CHAPTER 5

A Build for Every Check-In

Agile development focuses on catching bugs as soon as they are introduced. Optimally, the bugs are caught before changes are checked in, but there are classes of bugs that are expensive for the developer to verify. They happen infrequently, they are hassle to check for, and they can be painful to track down. The most visible relate to integration, platform dependencies, and external package dependencies.

The build must work on a freshly installed system, and it must contain everything that it needs to build itself. Products must frequently work with multiple versions of Python and across multiple platforms. A unit-testing module I maintain works with both Python 2.4 and 2.5, while another product I maintain is expected to work on any UNIX variant and Microsoft Windows.

Verifying these conditions before committing changes is expensive. It potentially involves many steps and a commitment of time that is guaranteed to break a programmer's flow. Just supporting one UNIX variant, Microsoft Windows, and two Python versions involves performing four sets of clean builds for every commit.

To make matters worse, most changes aren't going to cause these things to fail. With a mature product, the tests are likely to succeed dozens upon dozens of times before finally catching a failure. People aren't good at performing repetitive checks for infrequent failures—even more so when it derails their thought processes and they have to sit around waiting for the results. Eventually, vigilance lapses, a bug of this sort sneaks through, and it isn't found until deployment time. This frequently brings about a cascade of other failures.

Build servers address these problems. Rather than holding the developer responsible for verifying the correctness of the code on every system targeted, the job is given over to an automated system that is responsible for performing clean builds after every commit. Changes are validated immediately, and in case of failure, notifications are sent to the concerned parties. The build servers provide confidence that the software can always be built.

Many different build servers are available—both free and commercial. Among the more well known are CruiseControl and Anthill. This book focuses on Buildbot, an open source system written in Python. It supports build farms, in which builds are distributed to a number of client machines that then perform the builds and communicate the results back to the server. It has a centralized reporting system, and it is easily configured and extended using Python.

Buildbot Architecture

Buildbot is a common open build system. It is written in Python, but it will build anything. It uses a master-and-slave architecture. The central build master controls one or more build slaves. Builds are triggered by the master, and performed on the slaves. The slaves can be of a different architecture than the master. The slaves report build results to the master, and the master reports them to the users. The master contains a minimal web server showing the real-time build telemetry.

There are multiple options for triggering builds. The master can do it periodically, producing a nightly or hourly build. More interestingly, the master can be triggered to perform builds whenever new changes are committed.

The system demonstrated in this chapter contains a master, a slave, and a remote Subversion repository. The three systems are named `buildmaster`, `slave-lnx01`, and `source`. On my systems, these are DNS aliases for the underlying hosts. The slave performs builds against both Python 2.4 and 2.5., and these builds are triggered automatically after each commit.

Dedicated users will be created to run both Buildbot and Subversion. On the build systems, the application Buildbot will be run as the user `build`; and on the source server, Subversion will be run as the user `svn`.

ALIASING HOSTS

On my network, the names `buildmaster`, `slave-lnx01`, and `source` are aliases for the two hosts `phytoplankton` and `agile`. `buildmaster` and `source` are aliases for `phytoplankton`, and `slave-lnx01` is an alias for `agile`. The names refer to the service being provided, not the underlying host. This way, the service can be moved to another host without disrupting clients (both human and machine).

I might do this if I wanted to move `agile` to a real box rather than running it under a VM, as I currently do. I might also do this if `phytoplankton` died, or if the load of running the repository became too much for this one system to bear.

Installing Buildbot

Buildbot itself is a Setuptools package. It can be downloaded and installed using `easy_install`, but it is built on top of Twisted, which is "an event-driven networking engine." Twisted provides the bulk of the networking infrastructure for Buildbot. It's best to install Twisted before installing Buildbot.

Twisted is built with Distutils, and it must be installed carefully in multiple steps. It has its own dependency on a package called Zope Interface, which provides a limited typing system for Python. You could spend time chasing this package down, but that's not necessary, as it's bundled with Twisted. However, although it is bundled, it must be installed manually before Twisted.

I'll start by demonstrating how to install Buildbot on `buildmaster`. You'll be installing special Python installations just for the build slave's use, so it doesn't matter much where Buildbot and its dependencies are installed. I'm going to use the primary system installation:

```
$ curl -L -o Twisted-2.5.0.tar.bz2 ➥
 http://tmrc.mit.edu/mirror/twisted/Twisted/2.5/Twisted-2.5.0.tar.bz2
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 2683k  100 2683k    0     0   741k       0  0:00:03  0:00:03 --:--:--  787k
```

```
$ bunzip2 Twisted-2.5.0.tar.bz2
$ tar xvf Twisted-2.5.0.tar
```

```
Twisted-2.5.0/
Twisted-2.5.0/TwistedConch-0.8.0/
...
Twisted-2.5.0/LICENSE
Twisted-2.5.0/setup.py
```

```
$ cd Twisted-2.5.0
$ ls -F
```

```
LICENSE          TwistedMail-0.4.0/       TwistedWords-0.5.0/
README                   TwistedNames-0.4.0/     setup.py*
TwistedConch-0.8.0/      TwistedNews-0.3.0/       zope.interface-3.3.0/
TwistedCore-2.5.0/       TwistedRunner-0.2.0/
TwistedLore-0.3.0/       TwistedWeb-0.7.0/
```

```
$ cd zope.interface-3.3.0
$ sudo python ./setup.py install
```

```
running install
running bdist_egg
...
Processing dependencies for zope.interface==3.3.0
Finished processing dependencies for zope.interface==3.3.0
```

■ **Warning** With most packages, running `install` correctly invokes `build`, but that has not been my experience with Twisted. It is necessary to run build and install separately.

```
$ cd ..
$ python ./setup.py install
```

```
running install
running build
...
byte-compiling /usr/lib/python2.5/site-packages/twisted/➥
news/test/test_nntp.py to test_nntp.pyc
byte-compiling /usr/lib/python2.5/site-packages/twisted/➥
plugins/twisted_news.py to twisted_news.pyc
```

At this point, Twisted is installed, and Buildbot can now be installed. Buildbot is installed with `easy_install`, which is part of Setuptools. If it hasn't been installed yet, then you'll need to do this first. See Chapter 4 for more information.

There is one catch, though. Buildbot has contributed programs that are shipped with it, but that are not installed by `easy_install`. You'll use one later, so you'll want the source package to remain on the system. The build directory option `-b` specifies a directory where the installation is staged from. When `easy_install` completes, this directory will be left behind and the component files will be accessible.

```
$ easy_install -b /tmp/bbinst buildbot
```

```
Searching for buildbot
Reading http://pypi.python.org/simple/buildbot/
...
Installed /Users/jeff/Library/Python/2.5/site-packages/➥
buildbot-0.7.6-py2.5.egg
Processing dependencies for buildbot
Finished processing dependencies for buildbot
```

```
$ buildbot --version
```

```
Buildbot version: 0.7.6
Twisted version: 2.5.0
```

The identical process must now be performed on all machines communicating with Buildbot. This includes the Subversion host, too. Once the installations are complete, the master and slave can then be configured.

Configuring the Build System

As outlined earlier, there are two build hosts in our system: the Buildbot master, named `buildmaster`, and the Buildbot slave, named `slave-lnx01`. Buildbot runs on both systems as a dedicated user, which you'll name `build`. This provides administrative and security benefits. Startup configuration can be kept within the user's account. The user `build` has limited rights, so any compromises of the Buildbot server will be limited to `build`'s account, and any misconfigurations will be limited by filesystem permissions.

When a Buildbot slave starts, it contacts the build master. It needs three pieces of information to do this. First, it needs the name of the build master so that it can find it on the network. The port identifies the Buildbot instance running on the build master, and the password authenticates the slave to the master. The master must know which port to listen on, and it must know the password that slaves will present.

In the environment discussed here, the build master runs on the host `buildmaster` listening on port `4484` for the password `Fo74gh18` from instance `rsreader-full-py2.5`. The build server instances run from within the directory `/usr/local/buildbot`. RSReader is the project started in Chapter 4. The master lives in `/usr/local/buildbot/master/rsreader`, and the slave lives in `/usr/local/buildbot/slave/rsreader`. This directory structure allows you to intermix independent Buildbot instances for different projects on the same machines.

Setting up communications between the master and the slave is the first goal. Retrieving source code or performing a build is pointless until the two servers can speak to each other.

Mastering Buildbot

The build master is configured before the slave, as the slave's status is determined through its interactions with the build master. Creating the `build` user and the directories are the first steps.

■ **Tip** If you're trying to install Buildbot on Windows systems, it is started with `buildbot.bat`. This script is installed into `\Python25\scripts`. Unfortunately, it has a hard-coded reference to a nonexistent script in `\Python23`. This reference will need to be changed by hand.

```
$ useradd buid
$ sudo mkdir -p /usr/local/buildbot/master/rsreader
$ sudo chown build:build /usr/local/buildbot/master/rsreader
```

The next steps are performed as the newly created user `build`. They create the basic configuration files for a master.

```
$ su - build
$ buildbot create-master /usr/local/buildbot/master/rsreader
```

```
updating existing installation
chdir /usr/local/buildbot/master/rsreader
creating master.cfg.sample
populating public_html/
creating Makefile.sample
buildmaster configured in /usr/local/buildbot/master/rsreader
```

```
$ ls -F /usr/local/buildbot/master/rsreader
```

```
Makefile.sample         buildbot.tac
master.cfg.sample       public_html/
```

Once upon a time (Buildbot 0.6.5 and earlier), makefiles were used to start and stop Buildbot. This mechanism has been superseded by the `buildbot` command in current versions. Makefiles can still be used to override the startup process, but that's voodoo that I won't address, so you can safely forget that `Makefile.sample` exists.

`Buildbot.tac` is only of marginally more interest. It is used by the `buildbot` command to start the server. Essentially, it defines if this server is a client or a slave. It is necessary to Buildbot's operation, but you should never have to touch it.

The `public_html` directory is the document root for the build master's internal web server. It supplies the static content that will be served to your browser. Customizations to Buildbot's appearance go here, but they are strictly optional.

Of far more interest is `master.cfg.sample`. It is the template for the file `master.cfg`, which defines most of the master's behavior. It is a Python source file defining a single dictionary named `BuildmasterConfig`. This dictionary describes almost everything about the build master and the build process that ever needs changing. Much of this chapter is devoted to writing this file.

You'll start off with a minimal `master.cfg`. It defines the `BuildmasterConfig` dictionary and aliases it to the variable `c`. This is done to improve readability and save keystrokes (although rumors of an impending keystroke shortage have been determined to be false by reputable authorities).

```
# This is the dictionary that the buildmaster pays attention
# to. We also use a shorter alias to save typing.
c = BuildmasterConfig = {}
```

Next is the `slaves` property, which defines a list of `BuildSlave` objects. Each of these contains the name of a slave and the password that will be used to secure that connection. All slaves talk to the master on a single port, and the name is necessary to distinguish them from one another. Every slave has its own password, too. A separate password allows slaves to be controlled by different individuals without compromising the security of other slaves. In our case, we have one slave named `rsreader-linux`, and its password is `Fo74gh18`.

```
####### BUILDSLAVES

from buildbot.buildslave import BuildSlave
c['slaves'] = [BuildSlave("slave-lnx01", "Fo74gh18")]
```

The master listens for connections over a single port. The `slavePortnum` property defines this. This number is arbitrary, but it should be above 1024, as lower port numbers are reserved as rendezvous locations for well-known services (like mail) and web traffic. In our configuration, it will be 4484.

```
# 'slavePortnum' defines the TCP port to listen on. This must match the value
# configured into the buildslaves (with their --master option)

c['slavePortnum'] = 4484
```

When the source code changes, a build will be triggered. The build master needs to know how to find changes. Various classes within `buildbot.changes` supply these behaviors. The class `PBChangeSource` implements a listener that sits on `slavePortnum` and waits for externally generated change notifications. When it receives an appropriate notification, it triggers a build. In a few sections, you'll configure Subversion to send these notifications.

```
####### CHANGE SOURCES

from buildbot.changes.pb import PBChangeSource
c['change_source'] = PBChangeSource()
```

The `schedulers` property defines when builds are launched. It is a list of scheduler objects. These tie a scheduling policy to a builder that actually performs the build. You're going to schedule one build for Python 2.5. The scheduler will work on any branch, and it will run when there have been no more changes for 60 seconds.

```
####### SCHEDULERS

c['schedulers'] = []
c['schedulers'].append(Scheduler(name="rsreader",
                    branch=None,
                    treeStableTimer=60,
                    builderNames=["rsreader-full-py2.5"]))
```

Build factories describe the nitty-gritty details of building the application. They construct the instructions run by slaves. I shall be spending a lot of time on build factories, but right now a simple factory will suffice to test communication between the master and slave. The simple builder factory `f1` prints the message `build was run`.

```
####### BUILDERS

from buildbot.process import factory
from buildbot.steps.shell import ShellCommand

f1 = factory.BuildFactory()
f1.addStep(ShellCommand(command="echo 'build was run'"))
```

The `builders` property contains a list of builders. A *builder* is a dictionary associating the builder's name, the slave it runs on, and a builder factory. It also names the build directory. In this case, the builder is named `buildbot-full-py2.5`, and it runs on the slave `slave-lnx01` in the directory `full-py2.5` using the builder factory `f1`. The build directory is relative to the Buildbot root. In this case, the full path to the builder will be `/usr/local/buildbot/slave/rsreader/full-py2.5`.

```
b1 = {'name': "rsreader-full-py2.5",
      'slavename': "slave-lnx01",
      'builddir': "full-py2.5",
      'factory': f,
      }

c['builders'] = [b1]
```

The `status` property controls how build results are reported. We are implementing two. The `html.WebStatus` class implements a page referred to as the *waterfall display*, which shows the entire build system's recent activity. The web server port is configured with the `http_port` keyword. Here it's being configured to listen on port 8010.

The class `mail.MailNotifier` sends e-mail when a build fails. It is inventive and persistent in its actions. There are other notification classes, with the `words.IRC` class being perhaps the most interesting of those not being used in this example.

```
####### STATUS TARGETS

c['status'] = []

from buildbot.status.html import WebStatus
c['status'].append(WebStatus(http_port=8010))

from buildbot.status.mail import MailNotifier
c['status'].append(MailNotifier(
                    fromaddr="buildbot@phytoplankton.theblobshop.com",
                    extraRecipients=["builds@theblobshop.com"],
                    sendToInterestedUsers=False))
```

The properties `projectName`, `projectUrl`, and `buildbotUrl` configure communications with the user. The project name is used on the waterfall page. The project URL is the link from the waterfall page to the project's web site. `BuildbotURL` is the base URL to reach the Buildbot web server configured in the `status` property. Buildbot can't determine this URL on its own, so it must be configured here.

```
####### PROJECT IDENTITY

c['projectName'] = "RSReader"
c['projectURL'] = "http://www.theblobshop.com/rsreader"
c['buildbotURL'] = "http://buildmaster.theblobshop.com:8010/"
```

At this point, you can start the build master:

```
$ buildbot start /usr/local/buildbot/master/rsreader
```

```
Following twistd.log until startup finished..
2008-05-12 11:21:47-0700 [-] Log opened.
…
2008-05-12 11:21:47-0700 [-] BuildMaster listening on port tcp:4484
2008-05-12 11:21:47-0700 [-] configuration update started
2008-05-12 11:21:47-0700 [-] configuration update complete
The buildmaster appears to have (re)started correctly.
```

The messages indicate that Buildbot started correctly. In previous versions, the startup messages were untrustworthy and you often had to search through the file `twistd.log` in the application directory to determine if the reported status was accurate. This seems to have been remedied as of Buildbot 0.7.7. The landing screen is shown in Figure 5-1.

Figure 5-1. *The Buildbot landing page on the host buildmaster and port 8010*

Clicking the first link title, Waterfall Display, takes you to the page shown in Figure 5-2, which is a timeline. The top of the page represents now, and the screen extends down into the past. Each column represents a builder and the activity taking place. The builder's creation and the master's startup are both represented, so the display conveys information about the system's gross state, reducing the need to search through `twistd.log`. The red box at the top indicates that the build slave for `build-full-py2.5` is offline.

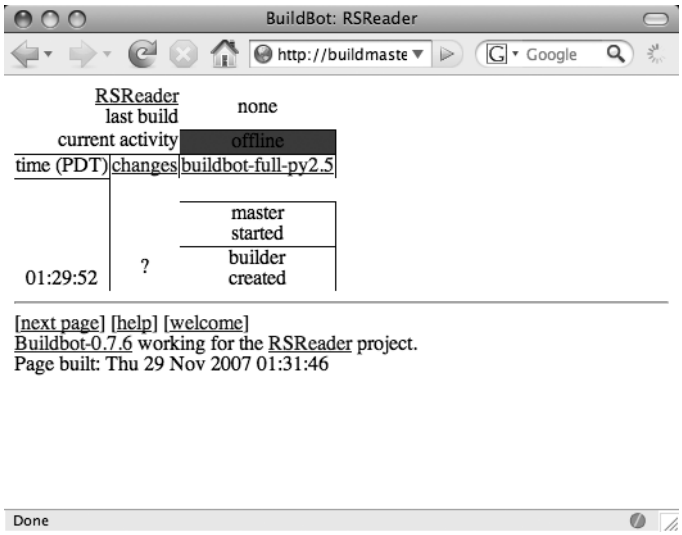

Figure 5-2. *The Buildbot waterfall display*

The properties `projectName` and `projectUrl` are used to produce the RSReader links at the top and bottom of the waterfall display. Clicking on either one is sufficient to verify the correct values. At this point, the basic server configuration is complete, and there is one last step.

The server must be started at reboot. Once upon a time, it was necessary to write a startup script and insert it into `/etc/init.d` on UNIX systems and create a few magically named symbolic links in the `/etc/rc` directories. These days, processes can be started from cron at reboot by adding the following line to the `build` user's crontab with the command `crontab -e`:

```
@reboot /path/to/buildbot start /usr/local/buildbot/master/rsreader
```

This technique should work on most modern UNIX systems, as well as Mac OS X. Cron doesn't have access to your full shell environment, so it is important to use the full path to the `buildbot` executable. Your shell may be able to locate `buildbot` when you are logged in, but it may not be able to when run from cron's extremely limited environment.

Enslaving Buildbot

In grand strokes, creating a basic Buildbot slave is similar to creating a master, but much simpler in the initial details. If running on a separate system, as in this example, then Buildbot must be installed first. Then the `build` user and Buildbot directories are created, a slave instance is created, the configuration files are updated, and Buildbot is started. In this test environment, the slave runs on `slave-lnx01`.

```
$ useradd build
$ sudo mkdir /usr/local/buildbot/slave/rsreader
$ sudo chown build:build /usr/local/buildbot/slave/rsreader
```

After creating the build directories, the client is configured from `build`'s account on `slave-lnx01`. Four pieces of information are necessary. The slave contacts the Buildbot master using the master's host name and port. In this case, the host name is `buildmaster` and the port is 4484. The slave identifies itself with a unique name. (The host name is insufficient, as there can be more than one slave running on a single host.) This is the name referred to on the master in both the builder definition and the `slaves` property. Finally, the slave needs the password to secure the connection. The `BuildSlave` object in `master.cfg` defines it; in this case, it's `Fo74gh18`.

```
$ su - build
$ buildbot create-slave /usr/local/buildbot/slave/rsreader ➥
 buildmaster:4484 rsreader-linux Fo74gh18
```

```
updating existing installation
chdir /usr/local/buildbot/slave/rsreader
creating Makefile.sample
mkdir /usr/local/buildbot/slave/rsreader/info
Creating info/admin, you need to edit it appropriately
Creating info/host, you need to edit it appropriately
Please edit the files in /usr/local/buildbot/slave/rsreader/info appropriately.
buildslave configured in /usr/local/buildbot/slave/rsreader
```

```
$ cd /usr/local/buildbot/slave/rsreader
$ ls -F
```

```
buildbot.tac info/ Makefile.sample
```

```
$ ls -F info
```

```
admin host
```

`Buildbot.tac` and `Makefile.sample` are analogous to those files on the build master. Buildbot uses `Buildbot.tac` to start the slave, but the slave's configuration is also in this file. Changes to the four configuration parameters can be made here. As with the master, `Makefile.sample` is a vestigial file lingering from previous generations of Buildbot.

The files in the `info` directory are of more interest. They are both text files containing information that is sent to the build master. `info/admin` contains this Buildbot administrator's name and e-mail address, while `info/host` contains a description of the slave.

The default for `info/admin` is `Your Name Here <admin@youraddress.invalid>`. In my environment, it is set to `Jeff Younker <buildmaster@theblobshop.com>`. The description in `slave-lnx01`'s `info/host` file reads `Produces pure Python 2.5 builds`. `info/host` is just a text file, and the information is to make your life, and the life of everyone who uses your build system, a little bit brighter and clearer, so make the description concise and informative. With these changes in place, the client can be started.

```
$ buildbot start /usr/local/buildbot/slave/rsreader
```

```
Following twistd.log until startup finished..
2007/11/27 02:18 -0700 [-] Log opened.
2007/11/27 02:18 -0700 [-] twistd 2.5.0 (/usr/bin/python 2.5.0) starting up
2007/11/27 02:18 -0700 [-] reactor class: ➥
<class 'twisted.internet.selectreactor.SelectReactor'>
2007/11/27 02:18 -0700 [-] Loading buildbot.tac...
2007/11/27 02:18 -0700 [-] Creating BuildSlave
2007/11/27 02:18 -0700 [-] Loaded.
2007/11/27 02:18 -0700 [-] Starting factory <buildbot.slave.bot.BotFactory➥
instance at 0xa0e484c>
2007/11/27 02:18 -0700 [broker,client] message from master: attached
The buildslave appears to have (re)started correctly.
```

The build slave has started and connected to the build master, which you can see on the waterfall display in Figure 5-3.

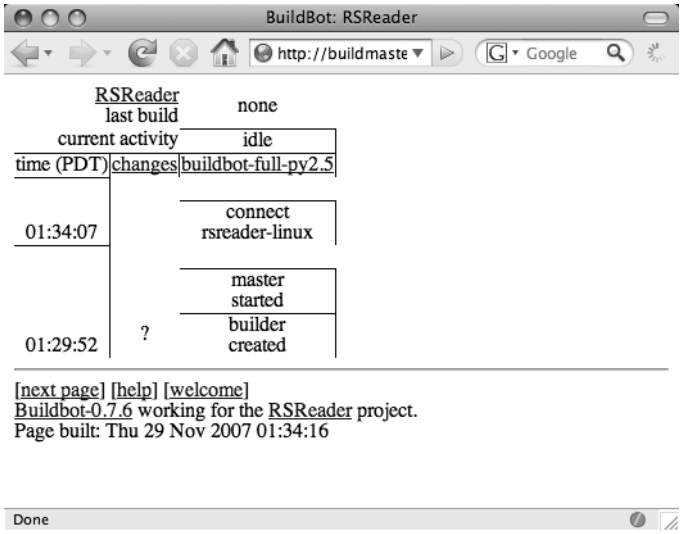

Figure 5-3. *The slave has successfully connected with the master.*

At this point, basic connectivity has been established and a build can be triggered with the command `buildbot sendchange`:

```
$ buildbot sendchange --master buildmaster:4484 -u jeff -n 30 setup.py
```

```
change sent successfully
```

The file name is arbitrary, but the change number specified with `-n` (in this case `30`) is not. The project branch (`rsreader/trunk` in this case) must exist at this revision, or else the build will fail.

The waterfall display immediately shows that the change has been received, and it shows a countdown timer until the build starts. Any changes submitted in this window will reset the timer. This is shown in Figure 5-4.

While the message is being, run the step is rendered in yellow. Once it completes, the step is rendered in green. If the step had failed, it would be red, and if an exception had been encountered, it would be purple. Once the timer expires, the build runs and the slave echoes its message. The output, shown in Figure 5-5, links from the build step.

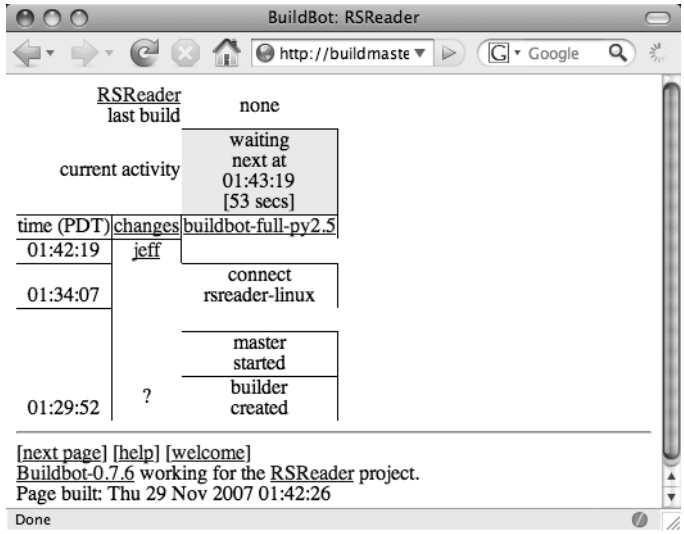

Figure 5-4. *The first build has been triggered.*

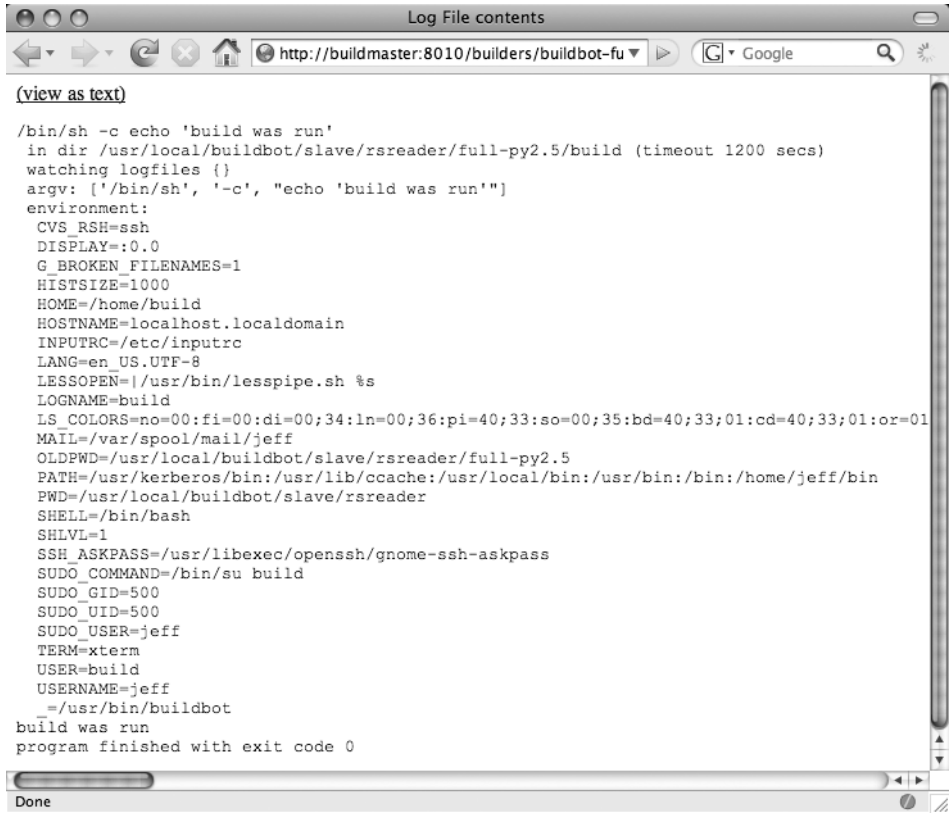

Figure 5-5. *The echo step output*

Information about the step is rendered in blue, and the actual output is rendered in black. It's clear that there is far more information about the step than actual output:

- The command run by the build step is shown in the first line.
- The present working directory follows. It indicates where the command runs from.
- The entire shell environment is displayed. Incorrect environment settings are a common source of build errors, so having this information recorded and available assists with debugging build problems.
- The command's output follows the environment. There's only one line in this case.
- The command's exit code is shown last. As with any UNIX shell command, `0` indicates success.

Hooking Up Source Control

The build master and build slave are now on separate hosts. They both need access to the Subversion repository. Until now, Subversion has been accessed directly through the filesystem, but this will no longer work. However, this isn't a simple choice. Subversion can be accessed remotely via a bewildering spectrum of methods. Repositories can be accessed through the Subversion network server, through WebDAV and the Apache web server, or by tunneling over a shell transport such as SSH. Enumerating the pros and cons of each approach would constitute a chapter's worth of material in itself.

I'm going to choose one pair of methods and stick with them, but if you'd like more information, then I encourage you to consult *Practical Subversion, Second Edition*, by Daniel Berlin and Garrett Rooney (Apress, 2006). The good news is that if you choose another method, then the changes on the client amount to nothing more than changing a URL.

For the rest of the book, I'm choosing a combination of `svnserve` running as a daemon for read-only access and `svnserve` over SSH for write access. This is a common configuration in which anyone can check out files anonymously, but committing changes requires a login and therefore authentication.

Committers need accounts on the Subversion server and write access to the Subversion repository. Authorization is done with group permissions. The repository tree is writable by the Subversion group, and all committers are members of that group. In this book, that group will be named `svn`.

Files created through `svnserve` over SSH are owned by the committer, but they must be writable by the Subversion group. `svnserve` sets the appropriate permissions, but those are affected by the user's umask. The umask turns off selected permissions when files are written. More frequently than not, the user's default umask turns off group write permissions, and it is therefore necessary to override it.

You do this with a wrapper script that replaces `svnserve`. The wrapper script sets the umask to `002` (which turns off writing by others) and calls the original `svnserve` while passing along all the arguments it received:

```
$ sudo mv /usr/local/bin/svnserve /usr/local/bin/svnserve-stock
$ sudo vi /usr/local/bin/svnserve
```

```
... set up the file as below ...
```

```
$ cat /usr/local/bin/svnserve
```

```
#!/bin/sh
umask 002
exec /usr/local/bin/svnserve-stock "$@"
```

```
$ sudo chmod a+x /usr/local/bin/svnserve
```

Now you'll create the user `svn` and change the ownership of the permissions of the Subversion repository. Remember that the repository is located in `/usr/local/svn/repos`.

```
$ sudo /usr/sbin/useradd svn
$ sudo chown svn:svn /usr/local/svn/repos
$ sudo chmod g+rw /usr/local/svn/repos
```

You can now start the Subversion server. The `--daemon` option tells the server to start as a daemon. The `-r` option tells it where to find the repository. It must be started from the Subversion user's account.

```
$ sudo -u svn /usr/local/bin/svnserve --daemon -r /usr/local/svn/repos
```

The Subversion server is now listening for requests on port 3690. Subversion clients use `file:///` URLs to access the local filesystem, and they access `svnserve` using `svn://` URLs. The local URL for the RSReader project is `file:///usr/local/svn/repos/rsreader`, and it maps to the remote URL `svn://source/rsreader`. The `source` component is the host name and the `rsreader` component is the path relative to the Subversion server's root.

You can verify the status with the `svn info` command:

```
$ svn info svn://source/rsreader
```

```
Path: rsreader
URL: svn://source/rsreader
Repository Root: svn://source
Repository UUID: e56658fc-2c3c-0410-b453-f6f88bcaf20d
Revision: 30
Node Kind: directory
Last Changed Author: jeff
Last Changed Rev: 30
Last Changed Date: 2007-11-20 13:38:06 -0800 (Tue, 20 Nov 2007)
```

Finally, it's necessary to add committers to the Subversion group. On my system, there are only two users to worry about: `jeff` and `doug`. On Linux systems, the `usermod` command adds users to groups.

```
$ sudo /usr/sbin/usermod -G svn jeff
$ sudo /usr/sbin/usermod -G svn doug
```

The depot is now ready for testing with Subversion over SSH. This access method runs `svnserve` on the repository machine, so it is a local access protocol like the `file:///`. The full directory path is used, yielding `svn+ssh://source/usr/local/svn/repos/rsreader`. Testing this from the command line shows that the URL is valid:

```
$ svn info svn+ssh://source/usr/local/svn/repos/rsreader
```

```
Password:

Path: rsreader
URL: svn+ssh://source/usr/local/svn/repos/rsreader
Repository Root: svn+ssh://source/usr/local/svn/repos
Repository UUID: e56658fc-2c3c-0410-b453-f6f88bcaf20d
Revision: 30
Node Kind: directory
Last Changed Author: jeff
Last Changed Rev: 30
Last Changed Date: 2007-11-20 13:38:06 -0800 (Tue, 20 Nov 2007)
```

Your password is requested before every new connection, but this can be circumvented using SSH keys. Setting up SSH trust relationships isn't very complicated, but it's outside the scope of this book. Tutorials can be found online, and a good one is provided by Linux Journal at `www.linuxjournal.com/article/8759`.

The output indicates that the repository is available, but we know that there is already a copy on the development machine that was checked out from the old file URL. The copy could just be abandoned, but if changes have already been made, then that course of action would be unpalatable. It would be better if there were a way of informing Subversion of the change in locations. The `svn switch` command does just that. The `--relocate` option maps the URLs from one location to another, transforming `file:///` URLs into `svn+ssh://` URLs. That step is performed on the development box, not the Subversion server.

```
$ svn switch --relocate file:/// svn+ssh://source/
```

```
Password:
```

Eclipse automatically recognizes the change in location. The next time Subversion is accessed, Subversive will ask for your credentials, but it may be necessary to open and close the project to get Subversive to correctly display the new URL next to the project.

The final step in setting up Subversion is ensuring that `svnserve` will start when the host machine reboots. As with Buildbot, you do this by putting an appropriate entry in the Subversion user's crontab. On my Subversion server, it looks like this:

```
$ sudo -u svn crontab -l
```

```
@reboot /usr/local/bin/svnserve --daemon -r /usr/local/svn/repos
```

Using the Source

The Subversion repository is now available across the network, so the build slave can now obtain the source code. You add the SVN step to the builder factory f1 in buildmaster's master.cfg. Currently, the relevant section reads as follows:

```
from buildbot.process import factory
from buildbot.steps.shell import ShellCommand

f1 = factory.BuildFactory()
f1.addStep(ShellCommand(command="echo 'build was run'"))
```

The SVN build step pulls down code from the repository. The baseURL points to the Subversion repository. The baseURL is concatenated with the branch, so the trailing slash is important. As configured, this branch defaults to trunk. The SVN step checks out a fresh copy each time when clobber mode is selected.

```
from buildbot.process import factory
from buildbot.steps.source import SVN

f1 = factory.BuildFactory()
f1.addStep(SVN, baseURL="svn://source/rsreader/",
    defaultBranch="trunk",
    mode="clobber",
    timeout=3600)
```

You reconfigure the build master using the command buildbot reconfig. This bypasses the need to restart Buildbot. Any error will be reported in twistd.log, and Buildbot will continue running with the old configuration.

```
$ sudo -u build buildbot reconfig /usr/local/buildbot/master/rsreader
```

```
sending SIGHUP to process 2152
2008-05-05 15:59:56-0700 [-] loading configuration from /usr/local/buildbot/master
…
Reconfiguration appears to have completed successfully.
```

The reconfiguration is reflected in the waterfall display too. This is shown in Figure 5-6.

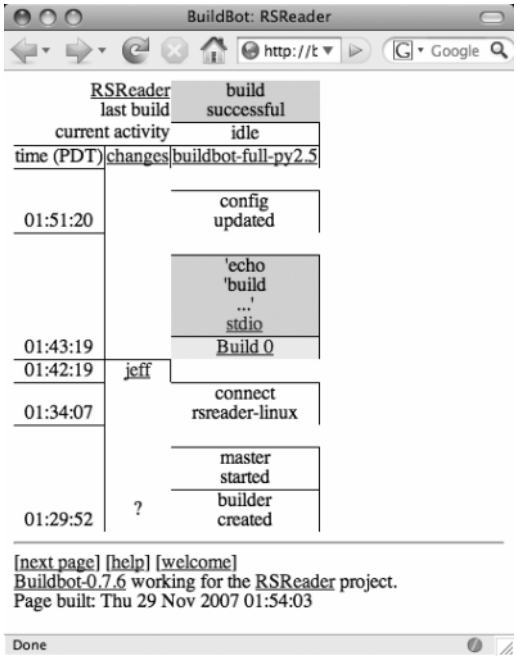

Figure 5-6. *Successful reconfiguration*

If all is configured correctly, then the next build will retrieve the source code from the repository. The build is again triggered using `buildbot sendchange`, and the waterfall display is monitored. If everything worked, then the `SVN` step will appear, and once it completes, it will be green, as shown in Figure 5-7.

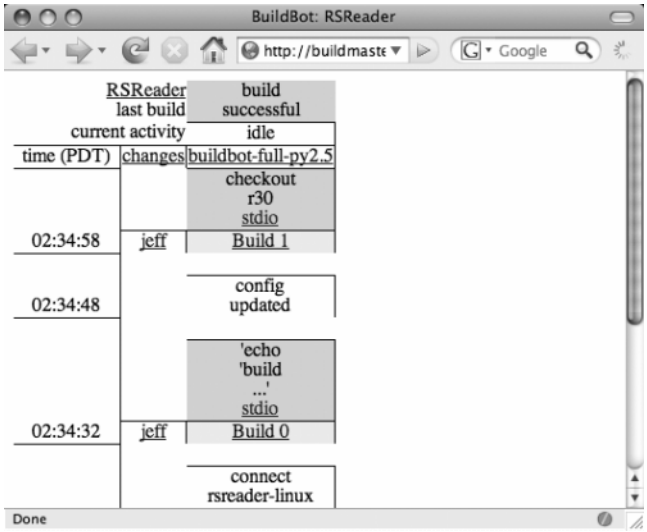

Figure 5-7. *The slave successfully checks out code.*

The successful checkout indicates that the reconfiguration was successful, that all the parameters are correct, and that the build slave can retrieve code from Subversion. It is useful to note that the checkout message in the SVN step includes the revision number; in this case revision 30 (r30).

Subversion to Buildbot, Over

Subversion should send notifications to Buildbot when changes are committed. This is done with hooks. *Hooks* are programs triggered by events in Subversion. The events are commits, locking, and revision property changes, and there are hooks for several steps in each kind of event. The commit event is the interesting one for our purpose.

The commit process has three hooks. The *start-commit* hook is called before the commit transaction is created. The *pre-commit* hook is called after the commit transaction has been created, but before it has been submitted. Both of these hooks can abort the commit. The *post-commit* hook is called after a successful commit, and it is the one of interest. It takes the repository path and the created revision as arguments, and its return code is ignored.

The hook sends notifications using `svn_buildbot.py`. This program ships in the Buildbot `contrib` directory. Recall that you installed Buildbot with `easy_install -b /tmp/bbinst buildbot`, and that left a copy of the full package in `/tmp/bbinst`. You can copy `svn_buildbot.py` from there to the Subversion directories.

The hooks themselves are stored in the directory `/usr/local/svn/repos/hooks`. As shipped, the directory contains templates demonstrating how each hook is used. The post-commit hook is named `/usr/local/svn/repos/hooks/post-commit`, and it must be executable.

```
$ sudo -u svn mkdir /usr/local/svn/bin
$ sudo -u svn cp /tmp/bbinst/buildbot/contrib/svn_buildbot.py➥
 /usr/local/svn/bin/
$ sudo -u svn vi /usr/local/svn/repos/hooks/post-commit
```

```
...some editing...
```

```
$ sudo -u svn chmod a+x /usr/local/svn/repos/hooks/post-commit
$ cat /usr/local/svn/repos/hooks/post-commit
```

```
#!/bin/sh
REPOS="$1"
REV="$2"
MASTER=buildmaster
PORT=4484
/usr/local/svn/bin/svn_buildbot.py --repository "$REPOS" \
                                   --revision "$REV" \
                                   --bbserver $MASTER \
                                   --bbport $PORT
```

The final step is testing the hook by submitting a change to the codeline and then checking the result on the waterfall display, as shown in Figure 5-8. This is done on the development machine:

```
$ cat " " >>  setup.py
$ svn commit -m "Just a minor change to trigger a build"
```

```
Password:

Sending        setup.py
Transmitting file data .
Committed revision 31.
```

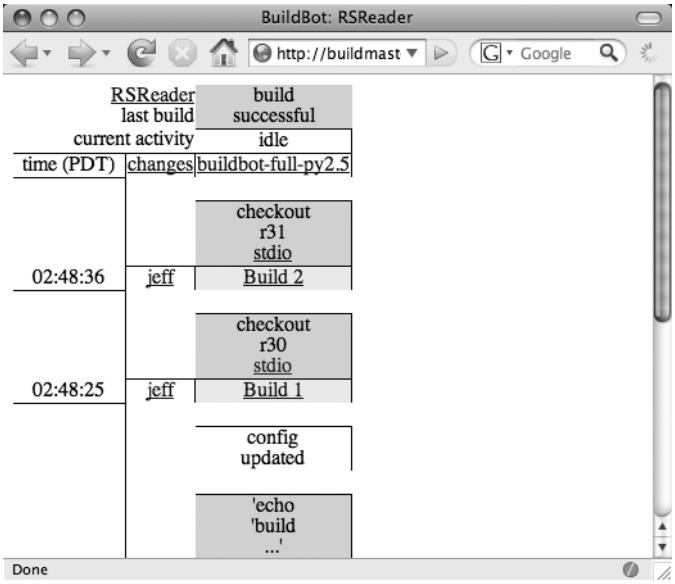

Figure 5-8. *The build was successfully triggered by a Subversion submission.*

A Python for Every Builder

I haven't said this for a while, so I'll say it again. The goal is to produce a clean build every time. This requires removing all packages and installed scripts from the Python installation. The easiest way of preventing builders from stepping on each other is to provide each one with its own interpreter. Some people may disagree with me, but disk space is cheap, and the cleansing process is straightforward and easily automated.

Python is installed into the build slave's root directory. The Python version is explicitly named so that multiple Python versions can be installed in the same build slave. In this case, the Python build prefix will be `/usr/local/buildbot/slave/rsreader/full-py2.5/python2.5`.

The decision not to track the minor version is a conscious one. If there comes a point where the minor Python revisions are important, then I will track them.

```
$ curl -L -o Python-2.5.1.tgz http://www.python.org/ftp/➥
python/2.5.1/Python-2.5.1.tgz
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 10.5M  100 10.5M    0     0  31388      0  0:05:52  0:05:52 --:--:-- 49899
```

```
$ tar xvfz Python-2.5.1.tgz
```

```
Python-2.5.1/
Python-2.5.1/Python/
...
Python-2.5.1/pyconfig.h.in
Python-2.5.1/install-sh
```

```
$ cd Python-2.5.1
$ ./configure --prefix=/usr/local/buildbot/slave/rsreader/full-py2.5/python2.5
```

```
checking MACHDEP... linux2
checking EXTRAPLATDIR...
... many minutes pass ...
creating Modules/Setup.local
creating Makefile
```

```
$ make
```

```
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall➥
-Wstrict-prototypes  -I. -I./Include   -DPy_BUILD_CORE➥
 -o Modules/python.o ./Modules/python.c
... many more minutes pass ...
changing mode of build/scripts-2.5/idle from 664 to 775
changing mode of build/scripts-2.5/smtpd.py from 664 to 775
```

```
$ make test
```

```
case $MAKEFLAGS in \
        *-s*)  CC='gcc -pthread' LDSHARED='gcc -pthread -shared'➥
 OPT='-DNDEBUG -g -O3 -Wall -Wstrict-prototypes' ./python -E
...
    test_timeout test_urllib2net test_urllibnet test_winreg
    test_winsound test_zipfile64
```

```
$ make install
```

```
/usr/bin/install -c python /usr/local/buildbot/slave/rsreader/full-py2.5/➥
python2.5/bin/python2.5
if test -f libpython2.5.so; then \
...
/usr/bin/install -c -m 644 ./Misc/python.man \
                /usr/local/buildbot/slave/rsreader/python2.5/man/man1/python.1
```

Finally, a Real Build Succeeds

Builds are produced using the `Compile` step. The `Compile` step will run the statement `python setup.py build` as if run from the command line. However, it should use the private Python interpreter installed in the previous section. Absolute paths are out of the question. The build clients may be rearranged in the future, or they may be relocated by others with good reasons for placing them elsewhere, so relative paths should be used. There are many paths to keep straight, so they're summarized in Table 5-1.

Table 5-1. *Paths Used on the Slave*

Path	Description
`/usr/local/buildbot/slave/rsreader`	The build slave's root directory
`/usr/local/buildbot/slave/rsreader/full-py2.5`	The *builder* directory defined in `master.cfg`
`/usr/local/buildbot/slave/rsreader/full-py2.5/build`	The *build* directory where the builder factory runs
`/usr/local/buildbot/slave/rsreader/full-py2.5/python2.5`	The slave's local Python 2.5 installation
`/usr/.../full-py2.5/python2.5/bin/python`	The Python interpreter
`/usr/.../full-py2.5/python2.5/lib/python2.5/site-packages`	Locally installed packages
`/usr/.../full-py2.5/python2.5/site-bin`	Locally installed executables
`../python2.5/bin/python`	The relative path from the build directory to the interpreter
`../python2.5/lib/python2.5/site-packages`	The relative path from the build directory to the locally installed packages
`../python2.5/site-bin`	The relative path from the build directory to the locally installed executables

The `SVN` step checks out the code into the directory `full-py2.5/build` relative to the slave's directory. The builder directory, `full-py2.5`, is specified in the builder's definition in `master.cfg`, and the last subdirectory is always `build`. The builder executes in this directory. The relative path from the build directory to the locally installed Python 2.5 interpreter is `../python2.5/bin/python`.

After adding the new step, the relevant section of `master.cfg` looks like this:

```
from buildbot.process import factory
from buildbot.steps.source import SVN
from buildbot.steps.shell import Compile

f1 = factory.BuildFactory()
f1.addStep(SVN, baseURL="svn://repos/rsreader/",
    defaultBranch="trunk",
    mode="clobber",
    timeout=3600)
f1.addStep(Compile, command=["../python2.5/bin/python",
                             "./setup.py",
                             "build"])
```

You reconfigure Buildbot and trigger a build with `buildbot sendchange`, and the change is reflected in the waterfall display. Figure 5-9 shows the completed build.

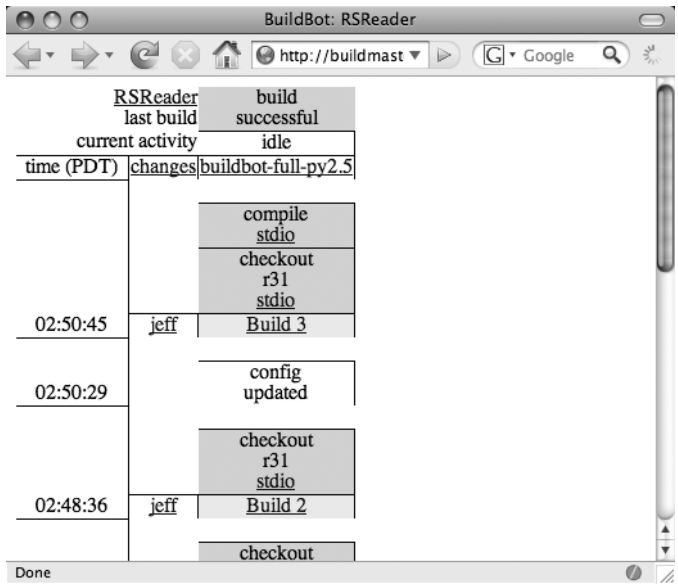

Figure 5-9. *The build step succeeds.*

Installing the Build

The `Install` step will generate executables. By default, these executables will be placed into the local Python `bin` directory along with the Python interpreter and other stock tools. The build will need to remove the generated artifacts—however, separating them from the preexisting tools is problematic. Fortunately, you can specify a different directory for executables with the `--install-script` option. This is not an issue for packages, as they are installed into `site-packages`. It contains no stock Python files, so it can be cleansed with impunity.

```
from buildbot.process import factory
from buildbot.steps.source import SVN
from buildbot.steps.shell import Compile, ShellCommand

f1 = factory.BuildFactory()
f1.addStep(SVN, baseURL="svn://repos/rsreader/",
    defaultBranch="trunk",
    mode="clobber",
    timeout=3600)
f1.addStep(ShellCommand, command=["mkdir",
                                  "../python2.5/site-bin"])
f1.addStep(Compile, command=["../python2.5/bin/python",
                             "./setup.py",
                             "build"])
f1.addStep(Compile, command=["../python2.5/bin/python",
                             "./setup.py",
                             "install",
                             "--install-scripts",
                             "../python2.5/site-bin"])
```

The `Install` step output shown in Figure 5-9 indicates that the `docutils` package was installed. Triggering the build a second time yields the log shown in Figure 5-10. It shows that `docutils` was not actually installed. Instead, the previous installation was used. This may seem a pedantic point, but I've encountered many situations in which a clean install would fail for one reason or another, but subsequent installations would succeed. It's not an acceptable answer to simply tell your customer, "Just reinstall it, and it will work." Each build should yield a clean result.

The build code functions, but it is getting messy. The build code is going to be around as long as the application—perhaps even longer. There is a tendency to neglect build configurations and build code. Normal programming practices aren't applied, and eventually the code rots under the weight of neglect. Changes to build code are easy to make if they're small, and the key to keeping them small is making the changes as the need is recognized.

Both the path to `python` and the path to the `site-bin` directory are replicated. We'll extract them into constants. This is refactoring—changing the structure of code to improve readability and maintainability without altering its function. Refactoring is best done when the code can be tested. Fortunately, the build configuration code has a built-in test and test harness, which is the build system itself. If you can't make a build, then your configuration changes are broken.

```
from buildbot.process import factory
from buildbot.steps.source import SVN
from buildbot.steps.shell import Compile, Install, ShellCommand

python = "../python2.5/bin/python"
site_bin = "../python2.5/site-bin"
```

```
f1 = factory.BuildFactory()
f1.addStep(SVN, baseURL="svn://repos/rsreader/",
    defaultBranch="trunk",
    mode="clobber",
    timeout=3600)
f1.addStep(ShellCommand, command=["mkdir", build_bin])
f1.addStep(Compile, command=[python, "./setup.py", "build"])
f1.addStep(Compile, command=[python, "./setup.py",  "install",
                             "--install-scripts", site-bin])
```

Here, we're saving `master.cfg`, reconfiguring Buildbot, and triggering a build. The results are the same, which is good and bad. It's good because we've verified your changes, and they work as expected. It's bad because the old build is still installed, so the installation directories `python2.5/lib/2.5/site-packages` and `python2.5/site-bin` must be removed and recreated.

```
from buildbot.process import factory
from buildbot.steps.source import SVN
from buildbot.steps.shell import Compile, ShellCommand

python = "../python2.5/bin/python"
site_bin = "../python2.5/site-bin"
site_pkgs = "../python2.5/lib/python2.5/site-packages"

f1 = factory.BuildFactory()
f1.addStep(SVN, baseURL="svn://repos/rsreader/",
    defaultBranch="trunk
    mode="clobber",
    timeout=3600)
f1.addStep(ShellCommand, command=["rm", "-rf", site_pkgs])
f1.addStep(ShellCommand, command=["mkdir", site_pkgs])
f1.addStep(ShellCommand, command=["rm", "-rf", site_bin])
f1.addStep(ShellCommand, command=["mkdir", site_bin])
f1.addStep(Compile, command=[python, "./setup.py", "build"])
f1.addStep(Compile, command=[python, "./setup.py",  "install",
                             "--install-scripts", site_bin])
```

Once again, we're saving the changes, reconfiguring Buildbot, and triggering a build. The waterfall display is shown in Figure 5-10. It clearly shows the new step and the last step's output—it is clear that the packages have been freshly installed.

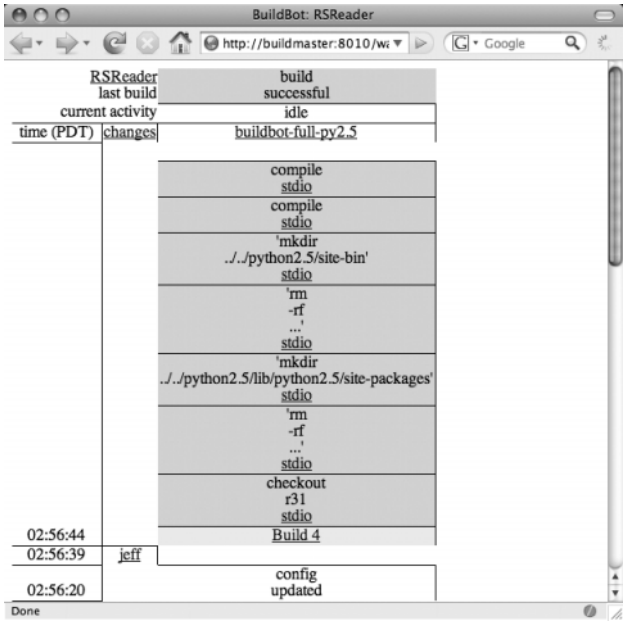

Figure 5-10. *A complete clean build*

Supporting Python 2.4 Builds

This chapter's ultimate goal is supporting both Python 2.4 and Python 2.5, and we're getting very close. Python 2.4 must be installed, a new scheduler added, and a new builder defined. The master configuration must also be refactored along the way.

Installing Python 2.4 is precisely analogous to installing Python 2.5. It is placed in a directory named `python2.4`, directly beneath the slave's root directory. Supplying the new directory to the `./configure` step is the only change in procedure.

```
$ curl -L -o Python-2.4.4.tgz http://www.python.org/ftp/python/2.4.4/➥
Python-2.4.4.tgz
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 10.5M  100 10.5M    0     0  31388      0  0:05:52  0:05:52 --:--:-- 49899
```

```
$ tar xvfz Python-2.4.4.tgz
```

```
Python-2.4.4/
Python-2.4.4/Python/
...
Python-2.4.4/pyconfig.h.in
Python-2.4.4/install-sh
```

```
$ cd Python-2.4.4
$ ./configure --prefix=/usr/local/buildbot/slave/rsreader/full-py2.4/python2.4
```

```
checking MACHDEP... linux2
checking EXTRAPLATDIR...
... many minutes pass ...
creating Modules/Setup.local
creating Makefile
```

```
$ make; make test; make install
```

```
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall➥
-Wstrict-prototypes  -I. -I./Include   -DPy_BUILD_CORE➥
 -o Modules/python.o ./Modules/python.c
... many more minutes pass ...
changing mode of build/scripts-2.4/idle from 664 to 775
changing mode of build/scripts-2.4/smtpd.py from 664 to 775
```

There are three things that must be done to a new builder. First, the factory producing it must be created. Then it must be created using that factory. Finally, the builder has to be scheduled. In this case, the process amounts to little more than duplicating the Python 2.5 definitions and changing the version number to 2.4, although some new constants will need to be created along the way.

The clarity-to-maintainability ratio for the scheduler and builder sections clearly favors duplication. Just as clearly, the clarity-to-maintainability ratio militates against duplicating the builder factory definition. It doubles the number of constants and the number of lines. If the build process is modified, it will need to be modified in both places, and I guarantee it will be modified before the chapter is out. There is much to be gained from refactoring here.

You'll encapsulate the builder factory in a function. That function will take the Python version as its argument, and it will return a builder factory for that Python version. Along the way, you'll extract many of the constants into functions.

The changes are made in two parts. You'll refactor the builder and builder factory and test them. Then the new 2.4 builder and schedulers will be added. This isolates the changes in each step, making debugging much easier.

```
####### BUILDERS

from buildbot.process import factory
from buildbot.steps.source import SVN
from buildbot.steps.shell import Compile, ShellCommand

def python_(version):
    return  "../python%s/bin/python" % version

def site_bin_(version):
    return  "../python%s/site-bin" % version
```

```
def site_pkgs_(version):
    subst = {'v': version}
    path = "../python%(v)s/lib/python%(v)s/site-packages"
    return path % subst

def pythonBuilder(version):
    python = python_(version)
    site_bin = site_bin_(version)
    site_pkgs = site_pkgs_(version)

    f = factory.BuildFactory()
    f.addStep(SVN, baseURL="svn://repos/rsreader/",
              defaultBranch="trunk",
              mode="clobber",
              timeout=3600)
    f.addStep(ShellCommand, command=["rm", "-rf", site_pkgs])
    f.addStep(ShellCommand, command=["mkdir", site_pkgs])
    f.addStep(ShellCommand, command=["rm", "-rf", site_bin])
    f.addStep(ShellCommand, command=["mkdir", site_bin])
    f.addStep(Compile, command=[python, "./setup.py", "build"])
    f.addStep(Compile, command=[python, "./setup.py", "install",
                                "--install-scripts", site_bin])
    return f

b1 = {'name': "buildbot-full-py2.5",
      'slavename': "rsreader-linux",
      'builddir': "full-py2.5",
      'factory': pythonBuilder('2.5'),
      }
```

You've now parameterized the builder factory. A reconfiguration and rebuild verifies that it works correctly. Now that you've made these changes, the Python 2.4 builder can be added. You'll add the schedule, define the builder, and add the builder to the `builders` property:

```
####### SCHEDULERS

from buildbot.scheduler import Scheduler
c['schedulers'] = []
c['schedulers'].append(Scheduler(name="rsreader under python 2.5",
                       branch=None,
                       treeStableTimer=60,
                       builderNames=["buildbot-full-py2.5"]))
c['schedulers'].append(Scheduler(name="rsreader under python 2.4",
                       branch=None,
                       treeStableTimer=60,
                       builderNames=["buildbot-full-py2.4"]))
```

```
...

####### BUILDERS

...

b1 = {'name': "buildbot-full-py2.5",
      'slavename': "rsreader-linux",
      'builddir': "full-py2.5",
      'factory': pythonBuilder('2.5'),
      }
b2 = {'name': "buildbot-full-py2.4",
      'slavename': "rsreader-linux",
      'builddir': "full-py2.4",
      'factory': pythonBuilder('2.4'),
      }

c['builders'] = [b1, b2]
```

This time when you run the build, the second builder shows up in a second column, as in Figure 5-11.

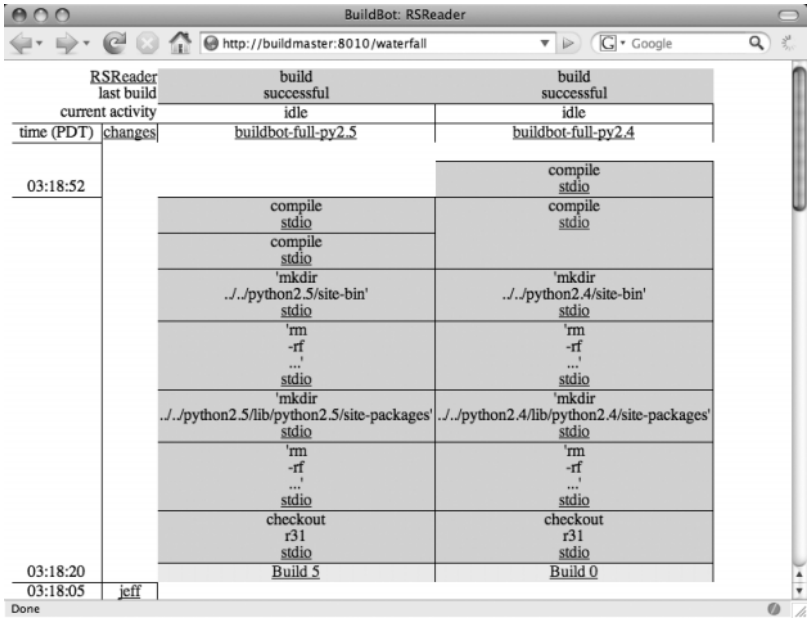

Figure 5-11. *Simultaneous Python 2.4 and 2.5 builds*

The second builder executes in parallel with the first. There are differences that are immediately apparent. The build step takes much longer under 2.4 than under 2.5. If you look at the log, the reason for this should quickly become clear: `ez_setup.py` is not using the locally provided copy of Setuptools; it is downloading Setuptools from the network instead.

Ensuring Local Dependency Processing

In Chapter 6, which deals with unit testing, I introduce a package called Nose. It will be a required dependency for RSReader. Adding it now gives me an opportunity to demonstrate how to restrict dependencies to local installation. This is done through `easy_install`'s `--allow-hosts` option. If the option is defined, then `easy_install` will only download eggs from servers whose host name matches its pattern. No hosts are matched if the pattern is `"None"`, so this effectively blocks all external access.

The Setuptools `install` command calls `easy_install` to process missing dependencies. It is the `easy_install` command that observes the `allow-hosts` option. Unfortunately, `install` knows nothing about the `--allow-hosts` option, and there is no way to hand the option directly from `install` to `easy_install`. However, it can be specified in the project's `setup.cfg` file.

The build server should always enforce this option to catch missing packages. It could be set in `setup.cfg` within the codeline—and indeed it may always be—but if it is removed, then the install will silently retrieve the packages from the network. Instead, we'll use the command `setup.py setopt` to fix the value before each build begins:

```
f = factory.BuildFactory()
f.addStep(SVN, baseURL="svn://repos/rsreader/",
          defaultBranch="trunk",
          mode="clobber",
          timeout=3600)
f.addStep(ShellCommand, command=["rm", "-rf", site_pkgs])
f.addStep(ShellCommand, command=["mkdir", site_pkgs])
f.addStep(ShellCommand, command=["rm", "-rf", site_bin])
f.addStep(ShellCommand, command=["mkdir", site_bin])
f.addStep(ShellCommand,
          command=[python, "./setup.py", "setopt",
                   ""--command", "easy_install",
                     "--option", "allow-hosts",
                     "--set-value", "None"])
f.addStep(Compile, command=[python, "./setup.py", "build"])
f.addStep(Compile, command=[python, "./setup.py", "install",
                            "--install-scripts", site_bin])
```

As always, when you make a change, you should perform a test build. Remember that the goal is ensuring build failures when a required package is missing. To check this, you add a requirement to the project's `setup.py` file without supplying the package in the project's `thirdparty` directory.

```
install_requires = [
                    'docutils == 0.4',
                    'nose == 0.10.0',
                    ],
```

You commit this change to Subversion, and it automatically triggers a build. The installation step fails, as in Figure 5-12. The failed step's output indicates that the package could not be located locally. As hoped, a missing package results in a build failure.

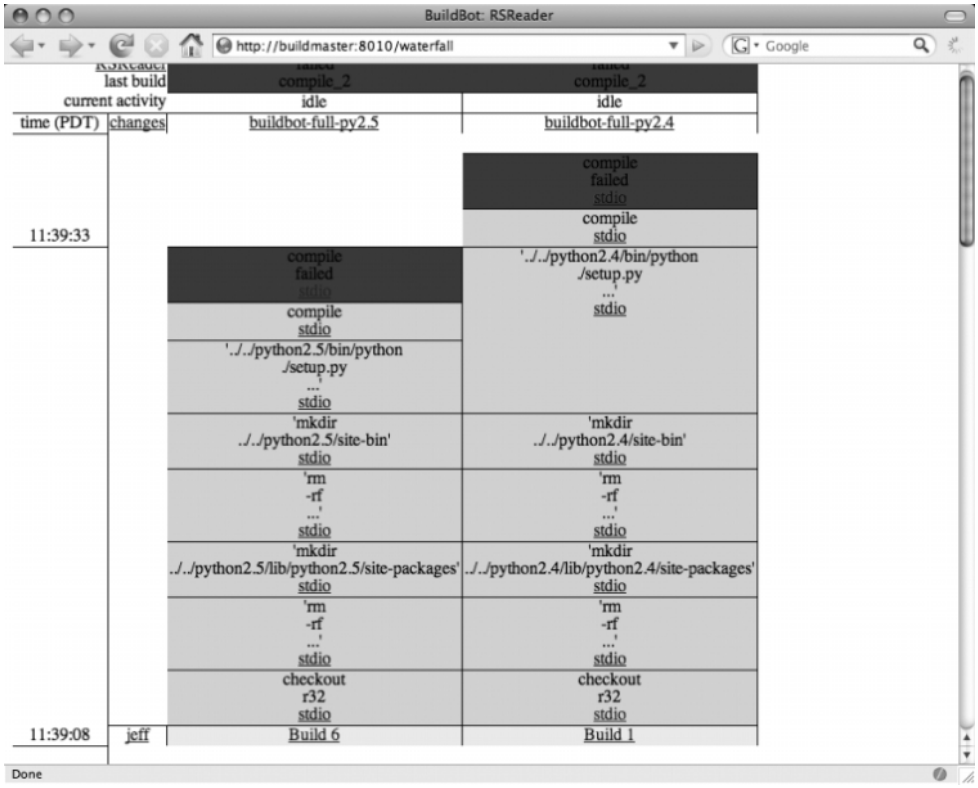

Figure 5-12. *The Install step fails when a dependency is missing.*

You download nose from `http://somethingaboutorange.com/mrl/projects/nose/nose-0.10.0.tar.gz`, and check the compressed archive into the `thirdparty` directory. You commit the change, and a build happens automatically. This time the build succeeds, as shown in Figure 5-13.

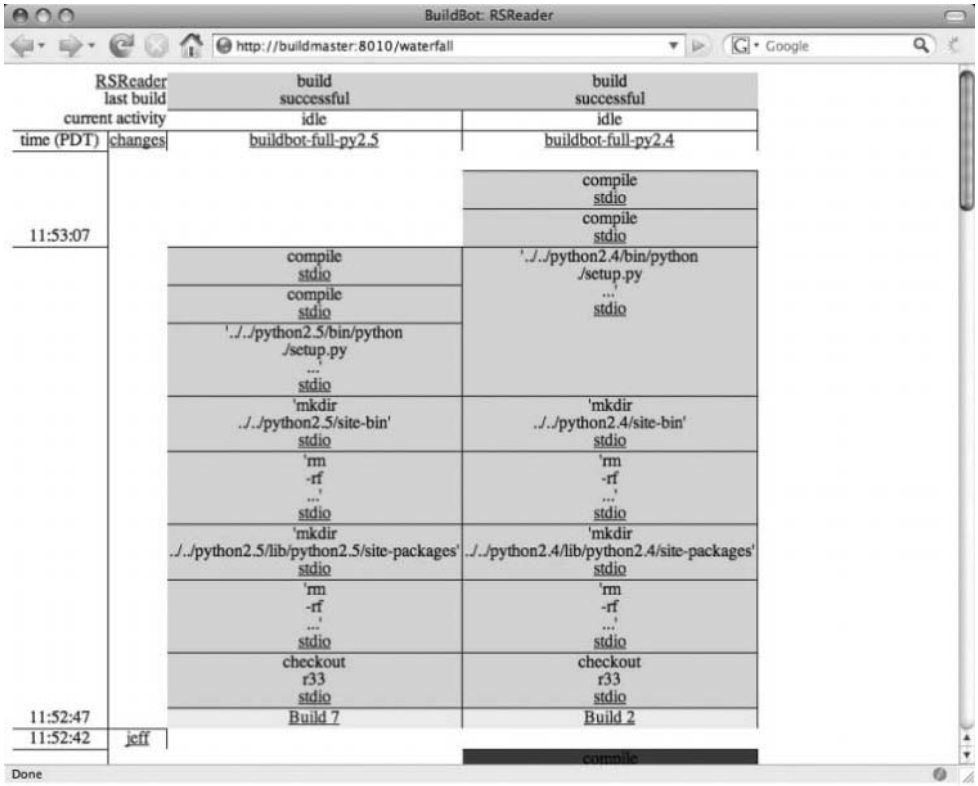

Figure 5-13. *The Install step succeeds when the dependency package is added.*

Keeping Up Appearances

The waterfall display records the name of each step in gory detail. That detailed information is available from each step's output. The information presented in the waterfall display should be understandable at a glance. The only step with an immediately clear meaning is `Compile`. The description presented for each step should be equally succinct and precise.

You accomplish this by modifying a pair of keyword properties in each build step. The message contained in the `description` keyword is shown while a step is in progress, and the message contained in the `descriptionDone` keyword is shown when a step is complete. We can add these keywords to each step to increase the waterfall display's clarity:

```
f = factory.BuildFactory()
f.addStep(SVN, baseURL="svn://repos/rsreader/",
   defaultBranch="trunk",
   mode="clobber",
   timeout=3600)
f.addStep(ShellCommand,
          command=["rm", "-rf", site_pkgs],
          description="removing old site-packages",
          descriptionDone="site-packages removed")
f.addStep(ShellCommand,
          command=["mkdir", site_pkgs],
          description="creating new site-packages",
          descriptionDone="site-packages created")
f.addStep(ShellCommand,
          command=["rm", "-rf", site_bin],
          description="removing old site-bin",
          description="site-bin removed")
f.addStep(ShellCommand,
          command=["mkdir", site_bin],
          description="creating new site-bin",
          descriptionDone="site-bin created")
f.addStep(ShellCommand,
          command=[python, "./setup.py", "setopt",
                   ""--command", "easy_install",
                     "--option", "allow-hosts",
                     "--set-value", "None"],
          description="Setting allow-hosts to None",
          descriptionDone="Allow-hosts set to None")
f.addStep(Compile, command=[python, "./setup.py", "build"])
f.addStep(ShellCommand,
          command=[python, "./setup.py",  "install",
                   "--install-scripts", site_bin],
          description="Installing",
          descriptionDone="Installed")
return f
```

The resulting waterfall display is shown in Figure 5-14. The labels are more concise and informative, and the resulting display uses less space. When using many builders, this can become a significant factor. It gives me a warm, fuzzy feeling, too, and that counts for a lot.

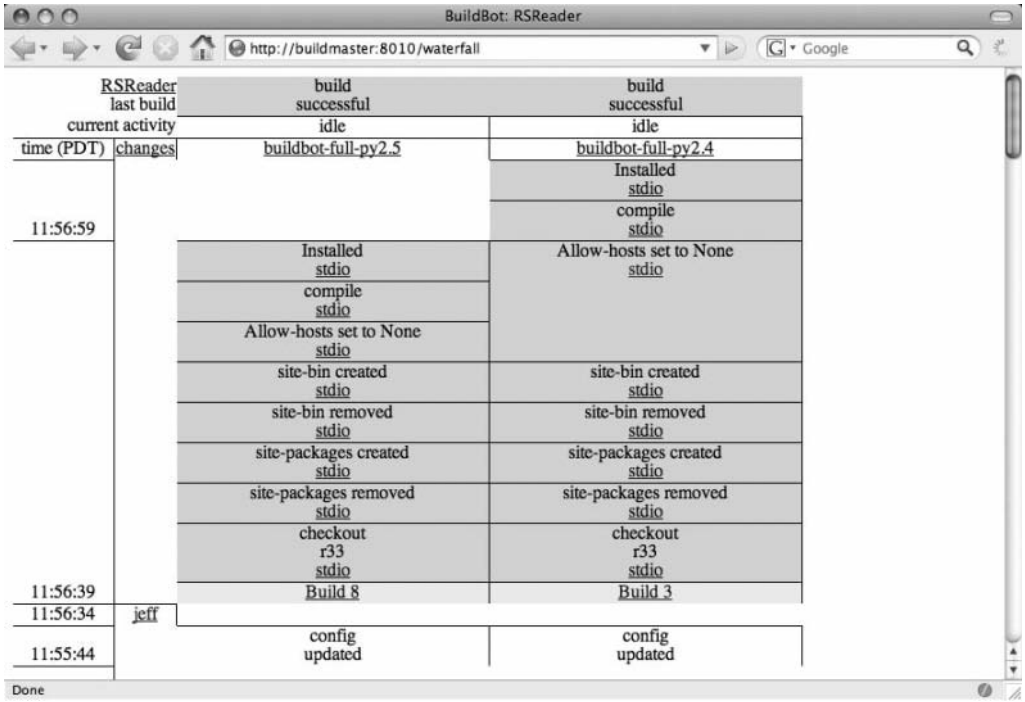

Figure 5-14. *More readable step descriptions*

Summary

Clean, repeatable builds are an easily achievable outcome using an external build system. Buildbot is one such system. A Buildbot system consists of four components: the Subversion server, the Buildbot master, one or more Buildbot slaves, and the development environments. A remotely accessible Subversion repository allows these roles to be distributed to many hosts.

Buildbot is implemented on top of Twisted. Buildbot and Twisted must be installed on the Buildbot master, the Buildbot slaves, and the Subversion repository server. It is not necessary to install it on the developers' machines. The Subversion repository triggers builds whenever code is committed. This is done through a post-commit hook.

The *build master* controls and configures *build slaves*, which are the machines that perform builds. The Buildbot master finds out about unprocessed changes through *change sources*. S*chedulers* trigger builders when certain conditions are satisfied. A *builder* ties together a *builder factory* and a build slave, and it defines a directory where the build will be performed. Builder factories generate the steps to perform a build, and a *build step* is an action that performs one step of a build.

Build steps include actions such as synching source from a repository, compiling an application, and installing the compiled application.

The system I discussed uses per-builder Python installations. These allow the build system to completely remove all installed packages and executables when a new build is performed, and this is done without impacting the rest of the system. This allows the build to verify that completely clean installations are self-contained.

Buildbot has additional capabilities you can configure. Among these is the ability to run unit tests for each build. I haven't demonstrated this yet, but I will in the next chapter, which covers the basics of unit testing.