Bazaar User Guide

Release 2.8.0dev1

Bazaar Developers

March 22, 2019

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Introducing Bazaar

1.1.1 What is Bazaar?

Bazaar is a tool for helping people collaborate. It tracks the changes that you and other people make to a group of files - such as software source code - to give you snapshots of each stage of their evolution. Using that information, Bazaar can effortlessly merge your work with other people's.

Tools like Bazaar are called version control systems (VCS) and have long been popular with software developers. Bazaar's ease of use, flexibility and simple setup make it ideal not only for software developers but also for other groups who work together on files and documents, such as technical writers, web designers and translators.

This guide takes you through installing Bazaar and how to use it, whether on your own or with a team of other people. If you're already familiar with distributed version control and want to dive straight in, you may wish to skim this section and jump straight to Learning more.

1.1.2 A brief history of version control systems

Version control tools have been evolving for several decades now. In simple terms, there have been 4 generations of tools:

1. file versioning tools, e.g. SCCS, RCS
2. tree versioning tools - central style, e.g. CVS
3. tree versioning tools - central style, take two, e.g. Subversion
4. tree versioning tools - distributed style, e.g. Bazaar.

The design and implementation of Bazaar builds on the lessons learned from all the previous generations of tools. In particular, Bazaar cleanly supports both the central and the distributed version control models so you can change models as it makes sense, without needing to change tools.

1.1.3 Central vs distributed VCS

Many traditional VCS tools require a central server which provides the change history or *repository* for a tree of files. To work on the files, users need to connect to the server and *checkout* the files. This gives them a directory or *working tree* in which a person can make changes. To record or *commit* these changes, the user needs access to the central server and they need to ensure they have merged their work with the latest version stored before trying to commit. This approach is known as the centralized model.

The centralized model has proven useful over time but it can have some notable drawbacks. Firstly, a centralized VCS requires that one is able to connect to the server whenever one wants to do version control work. Secondly, the centralized model tightly links the act of **snapshotting** changes with the act of **publishing** those changes. This can be good in some circumstances but it has a negative influence on quality in others.

Distributed VCS tools let users and teams have multiple repositories rather than just a single central one. In Bazaar's case, the history is normally kept in the same place as the code that is being version controlled. This allows the user to commit their changes whenever it makes sense, even when offline. Network access is only required when publishing changes or when accessing changes in another location.

In fact, using distributed VCS tools wisely can have advantages well beyond the obvious one of disconnected operations for developers. Other advantages include:

- easier for developers to create experimental branches
- easier ad-hoc collaboration with peers
- less time required on mechanical tasks - more time for creativity
- increased release management flexibility through the use of "feature-wide" commits
- trunk quality and stability can be kept higher, making everyone's job less stressful
- in open communities:
  - easier for non-core developers to create and maintain changes
  - easier for core developers to work with non-core developers and move them into the core
- in companies, easier to work with distributed and outsourced teams.

For a detailed look at the advantages of distributed VCS tools over centralized VCS tools, see http://wiki.bazaar.canonical.com/BzrWhy.

1.1.4 Key features of Bazaar

While Bazaar is not the only distributed VCS tool around, it does have some notable features that make it an excellent choice for many teams and communities. A summary of these and comparisons with other VCS tools can be found on the Bazaar Wiki, http://wiki.bazaar.canonical.com.

Of the many features, one in particular is worth highlighting: Bazaar is completely free software written in Python. As a result, it is easy to contribute improvements. If you wish to get involved, please see http://wiki.bazaar.canonical.com/BzrSupport.

1.1.5 Learning more

This manual provides an easy to read introduction to Bazaar and how to use it effectively. It is recommended that all users read at least the rest of this chapter as it:

- explains the core concepts users need to know
- introduces some popular ways of using Bazaar to collaborate.

Chapters 2-6 provide a closer look at how to use Bazaar to complete various tasks. It is recommended that most users read these in first-to-last order shortly after starting to use Bazaar. Chapter 7 and beyond provide additional information that helps you make the most of Bazaar once the core functionality is understood. This material can be read when required and in any order.

If you are already familiar with other version control tools, you may wish to get started quickly by reading the following documents:

- Bazaar in five minutes - a mini-tutorial
- Bazaar Quick Start Card - a one page summary of commonly used commands.

In addition, the online help and Bazaar User Reference provide all the details on the commands and options available.

We hope you find this manual useful. If you have suggestions on how it or the rest of Bazaar's documentation can be improved, please contact us on the mailing list, bazaar@lists.canonical.com.

1.2 Core concepts

1.2.1 A simple user model

To use Bazaar you need to understand four core concepts:

- **Revision** - a snapshot of the files you're working with.
- **Working tree** - the directory containing your version-controlled files and sub-directories.
- **Branch** - an ordered set of revisions that describe the history of a set of files.
- **Repository** - a store of revisions.

Let's look at each in more detail.

1.2.2 Revision

A revision is a *snapshot* of the state of a tree of files and directories, including their content and shape. A revision also has some metadata associated with it, including:

- Who committed it
- When it was committed
- A commit message
- Parent revisions from which it was derived

Revisions are immutable and can be globally, uniquely identified by a *revision-id*. An example revision-id is:

```
pqm@pqm.ubuntu.com-20071129184101-u9506rihe4zbzyyz
```

Revision-ids are generated at commit time or, for imports from other systems, at the time of import. While revision-ids are necessary for internal use and external tool integration, branch-specific *revision numbers* are the preferred interface for humans.

Revision numbers are dotted decimal identifiers like 1, 42 and 2977.1.59 that trace a path through the revision number graph for a branch. Revision numbers are generally shorter than revision-ids and, within a single branch, can be compared with each other to get a sense of their relationship. For example, revision 10 is the mainline (see below) revision immediately after revision 9. Revision numbers are generated on the fly when commands are executing, because they depend on which revision is the tip (i.e. most recent revision) in the branch.

See Specifying revisions in the appendices for a closer look at the numerous ways that revisions and ranges of revisions can be specified in Bazaar, and Understanding Revision Numbers for a more detailed description of revision numbering.

1.2.3 Working Tree

A working tree is a *version-controlled directory* holding files the user can edit. A working tree is associated with a *branch*.

Many commands use the working tree as their context, e.g. `commit` makes a new revision using the current content of files in the working tree.

1.2.4 Branch

In the simplest case, a branch is an *ordered series of revisions*. The last revision is known as the *tip*.

Branches may split apart and be *merged* back together, forming a *graph* of revisions. Technically, the graph shows directed relationships (between parent and child revisions) and there are no loops, so you may hear some people refer to it as a *directed acyclic graph* or DAG.

If this name sounds scary, don't worry. The important things to remember are:

- The primary line of development within the DAG is called the *mainline*, *trunk*, or simply the *left hand side* (LHS).
- A branch might have other lines of development and if it does, these other lines of development begin at some point and end at another point.

1.2.5 Repository

A repository is simply a *store of revisions*. In the simplest case, each branch has its own repository. In other cases, it makes sense for branches to share a repository in order to optimize disk usage.

1.2.6 Putting the concepts together

Once you have grasped the concepts above, the various ways of using Bazaar should become easier to understand. The simplest way of using Bazaar is to use a *standalone tree*, which has a working tree, branch, and repository all in a single location. Other common scenarios include:

- Shared repositories - working tree and branch are colocated, but the repository is in a higher level directory.
- Stacked branches - branch stores just its unique revisions, using its parent's repository for common revisions.
- Lightweight checkouts - branch is stored in a different location to the working tree.

The best way to use Bazaar, however, depends on your needs. Let's take a look at some common workflows next.

1.3 Workflows

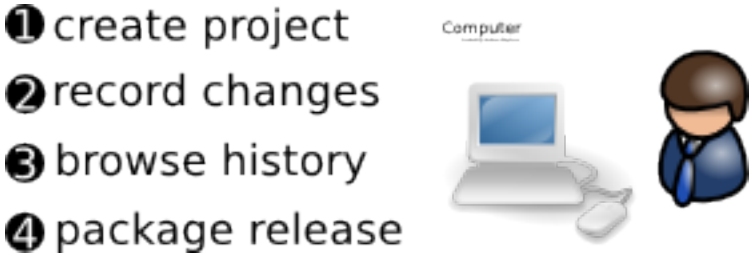
1.3.1 Bazaar is just a tool

Bazaar supports many different ways of working together. This means that you can start with one workflow and adapt it over time as circumstances change. There is no "one true way" that always makes sense and there never will be. This section provides a brief overview of some popular workflows supported by Bazaar.

Keep in mind that these workflow are just *some* examples of how Bazaar can be used. You may want to use a workflow not listed here, perhaps building on the ideas below.

1.3.2 Solo

Whether developing software, editing documents or changing configuration files, having an easy-to-use VCS tool can help. A single user can use this workflow effectively for managing projects where they are the only contributor.



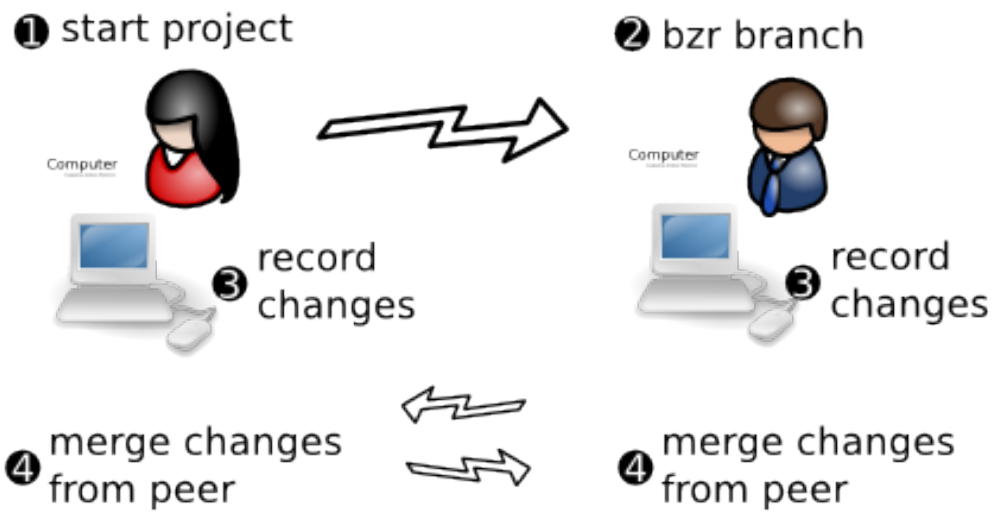
Advantages of this workflow over not using version control at all include:

- backup of old versions
- rollback to an earlier state
- tracking of history.

The key features of Bazaar appropriate for this workflow are low administration (no server setup) and ease of use.

1.3.3 Partner

Sometimes two people need to work together sharing changes as they go. This commonly starts off as a *Solo* workflow (see above) or a team-oriented workflow (see below). At some point, the second person takes a branch (copy including history) of what the first person has done. They can then work in parallel exchanging changes by merging when appropriate.



Advantages over *Solo* are:

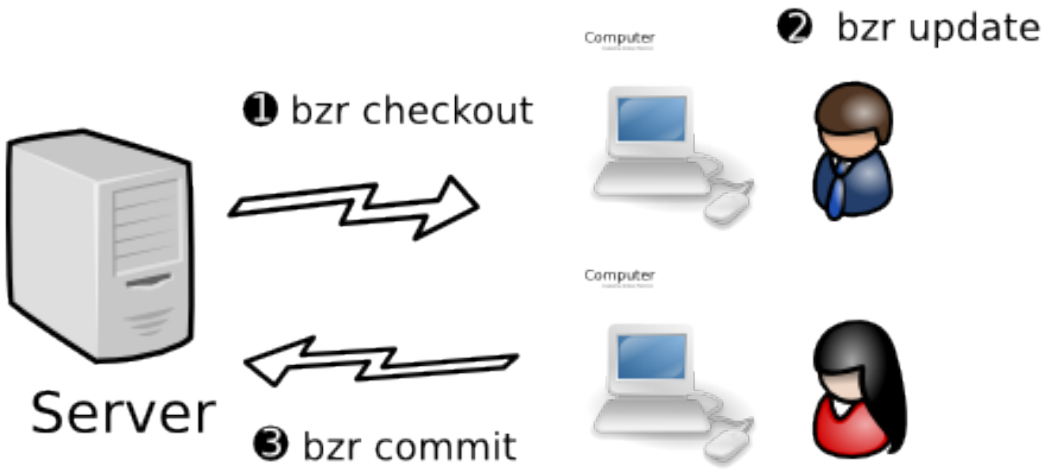
- easier sharing of changes
- each line of each text file can be attributed to a particular change including who changed it, when and why.

When implementing this workflow, Bazaar's advantages over CVS and Subversion include:

- no server to setup
- intelligent merging means merging multiple times isn't painful.

1.3.4 Centralized

Also known as *lock-step*, this is essentially the same as the workflow encouraged/enforced by CVS and Subversion. All developers work on the same branch (or branches). They run `bzr update` to get their checkout up-to-date, then `bzr commit` to publish their changes to the central location.

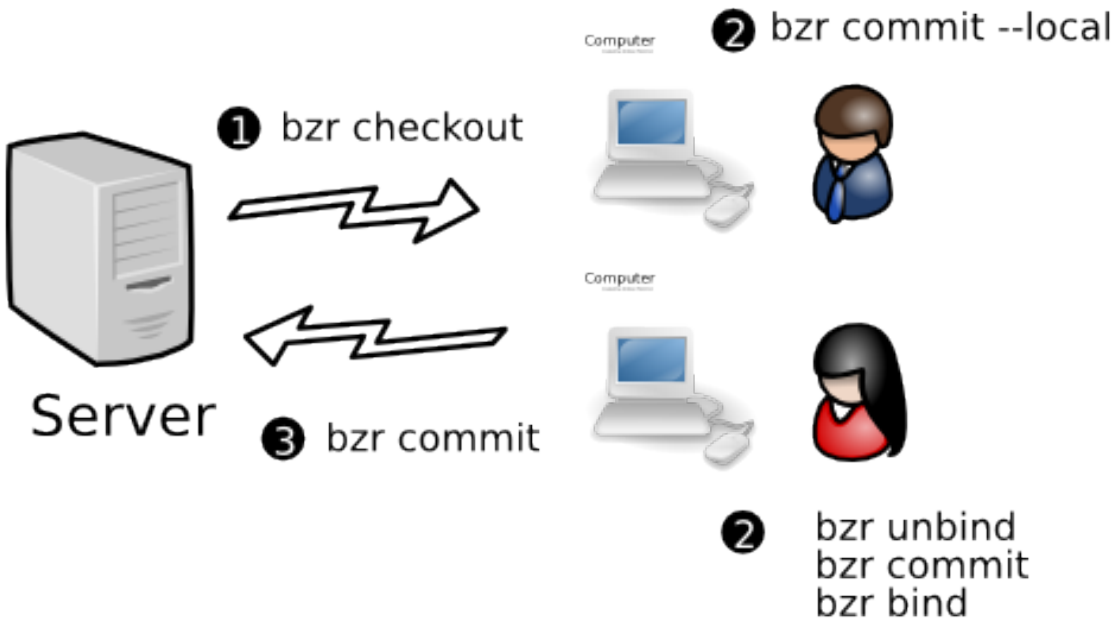


Subversion and CVS are good choices for implementing this workflow because they make it easy. Bazaar directly supports it as well while providing some important advantages over CVS and Subversion:

- better branching and merging
- better renaming support.

1.3.5 Centralized with local commits

This is essentially the same as the *Centralized* model, except that when developers are making a series of changes, they do `commit --local` or unbind their checkout. When it is complete, they commit their work to the shared mainline.



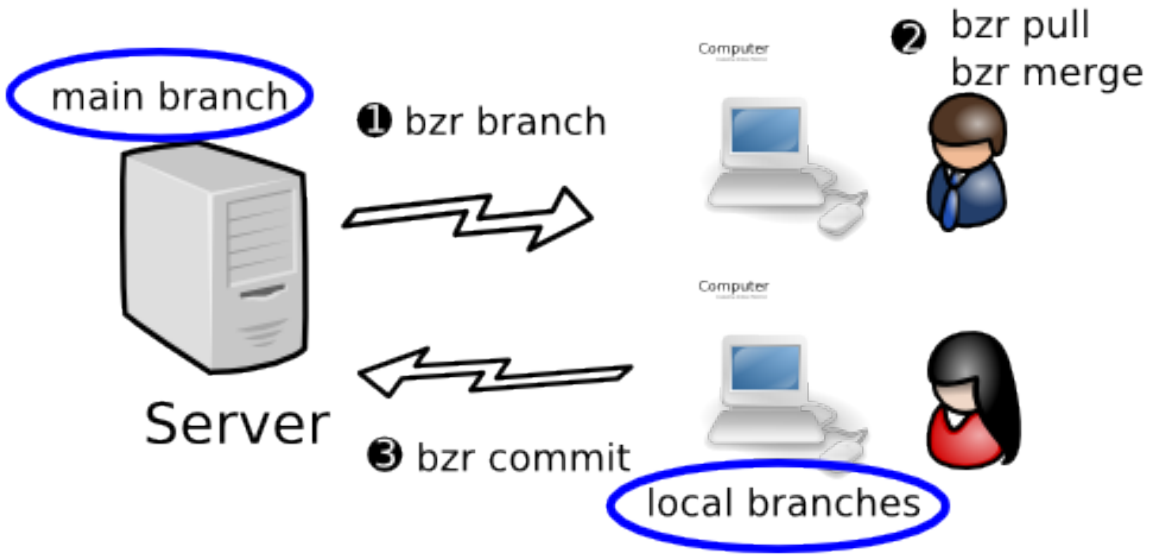
Advantages over *Centralized*:

- Can work offline, e.g. when disconnected during travel
- Less chance for a bad commit to interfere with everyone else's work

Subversion and CVS do not support this model. Other distributed VCS tools can support it but do so less directly than Bazaar does.

1.3.6 Decentralized with shared mainline

In this workflow, each developer has their own branch or branches, plus commit rights to the main branch. They do their work in their personal branch, then merge it into the mainline when it is ready.



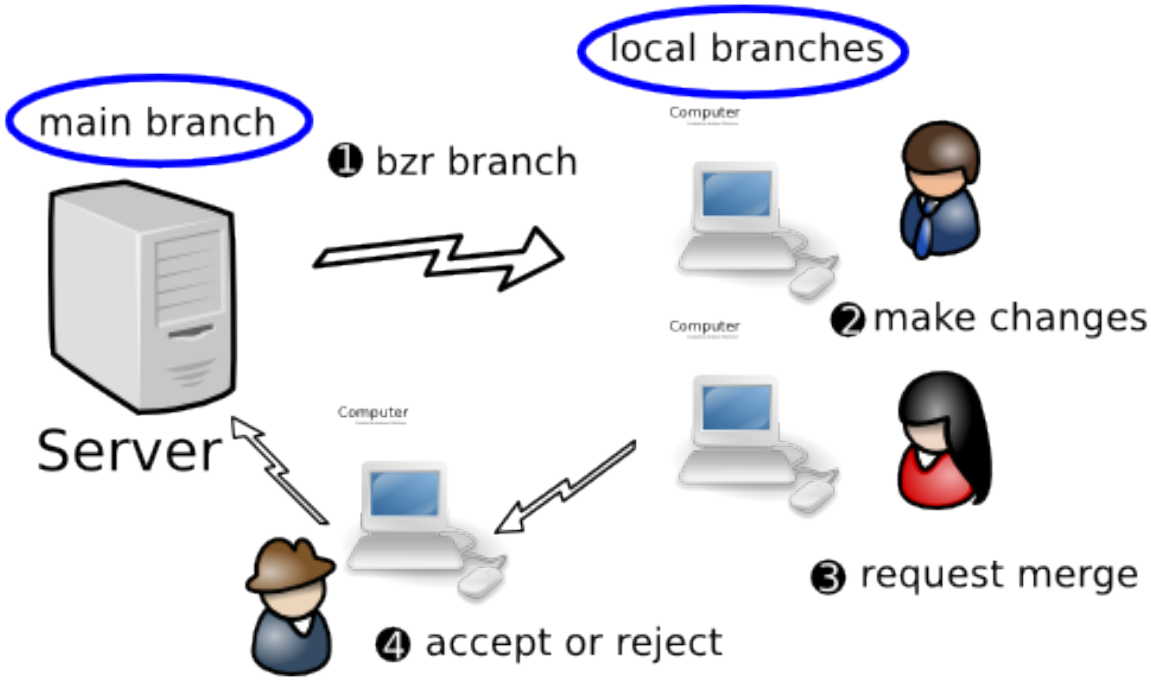
Advantage over *Centralized with local commits*:

- Easier organization of work - separate changes can be developed in their own branches
- Developers can merge one another's personal branches when working on something together.

Subversion and CVS do not support this model. Other distributed VCS tools support it. Many features of Bazaar are good for this workflow including ease of use, shared repositories, integrated merging and rich metadata (including directory rename tracking).

1.3.7 Decentralized with human gatekeeper

In this workflow, each developer has their own branch or branches, plus read-only access to the main branch. One developer (the gatekeeper) has commit rights to the main branch. When a developer wants their work merged, they ask the gatekeeper to merge it. The gatekeeper does code review, and merges the work into the main branch if it meets the necessary standards.



Advantage over *Decentralized with shared mainline*:

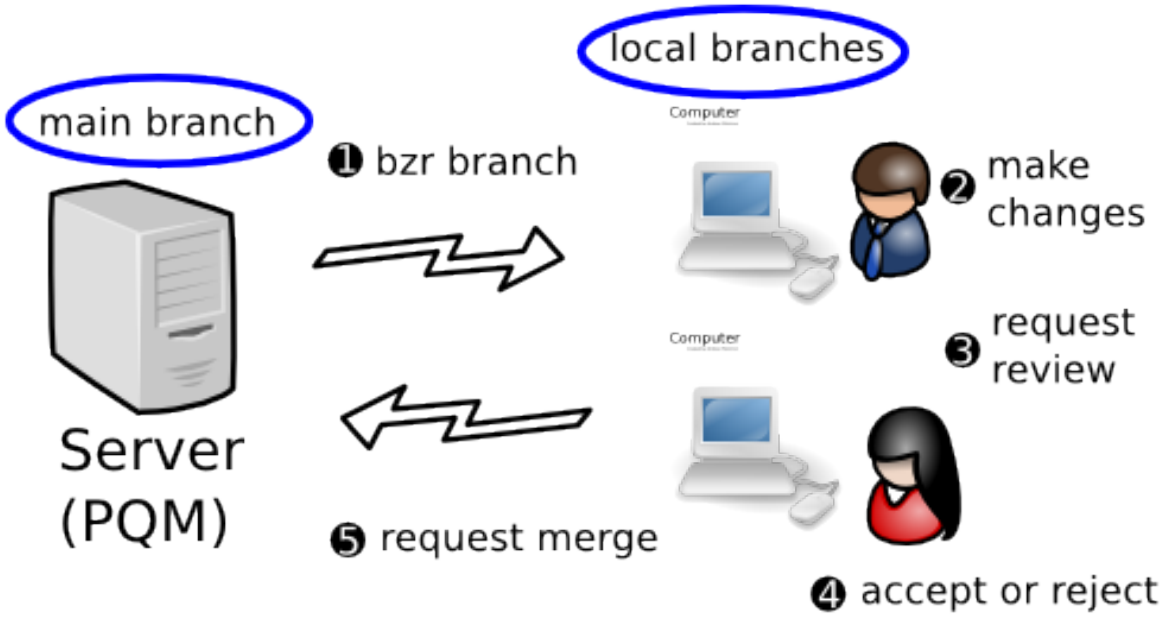
- Code is always reviewed before it enters the mainline
- Tighter control over when changes get incorporated into the mainline.

A companion tool of Bazaar's called Bundle Buggy can be very useful for tracking what changes are up for review, their status and reviewer comments.

1.3.8 Decentralized with automatic gatekeeper

In this workflow, each developer has their own branch or branches, plus read-only access to the mainline. A software gatekeeper has commit rights to the main branch. When a developer wants their work merged, they request another person to review it. Once it has passed review, either the original author or the reviewer asks the gatekeeper software to merge it, depending on team policies. The gatekeeper software does a merge, a compile, and runs the test suite. If and only if the code passes, it is merged into the mainline.

Note: As an alternative, the review step can be skipped and the author can submit the change to the automatic gatekeeper without it. (This is particularly appropriate when using practices such as Pair Programming that effectively promote just-in-time reviews instead of reviewing code as a separate step.)



Advantages over *Decentralized with human gatekeeper*:

- Code is always tested before it enters the mainline (so the integrity of the mainline is higher)
- Scales better as teams grow.

A companion tool of Bazaar's called Patch Queue Manager (PQM) can provide the automated gatekeeper capability.

1.3.9 Implementing a workflow

For an in-depth look at how to implement each of the workflows above, see chapters 3 to 6 in this manual. First though, chapter 2 explains some important pre-requisites including installation, general usage instructions and configuration tips.

CHAPTER

TWO

GETTING STARTED

2.1 Installing Bazaar

2.1.1 GNU/Linux

Bazaar packages are available for most popular GNU/Linux distributions including Ubuntu, Debian, Red Hat and Gentoo. See http://wiki.bazaar.canonical.com/Download for the latest instructions.

2.1.2 Windows

For Windows users, an installer is available that includes the core Bazaar package together with necessary pre-requisites and some useful plug-ins. See http://wiki.bazaar.canonical.com/Download for the latest instructions.

Note: If you are running Cygwin on Windows, a Bazaar for Cygwin package is available and ought to be used instead of the Windows version.

2.1.3 Other operating systems

Beyond Linux and Windows, Bazaar packages are available for a large range of other operating systems include Mac OS X, FreeBSD and Solaris. See http://wiki.bazaar.canonical.com/Download for the latest instructions.

2.1.4 Installing from scratch

If you wish to install Bazaar from scratch rather than using a pre-built package, the steps are:

1. If it is not installed already, install Python 2.6 or later.
2. Download the `bazaar-xxx.tar.gz` file (where xxx is the version number) from http://wiki.bazaar.canonical.com/Download or from Launchpad (https://launchpad.net/~bzr/).
3. Unpack the archive using tar, WinZip or equivalent.
4. Put the created directory on your PATH.

To test the installation, try running the **bzr** command like this:

```
bzr version
```

This will display the version of Bazaar you have installed. If this doesn't work, please contact us via email or IRC so we can help you get things working.

Installing into site-wide locations

Instead of adding the directory to your PATH, you can install bzr into the system locations using:

```
python setup.py install
```

If you do not have a compiler, or do not have the python development tools installed, bzr supplies a (slower) pure-python implementation of all extensions. You can install without compiling extensions with:

```
python setup.py install build_ext --allow-python-fallback
```

2.1.5 Running the development version

You may wish to always be using the very latest development version of Bazaar. Note that this is not recommended for the majority of users as there is an increased risk of bugs. On the other hand, the development version is remarkably solid (thanks to the processes we follow) and running it makes it easier for you to send us changes for bugs and improvements. It also helps us by having more people testing the latest software.

Here are the steps to follow:

1. Install Bazaar using one of the methods given above.
2. Get a copy of the development version like this:

   ```
   bzr branch lp:bzr
   ```

3. Put the created directory on your PATH.

Advanced users may also wish to build the optional C extensions for greater speed. This can be done using `make` and requires `pyrex` and a C compiler. Please contact us on email or IRC if you need assistance with this.

2.1.6 Running multiple versions

It's easy to have multiple versions of Bazaar installed and to switch between them. To do this, simply provide the full pathname to the **bzr** command you wish to run. The relevant libraries will be automatically detected and used. Of course, if you do not provide a pathname, then the **bzr** used will be the one found on your system path as normal.

Note that this capability is particularly useful if you wish to run (or test) both the latest released version and the development version say.

2.2 Entering commands

2.2.1 User interfaces

There are numerous user interfaces available for Bazaar. The core package provides a command line tool called **bzr** and graphical user interfaces (GUIs) are available as plug-ins.

2.2.2 Using bzr

The syntax is:

```
bzr [global-options] command [options and arguments]
```

Global options affect how Bazaar operates and can appear either before or after `command`. Command specific options must appear after the command but may be given either before, during or after any command-specific arguments.

2.2.3 Common options

Some options are legal for all commands as shown below.

Short form	Long form	Description
-h	–help	get help
-v	–verbose	be more verbose
-q	–quiet	be more quiet

Quiet mode implies that only errors and warnings are displayed. This can be useful in scripts.

Note: Most commands typically only support one level of verbosity though that may change in the future. To ask for a higher level of verbosity, simply specify the -v option multiple times.

2.3 Getting help

Bazaar comes with a built-in on-line help system, accessed through:

```
bzr help
```

You can ask for help on a command, or on non-command topics. To see a list of available help of each kind, use either:

```
bzr help commands
bzr help topics
```

For help on a particular command, use either of these forms:

```
bzr help status
bzr status --help
```

If you wish to search the help or read it as a larger document, the information is also available in the Bazaar User Reference.

2.4 Configuring Bazaar

2.4.1 Telling Bazaar about yourself

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 use the `whoami` command to set the identifier you want:

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

If `whoami` is used without an argument, the current value is displayed.

2.4.2 Using a network proxy

If your network requires that you use an HTTP proxy for outbound connections, you must set the `http_proxy` variable. If the proxy is also required for https connections, you need to set `https_proxy` too. If you need these and don't have them set, you may find that connections to Launchpad or other external servers fail or time out.

On Unix you typically want to set these in `/etc/environment` or `~/.bash_profile` and on Windows in the user profile.

```
http_proxy=http://proxy.example.com:3128/
https_proxy=http://proxy.example.com:3128/
```

The `no_proxy` variable can be set to a comma-separated list of hosts which shouldn't be reached by the proxy. (See <http://docs.python.org/library/urllib.html> for more details.)

2.4.3 Various ways to configure

As shown in the example above, there are various ways to configure Bazaar, they all share some common properties though. An option has:

- a name which is generally a valid python identifier,
- a value which is a string. In some cases, Bazaar will be able to recognize special values like 'True', 'False' to infer a boolean type, but basically, as a user, you will always specify a value as a string.

Options are grouped in various contexts so the option name uniquely identifies it in this context. When needed, options can be made persistent by recording them in a configuration file.

2.4.4 Configuration files

Configuration files are located in `$HOME/.bazaar` on Unix and `C:\Documents and Settings\<username>\Application Data\Bazaar\2.0` on Windows. There are three primary configuration files in this location:

- `bazaar.conf` describes default configuration options,
- `locations.conf` describes configuration information for specific branch locations,
- `authentication.conf` describes credential information for remote servers.

Each branch can also contain a configuration file that sets values specific to that branch. This file is found at `.bzr/branch/branch.conf` within the branch. This file is visible to **all users of a branch**. If you wish to override one of the values for a branch with a setting that is specific to you, then you can do so in `locations.conf`.

Here is sample content of `bazaar.conf` after setting an email address using the `whoami` command:

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

For further details on the syntax and configuration settings supported, see Configuration Settings in the Bazaar User Reference.

2.4.5 Looking at the active configuration

To look at all the currently defined options, you can use the following command:

```
bzr config
```

`bzr` implements some rules to decide where to get the value of a configuration option.

The current policy is to examine the existing configurations files in a given order for matching definitions.

- `locations.conf` is searched first for a section whose name matches the location considered (working tree, branch or remote branch),
- the current `branch.conf` is searched next,
- `bazaar.conf` is searched next,
- finally, some options can have default values generally defined in the code itself and not displayed by `bzr config` (see Configuration Settings).

This is better understood by using `'bzr config` with no arguments, which will display some output of the form:

```
locations:
  post_commit_to = commits@example.com
  news_merge_files = NEWS
branch:
  parent_location = bzr+ssh://bazaar.launchpad.net/+branch/bzr/
  nickname = config-modify
  push_location = bzr+ssh://bazaar.launchpad.net/~vila/bzr/config-modify/
bazaar:
  debug_flags = hpss,
```

Each configuration file is associated with a given scope whose name is displayed before each set of defined options.

If you need to look at a specific option, you can use:

```
bzr config <option>
```

This will display only the option value and is intended to be used in scripts.

2.4.6 Modifying the active configuration

To set an option to a given value use:

```
bzr config opt=value
```

An option value can reference another option by enclosing it in curly braces:

```
bzr config opt={other_opt}/subdir
```

If `other_opt` is set to `'root`, `bzr config opt` will display:

```
root/subdir
```

Note that when `--all` is used, the references are left as-is to better reflect the content of the config files and make it easier to organize them:

```
bzr config --all .*opt
```

```
bazaar:
  [DEFAULT]
  opt = {other_opt}/subdir
  other_opt = root
```

To remove an option use:

```
bzr config --remove opt
```

2.4.7 Rule-based preferences

Some commands and plugins provide custom processing on files matching certain patterns. Per-user rule-based preferences are defined in `BZR_HOME/rules`.

For further information on how rules are searched and the detailed syntax of the relevant files, see Rules in the Bazaar User Reference.

2.4.8 Escaping command lines

When you give a program name or command line in configuration, you can quote to include special characters or whitespace. The same rules are used across all platforms.

The rules are: strings surrounded by double-quotes are interpreted as single "words" even if they contain whitespace, and backslash may be used to quote quotation marks. For example:

```
BZR_EDITOR="C:\Program Files\My Editor\myeditor.exe"
```

2.5 Using aliases

2.5.1 What are aliases?

Aliases are an easy way to create shortcuts for commonly-typed commands, or to set defaults for commands.

2.5.2 Defining aliases

Command aliases can be defined in the `[ALIASES]` section of your `bazaar.conf` file. Aliases start with the alias name, then an equal sign, then a command fragment. Here's an example ALIASES section:

```
[ALIASES]
recentlog=log -r-3..-1
ll=log --line -r-10..-1
commit=commit --strict
diff=diff --diff-options -p
```

Here are the explanations of the examples above:

- The first alias makes a new `recentlog` command that shows the logs for the last three revisions
- The `ll` alias shows the last 10 log entries in line format.
- the `commit` alias sets the default for commit to refuse to commit if new files in the tree are not recognized.
- the `diff` alias adds the coveted -p option to diff

2.5.3 Using the aliases

The aliases defined above would be used like so:

```
% bzr recentlog
% bzr ll
% bzr commit
% bzr diff
```

2.5.4 Rules for aliases

- You can override a portion of the options given in an alias by specifying the new part on the command-line. For example, if you run `lastlog -r-5..`, you will only get five line-based log entries instead of 10. Note that all boolean options have an implicit inverse, so you can override the commit alias with `commit --no-strict`.
- Aliases can override the standard behaviour of existing commands by giving an alias name that is the same as the original command. For example, default commit is changed with `commit=commit --strict`.
- Aliases cannot refer to other aliases. In other words making a `lastlog` alias and referring to it with a `ll` alias will not work. This includes aliases that override standard commands.
- Giving the `--no-aliases` option to the bzr command will tell it to ignore aliases for that run. For example, running `bzr --no-aliases commit` will perform a standard commit instead, not do a `commit --strict`.

2.6 Using plugins

2.6.1 What is a plugin?

A plugin is an external component for Bazaar that is typically made by third parties. A plugin is capable of augmenting Bazaar by adding new functionality. A plugin can also change current Bazaar behavior by replacing current functionality. Sample applications of plugins are:

- overriding commands
- adding new commands
- providing additional network transports
- customizing log output.

The sky is the limit for the customization that can be done through plugins. In fact, plugins often work as a way for developers to test new features for Bazaar prior to inclusion in the official codebase. Plugins are helpful at feature retirement time as well, e.g. deprecated file formats may one day be removed from the Bazaar core and be made available as a plugin instead.

Plugins are good for users, good for external developers and good for Bazaar itself.

2.6.2 Where to find plugins

We keep our list of plugins on the http://wiki.bazaar.canonical.com/BzrPlugins page.

2.6.3 How to install a plugin

Installing a plugin is very easy! If not already created, create a `plugins` directory under your Bazaar configuration directory, `~/.bazaar/` on Unix and `C:\Documents and Settings\<username>\Application Data\Bazaar\2.0\` on Windows. Within this directory (referred to as $BZR_HOME below), each plugin is placed in its own subdirectory.

Plugins work particularly well with Bazaar branches. For example, to install the bzrtools plugins for your main user account on GNU/Linux, one can perform the following:

```
bzr branch http://panoramicfeedback.com/opensource/bzr/bzrtools
~/.bazaar/plugins/bzrtools
```

When installing plugins, the directories that you install them in must be valid python identifiers. This means that they can only contain certain characters, notably they cannot contain hyphens (-). Rather than installing `bzr-gtk` to `$BZR_HOME/plugins/bzr-gtk`, install it to `$BZR_HOME/plugins/gtk`.

2.6.4 Alternative plugin locations

If you have the necessary permissions, plugins can also be installed on a system-wide basis. One can additionally override the personal plugins location by setting the environment variable `BZR_PLUGIN_PATH` (see User Reference for a detailed explanation).

2.6.5 Listing the installed plugins

To do this, use the plugins command like this:

```
bzr plugins
```

The name, location and version of each plugin installed will be displayed.

New commands added by plugins can be seen by running `bzr help commands`. The commands provided by a plugin are shown followed by the name of the plugin in brackets.

2.6.6 Popular plugins

Here is a sample of some of the more popular plugins.

Category	Name	Description
GUI	QBzr	Qt-based GUI tools
GUI	bzr-gtk	GTK-based GUI tools
GUI	bzr-eclipse	Eclipse integration
General	bzrtools	misc. enhancements including shelf
General	difftools	external diff tool helper
General	extmerge	external merge tool helper
Integration	bzr-svn	use Subversion as a repository
Migration	cvsps	migrate CVS patch-sets

If you wish to write your own plugins, it is not difficult to do. See Writing a plugin in the appendices to get started.

2.7 Bazaar Zen

2.7.1 Grokking Bazaar

While Bazaar is similar to other VCS tools in many ways, there are some important differences that are not necessarily obvious at first glance. This section attempts to explain some of the things users need to know in order to "grok" Bazaar, i.e. to deeply understand it.

Note: It isn't necessary to fully understand this section to use Bazaar. You may wish to skim this section now and come back to it at a later time.

2.7.2 Understanding revision numbers

All revisions in the mainline of a branch have a simple increasing integer. (First commit gets 1, 10th commit gets 10, etc.) This makes them fairly natural to use when you want to say "grab the 10th revision from my branch", or "fixed in revision 3050".

For revisions which have been merged into a branch, a dotted notation is used (e.g., 3112.1.5). Dotted revision numbers have three numbers [1]. The first number indicates what mainline revision change is derived from. The second number is the branch counter. There can be many branches derived from the same revision, so they all get a unique number. The third number is the number of revisions since the branch started. For example, 3112.1.5 is the first branch from revision 3112, the fifth revision on that branch.

2.7.3 Hierarchical history is good

Imagine a project with multiple developers contributing changes where many changes consist of a series of commits. To give a concrete example, consider the case where:

- The tip of the project's trunk is revision 100.
- Mary makes 3 changes to deliver feature X.
- Bill makes 4 changes to deliver feature Y.

If the developers are working in parallel and using a traditional centralized VCS approach, the project history will most likely be linear with Mary's changes and Bill's changes interleaved. It might look like this:

```
107: Add documentation for Y
106: Fix bug found in testing Y
105: Fix bug found in testing X
104: Add code for Y
103: Add documentation for X
102: Add code and tests for X
101: Add tests for Y
100: ...
```

Many teams use this approach because their tools make branching and merging difficult. As a consequence, developers update from and commit to the trunk frequently, minimizing integration pain by spreading it over every commit. If you wish, you can use Bazaar exactly like this. Bazaar does offer other ways though that you ought to consider.

An alternative approach encouraged by distributed VCS tools is to create feature branches and to integrate those when they are ready. In this case, Mary's feature branch would look like this:

```
103: Fix bug found in testing X
102: Add documentation for X
101: Add code and tests for X
100: ...
```

And Bill's would look like this:

```
104: Add documentation for Y
103: Fix bug found in testing Y
102: Add code for Y
101: Add tests for Y
100: ...
```

[1] Versions prior to bzr 1.2 used a slightly different algorithm. Some nested branches would get extra numbers (such as 1.1.1.1.1) rather than the simpler 3-number system.

If the features were independent and you wanted to keep linear history, the changes could be pushed back into the trunk in batches. (Technically, there are several ways of doing that but that's beyond the scope of this discussion.) The resulting history might look like this:

```
107: Fix bug found in testing X
106: Add documentation for X
105: Add code and tests for X
104: Add documentation for Y
103: Fix bug found in testing Y
102: Add code for Y
101: Add tests for Y
100: ...
```

While this takes a bit more effort to achieve, it has some advantages over having revisions randomly intermixed. Better still though, branches can be merged together forming a non-linear history. The result might look like this:

```
102: Merge feature X
     100.2.3: Fix bug found in testing X
     100.2.2: Add documentation for X
     100.2.1: Add code and tests for X
101: Merge feature Y
     100.1.4: Add documentation for Y
     100.1.3: Fix bug found in testing Y
     100.1.2: Add code for Y
     100.1.1: Add tests for Y
100: ...
```

Or more likely this:

```
102: Merge feature X
     100.2.3: Fix bug
     100.2.2: Add documentation
     100.2.1: Add code and tests
101: Merge feature Y
     100.1.4: Add documentation
     100.1.3: Fix bug found in testing
     100.1.2: Add code
     100.1.1: Add tests
100: ...
```

This is considered good for many reasons:

- It makes it easier to understand the history of a project. Related changes are clustered together and clearly partitioned.
- You can easily collapse history to see just the commits on the mainline of a branch. When viewing the trunk history like this, you only see high level commits (instead of a large number of commits uninteresting at this level).
- If required, it makes backing out a feature much easier.
- Continuous integration tools can be used to ensure that all tests still pass before committing a merge to the mainline. (In many cases, it isn't appropriate to trigger CI tools after every single commit as some tests will fail during development. In fact, adding the tests first - TDD style - will guarantee it!)

In summary, the important points are:

Organize your work using branches.

Integrate changes using merge.

Ordered revision numbers and hierarchy make history easier to follow.

2.7.4 Each branch has its own view of history

As explained above, Bazaar makes the distinction between:

- mainline revisions, i.e. ones you committed in your branch, and
- merged revisions, i.e. ones added as ancestors by committing a merge.

Each branch effectively has its own view of history, i.e. different branches can give the same revision a different "local" revision number. Mainline revisions always get allocated single number revision numbers while merged revisions always get allocated dotted revision numbers.

To extend the example above, here's what the revision history of Mary's branch would look like had she decided to merge the project trunk into her branch after completing her changes:

```
104: Merge mainline
     100.2.1: Merge feature Y
     100.1.4: Add documentation
     100.1.3: Fix bug found in testing
     100.1.2: Add code
     100.1.1: Add tests
103: Fix bug found in testing X
102: Add documentation for X
101: Add code and tests for X
100: ...
```

Once again, it's easy for Mary to look at just *her* top level of history to see the steps she has taken to develop this change. In this context, merging the trunk (and resolving any conflicts caused by doing that) is just one step as far as the history of this branch is concerned.

It's important to remember that Bazaar is not changing history here, nor is it changing the global revision identifiers. You can always use the latter if you really want to. In fact, you can use the branch specific revision numbers when communicating *as long as* you provide the branch URL as context. (In many Bazaar projects, developers imply the central trunk branch if they exchange a revision number without a branch URL.)

Merges do not change revision numbers in a branch, though they do allocate local revision numbers to newly merged revisions. The only time Bazaar will change revision numbers in a branch is when you explicitly ask it to mirror another branch.

Note: Revisions are numbered in a stable way: if two branches have the same revision in their mainline, all revisions in the ancestry of that revision will have the same revision numbers. For example, if Alice and Bob's branches agree on revision 10, they will agree on all revisions before that.

2.7.5 Summary

In general, if you follow the advice given earlier - organise your work in branches and use merge to collaborate - you'll find Bazaar generally does what you expect.

In coming chapters, we examine various ways of using Bazaar beginning with the simplest: using Bazaar for personal projects.

CHAPTER

THREE

PERSONAL VERSION CONTROL

3.1 Going solo

3.1.1 A personal productivity tool

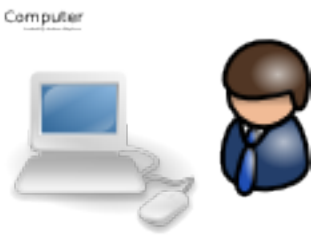
Some tools are designed to make individuals productive (e.g. editors) while other tools (e.g. back-end services) are focused on making teams or whole companies more productive. Version control tools have traditionally been in the latter camp.

One of the cool things about Bazaar is that it is so easy to setup that version control can now be treated as a personal productivity tool. If you wish to record changes to files for the purposes of checkpointing good known states or tracking history, it is now easy to do so. This chapter explains how.

3.1.2 The solo workflow

If you are creating your own masterpiece, whether that be a software project or a set of related documents, the typical workflow looks like this:

❶ create project
❷ record changes
❸ browse history
❹ package release



Even if you will always be working as part of a team, the tasks covered in this chapter will be the basis of what you'll be doing so it's a good place to start.

3.2 Starting a project

3.2.1 Putting an existing project under version control

If you already have a tree of source code (or directory of documents) you wish to put under version control, here are the commands to use:

```
cd my-stuff
bzr init
bzr add
bzr commit -m "Initial import"
```

`bzr init` creates a `.bzr` directory in the top level directory (`my-stuff` in the example above). Note that:

- Bazaar has everything it needs in that directory - you do **not** need to setup a database, web server or special service to use it
- Bazaar is polite enough to only create one `.bzr` in the directory given, not one in every subdirectory thereof.

`bzr add` then finds all the files and directories it thinks ought to be version controlled and registers them internally. `bzr commit` then records a snapshot of the content of these and records that information, together with a commit message.

More information on `init`, `add` and `commit` will be provided later. For now, the important thing to remember is the recipe above.

3.2.2 Starting a new project

If you are starting a project from scratch, you can also use the recipe above, after creating an empty directory first of course. For efficiency reasons that will be explored more in later chapters though, it is a good idea to create a repository for the project at the top level and to nest a *main* branch within it like this:

```
bzr init-repo my.repo
cd my.repo
bzr init my.main
cd my.main
hack, hack, hack
bzr add
bzr commit -m "Initial import"
```

Some users prefer a name like *trunk* or *dev* to *main*. Choose whichever name makes the most sense to you.

Note that the `init-repo` and `init` commands both take a path as an argument and will create that path if it doesn't already exist.

3.3 Controlling file registration

3.3.1 What does Bazaar track?

As explained earlier, `bzr add` finds and registers all the things in and under the current directory that Bazaar thinks ought to be version controlled. These things may be:

- files
- directories
- symbolic links.

Bazaar has default rules for deciding which files are interesting and which ones are not. You can tune those rules as explained in Ignoring files below.

Unlike many other VCS tools, Bazaar tracks directories as first class items. As a consequence, empty directories are correctly supported - you don't need to create a dummy file inside a directory just to ensure it gets tracked and included in project exports.

For symbolic links, the value of the symbolic link is tracked, not the content of the thing the symbolic link is pointing to.

Note: Support for tracking projects-within-projects ("nested trees") is currently under development. Please contact the Bazaar developers if you are interested in helping develop or test this functionality.

3.3.2 Selective registration

In some cases, you may want or need to explicitly nominate the things to register rather than leave it up to Bazaar to find things. To do this, simply provide paths as arguments to the `add` command like this:

```
bzr add fileX dirY/
```

Adding a directory implicitly adds all interesting things underneath it.

3.3.3 Ignoring files

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 Bazaar 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~           *~
```

Note that ignore patterns are only matched against non-versioned files, and control whether they are treated as "unknown" or "ignored". If a file is explicitly added, it remains versioned regardless of whether it matches an ignore pattern.

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"
```

The command `bzr ignore PATTERN` can be used to easily add PATTERN to the `.bzrignore file` (creating it if necessary and registering it to be tracked by Bazaar). Removing and modifying patterns are done by directly editing the `.bzrignore` file.

3.3.4 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.

3.4 Reviewing changes

3.4.1 Looking before you leap

Once you have completed some work, it's a good idea to review your changes prior to permanently recording it. This way, you can make sure you'll be committing what you intend to.

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

3.4.2 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.

3.4.3 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
```

To see the changes introduced by a single revision, you can use the `-c` option to diff.

```
% bzr diff -c 1000            # changes from r1000
                              # identical to -r999..1000
```

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

[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.

```
% 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`.

3.5 Recording changes

3.5.1 bzr commit

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

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.)

3.5.2 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.

3.5.3 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 .
```

3.5.4 Giving credit for a change

If you didn't actually write the changes that you are about to commit, for instance if you are applying a patch from someone else, you can use the `--author` commit option to give them credit for the change:

```
% bzr commit --author "Jane Rey <jrey@example.com>"
```

The person that you specify there will be recorded as the "author" of the revision, and you will be recorded as the "committer" of the revision.

If more than one person works on the changes for a revision, for instance if you are pair-programming, then you can record this by specifying `--author` multiple times:

```
% bzr commit --author "Jane Rey <jrey@example.com>" \
    --author "John Doe <jdoe@example.com>"
```

3.6 Browsing history

3.6.1 bzr log

The `bzr log` command shows a list of previous revisions.

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
```

3.6.2 Viewing merged revisions

As distributed VCS tools like Bazaar make merging much easier than it is in central VCS tools, the history of a branch may often contain lines of development splitting off the mainline and merging back in at a later time. Technically, the relationship between the numerous revision nodes is known as a Directed Acyclic Graph or DAG for short.

In many cases, you typically want to see the mainline first and drill down from there. The default behaviour of log is therefore to show the mainline and indicate which revisions have nested merged revisions. To explore the merged revisions for revision X, use the following command:

```
bzr log -n0 -rX
```

To see all revisions and all their merged revisions:

```
bzr log -n0
```

Note that the -n option is used to indicate the number of levels to display where 0 means all. If that is too noisy, you can easily adjust the number to only view down so far. For example, if your project is structured with a top level gatekeeper merging changes from team gatekeepers, `bzr log` shows what the top level gatekeeper did while `bzr log -n2` shows what the team gatekeepers did. In the vast majority of cases though, `-n0` is fine.

3.6.3 Tuning the output

The `log` command has several options that are useful for tuning the output. These include:

- `--forward` presents the log in chronological order, i.e. the most recent revisions are displayed last.

- the `--limit` option controls the maximum number of revisions displayed.

See the online help for the log command or the User Reference for more information on tuning the output.

3.6.4 Viewing the history for a file

It is often useful to filter the history so that it only applies to a given file. To do this, provide the filename to the `log` command like this:

```
bzr log foo.py
```

3.6.5 Viewing an old version of a file

To get the contents of a file at a given version, use the `cat` command like this:

```
bzr cat -r X file
```

where `X` is the revision identifier and `file` is the filename. This will send output to the standard output stream so you'll typically want to pipe the output through a viewing tool (like `less` or `more`) or redirect it like this:

```
bzr cat -r -2 foo.py | less
bzr cat -r 1 foo.py > /tmp/foo-1st-version.py
```

3.6.6 Graphical history viewers

History browsing is one area where GUI tools really make life easier. Bazaar has numerous plug-ins that provide this capability including QBzr and bzr-gtk. See Using plugins for details on how to install these if they are not already installed.

To use the graphical viewer from QBzr:

```
bzr qlog
```

To use the graphical viewer from bzr-gtk:

```
bzr viz
```

`viz` is actually a built-in alias for `visualize` so use the longer command name if you prefer.

3.7 Releasing a project

3.7.1 Packaging a release

The `export` command is used to package a release, i.e. to take a copy of the files and directories in a branch and package them into a fresh directory or archive. For example, this command will package the last committed version into a `tar.gz` archive file:

```
bzr export ../releases/my-stuff-1.5.tar.gz
```

The `export` command uses the suffix of the archive file to work out the type of archive to create as shown below.

Supported formats	Autodetected by extension
dir	(none)
tar	.tar
tbz2	.tar.bz2, .tbz2
tgz	.tar.gz, .tgz
zip	.zip

If you wish to package a revision other than the last one, use the `-r` option. If you wish to tune the root directory inside the archive, use the `--root` option. See the online help or User Reference for further details on the options supported by `export`.

3.7.2 Tagging a release

Rather than remembering which version was used to package a release, it's useful to define a symbolic name for a version using the `tag` command like this:

```
bzr tag version-1-5
```

That tag can be used later whenever a revision identifier is required, e.g.:

```
bzr diff -r tag:version-1-5
```

To see the list of tags defined in a branch, use the `tags` command.

3.8 Undoing mistakes

3.8.1 Mistakes happen

Bazaar has been designed to make it easy to recover from mistakes as explained below.

3.8.2 Dropping the revision history for a project

If you accidentally put the wrong tree under version control, simply delete the `.bzr` directory.

3.8.3 Deregistering a file or directory

If you accidentally register a file using `add` that you don't want version controlled, you can use the `remove` command to tell Bazaar to forget about it.

`remove` has been designed to *Do the Safe Thing* in that it will not delete a modified file. For example:

```
bzr add foo.html
(oops - didn't mean that)
bzr remove foo.html
```

This will complain about the file being modified or unknown. If you want to keep the file, use the `--keep` option. Alternatively, if you want to delete the file, use the `--force` option. For example:

```
bzr add foo.html
(oops - didn't mean that)
bzr remove --keep foo.html
(foo.html left on disk, but deregistered)
```

On the other hand, the unchanged `TODO` file is deregistered and removed from disk without complaint in this example:

```
bzr add TODO
bzr commit -m "added TODO"
(hack, hack, hack - but don't change TODO)
bzr remove TODO
(TODO file deleted)
```

Note: If you delete a file using your file manager, IDE or via an operating system command, the `commit` command will implicitly treat it as removed.

3.8.4 Undoing changes since the last commit

One of the reasons for using a version control tool is that it lets you easily checkpoint good tree states while working. If you decide that the changes you have made since the last `commit` ought to be thrown away, the command to use is `revert` like this:

```
bzr revert
```

As a precaution, it is good practice to use `bzr status` and `bzr diff` first to check that everything being thrown away really ought to be.

3.8.5 Undoing changes to a file since the last commit

If you want to undo changes to a particular file since the last commit but keep all the other changes in the tree, pass the filename as an argument to `revert` like this:

```
bzr revert foo.py
```

3.8.6 Undoing the last commit

If you make a commit and really didn't mean to, use the `uncommit` command to undo it like this:

```
bzr uncommit
```

Unlike `revert`, `uncommit` leaves the content of your working tree exactly as it is. That's really handy if you make a commit and accidently provide the wrong error message. For example:

```
bzr commit -m "Fix bug #11"
(damn - wrong bug number)
bzr uncommit
bzr commit -m "Fix bug #1"
```

Another common reason for undoing a commit is because you forgot to add one or more files. Some users like to alias `commit` to `commit --strict` so that commits fail if unknown files are found in the tree.

Tags for uncommitted revisions are removed from the branch unless `--keep-tags` was specified.

Note: While the `merge` command is not introduced until the next chapter, it is worth noting now that `uncommit` restores any pending merges. (Running `bzr status` after `uncommit` will show these.) `merge` can also be used to effectively undo just a selected commit earlier in history. For more information on `merge`, see Merging changes in the next chapter and the Bazaar User Reference.

3.8.7 Undoing multiple commits

You can use the -r option to undo several commits like this:

```
bzr uncommit -r -3
```

If your reason for doing this is that you really want to back out several changes, then be sure to remember that `uncommit` does not change your working tree: you'll probably need to run the `revert` command as well to complete the task. In many cases though, it's arguably better to leave your history alone and add a new revision reflecting the content of the last good state.

3.8.8 Reverting to the state of an earlier version

If you make an unwanted change but it doesn't make sense to uncommit it (because that code has been released to users say), you can use `revert` to take your working tree back to the desired state. For example:

```
% bzr commit "Fix bug #5"
Committed revision 20.
(release the code)
(hmm - bad fix)
bzr revert -r 19
bzr commit -m "Backout fix for bug #5"
```

This will change your entire tree back to the state as of revision 19, which is probably only what you want if you haven't made any new commits since then. If you have, the `revert` would wipe them out as well. In that case, you probably want to use Reverse cherrypicking instead to back out the bad fix.

Note: As an alternative to using an absolute revision number (like 19), you can specify one relative to the tip (-1) using a negative number like this:

```
bzr revert -r -2
```

3.8.9 Correcting a tag

If you have defined a tag prematurely, use the `--force` option of the `tag` command to redefine it. For example:

```
bzr tag 2.0-beta-1
(oops, we're not yet ready for that)
(make more commits to include more fixes)
bzr tag 2.0-beta-1 --force
```

3.8.10 Clearing a tag

If you have defined a tag and no longer want it defined, use the `--delete` option of the `tag` command to remove it. For example:

```
bzr tag 2.0-beta-4
(oops, we're not releasing a 4th beta)
bzr tag 2.0-beta-4 --delete
```

CHAPTER

FOUR

SHARING WITH PEERS

4.1 Working with another

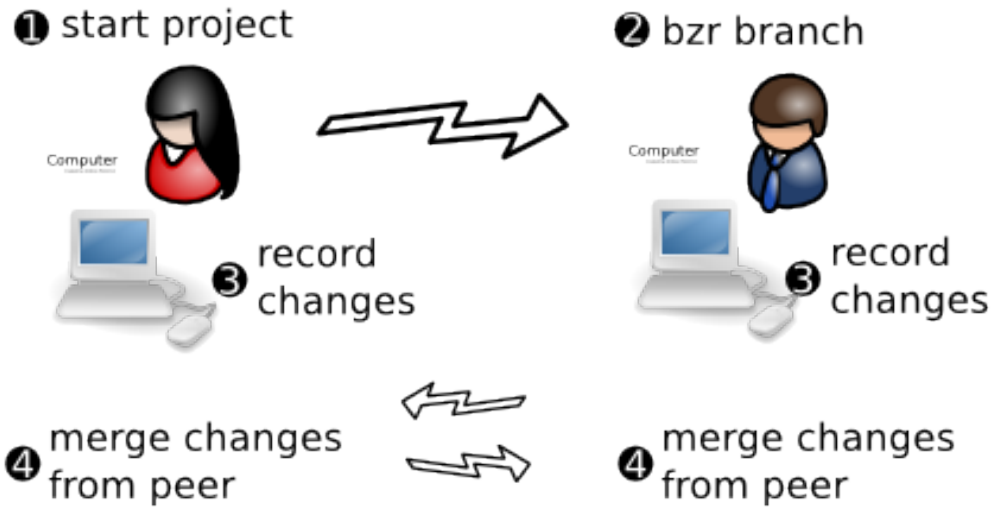
4.1.1 Peer-to-peer rocks

In many cases, two minds can be better than one. You may be the one who started a project and someone wishes to help, or perhaps it's you who wants to help another. Perhaps you are both members of a larger team that have been assigned a task together as pair programmers. Either way, two people need to agree on a process, a set of guidelines and a toolset in order to work together effectively.

Imagine if you were not allowed to call someone on the phone directly and the only way to talk to them was by registering a conference call first? Companies and communities that only share code via a central VCS repository are living with a similar straitjacket to that every day. There are times when central control makes a lot of sense and times when peer-to-peer rocks. Either way, Bazaar is designed to help.

4.1.2 The partner workflow

While it's certainly not the only way to do it, the *partner workflow* below is a good starting point for a pair of people who wish to collaborate using Bazaar.



Over and above the tasks covered in the previous chapter, this chapter introduces two essential collaboration activities:

- getting a copy of a branch
- merging changes between branches.

Even when it's just you working on a code base, it can be very useful to keep multiple branches around (for different releases say) and to merge changes between them as appropriate. Your "partner" may indeed be yourself.

4.2 Branching a project

4.2.1 Branch URLs

Before someone else can get a copy of your work, you need to agree on a transfer technology. You may decide to make the top level directory of your branch a network share, an approach familiar to Windows users. Unix users might prefer access to be via SSH, a secure protocol built-in to most SSH servers. Bazaar is *very* flexible in this regard with support for lots of protocols some of which are given below.

Prefix	Description
file://	Access using the standard filesystem (default).
bzr+ssh:/	Access over SSH (best remote option).
sftp://	Access using SFTP (most SSH servers provide SFTP).
bzr://	Fast access using the Bazaar smart server.
ftp://	Access using passive FTP.
http://	Access to branches exported by a web server.
https://	Encrypted access to branches exported by a web server.

As indicated above, branches are identified using URLs with the prefix indicating the transfer technology. If no prefix is given, normal filenames are assumed. For a complete list of supported protocols, see the `urlspec` online help topic or the URL Identifiers section of the Bazaar User Reference.

URLs are normally resolved relative to the root directory of the server, so `ftp://example.com/repo/foo` means the `/repo/foo` directory of that host. (We say 'normally' because some server software like Apache can be configured to remap URLs arbitrarily, in which case you'll need to look at the server configuration to find out which URL corresponds to which directory.)

To address a path relative to your home directory on the server, use a tilde like so: `bzr+ssh://example.com/~/public_html` should map to `public_html` within your home directory.

Note: Access over HTTP or HTTPS is read-only by default. See Pushing over the HTTP smart server for details on configuring read-write access.

4.2.2 A reminder about shared repositories

Before getting a copy of a branch, have a quick think about where to put it on your filesystem. For maximum storage efficiency down the track, it is recommended that branches be created somewhere under a directory that has been set up as a shared repository. (See Feature branches in Organizing your workspace for a commonly used layout.) For example:

```
bzr init-repo my-repo
cd my-repo
```

You are now ready to grab a branch from someone else and hack away.

4.2.3 The branch command

To get a branch based on an existing branch, use the `branch` command. The syntax is:

```
bzr branch URL [directory]
```

If a directory is not given, one is created based on the last part of the URL. Here are some examples showing a drive qualified path (M:/) and an SFTP URL respectively:

```
bzr branch M:/cool-trunk
bzr branch sftp://bill@mary-laptop/cool-repo/cool-trunk
```

This example shows explicitly giving the directory name to use for the new branch:

```
bzr branch /home/mary/cool-repo/cool-trunk cool
```

4.2.4 Time and space considerations

Depending on the size of the branch being transferred and the speed and latency of the network between your computer and the source branch, this initial transfer might take some time. Subsequent updates should be much faster as only the changes are transferred then.

Keep in mind that Bazaar is transferring the complete history of the branch, not just the latest snapshot. As a consequence, you can be off the network (or disconnected from the network share) after `branch` completes but you'll still be able to `log` and `diff` the history of the branch as much as you want. Furthermore, these operations are quick as the history is stored locally.

Note that Bazaar uses smart compression technology to minimize the amount of disk space required to store version history. In many cases, the complete history of a project will take up less disk space than the working copy of the latest version.

As explained in later chapters, Bazaar also has support for lightweight checkouts of a branch, i.e. working trees with no local storage of history. Of course, disconnected usage is not available then but that's a tradeoff you can decide to make if local disk space is really tight for you. Support for limited lookback into history - *history horizons* - is currently under development as well.

4.2.5 Viewing branch information

If you wish to see information about a branch including where it came from, use the `info` command. For example:

```
bzr info cool
```

If no branch is given, information on the current branch is displayed.

4.3 Merging changes

4.3.1 Parallel development

Once someone has their own branch of a project, they can make and commit changes in parallel to any development proceeding on the original branch. Pretty soon though, these independent lines of development will need to be combined again. This process is known as *merging*.

4.3.2 The merge command

To incorporate changes from another branch, use the `merge` command. Its syntax is:

```
bzr merge [URL]
```

If no URL is given, a default is used, initially the branch this branch originated from. For example, if Bill made a branch from Mary's work, he can merge her subsequent changes by simply typing this:

```
bzr merge
```

On the other hand, Mary might want to merge into her branch the work Bill has done in his. In this case, she needs to explicitly give the URL the first time, e.g.:

```
bzr merge bzr+ssh://mary@bill-laptop/cool-repo/cool-trunk
```

This sets the default merge branch if one is not already set. Use `--no-remember` to avoid setting it. To change the default after it is set, use the `--remember` option.

4.3.3 How does merging work?

A variety of algorithms exist for merging changes. Bazaar's default algorithm is a variation of *3-way merging* which works as follows. Given an ancestor A and two branches B and C, the following table provides the rules used.

A	B	C	Result	Comment
x	x	x	x	unchanged
x	x	y	y	line from C
x	y	x	y	line from B
x	y	z	?	conflict

Note that some merges can only be completed with the assistance of a human. Details on how to resolve these are given in Resolving conflicts.

4.3.4 Recording a merge

After any conflicts are resolved, the merge needs to be committed. For example:

```
bzr commit -m "Merged Mary's changes"
```

Even if there are no conflicts, an explicit commit is still required. Unlike some other tools, this is considered a feature in Bazaar. A clean merge is not necessarily a good merge so making the commit a separate explicit step allows you to run your test suite first to verify all is good. If problems are found, you should correct them before committing the merge or throw the merge away using `revert`.

4.3.5 Merge tracking

One of the most important features of Bazaar is distributed, high quality *merge tracking*. In other words, Bazaar remembers what has been merged already and uses that information to intelligently choose the best ancestor for a merge, minimizing the number and size of conflicts.

If you are a refugee from many other VCS tools, it can be really hard to "unlearn" the *please-let-me-avoid-merging-at-any-cost* habit. Bazaar lets you safely merge as often as you like with other people. By working in a peer-to-peer manner when it makes sense to do so, you also avoid using a central branch as an "integration swamp", keeping its quality higher. When the change you're collaborating on is truly ready for wider sharing, that's the time to merge and commit it to a central branch, not before.

Merging that Just Works truly can change how developers work together.

4.4 Resolving conflicts

4.4.1 Workflow

Unlike some other tools that force you to resolve each conflict during the merge process, Bazaar merges as much as it can and then reports the conflicts. This can make conflict resolution easier because the contents of the whole post-merge tree are available to help you decide how things ought to be resolved. You may also wish to selectively run tests as you go to confirm each resolution or group or resolutions is good.

4.4.2 Listing conflicts

As well as being reported by the `merge` command, the list of outstanding conflicts may be displayed at any time by using the `conflicts` command. It is also included as part of the output from the `status` command.

4.4.3 Resolving a conflict

When a conflict is encountered, the `merge` command puts embedded markers in each file showing the areas it couldn't resolve. It also creates 3 files for each file with a conflict:

- foo.BASE
- foo.THIS
- foo.OTHER

where `foo` is the name of the conflicted file. In many cases, you can resolve conflicts by simply manually editing each file in question, fixing the relevant areas and removing the conflict markers as you go.

After fixing all the files in conflict, and removing the markers, ask Bazaar to mark them as resolved using the `resolve` command like this:

```
bzr resolve
```

Alternatively, after fixing each file, you can mark it as resolved like this:

```
bzr resolve foo
```

Among other things, the `resolve` command cleans up the BASE, THIS and OTHER files from your working tree.

4.4.4 Using the remerge command

In some cases, you may wish to try a different merge algorithm on a given file. To do this, use the `remerge` command nominating the file like this:

```
bzr remerge --weave foo
```

where `foo` is the file and `weave` is one of the available merge algorithms. This algorithm is particularly useful when a so-called `criss-cross` merge is detected, e.g. when two branches merge the same thing then merge each other. See the online help for `criss-cross` and `remerge` for further details.

4.4.5 Using external tools to resolve conflicts

If you have a GUI tool you like using to resolve conflicts, be sure to install the *extmerge* plugin. Once installed, it can be used like this:

```
bzr extmerge foo
```

where `foo` is the conflicted file. Rather than provide a list of files to resolve, you can give the `--all` option to implicitly specify all conflicted files.

The `extmerge` command uses the tool specified by the `external_merge` setting in your `bazaar.conf` file. If not set, it will look for some popular merge tools such as `kdiff3` or `opendiff`, the latter being a command line interface to the FileMerge utility in OS X.

4.5 Annotating changes

4.5.1 Seeing the origin of content

When two or more people are working on files, it can be highly useful at times to see who created or last changed certain content. To do this, using the annotate command like this:

```
bzr annotate readme.txt
```

If you are a pessimist or an optimist, you might prefer to use one of built-in aliases for `annotate`: `blame` or `praise`. Either way, this displays each line of the file together with information such as:

- who changed it last
- when it was last changed
- the commit message.

4.5.2 GUI tools

The various GUI plugins for Bazaar provide graphical tools for viewing annotation information. For example, the bzr-gtk plugin provides a GUI tool for this that can be launched using the `gannotate` command:

```
bzr gannotate readme.txt
```

The GUI tools typically provide a much richer display of interesting information (e.g. all the changes in each commit) so you may prefer them over the text-based command.

CHAPTER

FIVE

TEAM COLLABORATION, CENTRAL STYLE

5.1 Centralized development

5.1.1 Motivation

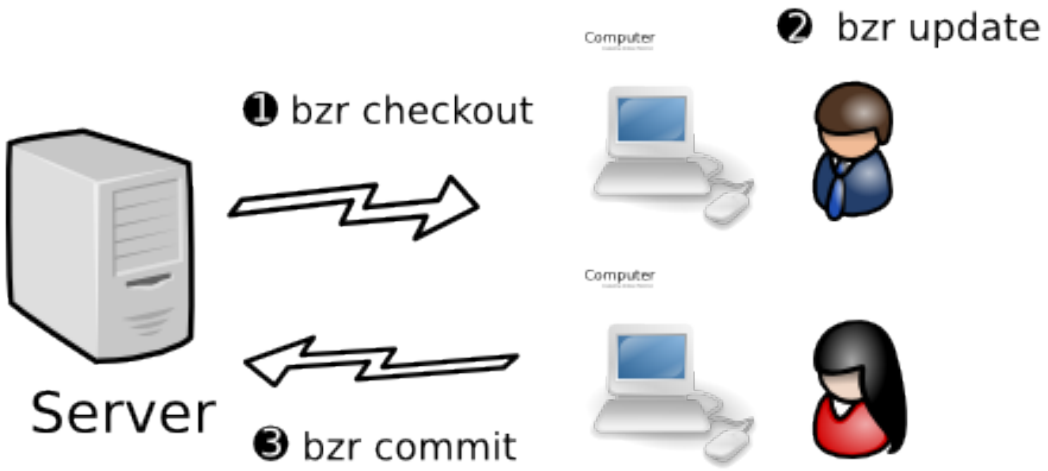
Rather than working in parallel and occasionally merging, it can be useful at times to work in lockstep, i.e. for multiple people to be continuously committing changes to a central location, merging their work with the latest content before every commit.

This workflow is very familiar to users of central VCS tools like Subversion and CVS. It is also applicable to a single developer who works on multiple machines, e.g. someone who normally works on a desktop computer but travels with a laptop, or someone who uses their (Internet connected) home computer to complete office work out of hours.

If centralized development works well for your team already, that's great. Many teams begin using Bazaar this way and experiment with alternative workflows later.

5.1.2 Centralized workflow

The diagram below provides an overview of the *centralized workflow*.



Even if your team is planning to use a more distributed workflow, many of the tasks covered in this chapter may be useful to you, particularly how to publish branches.

5.2 Publishing a branch

5.2.1 Setting up a central repository

While the centralized workflow can be used by socially nominating any branch on any computer as the central one, in practice most teams have a dedicated server for hosting central branches.

Just as it's best practice to use a shared repository locally, it's advisable to put central branches in a shared repository. Note that central shared branches typically only want to store history, not working copies of files, so their enclosing repository is usually creating using the `no-trees` option, e.g.:

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

You can think of this step as similar to setting up a new cvsroot or Subversion repository. However, Bazaar gives you more flexibility in how branches may be organised in your repository. See Advanced shared repository layouts in the appendices for guidelines and examples.

5.2.2 Starting a central branch

There are two ways of populating a central branch with some initial content:

1. Making a local branch and pushing it to a central location
2. Making an empty central branch then committing content to it.

Here is an example of the first way:

```
bzr init-repo PROJECT  (prepare local repository)
bzr init PROJECT/trunk
cd PROJECT/trunk
                       (copy development files)
cp -ar ~/PROJECT .     (copy files in using OS-specific tools)
bzr add                (populate repository; start version control)
bzr commit -m "Initial import"
                       (publish to central repository)
bzr push bzr+ssh://centralhost/srv/bzr/PROJECT/trunk
```

Here is an example of the second way:

```
bzr init-repo PROJECT  (prepare local repository)
cd PROJECT
bzr init bzr+ssh://centralhost/srv/bzr/PROJECT/trunk
bzr checkout bzr+ssh://centralhost/srv/bzr/PROJECT/trunk
cd trunk
cp -ar ~/PROJECT .     (copy files in using OS-specific tools)
bzr add                (populate repository; start version control)
bzr commit -m "Initial import"
                       (publish to central repository)
```

Note that committing inside a working tree created using the `checkout` command implicitly commits the content to the central location as well as locally. Had we used the `branch` command instead of `checkout` above, the content would have only been committed locally.

Working trees that are tightly bound to a central location like this are called *checkouts*. The rest of this chapter explains their numerous features in more detail.

5.3 Using checkouts

5.3.1 Turning a branch into a checkout

If you have a local branch and wish to make it a checkout, use the `bind` command like this:

```
bzr bind bzr+ssh://centralhost/srv/bzr/PROJECT/trunk
```

This is necessary, for example, after creating a central branch using `push` as illustrated in the previous section.

After this, commits will be applied to the bound branch before being applied locally.

5.3.2 Turning a checkout into a branch

If you have a checkout and wish to make it a normal branch, use the `unbind` command like this:

```
bzr unbind
```

After this, commits will only be applied locally.

5.3.3 Getting a checkout

When working in a team using a central branch, one person needs to provide some initial content as shown in the previous section. After that, each person should use the `checkout` command to create their local checkout, i.e. the sandbox in which they will make their changes.

Unlike Subversion and CVS, in Bazaar the `checkout` command creates a local full copy of history in addition to creating a working tree holding the latest content. This means that operations such as `diff` and `log` are fast and can still be used when disconnected from the central location.

5.3.4 Getting a lightweight checkout

While Bazaar does its best to efficiently store version history, there are occasions when the history is simply not wanted. For example, if your team is managing the content of a web site using Bazaar with a central repository, then your release process might be as simple as updating a checkout of the content on the public web server. In this case, you probably don't want the history downloaded to that location as doing so:

- wastes disk space holding history that isn't needed there
- exposes a Bazaar branch that you may want kept private.

To get a history-less checkout in Bazaar, use the `--lightweight` option like this:

```
bzr checkout --lightweight bzr+ssh://centralhost/srv/bzr/PROJECT/trunk
```

Of course, many of the benefits of a normal checkout are lost by doing this but that's a tradeoff you can make if and when it makes sense.

The `--lightweight` option only applies to checkouts, not to all branches.

Note: If your code base is really large and disk space on your computer is limited, lightweight checkouts may be the right choice for you. Be sure to consider all your options though including shared repositories, stacked branches, and reusing a checkout.

5.3.5 Updating to the latest content

One of the important aspects of working in lockstep with others is keeping your checkout up to date with the latest changes made to the central branch. Just as you would in Subversion or CVS, you do this in Bazaar by using the `update` command like this:

```
bzr update
```

This gets any new revisions available in the bound branch and merges your local changes, if any.

5.3.6 Handling commit failures

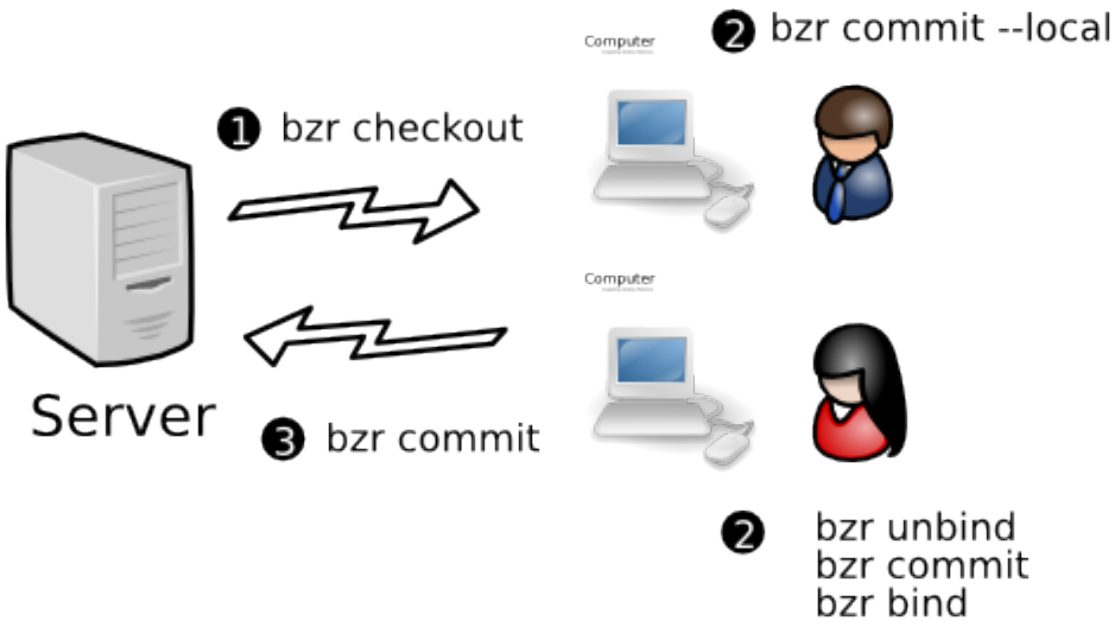
Note that your checkout *must* be up to date with the bound branch before running `commit`. Bazaar is actually stricter about this than Subversion or CVS - you need to be up to date with the full tree, not just for the files you've changed. Bazaar will ask you to run `update` if it detects that a revision has been added to the central location since you last updated.

If the network connection to the bound branch is lost, the commit will fail. Some alternative ways of working around that are outlined next.

5.4 Working offline on a central branch

5.4.1 The centralized with local commits workflow

If you lose your network connection because you are travelling, the central server goes down, or you simply want to snapshot changes locally without publishing them centrally just yet, this workflow is for you.



5.4.2 Committing locally

If you're working in a checkout and need/wish to commit locally only, add the `--local` option to the `commit` command like this:

```
bzr commit --local
```

5.4.3 Being disconnected for long time periods

If you will be or want to be disconnected from the bound branch for a while, then remembering to add `--local` to every `commit` command can be annoying. An alternative is to use the `unbind` command to make the checkout temporarily into a normal branch followed by the `bind` command at some later point in time when you want to keep in lockstep again.

Note that the `bind` command remembers where you were bound to last time this branch was a checkout so it isn't necessary to enter the URL of the remote branch when you use `bind` after an earlier `unbind`.

5.4.4 Merging a series of local commits

When you make commits locally independent of ongoing development on a central branch, then Bazaar treats these as two lines of development next time you `update`. In this case, `update` does the following:

- it brings the latest revisions from the bound branch down and makes that the mainline of development within your checkout
- it moves your local changes since you last updated into a logical parallel branch
- it merges these together so that your local changes are reported as a pending merge by `status`.

As always, you will need to run `commit` after this to send your work to the central branch.

5.5 Reusing a checkout

5.5.1 Motivation

At times, it can be useful to have a single checkout as your sandbox for working on multiple branches. Some possible reasons for this include:

- saving disk space when the working tree is large
- developing in a fixed location.

In many cases, working tree disk usage swamps the size of the `.bzr` directory. If you want to work on multiple branches but can't afford the overhead of a full working tree for each, reusing a checkout across multiples branches is the way to go.

On other occasions, the location of your sandbox might be configured into numerous development and testing tools. Once again, reusing a checkout across multiple branches can help.

5.5.2 Changing where a branch is bound to

To change where a checkout is bound to, follow these steps:

1. Make sure that any local changes have been committed centrally so that no work is lost.
2. Use the `bind` command giving the URL of the new remote branch you wish to work on.
3. Make your checkout a copy of the desired branch by using the `update` command followed by the `revert` command.

Note that simply binding to a new branch and running `update` merges in your local changes, both committed and uncommitted. You need to decide whether to keep them or not by running either `revert` or `commit`.

An alternative to the bind+update recipe is using the `switch` command. This is basically the same as removing the existing branch and running `checkout` again on the new location, except that any uncommitted changes in your tree are merged in.

Note: As `switch` can potentially throw away committed changes in order to make a checkout an accurate cache of a different bound branch, it will fail by design if there are changes which have been committed locally but are not yet committed to the most recently bound branch. To truly abandon these changes, use the `--force` option.

5.5.3 Switching a lightweight checkout

With a lightweight checkout, there are no local commits and `switch` effectively changes which branch the working tree is associated with. One possible setup is to use a lightweight checkout in combination with a local tree-less repository. This lets you switch what you are working on with ease. For example:

```
bzr init-repo --no-trees PROJECT
cd PROJECT
bzr branch bzr+ssh://centralhost/srv/bzr/PROJECT/trunk
bzr checkout --lightweight trunk my-sandbox
cd my-sandbox
(hack away)
```

Note that trunk in this example will have a `.bzr` directory within it but there will be no working tree there as the branch was created in a tree-less repository. You can grab or create as many branches as you need there and switch between them as required. For example:

```
(assuming in my-sandbox)
bzr branch bzr+ssh://centralhost/srv/bzr/PROJECT/PROJECT-1.0 ../PROJECT-1.0
bzr switch ../PROJECT-1.0
(fix bug in 1.0)
bzr commit -m "blah, blah blah"
bzr switch ../trunk
(go back to working on the trunk)
```

Note: The branches may be local only or they may be bound to remote ones (by creating them with `checkout` or by using `bind` after creating them with `branch`).

CHAPTER

SIX

TEAM COLLABORATION, DISTRIBUTED STYLE

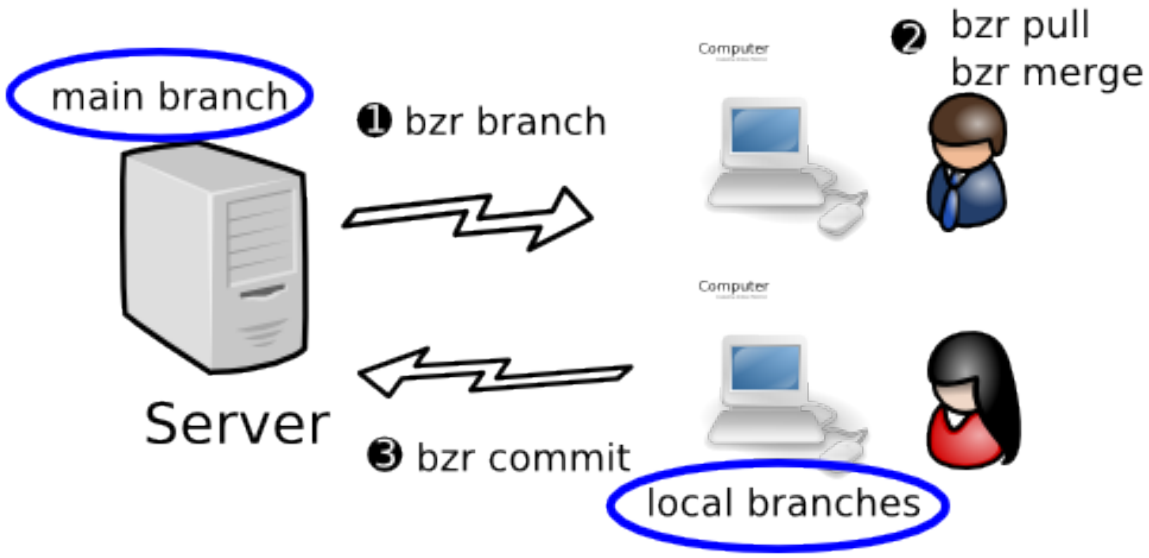
6.1 Distributed development

6.1.1 Motivation

Distributed VCS tools offer new ways of working together, ways that better reflect the modern world we live in and ways that enable higher quality outcomes.

6.1.2 The decentralized with shared mainline workflow

In this workflow, each developer has their own branch or branches, plus a checkout of the main branch. They do their work in their personal branch, then merge it into the mainline when it is ready.



Other distributed workflows are explored later in this chapter.

6.2 Organizing branches

6.2.1 Mirror branches

A primary difference when using distributed workflows to develop is that your main local branch is not the place to make changes. Instead, it is kept as a pristine copy of the central branch, i.e. it's a *mirror branch*.

To create a mirror branch, set-up a shared repository (if you haven't already) and then use the `branch` (or `checkout`) command to create the mirror. For example:

```
bzr init-repo PROJECT
cd PROJECT
bzr branch bzr+ssh://centralhost/srv/bzr/PROJECT/trunk
```

6.2.2 Task branches

Each new feature or fix is developed in its own branch. These branches are referred to as *feature branches* or *task branches* - the terms are used interchangeably.

To create a task branch, use the `branch` command against your mirror branch. For example:

```
bzr branch trunk fix-123
cd fix-123
(hack, hack, hack)
```

There are numerous advantages to this approach:

1. You can work on multiple changes in parallel
2. There is reduced coupling between changes
3. Multiple people can work in a peer-to-peer mode on a branch until it is ready to go.

In particular, some changes take longer to cook than others so you can ask for reviews, apply feedback, ask for another review, etc. By completing work to sufficient quality in separate branches before merging into a central branch, the quality and stability of the central branch are maintained at higher level than they otherwise would be.

6.2.3 Refreshing a mirror branch

Use the `pull` command to do this:

```
cd trunk
bzr pull
```

6.2.4 Merging the latest trunk into a feature branch

Use the `merge` command to do this:

```
cd fix-123
bzr merge
(resolve any conflicts)
bzr commit -m "merged trunk"
```

6.2.5 Merging a feature into the trunk

The policies for different distributed workflows vary here. The simple case where all developers have commit rights to the main trunk are shown below.

If your mirror is a checkout:

```
cd trunk
bzr update
bzr merge ../fix-123
(resolve any conflicts)
bzr commit -m "Fixed bug #123"
```

If your mirror is a branch:

```
cd trunk
bzr pull
bzr merge ../fix-123
(resolve any conflicts)
bzr commit -m "Fixed bug #123"
bzr push
```

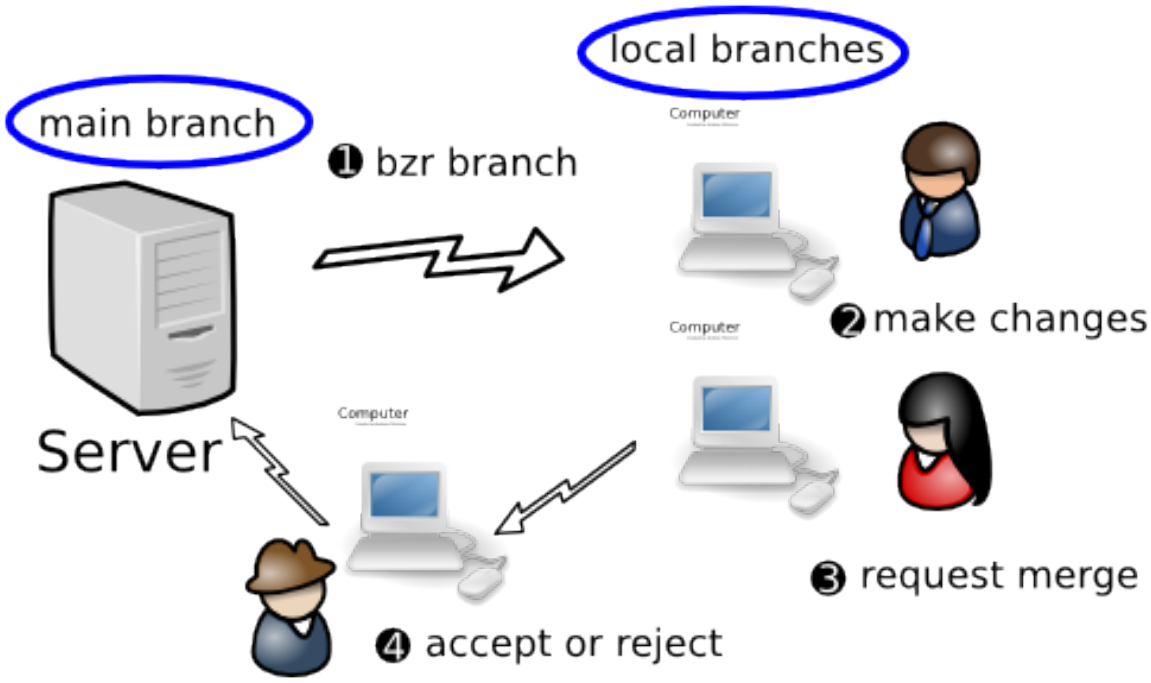
6.2.6 Backing up task branches

One of the side effects of centralized workflows is that changes get frequently committed to a central location which is backed up as part of normal IT operations. When developing on task branches, it is a good idea to publish your work to a central location (but not necessarily a shared location) that will be backed up. You may even wish to bind local task branches to remote ones established on a backup server just for this purpose.

6.3 Using gatekeepers

6.3.1 The decentralized with human gatekeeper workflow

In this workflow, one developer (the gatekeeper) has commit rights to the main branch while other developers have read-only access. All developers make their changes in task branches.



When a developer wants their work merged, they ask the gatekeeper to review their change and merge it if acceptable. If a change fails review, further development proceeds in the relevant task branch until it is good to go.

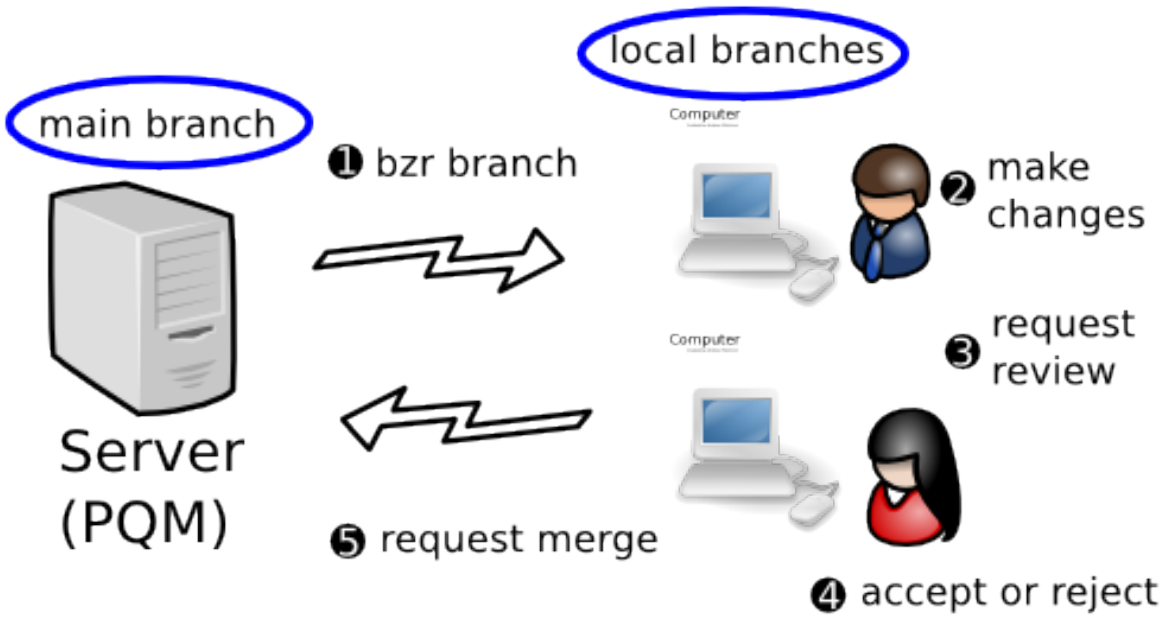
Note that a key aspect of this approach is the inversion of control that is implied: developers no longer decide when to "commit/push" changes into the central branch: the code base evolves by gatekeepers "merging/pulling" changes in a controlled manner. It's perfectly acceptable, indeed common, to have multiple central branches with different gatekeepers, e.g. one branch for the current production release and another for the next release. In this case, a task branch holding a bug fix will most likely be advertised to both gatekeepers.

One of the great things about this workflow is that it is hugely scalable. Large projects can be broken into teams and each team can have a *local master branch* managed by a gatekeeper. Someone can be appointed as the primary

gatekeeper to merge changes from the team master branches into the primary master branch when team leaders request it.

6.3.2 The decentralized with automatic gatekeeper workflow

To obtain even higher quality, all developers can be required to submit changes to an automated gatekeeper that only merges and commits a change if it passes a regression test suite. One such gatekeeper is a software tool called PQM.



For further information on PQM, see https://launchpad.net/pqm.

6.4 Sending changes

6.4.1 Motivation

In many distributed development scenarios, it isn't always feasible for developers to share task branches by advertising their URLs. For example, a developer working on a laptop might take it home overnight so his/her task branches could well be inaccessible when a gatekeeper in another timezone wants to review or merge it.

Bazaar provides a neat feature to assist here: *merge directives*.

6.4.2 Understanding merge directives

You can think of a merge directive as a "mini branch" - just the new growth on a branch since it was created. It's a software patch showing what's new but with added intelligence: metadata like interim commits, renames and digital signatures.

Another useful metaphor is a packet cake: a merge directive has a recipe together with the ingredients you need bundled inside it. To stretch the metaphor, the ingredients are all the metadata on the changes made to the branch; the recipe is instructions on how those changes ought to be merged, i.e. information for the `merge` command to use in selecting common ancestors.

Regardless of how you think of them, merge directives are neat. They are easy to create, suitable for mailing around as attachments and can be processed much like branches can on the receiving end.

6.4.3 Creating a merge directive

To create and optionally send a merge directive, use the `send` command.

By default, `send` will email the merge directive to the "submission address" for the branch, which is typically the lead developer or the development mailing list. `send` without options will create a merge directive, fire up your email tool and attach it, ready for you to add the explanatory text bit. (See the online help for `send` and Configuration Settings in the User Reference for further details on how to configure this.)

Most projects like people to add some explanation to the mail along with the patch, explaining the reason for the patch, and why it is done the way it is. This gives a reviewer some context before going into the line-by-line diff.

Alternatively, if the `--output` (or `-o`) option is given, `send` will write the merge directive to a file, so you can mail it yourself, examine it, or save it for later use. If an output file of `-` is given, the directive is written to stdout. For example:

```
cd X-fix-123
bzr send -o ../fix-123.patch
```

6.4.4 Applying a merge directive

Merge directives can be applied in much the same way as branches: by using the `merge` and `pull` commands.

They can also be useful when communicating with upstream projects that don't use Bazaar. In particular, the preview of the overall change in a merge directive looks like a vanilla software patch, so they can be applied using `patch -p0` for example.

CHAPTER

SEVEN

MISCELLANEOUS TOPICS

7.1 The journey ahead

We hope that earlier chapters have given you a solid understanding of how Bazaar can assist you in being productive on your own and working effectively with others. If you are learning Bazaar for the first time, it might be good to try the procedures covered already for a while, coming back to this manual once you have mastered them.

Remaining chapters covers various topics to guide you in further optimizing how you use Bazaar. Unless stated otherwise, the topics in this and remaining chapters are independent of each other and can therefore be read in whichever order you wish.

7.2 Pseudo merging

7.2.1 Cherrypicking

At times, it can be useful to selectively merge some of the changes in a branch, but not all of them. This is commonly referred to as *cherrypicking*. Here are some examples of where cherrypicking is useful:

- selectively taking fixes from the main development branch into a release branch
- selectively taking improvements out of an experimental branch into a feature branch.

To merge only the changes made by revision X in branch `foo`, the command is:

```
bzr merge -c X foo
```

To merge only the changes up to revision X in branch `foo`, the command is:

```
bzr merge -r X foo
```

To merge only the changes since revision X in branch `foo`, the command is:

```
bzr merge -r X.. foo
```

To merge only the changes from revision X to revision Y in branch `foo`, the command is:

```
bzr merge -r X..Y foo
```

Like a normal merge, you must explicitly commit a cherrypick. You may wish to see the changes made using `bzr diff`, and run your test suite if any, before doing this.

Unlike a normal merge, Bazaar does not currently track cherrypicks. In particular, the changes look like a normal commit and the (internal) revision history of the changes from the other branch is lost. In many cases where they are

useful (see above), this is not a major problem because there are good reasons why a full merge should never be done at a later time. In other cases, additional conflicts will need to be resolved when the changes are merged again.

7.2.2 Merging without parents

A related technique to cherrypicking, in that it makes changes without reference to the revisions that they came from is to perform a merge, but forget about the parent revisions before committing. This has the effect of making all of the changes that would have been in the merge happen in a single commit. After the merge and before the corresponding commit, you can do:

```
bzr revert --forget-merges
```

to keep the changes in the working tree, but remove the record of the revisions where the changes originated. The next commit would then record all of those changes without any record of the merged revisions.

This is desired by some users to make their history "cleaner", but you should be careful that the loss of history does not outweigh the value of cleanliness, particularly given Bazaar's capabilities for progressively disclosing merged revisions. In particular, because this will include the changes from the source branch, but without attribution to that branch, it can lead to additional conflicts on later merges that involve the same source and target branches.

7.2.3 Reverse cherrypicking

Cherrypicking can be used to reverse a set of changes made by giving an upper bound in the revision range which is *below* the lower bound. For example, to back-out changes made in revision 10, the command is:

```
bzr merge -r 10..9
```

If you want to take most changes, but not all, from somewhere else, you may wish to do a normal merge followed by a few reverse cherrypicks.

7.2.4 Merging uncommitted changes

If you have several branches and you accidentally start making changes in the wrong one, here are the steps to take to correct this. Assuming you began working in branch `foo` when you meant to work in branch `bar`:

1. Change into branch `bar`.
2. Run `bzr merge --uncommitted foo`
3. Check the changes came across (`bzr diff`)
4. Change into branch `foo`
5. Run `bzr revert`.

7.2.5 Rebasing

Another option to normal merging is *rebasing*, i.e. making it look like the current branch originated from a different point than it did. Rebasing is supported in Bazaar by the `rebase` command provided by the `rebase` plugin.

The `rebase` command takes the location of another branch on which the branch in the current working directory will be rebased. If a branch is not specified then the parent branch is used, and this is usually the desired result.

The first step identifies the revisions that are in the current branch that are not in the parent branch. The current branch is then set to be at the same revision as the target branch, and each revision is replayed on top of the branch. At the end of the process it will appear as though your current branch was branched off the current last revision of the target.

Each revision that is replayed may cause conflicts in the tree. If this happens the command will stop and allow you to fix them up. Resolve the commits as you would for a `merge`, and then run `bzr resolve` to marked them as resolved. Once you have resolved all the conflicts, you should run `bzr rebase-continue` to continue the rebase operation. If conflicts are encountered and you decide not to continue, you can run `bzr rebase-abort`. You can also use `rebase-todo` to show the list of commits still to be replayed.

Note: Some users coming from central VCS tools with poor merge tracking like rebasing because it's similar to how they are use to working in older tools, or because "perfectly clean" history seems important. Before rebasing in Bazaar, think about whether a normal merge is a better choice. In particular, rebasing a private branch before sharing it is OK but rebasing after sharing a branch with someone else is **strongly** discouraged.

7.3 Switch –store

In workflows that a single working tree, like co-located branches, sometimes you want to switch while you have uncommitted changes. By default, `switch` will apply your uncommitted changes to the new branch that you switch to. But often you don't want that. You just want to do some work in the other branch, and eventually return to this branch and work some more.

You could run `bzr shelve --all` before switching, to store the changes safely. So you have to know that there are uncommitted changes present, and you have to remember to run `bzr shelve --all`. Then when you switch back to the branch, you need to remember to unshelve the changes, and you need to know what their shelf-id was.

Using `switch --store` takes care of all of this for you. If there are any uncommitted changes in your tree, it stores them in your branch. It then restores any uncommitted changes that were stored in the branch of your target tree. It's almost like having two working trees and using `cd` to switch between them.

To take an example, first we'd set up a co-located branch:

```
$ bzr init foo
Created a standalone tree (format: 2a)
$ cd foo
$ bzr switch -b foo
```

Now create committed and uncommitted changes:

```
$ touch committed
$ bzr add
adding committed
$ bzr commit -m "Add committed"
Committing to: /home/abentley/sandbox/foo/
added committed
Committed revision 1.
$ touch uncommitted
$ bzr add
adding uncommitted
$ ls
committed  uncommitted
```

Now create a new branch using `--store`. The uncommitted changes are stored in "foo", but the committed changes are retained.

```
$ bzr switch -b --store bar
Uncommitted changes stored in branch "foo".
Tree is up to date at revision 1.
Switched to branch: /home/abentley/sandbox/foo/
abentley@speedy:~/sandbox/foo$ ls
committed
```

Now, create uncommitted changes in "bar":

```
$ touch uncommitted-bar
$ bzr add
adding uncommitted-bar
```

Finally, switch back to "foo":

```
$ bzr switch --store foo
Uncommitted changes stored in branch "bar".
Tree is up to date at revision 1.
Switched to branch: /home/abentley/sandbox/foo/
$ ls
committed  uncommitted
```

Each branch holds only one set of stored changes. If you try to store a second set, you get an error. If you use `--store` all the time, this can't happen. But if you use plain switch, then it won't restore the uncommitted changes already present:

```
$ bzr switch bar
Tree is up to date at revision 1.
Switched to branch: /home/abentley/sandbox/foo/
$ bzr switch --store foo
bzr: ERROR: Cannot store uncommitted changes because this branch already
stores uncommitted changes.
```

If you're working in a branch that already has stored changes, you can restore them with `bzr switch . --store`:

```
$ bzr shelve --all -m "Uncommitted changes from foo"
Selected changes:
-D  uncommitted
Changes shelved with id "1".
$ bzr switch . --store
Tree is up to date at revision 1.
Switched to branch: /home/abentley/sandbox/foo/
$ ls
committed  uncommitted-bar
```

7.4 Shelving Changes

Sometimes you will want to temporarily remove changes from your working tree and restore them later, For instance to commit a small bug-fix you found while working on something. Bazaar allows you to put changes on a `shelf` to achieve this. When you want to restore the changes later you can use `unshelve` to apply them to your working tree again.

For example, consider a working tree with one or more changes made ...

```
$ bzr diff
=== modified file 'description.txt'
--- description.txt
+++ description.txt
@@ -2,7 +2,7 @@
 ===============

 These plugins
-by Michael Ellerman
```

```
+written by Michael Ellerman
 provide a very
 fine-grained 'undo'
 facility
@@ -11,6 +11,6 @@
 This allows you to
 undo some of
 your changes,
-commit, and get
+perform a commit, and get
 back to where you
 were before.
```

The `shelve` command interactively asks which changes you want to retain in the working tree:

```
$ bzr shelve
--- description.txt
+++ description.txt
@@ -2,7 +2,7 @@
 ===============

 These plugins
-by Michael Ellerman
+written by Michael Ellerman
 provide a very
 fine-grained 'undo'
 facility

Shelve? [yNfrq?]: y
--- description.txt
+++ description.txt
@@ -11,6 +11,6 @@
 This allows you to
 undo some of
 your changes,
-commit, and get
+perform a commit, and get
 back to where you
 were before.

Shelve? [yNfrq?]: n
Shelve 2 change(s)? [yNfrq?]', 'y'
Selected changes:
 M  description.txt
Changes shelved with id "1".
```

If there are lots of changes in the working tree, you can provide the `shelve` command with a list of files and you will only be asked about changes in those files. After shelving changes, it's a good idea to use `diff` to confirm the tree has just the changes you expect:

```
$ bzr diff
=== modified file 'description.txt'
--- description.txt
+++ description.txt
@@ -2,7 +2,7 @@
 ===============

 These plugins
-by Michael Ellerman
```

```
+written by Michael Ellerman
 provide a very
 fine-grained 'undo'
 facility
```

Great - you're ready to commit:

```
$ bzr commit -m "improve first sentence"
```

At some later time, you can bring the shelved changes back into the working tree using `unshelve`:

```
$ bzr unshelve
Unshelving changes with id "1".
 M  description.txt
All changes applied successfully.
```

If you want to, you can put multiple items on the shelf. Normally each time you run `unshelve` the most recently shelved changes will be reinstated. However, you can also unshelve changes in a different order by explicitly specifying which changes to unshelve.

Bazaar merges the changes in to your working tree, so they will apply even if you have edited the files since you shelved them, though they may conflict, in which case you will have to resolve the conflicts in the same way you do after a conflicted merge.

7.5 Filtered views

7.5.1 Introducing filtered views

Views provide a mask over the tree so that users can focus on a subset of a tree when doing their work. There are several cases where this masking can be helpful. For example, technical writers and testers on many large projects may prefer to deal with just the directories/files in the project of interest to them.

Developers may also wish to break a large set of changes into multiple commits by using views. While `shelve` and `unshelve` let developers put some changes aside for a later commit, views let developers specify what to include in (instead of exclude from) the next commit.

After creating a view, commands that support a list of files - status, diff, commit, etc - effectively have that list of files implicitly given each time. An explicit list of files can still be given to these commands but the nominated files must be within the current view. In contrast, tree-centric commands - pull, merge, update, etc. - continue to operate on the whole tree but only report changes relevant to the current view. In both cases, Bazaar notifies the user each time it uses a view implicitly so that it is clear that the operation or output is being masked accordingly.

Note: Filtered views are only supported in format 2a, the default in Bazaar 2.0, or later.

7.5.2 Creating a view

This is done by specifying the files and directories using the `view` command like this:

```
bzr view file1 file2 dir1 ...
```

The output is:

```
Using 'my' view: file1, file2, dir1
```

7.5.3 Listing the current view

To see the current view, use the `view` command without arguments:

```
bzr view
```

If no view is current, a message will be output saying `No current view.`. Otherwise the name and content of the current view will be displayed like this:

```
'my' view is: a, b, c
```

7.5.4 Switching between views

In most cases, a view has a short life-span: it is created to make a selected change and is deleted once that change is committed. At other times, you may wish to create one or more named views and switch between them.

To define a named view and switch to it:

```
bzr view --name view-name file1 dir1 ...
```

For example:

```
bzr view --name doc NEWS doc/
Using doc view: NEWS, doc/
```

To list a named view:

```
bzr view --name view-name
```

To switch to a named view:

```
bzr view --switch view-name
```

To list all views defined:

```
bzr view --all
```

7.5.5 Temporarily disabling a view

To disable the current view without deleting it, you can switch to the pseudo view called `off`. This can be useful when you need to see the whole tree for an operation or two (e.g. merge) but want to switch back to your view after that.

To disable the current view without deleting it:

```
bzr view --switch off
```

After doing the operations you need to, you can switch back to the view you were using by name. For example, if the previous view used the default name:

```
bzr view --switch my
```

7.5.6 Deleting views

To delete the current view:

```
bzr view --delete
```

To delete a named view:

```
bzr view --name view-name --delete
```

To delete all views:

```
bzr view --delete --all
```

7.5.7 Things to be aware of

Defining a view does not delete the other files in the working tree - it merely provides a "lens" over the working tree.

Views are stored as working tree metadata. They are not propagated by branch commands like pull, push and update.

Views are defined in terms of file paths. If you move a file in a view to a location outside of the view, the view will no longer track that path. For example, if a view is defined as `doc/` and `doc/NEWS` gets moved to `NEWS`, the views stays defined as `doc/` and does not get changed to `doc/ NEWS`. Likewise, deleting a file in a view does not remove the file from that view.

The commands that use the current view are:

- status
- diff
- commit
- add
- remove
- revert
- mv
- ls.

Commands that operate on the full tree but only report changes inside the current view are:

- pull
- update
- merge.

Many commands currently ignore the current view. Over time, some of these commands may be added to the lists above as the need arises. By design, some commands will most likely always ignore the current view because showing the whole picture is the better thing to do. Commands in this group include:

- log
- info.

7.6 Using stacked branches

7.6.1 Motivation

If you are working on a project, and you have read access to whose public repository but do not have write access to it, using stacked branches to backup/publish your work onto the same host of the public repository might be an option

for you.

Other scenarios for stacked branch usage include experimental branches and code hosting sites. For these scenarios, stacked branches are ideal because of the benefits it provides.

7.6.2 What is a stacked branch?

A stacked branch is a branch that knows how to find revisions in another branch (the stacked-on branch). Stacked branches store just the unique revisions that are not in the stacked-on branch, making them faster to create and more storage efficient. In these respects, stacked branches are similar to shared repositories. However, stacked branches have additional benefits:

- The new branch can be in a completely different location to the branch being stacked on.
- Deleting the stacked branch really deletes the revisions (rather than leaving them in a shared repository).
- Security is improved over shared repositories, because the stacked-on repository can be physically readonly to developers committing to stacked branches.

7.6.3 Creating a stacked branch

To create a stacked branch, use the `stacked` option of the branch command. For example:

```
bzr branch --stacked source-url my-dir
```

This will create `my-dir` as a stacked branch with no local revisions. If it is defined, the public branch associated with `source-url` will be used as the *stacked-on* location. Otherwise, `source-url` will be the *stacked-on* location.

7.6.4 Creating a stacked checkout

Direct creation of a stacked checkout is expected to be supported soon. In the meantime, a two step process is required:

1. Create a stacked branch as shown above.
2. Convert the branch into a checkout using either the `reconfigure` or `bind` command.

7.6.5 Pushing a stacked branch

Most changes on most projects build on an existing branch such as the *development trunk* or *current stable* branch. Creating a new branch stacked on one of these is easy to do using the `push` command like this:

```
bzr push --stacked-on reference-url my-url
```

This creates a new branch at `my-url` that is stacked on `reference-url` and only contains the revisions in the current branch that are not already in the branch at `reference-url`. In particular, `my-url` and `reference-url` can be on the same host, and the `--stacked-on` option can be used additionally to inform `push` to reference the revisions in `reference-url`. For example:

```
bzr push --stacked-on bzr+ssh://host/project bzr+ssh://host/user/stacked-branch
```

This usage fits the scenario described in the Motivation section.

You can also use the `--stacked` option without specifying `--stacked-on`. This will automatically set the *stacked-on* location to the parent branch of the branch you are pushing (or its `public_location` if configured). For example:

```
bzr branch source-url my-dir
cd my-dir
(hack, hack, hack)
bzr commit -m "fix bug"
bzr push --stacked
```

You can combine `bzr branch --stacked` and `bzr push --stacked` to work on a branch without downloading or uploading the whole history:

```
bzr branch --stacked source-url my-dir
cd my-dir
(hack, hack, hack)
bzr commit -m "fix bug"
bzr push --stacked
```

7.6.6 Limitations of stacked branches

The important thing to remember about a stacked branch is that the stacked-on branch needs to be accessible for almost all operations. This is not an issue when both branches are local, or when both branches are on the same server and the stacked-on location is a relative path. But clearly a branch hosted on a server with a stacked-on location of `file:///...` is not going to work for anyone except the user that originally pushed it. It's a good idea to configure `public_location` to help prevent that.

Similarly, because most of the history is stored in the stacked-on repository, operations like `bzr log` can be slower when the stacked-on repository is accessed via a network.

If a stacked branch is in a format older than 2a, you cannot commit to it due to bug 375013.

7.6.7 Changing branch stacking

Stacking of existing branches can be changed using the `bzr reconfigure` command to either stack on an existing branch, or to turn off stacking. Be aware that when `bzr reconfigure --unstacked` is used, bzr will copy all the referenced data from the stacked-on repository into the previously stacked repository. For large repositories this may take considerable time and may substantially increase the size of the repository.

7.7 Running a smart server

Bazaar does not require a specialised server because it operates over HTTP, FTP or SFTP. There is an optional smart server that can be invoked over SSH, from inetd, or in a dedicated mode.

7.7.1 Dumb servers

We describe HTTP, FTP, SFTP and HTTP-WebDAV as "dumb" servers because they do not offer any assistance to Bazaar. If you make a Bazaar repository available over any of these protocols, Bazaar will allow you to read it remotely. Just enter the URL to the branch in the Bazaar command you are running.:

```
bzr log http://bazaar.launchpad.net/~bzr-pqm/bzr/bzr.dev
```

Bazaar supports writing over FTP, SFTP and (via a plugin) over HTTP-WebDAV.

7.7.2 High-performance smart server

The high-performance smart server (hpss) performs certain operations much faster than dumb servers are capable of. In future releases, the range of operations that are improved by using the smart server will increase as we continue to tune performance.

To maintain the highest security possible, the current smart server provides read-only access by default. To enable read-write access, run it with `--allow-writes`. When using the SSH access method, bzr automatically runs with the `--allow-writes` option.

The alternative ways of configuring a smart server are explained below.

SSH

Using Bazaar over SSH requires no special configuration on the server; so long as Bazaar is installed on the server you can use `bzr+ssh` URLs, e.g.:

```
bzr log bzr+ssh://host/path/to/branch
```

If *bzr* is not installed system-wide on the server you may need to explicitly tell the local *bzr* where to find the remote *bzr*:

```
BZR_REMOTE_PATH=~/bin/bzr bzr log bzr+ssh://host/path/to/branch
```

The `BZR_REMOTE_PATH` environment variable adjusts how *bzr* will be invoked on the remote system. By default, just *bzr* will be invoked, which requires the *bzr* executable to be on the default search path. You can also set this permanently per-location in `locations.conf`.

Like SFTP, paths starting with `~` are relative to your home directory, e.g. `bzr+ssh://example.com/~/code/proj`. Additionally, paths starting with `~user` will be relative to that user's home directory.

inetd

This example shows how to run *bzr* with a dedicated user *bzruser* for a shared repository in `/srv/bzr/repo` which has a branch at `/srv/bzr/repo/branchname`.

Running a Bazaar server from inetd requires an inetd.conf entry:

```
4155  stream  TCP  nowait  bzruser  /usr/bin/bzr /usr/bin/bzr serve --inet --directory=/srv/bzr/repo
```

When running client commands, the URL you supply is a *bzr://* URL relative to the `--directory` option given in inetd.conf:

```
bzr log bzr://host/branchname
```

If possible, paths starting with `~` and `~user` will be expanded as for `bzr+ssh`. Home directories outside the `--directory` specified to `bzr serve` will not be accessible.

Dedicated

This mode has the same path and URL behaviour as the inetd mode. To run as a specific user, you should use `su` or login as that user.

This example runs bzr on its official port number of *4155* and listens on all interfaces. This allows connections from anywhere in the world that can reach your machine on port *4155*.

server:

```
bzr serve --directory=/srv/bzr/repo
```

client:

```
bzr log bzr://host/branchname
```

This example runs `bzr serve` on *localhost* port *1234*.

server:

```
bzr serve --listen=localhost --port=1234 --directory=/srv/bzr/repo
```

client:

```
bzr log bzr://localhost:1234/branchname
```

7.8 Using hooks

7.8.1 What is a hook?

One way to customize Bazaar's behaviour is with *hooks*. Hooks allow you to perform actions before or after certain Bazaar operations. The operations include `commit`, `push`, `pull`, and `uncommit`. For a complete list of hooks and their parameters, see Hooks in the User Reference.

Most hooks are run on the client, but a few are run on the server. (Also see the push-and-update plugin that handles one special case of server-side operations.)

7.8.2 Using hooks

To use a hook, you should write a plugin. Instead of creating a new command, this plugin will define and install the hook. Here's an example:

```
from bzrlib import branch


def post_push_hook(push_result):
    print "The new revno is %d" % push_result.new_revno


branch.Branch.hooks.install_named_hook('post_push', post_push_hook,
                                       'My post_push hook')
```

To use this example, create a file named `push_hook.py`, and stick it in `plugins` subdirectory of your configuration directory. (If you have never installed any plugins, you may need to create the `plugins` directory).

That's it! The next time you push, it should show "The new revno is...". Of course, hooks can be much more elaborate than this, because you have the full power of Python at your disposal. Now that you know how to use hooks, what you do with them is up to you.

The plugin code does two things. First, it defines a function that will be run after `push` completes. (It could instead use an instance method or a callable object.) All push hooks take a single argument, the `push_result`.

Second, the plugin installs the hook. The first argument `'post_push'` identifies where to install the hook. The second argument is the hook itself. The third argument is a name `'My post_push hook'`, which can be used in progress messages and error messages.

To reduce the start-up time of Bazaar it is also possible to "lazily" install hooks, using the `bzrlib.hooks.install_lazy_named_hook` function. This removes the need to load the module that contains the hook point just to install the hook. Here's lazy version of the example above:

```
from bzrlib import hooks

def post_push_hook(push_result):
    print "The new revno is %d" % push_result.new_revno


hooks.install_lazy_named_hook('bzrlib.branch', 'Branch.hooks',
    'post_push', post_push_hook, 'My post_push hook')
```

7.8.3 Debugging hooks

To get a list of installed hooks (and available hook points), use the hidden `hooks` command:

```
bzr hooks
```

7.8.4 Example: a merge plugin

Here's a complete plugin that demonstrates the `Merger.merge_file_content` hook. It installs a hook that forces any merge of a file named `*.xml` to be a conflict, even if Bazaar thinks it can merge it cleanly.

`merge_xml.py`:

```
"""Custom 'merge' logic for *.xml files.

Always conflicts if both branches have changed the file.
"""

from bzrlib.merge import PerFileMerger, Merger

def merge_xml_files_hook(merger):
    """Hook to merge *.xml files"""
    return AlwaysConflictXMLMerger(merger)

class AlwaysConflictXMLMerger(PerFileMerger):

    def file_matches(self, params):
        filename = self.get_filename(params, self.merger.this_tree)
        return filename.endswith('.xml')

    def merge_matching(self, params):
        return 'conflicted', params.this_lines

Merger.hooks.install_named_hook(
    'merge_file_content', merge_xml_files_hook, '*.xml file merge')
```

`merge_file_content` hooks are executed for each file to be merged. For a more a complex example look at the `news_merge` plugin that's bundled with Bazaar in the `bzrlib/plugins` directory.

7.9 Exporting version information

7.9.1 Getting the last revision number

If you only need the last revision number in your build scripts, you can use the `revno` command to get that value like this:

```
$ bzr revno
3104
```

7.9.2 Getting more version information

The `version-info` command can be used to output more information about the latest version like this:

```
$ bzr version-info
revision-id: pqm@pqm.ubuntu.com-20071211175118-s94sizduj201hrs5
date: 2007-12-11 17:51:18 +0000
build-date: 2007-12-13 13:14:51 +1000
revno: 3104
branch-nick: bzr.dev
```

You can easily filter that output using operating system tools or scripts. For example:

```
$ bzr version-info | grep ^date
date: 2007-12-11 17:51:18 +0000
```

The `--all` option will actually dump version information about every revision if you need that information for more advanced post-processing.

7.9.3 Python projects

If using a Makefile to build your project, you can generate the version information file as simply as:

```
library/_version.py:
      bzr version-info --format python > library/_version.py
```

This generates a file which contains 3 dictionaries:

- *version_info*: A dictionary containing the basic information about the current state.
- *revisions*: A dictionary listing all of the revisions in the history of the tree, along with the commit times and commit message. This defaults to being empty unless `--all` or `--include-history` is supplied. This is useful if you want to track what bug fixes, etc, might be included in the released version. But for many projects it is more information than needed.
- *file_revisions*: A dictionary listing the last-modified revision for all files in the project. This can be used similarly to how `$Id$` keywords are used in CVS-controlled files. The last modified date can be determined by looking in the `revisions` map. This is also empty by default, and enabled only by `--all` or `--include-file-revisions`.

7.9.4 Getting version info in other formats

Bazaar supports a template-based method for getting version information in arbitrary formats. The `--custom` option to `version-info` can be used by providing a `--template` argument that contains variables that will be expanded

based on the status of the working tree. For example, to generate a C header file with a formatted string containing the current revision number:

```
bzr version-info --custom \
     --template="#define VERSION_INFO \"Project 1.2.3 (r{revno})\"\n" \
     > version_info.h
```

where the `{revno}` will be replaced by the revision number of the working tree. (If the example above doesn't work on your OS, try entering the command all on one line.) For more information on the variables that can be used in templates, see Version Info in the Bazaar User Reference.

Predefined formats for dumping version information in specific languages are currently in development. Please contact us on the mailing list about your requirements in this area.

7.9.5 Check clean

Most information about the contents of the project can be cheaply determined by just reading the revision entry. However, it can be useful to know if the working tree was completely up-to-date when it was packaged, or if there was a local modification. By supplying either `--all` or `--check-clean`, `bzr` will inspect the working tree, and set the `clean` flag in `version_info`, as well as set entries in `file_revisions` as `modified` where appropriate.

7.10 GnuPG Signatures

7.10.1 Reasons to Sign Your Repository

Bazaar can sign revisions using GnuPG, a Free Software implementation of the OpenPGP digital signature format. By signing commits a person wanting to make use of a branch can be confident where the code came from, assuming the GnuPG keys used can be verified. This could for example prevent worry about compromised code in the case where a server hosting Bazaar branches has been hacked into. It could also be used to verify that all code is written by a select group of people, such as if contributor agreements are needed.

Signatures are passed around with commits during branch, push, merge and other operations.

7.10.2 Setting up GnuPG

There are many guides to creating a digital signature key with GnuPG. See for example the GnuPG Handbook or the Launchpad Wiki.

7.10.3 Signing Commits

To sign commits as they are made turn on the `create_signatures` configuration option in your `bazaar.conf` or `locations.conf` file:

```
create_signatures = always
```

When you next make a commit it will ask for the pass phrase for your GnuPG key. If you want GnuPG to remember your password ensure you have `gnupg-agent` installed.

To sign previous commits to a branch use `sign-my-commits`. This will go through all revisions in the branch and sign any which match your commit name. You can also pass the name of a contributor to `sign-my-commits` to sign someone else's commits or if your GnuPG key does not match your Bazaar name and e-mail:

```
bzr sign-my-commits . "Amy Pond <amy@example.com>"
```

It will not sign commits which already have a signature.

To sign a single commit or a range of commits use the (hidden) command `re-sign`:

```
bzr re-sign -r 24
```

`re-sign` is also useful to change an existing signature.

By default Bazaar will tell GnuPG to use a key with the same user identity as the one set with `whoami`. To override this set `gpg_signing_key` in bazaar.conf or locations.conf.

```
gpg_signing_key=DD4D5088
```

```
gpg_signing_key=amy@example.com
```

7.10.4 Verifying Commits

Signatures can be verified with the `bzr verify-signatures` command. By default this will check all commits in the branch and notify that all commits are signed by known trusted signatures. If not all commits have trusted signatures it will give a summary of the number of commits which are invalid, having missing keys or are not signed.

The `verify-signatures` command can be given a comma separated list of key patters to specify a list of acceptable keys. It can also take a range of commits to verify in the current branch. Finally using the verbose option will list each key that is valid or authors for commits which failed:

```
$bzr verify-signatures -kamy -v -r 1..5
1 commit with valid signature
  Amy Pond <amy@example.com> signed 4 commits
0 commits with unknown keys
1 commit not valid
  1 commit by author The Doctor <doctor@example.com>
0 commits not signed
```

7.10.5 Work in Progress

There is still a number of digital signature related features which are hoped to be added to Bazaar soon. These include bzr explorer integration and setting branches to require signatures.

CHAPTER

EIGHT

A BRIEF TOUR OF SOME POPULAR PLUGINS

8.1 BzrTools

8.1.1 Overview

BzrTools is a collection of useful enhancements to Bazaar. For installation instructions, see the BzrTools home page: http://wiki.bazaar.canonical.com/BzrTools. Here is a sample of the frequently used commands it provides.

8.1.2 shell

`bzr shell` starts up a command interpreter than understands Bazaar commands natively. This has several advantages:

- There's no need to type `bzr` at the front of every command.
- Intelligent auto-completion is provided.
- Commands run slightly faster as there's no need to load Bazaar's libraries each time.

8.1.3 cdiff

`bzr cdiff` provides a colored version of `bzr diff` output. On GNU/Linux, UNIX and OS X, this is often used like this:

```
bzr cdiff | less -R
```

8.2 bzr-svn

8.2.1 Overview

bzr-svn lets developers use Bazaar as their VCS client on projects still using a central Subversion repository. Access to Subversion repositories is largely transparent, i.e. you can use most `bzr` commands directly on Subversion repositories exactly the same as if you were using `bzr` on native Bazaar branches.

Many bzr-svn users create a local mirror of the central Subversion trunk, work in local feature branches, and submit their overall change back to Subversion when it is ready to go. This lets them gain many of the advantages of distributed

VCS tools without interrupting existing team-wide processes and tool integration hooks currently built on top of Subversion. Indeed, this is a common interim step for teams looking to adopt Bazaar but who are unable to do so yet for timing or non-technical reasons.

For installation instructions, see the bzr-svn home page: http://wiki.bazaar.canonical.com/BzrForeignBranches/Subversion.

8.2.2 A simple example

Here's a simple example of how you can use bzr-svn to hack on a GNOME project like **beagle**. Firstly, setup a local shared repository for storing your branches in and checkout the trunk:

```
bzr init-repo beagle-repo
cd beagle-repo
bzr checkout svn+ssh://svn.gnome.org/svn/beagle/trunk beagle-trunk
```

Next, create a feature branch and hack away:

```
bzr branch beagle-trunk beagle-feature1
cd beagle-feature1
(hack, hack, hack)
bzr commit -m "blah blah blah"
(hack, hack, hack)
bzr commit -m "blah blah blah"
```

When the feature is cooked, refresh your trunk mirror and merge your change:

```
cd ../beagle-trunk
bzr update
bzr merge ../beagle-feature1
bzr commit -m "Complete comment for SVN commit"
```

As your trunk mirror is a checkout, committing to it implicitly commits to the real Subversion trunk. That's it!

8.2.3 Using a central repository mirror

For large projects, it often makes sense to tweak the recipe given above. In particular, the initial checkout can get quite slow so you may wish to import the Subversion repository into a Bazaar one once and for all for your project, and then branch from that native Bazaar repository instead. bzr-svn provides the `svn-import` command for doing this repository-to-repository conversion. Here's an example of how to use it:

```
bzr svn-import svn+ssh://svn.gnome.org/svn/beagle
```

Here's the recipe from above updated to use a central Bazaar mirror:

```
bzr init-repo beagle-repo
cd beagle-repo
bzr branch bzr+ssh://bzr.gnome.org/beagle.bzr/trunk beagle-trunk
bzr branch beagle-trunk beagle-feature1
cd beagle-feature1
(hack, hack, hack)
bzr commit -m "blah blah blah"
(hack, hack, hack)
bzr commit -m "blah blah blah"
cd ../beagle-trunk
bzr pull
bzr merge ../beagle-feature1
bzr commit -m "Complete comment for SVN commit"
bzr push
```

In this case, committing to the trunk only commits the merge locally. To commit back to the master Subversion trunk, an additional command (`bzr push`) is required.

Note: You'll need to give `pull` and `push` the relevant URLs the first time you use those commands in the trunk branch. After that, bzr remembers them.

The final piece of the puzzle in this setup is to put scripts in place to keep the central Bazaar mirror synchronized with the Subversion one. This can be done by adding a cron job, using a Subversion hook, or whatever makes sense in your environment.

8.2.4 Limitations of bzr-svn

Bazaar and Subversion are different tools with different capabilities so there will always be some limited interoperability issues. Here are some examples current as of bzr-svn 0.5.4:

- Bazaar doesn't support versioned properties
- Bazaar doesn't support tracking of file copies.

See the bzr-svn web page, http://wiki.bazaar.canonical.com/BzrForeignBranches/Subversion, for the current list of constraints.

CHAPTER

NINE

INTEGRATING BAZAAR INTO YOUR ENVIRONMENT

9.1 Web browsing

9.1.1 Overview

There are a range of options available for providing a web view of a Bazaar repository, the main one being Loggerhead. The homepage of Loggerhead can be found at https://launchpad.net/loggerhead.

A list of alternative web viewers including download links can be found on http://wiki.bazaar.canonical.com/WebInterface.

Note: If your project is hosted or mirrored on Launchpad, Loggerhead code browsing is provided as part of the service.

9.2 Bug trackers

Bazaar has a facility that allows you to associate a commit with a bug in the project's bug tracker. Other tools (or hooks) can then use this information to generate hyperlinks between the commit and the bug, or to automatically mark the bug closed in the branches that contain the commit.

9.2.1 Associating commits and bugs

When you make a commit, you can associate it with a bug by using the `--fixes` option of `commit`. For example:

```
$ bzr commit --fixes lp:12345 -m "Properly close the connection"
```

This records metadata in Bazaar linking the commit with bug 12345 in Launchpad. If you use a different bug tracker, it can be given its own tracker code (instead of `lp`) and used instead. For details on how to configure this for Bugzilla, Trac, Roundup and other bug/issue trackers, refer to Bug Tracker Settings in the Bazaar User Reference.

9.2.2 Metadata recording vs bug tracker updating

Recording metadata about bugs fixed at commit time is only one of the features needed for complete bug tracker integration. As Bazaar is a distributed VCS, users may be offline while committing so accessing the bug tracker itself at that time may not be possible. Instead, it is recommended that a hook be installed to update the bug tracker when changes are pushed to a central location appropriate for your project's workflow.

Note: This second processing stage is part of the integration provided by Launchpad when it scans external or hosted branches.

9.2.3 Making corrections

This method of associating revisions and bugs does have some limitations. The first is that the association can only be made at commit time. This means that if you forget to make the association when you commit, or the bug is reported after you fix it, you generally cannot go back and add the link later.

Related to this is the fact that the association is immutable. If a bug is marked as fixed by one commit but that revision does not fully solve the bug, or there is a later regression, you cannot go back and remove the link.

Of course, `bzr uncommit` can always be used to undo the last commit in order to make it again with the correct options. This is commonly done to correct a bad commit message and it equally applies to correcting metadata recorded (via `--fixes` for example) on the last commit.

Note: `uncommit` is best done before incorrect revisions become public.

CHAPTER

TEN

APPENDICES

10.1 Specifying revisions

10.1.1 Revision identifiers and ranges

Bazaar has a very expressive way to specify a revision or a range of revisions. To specify a range of revisions, the upper and lower bounds are separated by the `..` symbol. For example:

```
$ bzr log -r 1..4
```

You can omit one bound like:

```
$ bzr log -r 1..
$ bzr log -r ..4
```

Some commands take only one revision, not a range. For example:

```
$ bzr cat -r 42 foo.c
```

In other cases, a range is required but you want the length of the range to be one. For commands where this is relevant, the `-c` option is used like this:

```
$ bzr diff -c 42
```

10.1.2 Available revision identifiers

The revision, or the bounds of the range, can be given using different format specifications as shown below.

argument type	description
number	revision number
revno:*number*	revision number
last:*number*	negative revision number
guid	globally unique revision id
revid:*guid*	globally unique revision id
before:*rev*	leftmost parent of ''rev''
date-value	first entry after a given date
date:*date-value*	first entry after a given date
tag-name	revision matching a given tag
tag:*tag-name*	revision matching a given tag
ancestor:*path*	last merged revision from a branch
branch:*path*	latest revision on another branch
submit:*path*	common ancestor with submit branch

A brief introduction to some of these formats is given below. For complete details, see Revision Identifiers in the Bazaar User Reference.

Numbers

Positive numbers denote revision numbers in the current branch. Revision numbers are labelled as "revno" in the output of `bzr log`. To display the log for the first ten revisions:

```
$ bzr log -r ..10
```

Negative numbers count from the latest revision, -1 is the last committed revision.

To display the log for the last ten revisions:

```
$ bzr log -r -10..
```

revid

revid allows specifying an internal revision ID, as shown by `bzr log --show-ids` and some other commands.

For example:

```
$ bzr log -r revid:Matthieu.Moy@imag.fr-20051026185030-93c7cad63ee570df
```

before

before ''rev" specifies the leftmost parent of ''rev'', that is the revision that appears before ''rev" in the revision history, or the revision that was current when ''rev" was committed.

''rev" can be any revision specifier and may be chained.

For example:

```
$ bzr log -r before:before:4
...
revno: 2
...
```

date

date ''value" matches the first history entry after a given date, either at midnight or at a specified time.

Legal values are:

- **yesterday**
- **today**
- **tomorrow**
- A **YYYY-MM-DD** format date.
- A **YYYY-MM-DD,HH:MM:SS** format date/time, seconds are optional (note the comma)

The proper way of saying "give me all the log entries for today" is:

```
$ bzr log -r date:yesterday..date:today
```

Ancestor

ancestor:*path* specifies the common ancestor between the current branch and a different branch. This is the same ancestor that would be used for merging purposes.

path may be the URL of a remote branch, or the file path to a local branch.

For example, to see what changes were made on a branch since it was forked off `../parent`:

```
$ bzr diff -r ancestor:../parent
```

Branch

branch `path` specifies the latest revision in another branch.

`path` may be the URL of a remote branch, or the file path to a local branch.

For example, to get the differences between this and another branch:

```
$ bzr diff -r branch:http://example.com/bzr/foo.dev
```

10.2 Organizing your workspace

10.2.1 Common workspace layouts

The best way for a Bazaar user to organize their workspace for a project depends on numerous factors including:

- user role: project owner vs core developer vs casual contributor
- workflows: particularly the workflow the project encourages/mandates for making contributions
- size: large projects have different resource requirements to small ones.

There are at least 4 common ways of organizing one's workspace:

- lightweight checkout
- standalone tree
- feature branches

- switchable sandbox.

A brief description of each layout follows.

10.2.2 Lightweight checkout

In this layout, the working tree is local and the branch is remote. This is the standard layout used by CVS and Subversion: it's simple and well understood.

To set up:

```
bzr checkout --lightweight URL project
cd project
```

To work:

```
(make changes)
bzr commit
(make changes)
bzr commit
```

Note that each commit implicitly publishes the change to everyone else working from that branch. However, you need to be up to date with changes in the remote branch for the commit to succeed. To grab the latest code and merge it with your changes, if any:

```
bzr update
```

10.2.3 Standalone tree

In this layout, the working tree & branch are in the one place. Unless a shared repository exists in a higher level directory, the repository is located in that same place as well. This is the default layout in Bazaar and it's great for small to moderately sized projects.

To set up:

```
bzr branch URL project
cd project
```

To work:

```
(make changes)
bzr commit
(make changes)
bzr commit
```

To publish changes to a central location:

```
bzr push [URL]
```

The URL for push is only required the first time.

If the central location has, in the meantime, received changes from other users, then you'll need to merge those changes into your local branch before you try to push again:

```
bzr merge
(resolve conflicts)
bzr commit
```

As an alternative, a checkout can be used. Like a branch, a checkout has a full copy of the history stored locally but the local branch is bound to the remote location so that commits are published to both locations at once.

Note: A checkout is actually smarter than a local commit followed by a push. In particular, a checkout wil commit to the remote location first and only commit locally if the remote commit succeeds.

10.2.4 Feature branches

In this layout, there are multiple branches/trees, typically sharing a repository. One branch is kept as a mirror of "trunk" and each unit-of-work (i.e. bug-fix or enhancement) gets its own "feature branch". This layout is ideal for most projects, particularly moderately sized ones.

To set up:

```
bzr init-repo project
cd project
bzr branch URL trunk
```

To start a feature branch:

```
bzr branch trunk featureX
cd featureX
```

To work:

```
(make changes)
bzr commit
(make changes)
bzr commit
```

To publish changes to a mailing list for review & approval:

```
bzr send
```

To publish changes to a public branch (that can then be registered as a Launchpad merge request, say):

```
bzr push [URL]
```

As a variation, the trunk can be created as a checkout. If you have commit privileges on trunk, that lets you merge into trunk and the commit of the merge will implicitly publish your change. Alternatively, if the trunk URL is read-only (e.g. an HTTP address), that prevents accidental submission this way - ideal if the project workflow uses an automated gatekeeper like PQM, say.

10.2.5 Local sandbox

This layout is very similar to the feature branches layout except that the feature branches share a single working tree rather than having one each. This is similar to git's default layout and it's useful for projects with really large trees (> 10000 files say) or for projects with lots of build artifacts (like .o or .class files).

To set up:

```
bzr init-repo --no-trees project
cd project
bzr branch URL trunk
bzr checkout --lightweight trunk sandbox
cd sandbox
```

While you *could* start making changes in sandbox now, committing while the sandbox is pointing to the trunk would mean that trunk is no longer a mirror of the upstream URL (well unless the trunk is a checkout). Therefore, you usually want to immediately create a feature branch and switch your sandbox to it like this:

```
bzr branch ../trunk ../featureX
bzr switch ../featureX
```

The processes for making changes and submitting them are otherwise pretty much the same as those used for feature branches.

10.2.6 Advanced layouts

If you wish, you can put together your own layout based on how **you** like things organized. See Advanced shared repository layouts for examples and inspiration.

10.3 Advanced shared repository layouts

Bazaar is designed to give you flexibility in how you layout branches inside a shared repository. This flexibility allows users to tailor Bazaar to their workflow, but it also leads to questions about what is a "good" layout. We present some alternatives and give some discussion about the benefits of each.

One key point which should be mentioned is that any good layout should somehow highlight what branch a "general" user should grab. In SVN this is deemed the "`trunk/`" branch, and in most of the layouts this naming convention is preserved. Some would call this "`mainline`" or "`dev`", and people from CVS often refer to this as "`HEAD`".

10.3.1 "SVN-Style" (`trunk/`, `branches/`)

Most people coming from SVN will be familiar with their "standard" project layout. Which is to layout the repository as:

```
repository/       # Overall repository
 +- trunk/        # The mainline of development
 +- branches/     # A container directory
 |   +- foo/      # Branch for developing feature foo
 |     ...
 +- tags/         # Container directory
     +- release-X # A branch specific to mark a given release version
        ...
```

With Bazaar, that is a perfectly reasonable layout. It has the benefit of being familiar to people coming from SVN, and making it clear where the development focus is.

When you have multiple projects in the same repository, the SVN layout is a little unclear what to do.

`project/trunk`

The preferred method for SVN seems to be to give each project a top level directory for a layout like:

```
repository/            # Overall repository
 +- project1/          # A container directory
 |   +- trunk/         # The mainline of development of project1
 |   +- branches/      # A container directory
 |       +- foo/       # Branch for developing feature foo of project1
```

```
 |         ...
 |
 +- project2/          # Container for project2
     +- trunk/         # Mainline for project2
     +- branches/      # Container for project2 branches
```

This also works with Bazaar. However, with Bazaar repositories are cheap to create (a simple `bzr init-repo` away), and their primary benefit is when the branches share a common ancestry.

So the preferred way for Bazaar would be:

```
project1/          # A repository for project1
 +- trunk/         # The mainline of development of project1
 +- branches/      # A container directory
     +- foo/       # Branch for developing feature foo of project1
       ...

project2/          # A repository for project2
 +- trunk/         # Mainline for project2
 +- branches/      # Container for project2 branches
```

trunk/project

There are also a few projects who use this layout in SVN:

```
repository/             # Overall repository
  +- trunk/             # A container directory
  |   +- project1       # Mainline for project 1
  |   +- project2       # Mainline for project 2
  |         ...
  |
  +- branches/          # Container
      +- project1/      # Container (?)
      |   +- foo        # Branch 'foo' of project1
      +- project2/
          +- bar        # Branch 'bar' of project2
```

A slight variant is:

```
repository/             # Overall repository
  +- trunk/             # A container directory
  |   +- project1       # Mainline for project 1
  |   +- project2       # Mainline for project 2
  |         ...
  |
  +- branches/          # Container
      +- project1-foo/  # Branch 'foo' of project1
      +- project2-bar/  # Branch 'bar' of project2
```

I believe the reason for this in SVN, is so that someone can checkout all of "`trunk/`" and get the all the mainlines for all projects.

This layout can be used for Bazaar, but it is not generally recommended.

1. `bzr branch/checkout/get` is a single branch at a time. So you don't get the benefit of getting all mainlines with a single command. [1]

[1] Note: NestedTreeSupport can provide a way to create "meta-projects" which aggregate multiple projects regardless of the repository layout. Letting you `bzr checkout` one project, and have it grab all the necessary sub-projects.

2. It is less obvious of whether `repository/trunk/foo` is the `trunk` of project `foo` or it is just the `foo` directory in the `trunk` branch. Some of this confusion is due to SVN, because it uses the same "namespace" for files in a project that it uses for branches of a project. In Bazaar, there is a clear distinction of what files make up a project, versus the location of the Branch. (After all, there is only one `.bzr/` directory per branch, versus many `.svn/` directories in the checkout).

10.3.2 Nested Style (`project/branch/sub-branch/`)

Another style with Bazaar, which is not generally possible in SVN is to have branches nested within each-other. This is possible because Bazaar supports (and recommends) creating repositories with no working trees (`--no-trees`). With a `--no-trees` repository, because the working files are not intermixed with your branch locations, you are free to put a branch in whatever namespace you want.

One possibility is:

```
project/             # The overall repository, *and* the project's mainline branch
 + joe/              # Developer Joe's primary branch of development
 |  +- feature1/     # Developer Joe's feature1 development branch
 |  |   +- broken/   # A staging branch for Joe to develop feature1
 |  +- feature2/     # Joe's feature2 development branch
 |    ...
 + barry/            # Barry's development branch
 |  ...
 + releases/
    +- 1.0/
        +- 1.1.1/
```

The idea with this layout is that you are creating a hierarchical layout for branches. Where changes generally flow upwards in the namespace. It also gives people a little corner of the namespace to work on their stuff. One nice feature of this layout, is it makes branching "cheaper" because it gives you a place to put all the mini branches without cluttering up the global `branches/` namespace.

The other power of this is that you don't have to repeat yourself when specifying more detail in the branch name.

For example compare:

```
bzr branch http://host/repository/project/branches/joe-feature-foo-bugfix-10/
```

Versus:

```
bzr branch http://host/project/joe/foo/bugfix-10
```

Also, if you list the `repository/project/branches/` directory you might see something like:

```
barry-feature-bar/
barry-bugfix-10/
barry-bugfix-12/
joe-bugfix-10/
joe-bugfix-13/
joe-frizban/
```

Versus having these broken out by developer. If the number of branches are small, `branches/` has the nice advantage of being able to see all branches in a single view. If the number of branches is large, `branches/` has the distinct disadvantage of seeing all the branches in a single view (it becomes difficult to find the branch you are interested in, when there are 100 branches to look through).

Nested branching seems to scale better to larger number of branches. However, each individual branch is less discoverable. (eg. "Is Joe working on bugfix 10 in his feature foo branch, or his feature bar branch?")

One other small advantage is that you can do something like:

```
 bzr branch http://host/project/release/1/1/1
or
 bzr branch http://host/project/release/1/1/2
```

To indicate release 1.1.1 and 1.1.2. This again depends on how many releases you have and whether the gain of splitting things up outweighs the ability to see more at a glance.

10.3.3 Sorted by Status (`dev/`, `merged/`, `experimental/`)

One other way to break up branches is to sort them by their current status. So you would end up with a layout something like:

```
project/                # Overall layout
 +- trunk/              # The development focus branch
 +- dev/                # Container directory for in-progress work
 |   +- joe-feature1    # Joe's current feature-1 branch
 |   +- barry-bugfix10 # Barry's work for bugfix 10
 |    ...
 +- merged/             # Container indicating these branches have been merged
 |   +- bugfix-12       # Bugfix which has already been merged.
 +- abandonded/         # Branches which are considered 'dead-end'
```

This has a couple benefits and drawbacks. It lets you see what branches are actively being developed on, which is usually only a small number, versus the total number of branches ever created. Old branches are not lost (versus deleting them), but they are "filed away", such that the more likely you are to want a branch the easier it is to find. (Conversely, older branches are likely to be harder to find).

The biggest disadvantage with this layout, is that branches move around. Which means that if someone is following the `project/dev/new-feature` branch, when it gets merged into `trunk/` suddenly `bzr pull` doesn't mirror the branch for them anymore because the branch is now at `project/merged/new-feature`. There are a couple ways around this. One is to use HTTP redirects to point people requesting the old branch to the new branch. `bzr` >= 0.15 will let users know that `http://old/path redirects to http://new/path`. However, this doesn't help if people are accessing a branch through methods other than HTTP (SFTP, local filesystem, etc).

It would also be possible to use a symlink for temporary redirecting (as long as the symlink is within the repository it should cause little trouble). However eventually you want to remove the symlink, or you don't get the clutter reduction benefit. Another possibility instead of a symlink is to use a `BranchReference`. It is currently difficult to create these through the `bzr` command line, but if people find them useful that could be changed. This is actually how Launchpad allows you to `bzr checkout https://launchpad.net/bzr`. Effectively a `BranchReference` is a symlink, but it allows you to reference any other URL. If it is extended to support relative references, it would even work over HTTP, SFTP, and local paths.

10.3.4 Sorted by date/release/etc (`2006-06/`, `2006-07/`, `0.8/`, `0.9`)

Another method of allowing some scalability while also allowing the browsing of "current" branches. Basically, this works on the assumption that actively developed branches will be "new" branches, and older branches are either merged or abandoned.

Basically the date layout looks something like:

```
project/                # Overall project repository
 +- trunk/              # General mainline
 +- 2006-06/            # containing directory for branches created in this month
 |   +- feature1/       # Branch of "project" for "feature1"
```

```
 |   +- feature2/       # Branch of "project" for "feature2"
 +- 2005-05/            # Containing directory for branches create in a different month
     +- feature3/
     ...
```

This answers the question "Where should I put my new branch?" very quickly. If a feature is developed for a long time, it is even reasonable to copy a branch into the newest date, and continue working on it there. Finding an active branch generally means going to the newest date, and going backwards from there. (A small disadvantage is that most directory listings sort oldest to the top, which may mean more scrolling). If you don't copy old branches to newer locations, it also has the disadvantage that searching for a branch may take a while.

Another variant is by release target:

```
project/          # Overall repository
 +- trunk/        # Mainline development branch
 +- releases/     # Container for release branches
 |   +- 0.8/      # The branch for release 0.8
 |   +- 0.9/      # The branch for release 0.9
 +- 0.8/          # Container for branches targeting release 0.8
 |   +- feature1/ # Branch for "feature1" which is intended to be merged into 0.8
 |   +- feature2/ # Branch for "feature2" which is targeted for 0.8
 +- 0.9/
     +- feature3/ # Branch for "feature3", targeted for release 0.9
```

Some possible variants include having the `0.9` directory imply that it is branched *from* 0.9 rather than *for* 0.9, or having the `0.8/release` as the official release 0.8 branch.

The general idea is that by targeting a release, you can look at what branches are waiting to be merged. It doesn't necessarily give you a good idea of what the state of the branch (is it in development or finished awaiting review). It also has a history-hiding effect, and otherwise has the same benefits and deficits as a date-based sorting.

10.3.5 Simple developer naming (`project/joe/foo`, `project/barry/bar`)

Another possibly layout is to give each developer a directory, and then have a single sub-directory for branches. Something like:

```
project/      # Overall repository
 +- trunk/    # Mainline branch
 +- joe/      # A container for Joe's branches
 |   +- foo/  # Joe's "foo" branch of "project"
 +- barry/
     +- bar/  # Barry's "bar" branch of "project"
```

The idea is that no branch is "nested" underneath another one, just that each developer has his/her branches grouped together.

A variant which is used by Launchpad is:

```
repository/
 +- joe/             # Joe's branches
 |   +- project1/    # Container for Joe's branches of "project1"
 |   |   +- foo/     # Joe's "foo" branch of "project1"
 |   +- project2/    # Container for Joe's "project2" branches
 |       +- bar/     # Joe's "bar" branch of "project2"
 |        ...
 |
 +- barry/
 |   +- project1/    # Container for Barry's branches of "project1"
```

```
|       +- bug-10/  # Barry's "bug-10" branch of "project1"
|   ...
+- group/
    +- project1/
        +- trunk/   # The main development focus for "project1"
```

This lets you easily browse what each developer is working on. Focus branches are kept in a "group" directory, which lets you see what branches the "group" is working on.

This keeps different people's work separated from each-other, but also makes it hard to find "all branches for project X". Launchpad compensates for this by providing a nice web interface with a database back end, which allows a "view" to be put on top of this layout. This is closer to the model of people's home pages, where each person has a "`~/public_html`" directory where they can publish their own web-pages. In general, though, when you are creating a shared repository for centralization of a project, you don't want to split it up by person and then project. Usually you would want to split it up by project and then by person.

10.3.6 Summary

In the end, no single naming scheme will work for everyone. It depends a lot on the number of developers, how often you create a new branch, what sort of lifecycles your branches go through. Some questions to ask yourself:

1. Do you create a few long-lived branches, or do you create lots of "mini" feature branches (Along with this is: Would you *like* to create lots of mini feature branches, but can't because they are a pain in your current VCS?)
2. Are you a single developer, or a large team?
3. If a team, do you plan on generally having everyone working on the same branch at the same time? Or will you have a "stable" branch that people are expected to track.

10.4 Configuring email

10.4.1 Why set up an email address with Bazaar?

Bazaar stores the specified email address in revisions when they're created so that people can tell who committed which revisions. The email addresses are not verified, therefore they could be bogus, so you have to trust the people involved in your project. Additionally, the email address in a revision gives others a way to contact the author of a revision for credit and/or blame. :)

10.4.2 How to set up your email address

Bazaar will try to guess an email address based on your username and the hostname if none is set. This will probably not be what you want, so three ways exist to tell Bazaar what email to use:

You can set your email in one of several configuration files. Like other configuration values, you can set it in `bazaar.conf` as a general setting. If you want to override the value for a particular branch, or set of branches, you can use `locations.conf`. `.bzr/branch/branch.conf` will also work, but will cause all commits to that branch to use the same email address, even if someone else does them.

The order of precedence is

1. If the `BZR_EMAIL` environment variable is set.
2. If an email is set for your current branch in the `locations.conf` file.
3. If an email is set four your current branch in the `.bzr/branch/branch.conf` file.

4. If an email is set in the `bazaar.conf` default configuration file.
5. If the *EMAIL* environment variable is set.
6. Bazaar will try to guess based on your username and the hostname.

To check on what Bazaar thinks your current email is, use the `whoami` ("who am i?") command:

```
% bzr whoami
Joe Cool <joe@example.com>
```

10.4.3 Setting email via the 'whoami' command

You can use the whoami command to set your email globally:

```
% bzr whoami "Joe Cool <joe@example.com>"
```

or only for the current branch:

```
% bzr whoami --branch "Joe Cool <joe@example.com>"
```

These modify your global `bazaar.conf` or branch `branch.conf` file, respectively.

10.4.4 Setting email via default configuration file

To use the default ini file, create or edit the `bazaar.conf` file (in `~/.bazaar/` on Unix and in `%APPDATA%\bazaar\2.0\` in Windows) and set an email address as shown below. Please note that the word DEFAULT is case sensitive, and must be in upper-case.

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

For more information on the ini file format, see Configuration Settings in the Bazaar User Reference.

10.4.5 Setting email on a per-branch basis

The second approach is to set email on a branch by branch basis by using the `locations.conf` configuration file like this:

```
[/some/branch/location]
email=Your Name <name@other-isp.com>
```

This will set your email address in the branch at `/some/branch/location`, overriding the default specified in the `bazaar.conf` above.

10.4.6 Setting email via environment variable

The final method Bazaar will use is checking for the `BZR_EMAIL` and `EMAIL` environment variables. Generally, you would use this method to override the email in a script context. If you would like to set a general default, then please see the ini methods above.

10.4.7 Concerns about spam

Some people want to avoid sharing their email address so as not to get spam. Bazaar will never disclose your email address, unless you publish a branch or changeset in a public location. It's recommended that you *do* use a real address, so that people can contact you about your work, but it's not required. You can use an address which is obfuscated, which bounces, or which goes through an anti-spam service such as *spamgourmet.com*.

10.5 Serving Bazaar with Apache

This document describes one way to set up a Bazaar HTTP smart server, using Apache 2.0 and FastCGI or mod_python or mod_wsgi.

For more information on the smart server, and other ways to configure it see the main smart server documentation.

10.5.1 Example

You have a webserver already publishing */srv/example.com/www/code* as *http://example.com/code/...* with plain HTTP. It contains bzr branches and directories like */srv/example.com/www/code/branch-one* and */srv/example.com/www/code/my-repo/branch-two*. You want to provide read-only smart server access to these directories in addition to the existing HTTP access.

10.5.2 Configuring Apache 2.0

FastCGI

First, configure mod_fastcgi, e.g. by adding lines like these to your httpd.conf:

```
LoadModule fastcgi_module /usr/lib/apache2/modules/mod_fastcgi.so
FastCgiIpcDir /var/lib/apache2/fastcgi
```

In our example, we're already serving */srv/example.com/www/code* at *http://example.com/code*, so our existing Apache configuration would look like:

```
Alias /code /srv/example.com/www/code
<Directory /srv/example.com/www/code>
    Options Indexes
    # ...
</Directory>
```

We need to change it to handle all requests for URLs ending in *.bzr/smart*. It will look like:

```
Alias /code /srv/example.com/www/code
<Directory /srv/example.com/www/code>
    Options Indexes FollowSymLinks
    RewriteEngine On
    RewriteBase /code
    RewriteRule ^(.*/)?\.bzr/smart$ /srv/example.com/scripts/bzr-smart.fcgi
</Directory>

# bzr-smart.fcgi isn't under the DocumentRoot, so Alias it into the URL
# namespace so it can be executed.
Alias /srv/example.com/scripts/bzr-smart.fcgi /srv/example.com/scripts/bzr-smart.fcgi
<Directory /srv/example.com/scripts>
    Options ExecCGI
```

```
    <Files bzr-smart.fcgi>
        SetHandler fastcgi-script
    </Files>
</Directory>
```

This instructs Apache to hand requests for any URL ending with */.bzr/smart* inside */code* to a Bazaar smart server via FastCGI.

Refer to the mod_rewrite and mod_fastcgi documentation for further information.

mod_python

First, configure mod_python, e.g. by adding lines like these to your httpd.conf:

```
LoadModule python_module /usr/lib/apache2/modules/mod_python.so
```

Define the rewrite rules with mod_rewrite the same way as for FastCGI, except change:

```
RewriteRule ^(.*/)?\.bzr/smart$ /srv/example.com/scripts/bzr-smart.fcgi
```

to:

```
RewriteRule ^(.*/)?\.bzr/smart$ /srv/example.com/scripts/bzr-smart.py
```

Like with mod_fastcgi, we also define how our script is to be handled:

```
Alias /srv/example.com/scripts/bzr-smart.py /srv/example.com/scripts/bzr-smart.py
<Directory /srv/example.com/scripts>
    <Files bzr-smart.py>
        PythonPath "sys.path+['/srv/example.com/scripts']"
        AddHandler python-program .py
        PythonHandler bzr-smart::handler
    </Files>
</Directory>
```

This instructs Apache to hand requests for any URL ending with */.bzr/smart* inside */code* to a Bazaar smart server via mod_python.

NOTE: If you don't have bzrlib in your PATH, you will be need to change the following line:

```
PythonPath "sys.path+['/srv/example.com/scripts']"
```

To:

```
PythonPath "['/path/to/bzr']+sys.path+['/srv/example.com/scripts']"
```

Refer to the mod_python documentation for further information.

mod_wsgi

First, configure mod_wsgi, e.g. enabling the mod with a2enmod wsgi. We need to change it to handle all requests for URLs ending in *.bzr/smart*. It will look like:

```
WSGIScriptAliasMatch ^/code/.*/\.bzr/smart$ /srv/example.com/scripts/bzr.wsgi

#The three next lines allow regular GETs to work too
RewriteEngine On
RewriteCond %{REQUEST_URI} !^/code/.*/\.bzr/smart$
```

```
RewriteRule ^/code/(.*/\.bzr/.*)$ /srv/example.com/www/code/$1 [L]

<Directory /srv/example.com/www/code>
    WSGIApplicationGroup %{GLOBAL}
</Directory>
```

This instructs Apache to hand requests for any URL ending with */.bzr/smart* inside */code* to a Bazaar smart server via WSGI, and any other URL inside */code* to be served directly by Apache.

Refer to the mod_wsgi documentation for further information.

10.5.3 Configuring Bazaar

FastCGI

We've configured Apache to run the smart server at */srv/example.com/scripts/bzr-smart.fcgi*. This is just a simple script we need to write to configure a smart server, and glue it to the FastCGI gateway. Here's what it looks like:

```
import fcgi
from bzrlib.transport.http import wsgi

smart_server_app = wsgi.make_app(
    root='/srv/example.com/www/code',
    prefix='/code/',
    path_var='REQUEST_URI',
    readonly=True,
    load_plugins=True,
    enable_logging=True)

fcgi.WSGIServer(smart_server_app).run()
```

The *fcgi* module can be found at http://svn.saddi.com/py-lib/trunk/fcgi.py. It is part of flup.

mod_python

We've configured Apache to run the smart server at */srv/example.com/scripts/bzr-smart.py*. This is just a simple script we need to write to configure a smart server, and glue it to the mod_python gateway. Here's what it looks like:

```
import modpywsgi
from bzrlib.transport.http import wsgi

smart_server_app = wsgi.make_app(
    root='/srv/example.com/www/code',
    prefix='/code/',
    path_var='REQUEST_URI',
    readonly=True,
    load_plugins=True,
    enable_logging=True)

def handler(request):
    """Handle a single request."""
    wsgi_server = modpywsgi.WSGIServer(smart_server_app)
    return wsgi_server.run(request)
```

The *modpywsgi* module can be found at http://ice.usq.edu.au/svn/ice/trunk/apps/ice-server/modpywsgi.py. It was part of pocoo. You sould make sure you place modpywsgi.py in the same directory as bzr-smart.py (ie. /srv/example.com/scripts/).

mod_wsgi

We've configured Apache to run the smart server at */srv/example.com/scripts/bzr.wsgi*. This is just a simple script we need to write to configure a smart server, and glue it to the WSGI gateway. Here's what it looks like:

```
from bzrlib.transport.http import wsgi

def application(environ, start_response):
    app = wsgi.make_app(
        root="/srv/example.com/www/code/",
        prefix="/code",
        readonly=True,
        enable_logging=False)
    return app(environ, start_response)
```

10.5.4 Clients

Now you can use *bzr+http://* URLs or just *http://* URLs, e.g.:

```
bzr log bzr+http://example.com/code/my-branch
```

Plain HTTP access should continue to work:

```
bzr log http://example.com/code/my-branch
```

10.5.5 Advanced configuration

Because the Bazaar HTTP smart server is a WSGI application, it can be used with any 3rd-party WSGI middleware or server that conforms the WSGI standard. The only requirements are:

- to construct a *SmartWSGIApp*, you need to specify a **root transport** that it will serve.
- each request's *environ* dict must have a **'bzrlib.relpath'** variable set.

The *make_app* helper used in the example constructs a *SmartWSGIApp* with a transport based on the *root* path given to it, and calculates the 'bzrlib.relpath' for each request based on the *prefix* and *path_var* arguments. In the example above, it will take the 'REQUEST_URI' (which is set by Apache), strip the '/code/' prefix and the '/.bzr/smart' suffix, and set that as the 'bzrlib.relpath', so that a request for '/code/foo/bar/.bzr/smart' will result in a 'bzrlib.relpath' of 'foo/bzr'.

It's possible to configure a smart server for a non-local transport, or that does arbitrary path translations, etc, by constructing a *SmartWSGIApp* directly. Refer to the docstrings of *bzrlib.transport.http.wsgi* and the WSGI standard for further information.

Pushing over the HTTP smart server

It is possible to allow pushing data over the HTTP smart server. The easiest way to do this, is to just supply `readonly=False` to the `wsgi.make_app()` call. But be careful, because the smart protocol does not contain any Authentication. So if you enable write support, you will want to restrict access to `.bzr/smart` URLs to restrict who can actually write data on your system, e.g. in apache it looks like:

```
<Location /code>
    AuthType Basic
    AuthName "example"
    AuthUserFile /srv/example.com/conf/auth.passwd
    <LimitExcept GET>
        Require valid-user
    </LimitExcept>
</Location>
```

At this time, it is not possible to allow some people to have read-only access and others to have read-write access to the same URLs. Because at the HTTP layer (which is doing the Authenticating), everything is just a POST request. However, it would certainly be possible to have HTTPS require authentication and use a writable server, and plain HTTP allow read-only access.

If bzr gives an error like this when accessing your HTTPS site:

```
bzr: ERROR: Connection error: curl connection error (server certificate verification failed.
CAfile:/etc/ssl/certs/ca-certificates.crt CRLfile: none)
```

You can workaround it by using `https+urllib` rather than `http` in your URL, or by uninstalling pycurl. See bug 82086 for more details.

10.6 Writing a plugin

10.6.1 Introduction

Plugins are very similar to bzr core functionality. They can import anything in bzrlib. A plugin may simply override standard functionality, but most plugins supply new commands.

10.6.2 Creating a new command

To create a command, make a new object that derives from `bzrlib.commands.Command`, and name it `cmd_foo`, where foo is the name of your command. If you create a command whose name contains an underscore, it will appear in the UI with the underscore turned into a hyphen. For example, *cmd_baz_import* will appear as *baz-import*. For examples of how to write commands, please see `builtins.py`.

Once you've created a command you must register the command with `bzrlib.commands.register_command(cmd_foo)`. You must register the command when your file is imported, otherwise bzr will not see it.

10.6.3 Installing a hook

See Using hooks.

10.6.4 Specifying a plugin version number

Simply define `version_info` to be a tuple defining the current version number of your plugin. eg. `version_info = (0, 9, 0)` `version_info = (0, 9, 0, 'dev', 0)`

10.6.5 Plugin searching rules

Bzr will scan `~/.bazaar/plugins` and `bzrlib/plugins` for plugins by default. You can override this with `BZR_PLUGIN_PATH` (see User Reference for details).

Plugins may be either modules or packages. If your plugin is a single file, you can structure it as a module. If it has multiple files, or if you want to distribute it as a bzr branch, you should structure it as a package, i.e. a directory with an `__init__.py` file.

10.6.6 More information

Please feel free to contribute your plugin to BzrTools, if you think it would be useful to other people.

See the Bazaar Developer Guide for details on Bazaar's development guidelines and policies.

10.7 Licence

Copyright 2009-2011 Canonical Ltd. Bazaar is free software, and you may use, modify and redistribute both Bazaar and this document under the terms of the GNU General Public License version 2 or later. See <http://www.gnu.org/licenses/>.