Continuous integration with Buildbot
CI in theory and practice using a Python-based tool

Noah Gift
June 02, 2010

The days of cowboy coding are long gone at most organizations, replaced by a renewed interest in generating quality software. Continuous integration (CI) testing is a vital component in the practice of agile programming techniques that lead to high-quality software. Learn the theory and practice of CI testing by exploring Buildbot, an open source CI system written in Python.

Continuous integration (CI) is a software development process that promotes the following principles:

- Maintain a single-source repository
- Automate the build
- Make your build self-testing
- Everyone commits every day
- Every commit should build the mainline on an integration machine
- Keep the build fast
- Test in a clone of the production environment
- Make it easy for anyone to get the latest executable
- Everyone can see what's happening
- Automate deployment

Largely popularized by Martin Fowler, the basic idea of CI is to continuously test and build your software at each branch and when code is merged into the trunk. This leads to an overall increase in the health of the code base. It can also lead to an increase in communication with team members and an opportunity to obtain feedback about the overall quality of your code. Often, people use this cycle to generate code coverage reports and other statistics.

Buildbot, like other CI systems, helps to automate this cycle of checkout, build, test. Buildbot slaves usually run on different platforms such as Win32, Solaris, Intelx64, etc. Buildbot can send an email notification when a build breaks, and it keeps track of all running builds so the developer can get a bird’s-eye view of the whole process. Finally, people often tap into the automated cycle to generate metrics about the quality of their software at any given time. The end of this article touches on metrics and why it makes sense to run them within a CI system.
Introduction to Buildbot

Before we get into the nuts and bolts of Buildbot, let's look at the architecture. As shown in Figure 1, there are essentially three layers on top of the build process. There is a version control layer, which hooks into notifications from a version control system. There is a build layer, which takes communication from the build master and returns the build results. Finally, there is a notification layer, which is often configured to send emails, or an IRC message, when the build has failed, or to have a web page display the collected results of the builds over time.

Figure 1. Buildbot architectural overview

One of the other central features of Buildbot architecture is the reliance on the Python-based Twisted library to handle asynchronous communication between the master and slaves. This callback-based architecture allows for a very simple yet robust master/slave feedback loop. Find more details on Twisted in the Related topics section later in this article.

If you haven't heard of Buildbot yet, a few Google searches will reveal a large collection of masters and slaves associated with open source projects both large and small. A slave, which I referred to briefly before, is literally a slave machine controlled by the master Buildbot server. Typically, a slave is one of many slaves that each run a different test platform. This is an important concept in the Buildbot server world. For example, you might be on the mailing list for an open source project and hear someone say, "Does anyone want to volunteer a virtual machine for a Windows slave"?

The Python language project itself uses a large collection of Buildbot slaves to continuously build and test the latest version of Python on as many platforms as possible. Figure 2 shows a large assortment of machines running these slave builds of the Python trunk, as well as the tests. With
Continuous integration with Buildbot

the recent advances in virtualization, it's now common to ask members of your development community to host a Buildbot slave, or to simply run several virtual machines that emulate different hardware configurations.

Figure 2. Python Buildbot

Another high-profile user of Buildbot is the Google Chrome browser project. Figure 3 shows a highly customized version of Buildbot that significantly enhances the look and feel of the Buildbot user interface. Fortunately, Google open sourced these enhancements to Buildbot, and from the Related topics section below, you can obtain the source and build this customized version.

Figure 3. Google Chrome-enhanced Buildbot

Building this specific configuration is outside the scope of this article, but I recommend that you take a look on your own. Now let's get a Buildbot master server running quickly.

Setting up Buildbot in five minutes

I ran these steps on Ubuntu 8.10, but they should work on most Linux distributions:
1. Download ez_setup.py:
   
   ```bash
   wget http://peak.telecommunity.com/dist/ez_setup.py
   ```

2. Install easy_install:
   
   ```bash
   sudo python ez_setup.py
   ```

3. Install the Python Twisted package with apt-get:
   
   ```bash
   sudo apt-get install python-Twisted
   ```

4. Follow this collective.buildbot "recipe":
   
   ```bash
   sudo easy_install collective.buildbot
   ```

At this point, a lot of stuff will start spewing out of the shell, as a bunch of packages get downloaded and installed automatically. Once this has finished, you are ready to create a Buildbot! If the installation went correctly, type at the shell prompt:

```bash
$ paster create -t buildbot my.project
$ cd my.project
```

You are almost done, really, but before you finish, let me point out a couple of gotchas that can trip you up when you first configure Buildbot. Inside your my.project/master.cfg file, you should see something like this:

### Listing 1. Contents of master.cfg

```ini
[buildout]
master-parts =
    master
    passing.project
# uncomment this to enable polling
    poller

[master]
recipe = collective.buildbot:master
project-name = passing.project project

# allow to force build with the web interface
allow-force = true

# internal port
port = 9051

# http port
wport = 9081

# buildbot url. change this if you use a virtualhost
url = http://localhost:9081/

# static files
public-html = ${buildout:directory}/public_html

slaves =
    localhost NaOaPSWb

[passing.project]
recipe = collective.buildbot:project
slave-names = localhost
vcs = hg
repositories = /home/ngift/myhgrepo

# notifications
```
mail-host = localhost
email-notification-sender = buildbot@cortese
email-notification-recipient =
  super@example.com

# run test each hour
periodic-scheduler=60

# cron build
cron-scheduler = 0 8 * * *

# You can change the sequences to build / test your app
# default options should work for most buildout based projects
build-sequence =
  /usr/bin/python2.5 bootstrap.py -c project.cfg
  /usr/bin/python2.5 bin/buildout -c project.cfg

test-sequence =
  nosetests
# zope.testing require exit with status
#  bin/test --exit-with-status

[poller]
recipe = collective.buildbot:poller
# don't forget to check this
# since it's generated from the paster template it may be a wrong url
repositories = /home/ngift/myhgrepo
#user = h4x0r
#password = passwd
poll-interval = 120

The most important things to check initially are that you have the proper source control repository, that you leave build-sequence blank at first, and that your test-sequence, "nose" in my case, will pass tests when the code is checked out of the repository you gave it. Look at the resource guide for collective.buildbot if you have any additional questions (see Related topics for a link).

Once your config file has been set up, you simply run the following two commands:

$ python bootstrap.py
$ ./bin/buildout

When you run the buildout command, you will get quite a bit of output that looks something like this:

Listing 2. Output from buildout command

{673} > ./bin/buildout
Unused options for buildout: 'master-parts'.
Installing master.
New python executable in /home/ngift/my.project
Installing setuptools.............done.
[output suppressed for space]

Once this command finishes, you're done installing Buildbot and you are ready to fire things up. Start both of the Buildbot daemons by running the following commands from the shell:

$. ./bin/master start $ ./bin/yourhostname start
If you then point a browser to the URL you set in the master.cfg file, which by default is http://localhost:9081/, you will see your shiny new Buildbot. Of course, it probably won't be doing much yet. If you give it a build script and a test runner, it will then happily check out your code, build the code, then test it automatically. You should, of course, browse through some of the configuration options later, but the hard work is essentially done.

### Generating code metrics reports

One recent intellectual development among "testing nerds" is to take advantage of the continuous integration cycle to also generate metrics about the source code. One of the most popular techniques is running the nosetest test collector with the coverage option. If you had a project named "foo", you would typically run:

```bash
nosetests --with-coverage --cover-package=example --cover-html --cover-html-dir=example_report.html test_example.py
```

This would generate both an HTML report showing all of the lines of code that weren't covered, along with output to stdout that looked like this:

#### Listing 3. Output from nosetest

```
nglep% nosetests --with-coverage --cover-package=example --cover-html test_example.py
.
Name          Stmts  Exec  Cover  Missing
--------------------------
example        2      2   100%    
--------------------------
Ran 1 test in 0.004s
OK
```

You can download example.py and test_example.py from the [Download](#) section.

Running this report every time the code is revised gives the developer and manager metadata about what is actually happening in the code. This is a perfect example of why running metrics at the same time as CI can be advantageous to a project.

One other metric tool that gives metadata about your code is PyMetrics's McCabe rating. Back in the 1970s, Thomas McCabe came up with a simple but ingenious observation about code: the more complex a piece of code, the more often it breaks. While this may seem obvious, many developers unfortunately don't seem to see the connection. By using the PyMetrics command line tool, you can determine the total number of branches per function.

Typically, you want to keep the number of branches to fewer than 10 for every method or function that you write, as it is tough to keep more than seven or eight things in context in the human brain at any one time. As a point of reference, code that scores higher than 50 is basically untestable/unmaintainable.
I have personally seen code as score high as 140 in production, and the code was pretty bad, so it did actually validate McCabe's theory. If you can catch and flag this complex, fragile code early in the development process, then it never sneaks into production even if all the tests pass.

**Conclusion**

The major benefit of continuous integration is the ability to streamline the quality-assurance cycle with automated builds of software, along with tests, and, optionally, software metrics. These builds get triggered on each change to the source and give instant feedback as well as reports for the life of the project. When CI is configured correctly, it actually becomes integrated into the process of making code, as much as writing the code itself.

Buildbot isn't the only game in town for CI testing. You might also take a look at Hudson and Bitten. Each of these allows customization with plug-ins in Python, even though Hudson was written in Python. Refer to the resources below for more information about these systems.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample Python scripts</td>
<td>example.zip</td>
<td>1KB</td>
</tr>
</tbody>
</table>
Related topics

- Get the Google Chrome Buildbot source.
- Read Martin Fowler's article on Continuous Integration.
- Wikipedia offers a backgrounder on continuous integration testing.
- The Twisted core documentation gives an overview and tutorial on the Twisted framework.
- For alternatives to Buildbot, check out Bitten.
- The collective.buildbot site gives a collection of Buildbot "buildout" recipes.
- Statement coverage for Python introduces and explains the Python coverage module coverage.py.
- PyMetrics is a tool to generate a Cyclomatic Complexity Score.
- David Stanek discusses measurement of Python Cyclomatic Complexity.
- "Using Python to create UNIX command line tools" (developerWorks, March 2008) discusses how to write command line tools in Python for admins and developers.
- "Agile planning in real life" (developerWorks, April 2009) provides hard-won advice for making agile development work in your company.
- "Automation for the people" is a developerWorks series—targeted at Java developers but easily applicable to any language—that covers numerous ways to automate and streamline various aspects of the software development process.
- Also aimed at Java developers, "Spot defects early with Continuous Integration" (developerWorks, November 2007) gives an overview of CI.
- In the developerWorks Linux zone, find hundreds of how-to articles and tutorials, as well as downloads, discussion forums, and a wealth other resources for Linux developers and administrators.
- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online.
- Follow developerWorks on Twitter.