# Bazaar Upgrade Guide

***Release 2.8.0dev1***

**Bazaar Developers**

August 11, 2021

# CONTENTS

CHAPTER

ONE

# OVERVIEW

## 1.1 High level upgrade process

In broad terms, there are 3 steps involved in upgrading a new format:

1. Upgrade the core software
2. Upgrade required plugins
3. Migrate data to the new default format.

Bazaar supports branches in earlier formats so the third step is strictly not required. However, when new default formats are introduced, they are more space efficient, faster on large projects and/or provide new features. So it is recommended that most projects migrate to it at a convenient time.

For most users, upgrading and migrating to the new format is straight forward. For projects with a large community of developers though, things become more complex. In these cases, careful planning and good communications become essential. This document provides general advice which aims to assist in this regard. If in doubt, please contact us on our mailing list or IRC channel with any questions or concerns you have.

## 1.2 Upgrading the core software

The steps required to upgrade the core software vary from operating system to operating system. A brief outline of the steps is given below.

To upgrade Bazaar on Ubuntu:

1. Ensure your package manager is configured with the required software sources, e.g. the official stable release PPA for Ubuntu: https://launchpad.net/~bzr/+archive
2. Use your package manager to upgrade to the latest version.

To upgrade Bazaar on Windows:

1. Uninstall the existing version using Add/Remove Programs.
2. Install the new version using the relevant installer.

To upgrade Bazaar on OS X (via the installer):

1. Install the new version using the relevant installer.

To upgrade Bazaar on OS X (via MacPorts):

1. Refresh the package metadata using **sudo port selfupdate**

2. Upgrade to the latest version using **sudo port upgrade bzr**

For further information on installing and upgrading, see http://wiki.bazaar.canonical.com/Download.

## 1.3 Upgrading required plugins

Many plugins are not dependent on a particular Bazaar version so upgrading them is optional. Other plugins, notably bzrtools and bzr-svn, are more tightly associated with Bazaar's APIs so these typically need to be upgraded in lockstep with the core software.

For Windows and OS X users, bzrtools and bzr-svn are typically included in the installer so no special steps are required to upgrade these. For Ubuntu and other GNU/Linux or Unix systems users, bztrools, bzr-svn and many other popular plugins can be installed and upgraded using your platform's package manager, e.g. Synaptic on Ubuntu.

## 1.4 Migrating data to the new default format

As mentioned earlier, the complexity of migrating to a new format depends on several factors, particularly project community size. It also depends on how data is currently stored, e.g. in a standalone branch, multiple branches in a shared repository, stacked branches on Launchpad, etc. These various scenarios are covered in the next chapter.

CHAPTER

TWO

# DATA MIGRATION

## 2.1 Preparing for data migration

Before starting a migration, there are a few important things to do first:

1. Take a complete backup.
2. Take some time to purge obsolete branches.

A complete backup gives you a safety net in case anything goes wrong.

Purging obsolete branches reduces the amount of data that needs to be migrated. See Finding obsolete branches later for some tips on doing this.

## 2.2 Introducing the upgrade-related commands

There are 3 important commands to be aware of when migrating data.

- **check** - check a repository, branch or tree for data integrity errors
- **reconcile** - fix data integrity errors
- **upgrade** - migrate data to a different format.

**reconcile** is rarely needed but it's good practice to run **check** before and after running **upgrade**.

For detailed help on these commands, see the Bazaar User Reference.

## 2.3 Communicating with your community

To enable a smooth transition to the new format, you should:

1. Make one person responsible for migrating the trunk.
2. Test the migration of trunk works successfully.
3. Schedule a time for the trunk migration and notify your community in advance.

This advance warning should be long enough for users to have time to upgrade Bazaar and any required plugins before the migration date.

For larger projects, allow some time for the migration itself. You should have a good idea of how long the migration will take after doing the test migration. It may make sense to do the migration on a weekend or a Friday, giving yourself some breathing space if things go wrong.

After the trunk is migrated, you'll need to notify your community accordingly, giving them instructions as to how to migrate their local branches. Sample instructions are provided later in this document.

## 2.4 Migrating a standalone branch

The steps are:

1. Run **bzr check**.
2. If there are errors, try using **bzr reconcile** to fix them. If that fails, file a bug so we can help you resolve the issue and get your trunk clean. If it works, take a backup copy of your now clean trunk.
2. Run **bzr upgrade –format** where *format* is 2a or later.
3. Run **bzr check** to confirm the final result is good.

## 2.5 Migrating branches in a shared repository

Upgrade things in the following order:

1. Upgrade the shared repository.
2. Upgrade the branches.
3. Upgrade any lightweight checkouts.

As in the standalone branch case, be sure to run **check** before and after the upgrade to check for any existing or introduced issues.

## 2.6 Migrating branches on Launchpad

You have two options for upgrading your Launchpad branches. You can either upgrade them remotely or you can upgrade them locally and push the migrated branch to Launchpad. We recommend the latter. Upgrading remotely currently requires a fast, rock solid network connection to the Launchpad servers, and any interruption in that connection can leave you with a partially upgraded branch. The instructions below are the safest and often fastest way to upgrade your Launchpad branches.

To allow isolation between public and private branches, Launchpad uses stacked branches rather than shared repositories as the core technology for efficient branch storage. The process for migrating to a new format for projects using Launchpad code hosting is therefore different to migrating a personal or in-house project.

In Launchpad, a project can define a *development series* and associate a branch with that series. The branch then becomes the *focus of development* and gets special treatment and a shortcut URL. By default, if anybody branches your project's focus of development and pushes changes back to Launchpad, their branch will be stacked on your development focus branch. Also, branches can be associated with other Launchpad artifacts such as bugs and merge proposals. All of these things mean that upgrading your focus of development branch is trickier.

Here are the steps to follow:

1. The nominated person grabs a copy of trunk and does the migration locally.
2. On Launchpad, unset the current trunk from being the development focus. (This *must* be done or the following step won't work as expected.)
   (a) Go to your project's home page on Launchpad

(b) Look for "XXX is the current focus of development"

(c) Click on the edit (pencil) icon

(d) Click on "Change details" in the portlet on the right

(e) Scroll down to where it says "Branch: (Optional)"

(f) Blank out this input field and click "Change"

3. Push the migrated trunk to Launchpad. See below if you want your new migrated development focus branch to have the same name as your old pre-migration development focus branch.
4. Set it as the development focus. Follow the instructions above but at step 5, enter the name of the newly migrated branch you just pushed.
5. Ask users subscribed to the old trunk to subscribe to the new one.

In summary, these steps mean that the old trunk is still available and existing branches stacked on it will continue to be so. However, the development focus has switched to the migrated trunk and any new branches pushed to Launchpad for your project will now stack on it.

You are now ready to tell your community that the new trunk is available and to give them instructions on migrating any local branches they have.

If you want your new migrated development focus branch to have the same name as your old pre-migration branch, you need to do a few extra things before you establish the new development focus.

1. Rename your old pre-migration branch; use something like **foo-obsolete-do-not-use**. You will really not want to delete this because there will be artifacts (bugs, merge proposals, etc.) associated with it.
2. Rename the new migrated branch to the pre-migration branch's old name.
3. Re-establish the development focus branch using the new migrated branch's new name (i.e. the old pre-migration branch's original name).

## 2.7 Migrating local branches after a central trunk has migrated

To migrate a standalone branch:

1. Grab the latest branch from the central location into a new directory.
2. Pull or merge any changes you've made in your existing branch into the new branch.

To migrate branches in a shared repository:

1. Create a fresh shared repository in the new format (2a or later).
2. Grab the latest branch from the central location into a new directory inside the shared repository.
3. Decide which of your local branches you want to migrate. (If you haven't already, now's a good time for Finding obsolete branches and purging them, after backing up first of course.)
4. To migrate each local branch of interest, there are 2 options:

- **init** an empty branch in the new repository and **pull** the revisions from the branch in the old repository across.
- In the new repository, **branch** from trunk to the new branch name then **merge** your changes from the matching branch in the old repository.

The first method will give you a branch which is identical (in terms of revision history) to the old branch, but it's parent branch will be set to the old branch, not your new trunk. If you use this method, you'll probably update the `parent_location` configuration variable in the `branch.conf` file with:

```
bzr config parent_location=XXX
```

XXX being the URL to your new trunk.

In contrast, the second approach sets up the parent branch correctly. However, it isn't ideal if you're not ready to include all the latest revisions from trunk into that branch yet.

CHAPTER

THREE

# TIPS AND TRICKS

## 3.1 Finding obsolete branches

If you use feature branching for developing each fix and enhancement separately, you may have several old branches that are no longer required. In many cases, the relevant changes may now be merged into trunk. In other cases, a branch may be obsolete thanks to another change made by yourself or others.

When checking for an obsolete branch, there are three things in particular to confirm:

1. The working tree has no in-progress changes.
2. The working tree has no shelved changes.
3. Any locally committed revisions have been merged into the parent branch.

After changing into the root of a branch, the commands to check these things respectively are:

```
bzr status
bzr shelve --list
bzr missing --mine-only
```

If your branches are stored in a shared repository locally, you may find the *Local Changes* tab in Bazaar Explorer's repository view helpful here (revision 159 or later) as it shows a summary of these things, excluding the shelve information currently, for each branch as you select it.

### 3.1.1 Licence