the *doc/* subdirectory of the Bazaar source tree.

When you are done, make sure that you commit your last set of changes as well! Once you are happy with your changes, ask for them to be merged, as described below.

1.4.2 Making a Merge Proposal

The Bazaar developers use Launchpad to further enable a truly distributed style of development. Anyone can propose a branch for merging into the Bazaar trunk. To start this process, you need to push your branch to Launchpad. To do this, you will need a Launchpad account and user name, e.g. *your_lp_username*. You can push your branch to Launchpad directly from Bazaar:

```
$ bzr push lp:~<your_lp_username>/bzr/meaningful_name_here
```

After you have pushed your branch, you will need to propose it for merging to the Bazaar trunk. Go to <https://launchpad.net/~<your_lp_username>/bzr/meaningful_name_here> and choose "Propose for merging into another branch". Select "lp:bzr" to hand your changes off to the Bazaar developers for review and merging.

Alternatively, after pushing you can use the `lp-propose` command to create the merge proposal.

Using a meaningful name for your branch will help you and the reviewer(s) better track the submission. Use a very succint description of your submission and prefix it with bug number if needed (lp:~mbp/bzr/484558-merge-directory for example). Alternatively, you can suffix with the bug number (lp:~jameinel/bzr/export-file-511987).

1.4.3 Review cover letters

Please put a "cover letter" on your merge request explaining:

- the reason **why** you're making this change
- **how** this change achieves this purpose
- anything else you may have fixed in passing

- anything significant that you thought of doing, such as a more extensive fix or a different approach, but didn't or couldn't do now

A good cover letter makes reviewers' lives easier because they can decide from the letter whether they agree with the purpose and approach, and then assess whether the patch actually does what the cover letter says. Explaining any "drive-by fixes" or roads not taken may also avoid queries from the reviewer. All in all this should give faster and better reviews. Sometimes writing the cover letter helps the submitter realize something else they need to do. The size of the cover letter should be proportional to the size and complexity of the patch.

1.4.4 Why make a local copy of bzr.dev?

Making a local mirror of bzr.dev is not strictly necessary, but it means

- You can use that copy of bzr.dev as your main bzr executable, and keep it up-to-date using `bzr pull`.
- Certain operations are faster, and can be done when offline. For example:
  - `bzr bundle`
  - `bzr diff -r ancestor:...`
  - `bzr merge`
- When it's time to create your next branch, it's more convenient. When you have further contributions to make, you should do them in their own branch:

```
$ cd ~/bzr
$ bzr branch bzr.dev additional_fixes
$ cd additional_fixes # hack, hack, hack
```

1.5 Understanding the Development Process

The development team follows many practices including:

- a public roadmap and planning process in which anyone can participate
- time based milestones everyone can work towards and plan around
- extensive code review and feedback to contributors
- complete and rigorous test coverage on any code contributed
- automated validation that all tests still pass before code is merged into the main code branch.

The key tools we use to enable these practices are:

- Launchpad - https://launchpad.net/
- Bazaar - http://bazaar.canonical.com/
- Patch Queue Manager - https://launchpad.net/pqm/

For further information, see <http://wiki.bazaar.canonical.com/BzrDevelopment>.

1.6 Preparing a Sandbox for Making Changes to Bazaar

Bazaar supports many ways of organising your work. See http://wiki.bazaar.canonical.com/SharedRepositoryLayouts for a summary of the popular alternatives.

Of course, the best choice for you will depend on numerous factors: the number of changes you may be making, the complexity of the changes, etc. As a starting suggestion though:

- create a local copy of the main development branch (bzr.dev) by using this command:

  ```
  bzr branch lp:bzr bzr.dev
  ```

- keep your copy of bzr.dev pristine (by not developing in it) and keep it up to date (by using bzr pull)
- create a new branch off your local bzr.dev copy for each issue (bug or feature) you are working on.

This approach makes it easy to go back and make any required changes after a code review. Resubmitting the change is then simple with no risk of accidentally including edits related to other issues you may be working on. After the changes for an issue are accepted and merged, the associated branch can be deleted or archived as you wish.

1.7 Navigating the Code Base

Some of the key files in this directory are:

bzr The command you run to start Bazaar itself. This script is pretty short and just does some checks then jumps into bzrlib.

README This file covers a brief introduction to Bazaar and lists some of its key features.

setup.py Installs Bazaar system-wide or to your home directory. To perform development work on Bazaar it is not required to run this file - you can simply run the bzr command from the top level directory of your development copy. Note: That if you run setup.py this will create a 'build' directory in your development branch. There's nothing wrong with this but don't be confused by it. The build process puts a copy of the main code base into this build directory, along with some other files. You don't need to go in here for anything discussed in this guide.

bzrlib Possibly the most exciting folder of all, bzrlib holds the main code base. This is where you will go to edit python files and contribute to Bazaar.

doc Holds documentation on a whole range of things on Bazaar from the origination of ideas within the project to information on Bazaar features and use cases. Within this directory there is a subdirectory for each translation into a human language. All the documentation is in the ReStructuredText markup language.

doc/developers Documentation specifically targeted at Bazaar and plugin developers. (Including this document.)

doc/en/release-notes/

Detailed changes in each Bazaar release (there is one file by series: bzr-2.3.txt, bzr-2.4.txt, etc) that can affect users or plugin developers.

doc/en/whats-new/

High-level summaries of changes in each Bazaar release (there is one file by series: whats-new-in-2.3.txt, whats-new-in-2.4.txt, etc).

Automatically-generated API reference information is available at <http://people.canonical.com/~mwh/bzrlibapi/>.

See also the Bazaar Architectural Overview.

CHAPTER

TWO

CORE TOPICS

2.1 Evolving Interfaces

We don't change APIs in stable branches: any supported symbol in a stable release of bzr must not be altered in any way that would result in breaking existing code that uses it. That means that method names, parameter ordering, parameter names, variable and attribute names etc must not be changed without leaving a 'deprecated forwarder' behind. This even applies to modules and classes.

If you wish to change the behaviour of a supported API in an incompatible way, you need to change its name as well. For instance, if I add an optional keyword parameter to branch.commit - that's fine. On the other hand, if I add a keyword parameter to branch.commit which is a *required* transaction object, I should rename the API - i.e. to 'branch.commit_transaction'.

> (Actually, that may break code that provides a new implementation of `commit` and doesn't expect to receive the parameter.)

When renaming such supported API's, be sure to leave a deprecated_method (or _function or ...) behind which forwards to the new API. See the bzrlib.symbol_versioning module for decorators that take care of the details for you - such as updating the docstring, and issuing a warning when the old API is used.

For unsupported API's, it does not hurt to follow this discipline, but it's not required. Minimally though, please try to rename things so that callers will at least get an AttributeError rather than weird results.

2.1.1 Deprecation decorators

`bzrlib.symbol_versioning` provides decorators that can be attached to methods, functions, and other interfaces to indicate that they should no longer be used. For example:

```
@deprecated_method(deprecated_in((0, 1, 4)))
def foo(self):
     return self._new_foo()
```

To deprecate a static method you must call `deprecated_function` (**not** method), after the staticmethod call:

```
@staticmethod
@deprecated_function(deprecated_in((0, 1, 4)))
def create_repository(base, shared=False, format=None):
```

When you deprecate an API, you should not just delete its tests, because then we might introduce bugs in them. If the API is still present at all, it should still work. The basic approach is to use `TestCase.applyDeprecated` which in one step checks that the API gives the expected deprecation message, and also returns the real result from the method, so that tests can keep running.

Deprecation warnings will be suppressed for final releases, but not for development versions or release candidates, or when running `bzr selftest`. This gives developers information about whether their code is using deprecated functions, but avoids confusing users about things they can't fix.

2.2 General Guidelines

2.2.1 Copyright

The copyright policy for bzr was recently made clear in this email (edited for grammatical correctness):

```
The attached patch cleans up the copyright and license statements in
the bzr source. It also adds tests to help us remember to add them
with the correct text.

We had the problem that lots of our files were "Copyright Canonical
Development Ltd" which is not a real company, and some other variations
on this theme. Also, some files were missing the GPL statements.

I want to be clear about the intent of this patch, since copyright can
be a little controversial.

1) The big motivation for this is not to shut out the community, but
just to clean up all of the invalid copyright statements.

2) It has been the general policy for bzr that we want a single
copyright holder for all of the core code. This is following the model
set by the FSF, which makes it easier to update the code to a new
license in case problems are encountered. (For example, if we want to
upgrade the project universally to GPL v3 it is much simpler if there is
a single copyright holder). It also makes it clearer if copyright is
ever debated, there is a single holder, which makes it easier to defend
in court, etc. (I think the FSF position is that if you assign them
copyright, they can defend it in court rather than you needing to, and
I'm sure Canonical would do the same).
As such, Canonical has requested copyright assignments from all of the
major contributers.

3) If someone wants to add code and not attribute it to Canonical, there
is a specific list of files that are excluded from this check. And the
test failure indicates where that is, and how to update it.

4) If anyone feels that I changed a copyright statement incorrectly, just
let me know, and I'll be happy to correct it. Whenever you have large
mechanical changes like this, it is possible to make some mistakes.

Just to reiterate, this is a community project, and it is meant to stay
that way. Core bzr code is copyright Canonical for legal reasons, and
the tests are just there to help us maintain that.
```

CHAPTER

THREE

MISCELLANEOUS TOPICS

3.1 Debugging

Bazaar has a few facilities to help debug problems by going into pdb, the Python debugger.

If the `BZR_PDB` environment variable is set then bzr will go into pdb post-mortem mode when an unhandled exception occurs.

If you send a SIGQUIT or SIGBREAK signal to bzr then it will drop into the debugger immediately. SIGQUIT can be generated by pressing Ctrl-\ on Unix. SIGBREAK is generated with Ctrl-Pause on Windows (some laptops have this as Fn-Pause). You can continue execution by typing `c`. This can be disabled if necessary by setting the environment variable `BZR_SIGQUIT_PDB=0`.

All tests inheriting from bzrlib.tests.TestCase can use `self.debug()` instead of the longer `import pdb; pdb.set_trace()`. The former also works when `stdin/stdout` are redirected (by using the original `stdin/stdout` file handles at the start of the `bzr` script) while the later doesn't. `bzrlib.debug.set_trace()` also uses the original `stdin/stdout` file handles.

3.2 Debug Flags

Bazaar accepts some global options starting with `-D` such as `-Dhpss`. These set a value in *bzrlib.debug.debug_flags*, and typically cause more information to be written to the trace file. Most *mutter* calls should be guarded by a check of those flags so that we don't write out too much information if it's not needed.

Debug flags may have effects other than just emitting trace messages.

Run `bzr help global-options` to see them all.

These flags may also be set as a comma-separated list in the `debug_flags` option in e.g. `~/.bazaar/bazaar.conf`. (Note that it must be in this global file, not in the branch or location configuration, because it's currently only loaded at startup time.) For instance you may want to always record hpss traces and to see full error tracebacks:

```
debug_flags = hpss, error
```

3.3 Jargon

revno Integer identifier for a revision on the main line of a branch. Revision 0 is always the null revision; others are 1-based indexes into the branch's revision history.

3.4 Unicode and Encoding Support

This section discusses various techniques that Bazaar uses to handle characters that are outside the ASCII set.

3.4.1 `Command.outf`

When a `Command` object is created, it is given a member variable accessible by `self.outf`. This is a file-like object, which is bound to `sys.stdout`, and should be used to write information to the screen, rather than directly writing to `sys.stdout` or calling `print`. This file has the ability to translate Unicode objects into the correct representation, based on the console encoding. Also, the class attribute `encoding_type` will effect how unprintable characters will be handled. This parameter can take one of 3 values:

replace Unprintable characters will be represented with a suitable replacement marker (typically '?'), and no exception will be raised. This is for any command which generates text for the user to review, rather than for automated processing. For example: `bzr log` should not fail if one of the entries has text that cannot be displayed.

strict Attempting to print an unprintable character will cause a UnicodeError. This is for commands that are intended more as scripting support, rather than plain user review. For example: `bzr ls` is designed to be used with shell scripting. One use would be `bzr ls --null --unknowns | xargs -0 rm`. If `bzr` printed a filename with a '?', the wrong file could be deleted. (At the very least, the correct file would not be deleted). An error is used to indicate that the requested action could not be performed.

exact Do not attempt to automatically convert Unicode strings. This is used for commands that must handle conversion themselves. For example: `bzr diff` needs to translate Unicode paths, but should not change the exact text of the contents of the files.

3.4.2 `bzrlib.urlutils.unescape_for_display`

Because Transports work in URLs (as defined earlier), printing the raw URL to the user is usually less than optimal. Characters outside the standard set are printed as escapes, rather than the real character, and local paths would be printed as `file://` URLs. The function `unescape_for_display` attempts to unescape a URL, such that anything that cannot be printed in the current encoding stays an escaped URL, but valid characters are generated where possible.

3.5 C Extension Modules

We write some extensions in C using pyrex. We design these to work in three scenarios:

- User with no C compiler
- User with C compiler
- Developers

The recommended way to install bzr is to have a C compiler so that the extensions can be built, but if no C compiler is present, the pure python versions we supply will work, though more slowly.

For developers we recommend that pyrex be installed, so that the C extensions can be changed if needed.

For the C extensions, the extension module should always match the original python one in all respects (modulo speed). This should be maintained over time.

To create an extension, add rules to setup.py for building it with pyrex, and with distutils. Now start with an empty .pyx file. At the top add "include 'yourmodule.py"'. This will import the contents of foo.py into this file at build time - remember that only one module will be loaded at runtime. Now you can subclass classes, or replace functions, and only your changes need to be present in the .pyx file.

Note that pyrex does not support all 2.4 programming idioms, so some syntax changes may be required. I.e.

- 'from foo import (bar, gam)' needs to change to not use the brackets.
- 'import foo.bar as bar' needs to be 'import foo.bar; bar = foo.bar'

If the changes are too dramatic, consider maintaining the python code twice - once in the .pyx, and once in the .py, and no longer including the .py file.

3.6 Making Installers for OS Windows

To build a win32 installer, see the instructions on the wiki page: http://wiki.bazaar.canonical.com/BzrWin32Installer

CHAPTER

FOUR

CORE DEVELOPER TASKS

4.1 Overview

4.1.1 What is a Core Developer?

While everyone in the Bazaar community is welcome and encouraged to propose and submit changes, a smaller team is reponsible for pulling those changes together into a cohesive whole. In addition to the general developer stuff covered above, "core" developers have responsibility for:

- reviewing changes
- planning releases
- managing releases (see Releasing Bazaar)

Note: Removing barriers to community participation is a key reason for adopting distributed VCS technology. While DVCS removes many technical barriers, a small number of social barriers are often necessary instead. By documenting how the above things are done, we hope to encourage more people to participate in these activities, keeping the differences between core and non-core contributors to a minimum.

4.1.2 Communicating and Coordinating

While it has many advantages, one of the challenges of distributed development is keeping everyone else aware of what you're working on. There are numerous ways to do this:

1. Assign bugs to yourself in Launchpad
2. Mention it on the mailing list
3. Mention it on IRC

As well as the email notifcations that occur when merge requests are sent and reviewed, you can keep others informed of where you're spending your energy by emailing the **bazaar-commits** list implicitly. To do this, install and configure the Email plugin. One way to do this is add these configuration settings to your central configuration file (e.g. `~/.bazaar/bazaar.conf`):

```
[DEFAULT]
email = Joe Smith <joe.smith@internode.on.net>
smtp_server = mail.internode.on.net:25
```

Then add these lines for the relevant branches in `locations.conf`:

```
post_commit_to = bazaar-commits@lists.canonical.com
post_commit_mailer = smtplib
```

While attending a sprint, RobertCollins' Dbus plugin is useful for the same reason. See the documentation within the plugin for information on how to set it up and configure it.

4.2 Planning Releases

4.2.1 Bug Triage

Keeping on top of bugs reported is an important part of ongoing release planning. Everyone in the community is welcome and encouraged to raise bugs, confirm bugs raised by others, and nominate a priority. Practically though, a good percentage of bug triage is often done by the core developers, partially because of their depth of product knowledge.

With respect to bug triage, core developers are encouraged to play an active role with particular attention to the following tasks:

- keeping the number of unconfirmed bugs low
- ensuring the priorities are generally right (everything as critical - or medium - is meaningless)
- looking out for regressions and turning those around sooner rather than later.

Note: As well as prioritizing bugs and nominating them against a target milestone, Launchpad lets core developers offer to mentor others in fixing them.
