Continuous Integration: Introduction

Authors:
Jennifer Beattie
Edmund Sutcliffe

Summary

This document illustrates the motivation for using a continuous integration system to track and build changes to your infrastructure environment.

We use the Hudson tool and a small skeleton example to show how we can integrate our automated build processes in a simple manner.

Overview

Previous documents in this series have illustrated how we can use well-known, open technology to eliminate a number of long-winded manual infrastructure tasks, by making them automated and repeatable.

We can also make it easy to integrate multiple changes to the infrastructure, by using a continuous integration (CI) tool. We use a skeleton example in this document, to show how easy it can be to use Hudson (a popular CI tool) to handle three key automation aspects:

- Monitoring for changes to infrastructure scripts in our Subversion repository
- Rebuilding the environment based on the changes
- Tracking the build history and alerting to any failures

Prerequisites

You must have a build machine with a PXE boot environment installed as per an earlier document in this series, the Hive PXE Install document.

You must have created a Subversion epository as per another earlier document in this series, the Hive Versioned Infrastructure Templates document. We reuse paths to the notional BuildTools project in that server in this document.

Installing Hudson

Hudson is a Java web application, so the WAR file which contains it can be easily installed inside Tomcat or any other standard Java application server.
The Hudson package is also directly launchable as a Java application, so we will demonstrate this ease of use here. The Hudson web site also includes an RPM-packaged version of the code.

On a suitable Linux server such as the build machine, gain root access, move into `/root` (for the purposes of this example), and download the Hudson WAR file.

Assuming Java is already installed on your local machine, launch the WAR file and you should see it start up and begin listening on port 8080:

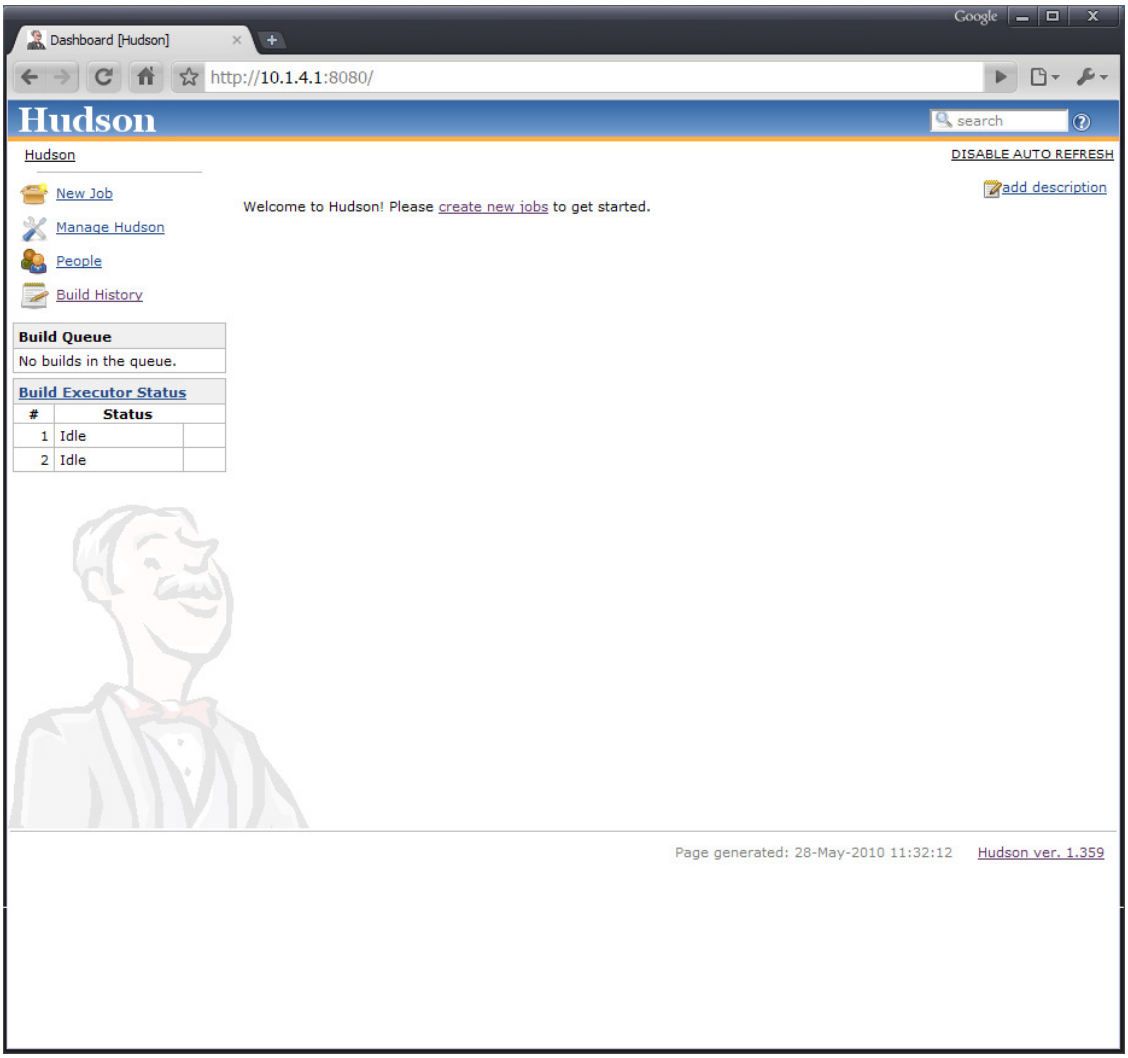
```
cd /root
wget http://hudson-ci.org/latest/hudson.war
java -jar hudson.war
```

produces output like:

```
[root@core root]# java -jar hudson.war
Running from: /root/hudson.war
[Winstone 2010/05/27 12:59:15] - Beginning extraction from war file
hudson home directory: /root/.hudson
Using one-time self-signed certificate
[Winstone 2010/05/27 12:59:19] - AJP13 Listener started: port=8009
[Winstone 2010/05/27 12:59:20] - HTTP Listener started: port=8080
[Winstone 2010/05/27 12:59:20] - Winstone Servlet Engine v0.9.10
running: controlPort=disabled
May 27, 2010 12:59:20 PM hudson.model.Hudson$4 onAttained
INFO: Started initialization
May 27, 2010 12:59:20 PM hudson.model.Hudson$4 onAttained
INFO: Listed all plugins
May 27, 2010 12:59:20 PM hudson.model.Hudson$4 onAttained
INFO: Prepared all plugins
May 27, 2010 12:59:20 PM hudson.model.Hudson$4 onAttained
INFO: Started all plugins
May 27, 2010 12:59:23 PM hudson.model.Hudson$4 onAttained
INFO: Loaded all jobs
May 27, 2010 12:59:23 PM hudson.model.Hudson$4 onAttained
INFO: Completed initialization
May 27, 2010 12:59:23 PM hudson.TcpSlaveAgentListener <init>
INFO: JNLP slave agent listener started on TCP port 32769
```

Even this simple method of launching Hudson will create a data directory under `/root/.hudson` , so that it will remember persistent state like job history across multiple instances.

Now, assuming port 8080 is open in iptables, we can point our web browser at the Hudson interface. Although it is possible to write job configuration files by hand if needed, all the useful Hudson actions can be performed through this web UI:



In the next section we will create a simple job to demonstrate how an automated environment build can run.

Configuring Hudson jobs

For demonstration purposes, we will assume that a repository is available under the folder `https://buildtools.test.example.com/svn/BuildTools/`
and we will create a one line dummy bash shell script that will indicate when the build details actually run:

```
#/!bin/bash
echo "Build runs here"
```

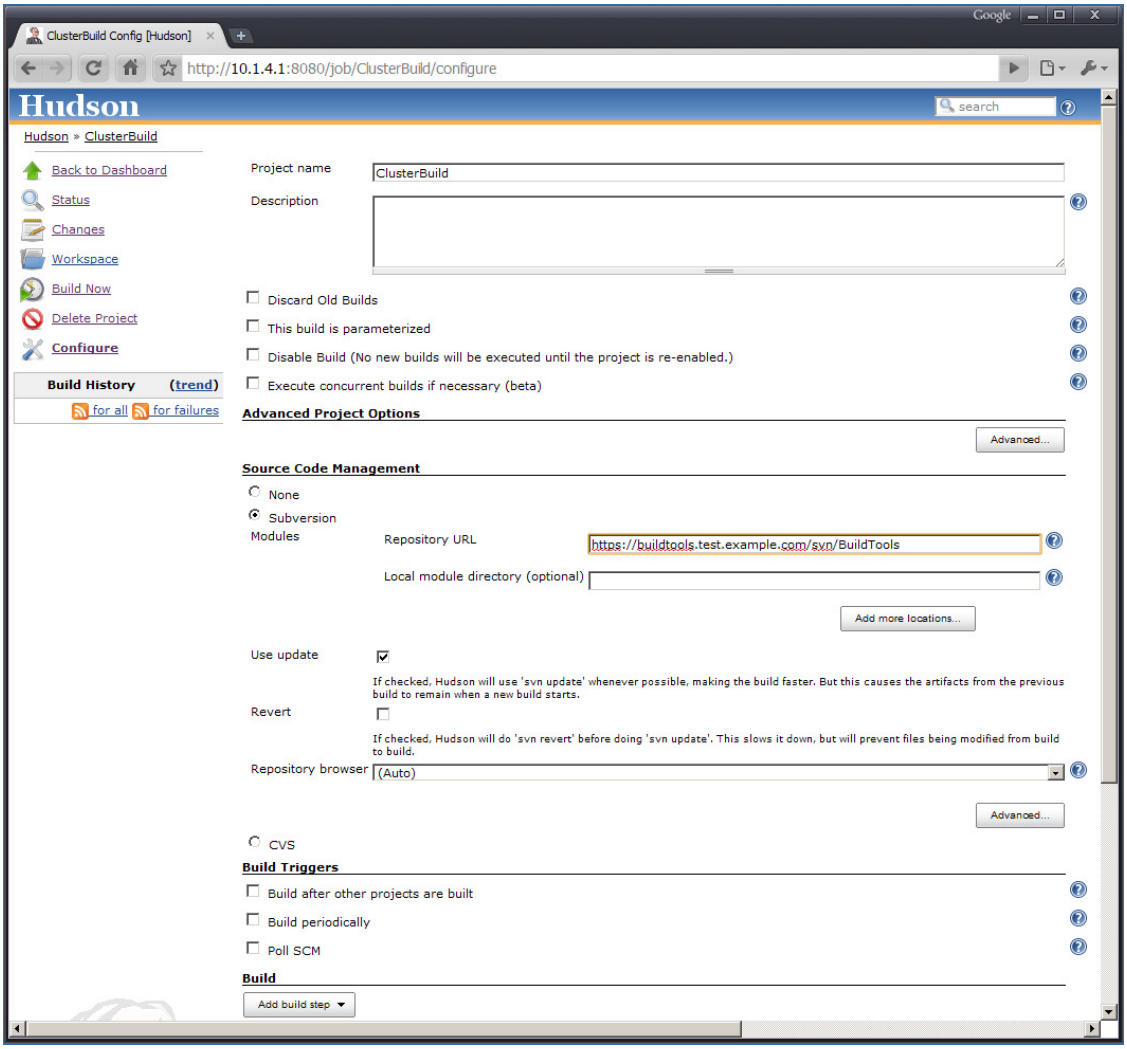
We check this into our `BuildTools` folder in the repository as `autobuild.sh` .

Note: We must also set the `svn:executable` flag on this script, so that it will be given file-system executable permissions when it is checked out. This will allow Hudson (or a human) to execute the script and run a build without errors.
The exact means of setting this flag varies depending on your client. From a command line client working on a base copy you could do

```
svn propset svn:executable true autobuild.sh
svn commit –m "Marking as executable"
```

We can now configure a Hudson job to periodically check the repository for any changes, and create a new build if any are needed -- we click the "**New Job**" link, and create a "**free-style software project**" called `ClusterBuild` .

Under **Source Code Management** we can now select "**Subversion**" and enter the URL `https://buildtools.test.example.com/svn/BuildTools`



If the repository requires authentication, as it should, then Hudson will pop up a warning message and ask us to enter credentials, which will normally be a simple userID and password combination. Note that it might be necessary to go back into the ClusterBuild configuration tab and re-enter the Repository URL in this scenario.

Under **Build Triggers** we can check "Poll SCM" so that Hudson only rebuilds when it detects a change in the repository. It will request a schedule, which is a cron-style specification of when to poll, although it provides some useful help text under the input field.

For example, we can enter
`*/5 * * * *`
in the schedule field in order to poll every 5 minutes.

Finally, under **Build**, we can click "**Add Build Step**" and choose **Execute Shell** as the action to take. Hudson creates a workspace for each job, and it will check out the

BuildTools folder into this workspace. Therefore, the shell script path we want to configure is `BuildTools/autobuild.sh` since it is relative to the workspace.

Finally, we can click **Save** to create the job and store it in Hudson.

Hudson filesystem structure

At this point we can see what Hudson is doing for us, beyond us running our own cron jobs to periodically rebuild our infrastructure.

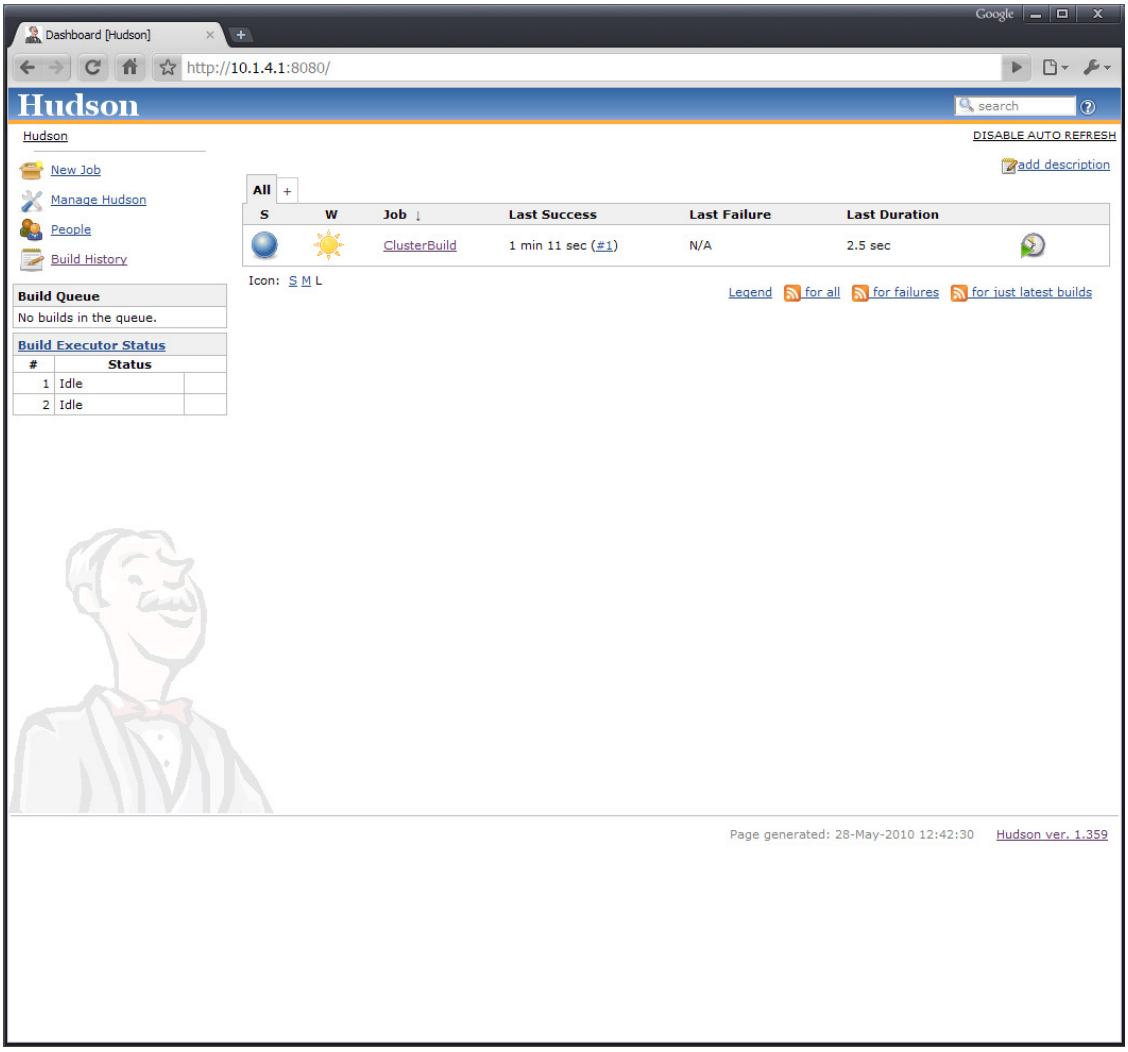
Inside its filesystem space (which in our temporary example is under `/root/.hudson` but if you are using the RPM-installed version of Hudson will be under `/var/lib/hudson`) we see a new folder `/root/.hudson/jobs/ClusterBuild` with a file `config.xml` representing this build job:

```
<?xml version='1.0' encoding='UTF-8'?>
<project>
  <actions/>
  <description></description>
  <keepDependencies>false</keepDependencies>
  <properties/>
  <scm class="hudson.scm.SubversionSCM">
    <locations>
      <hudson.scm.SubversionSCM_-ModuleLocation>

<remote>https://buildtools.test.example.com/svn/BuildTools</remote>
      </hudson.scm.SubversionSCM_-ModuleLocation>
    </locations>
    <useUpdate>true</useUpdate>
    <doRevert>false</doRevert>
    <excludedRegions></excludedRegions>
    <includedRegions></includedRegions>
    <excludedUsers></excludedUsers>
    <excludedRevprop></excludedRevprop>
    <excludedCommitMessages></excludedCommitMessages>
  </scm>
  <canRoam>true</canRoam>
  <disabled>false</disabled>

<blockBuildWhenUpstreamBuilding>false</blockBuildWhenUpstreamBuilding>
  <triggers class="vector">
    <hudson.triggers.SCMTrigger>
      <spec>*/5 * * * *</spec>
    </hudson.triggers.SCMTrigger>
  </triggers>
  <concurrentBuild>false</concurrentBuild>
  <builders>
    <hudson.tasks.Shell>
      <command>BuildTools/autobuild.sh</command>
    </hudson.tasks.Shell>
  </builders>
  <publishers/>
  <buildWrappers/>
</project>
```

It is also possible, therefore, to edit these jobs by hand rather than via the UI, and save the job information elsewhere if we wish to harden the Hudson build process itself for better availability.

Using Hudson

If we return to the Hudson home page, we will see the new job and its recent status:

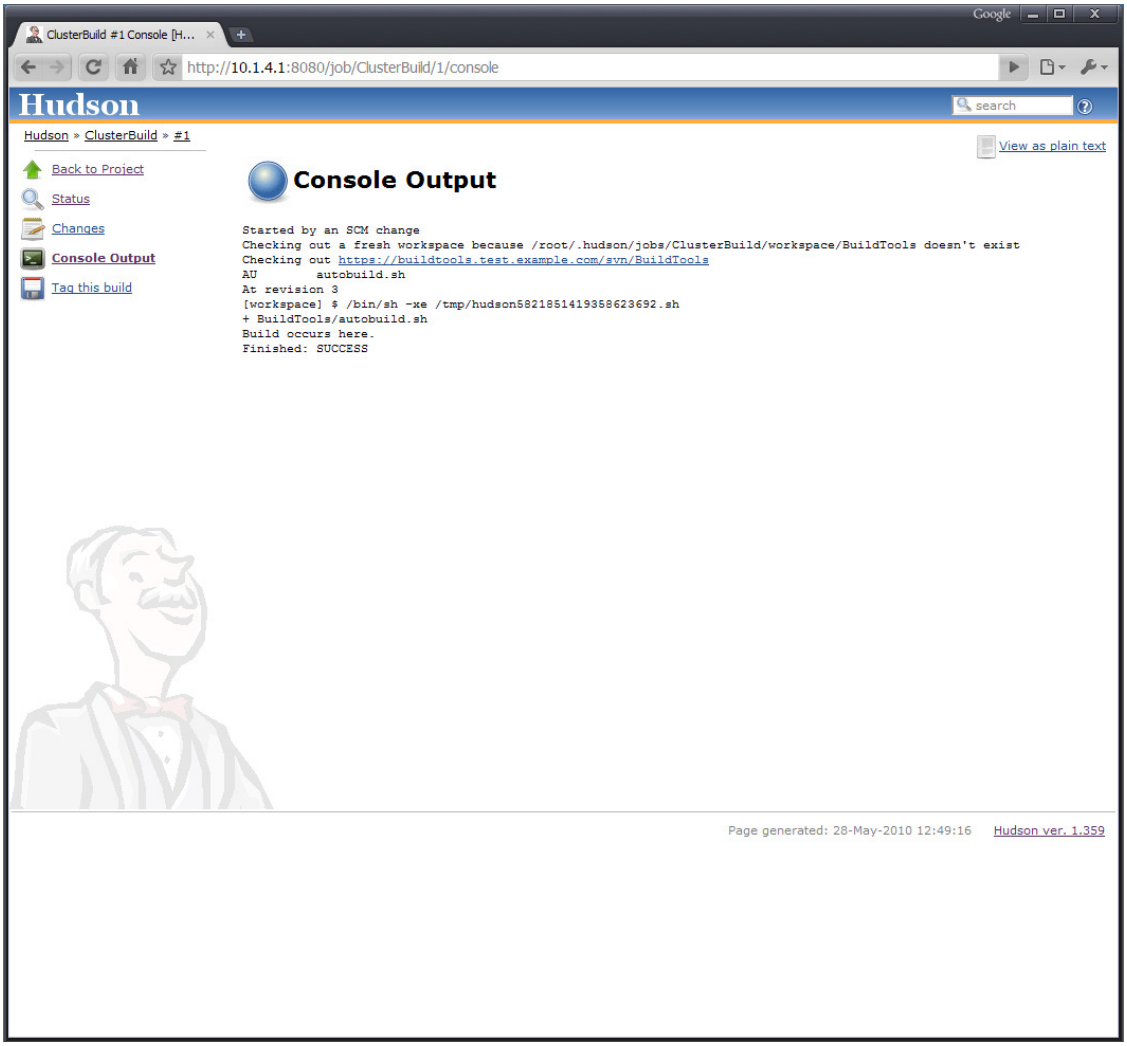


The blue ball indicates a successful build (red would indicate failure - the build could not complete; yellow would indicate an unstable build - the build completed but a follow-on publisher action such as a test suite reported failure).

The weather icon is used to indicate the *history* of recent builds. The "sunny" icon shows that recent builds have all succeeded. If recent builds had contained failures, the weather icon might be cloudy or stormy.

Now, if we click on the job link to enter the job specific page, we can see links to **Configure** the job, **Build Now** to force a build, or check the **Subversion Polling Log** to make sure it is tracking changes.

Moreover, the **Build History** in the left column allows us to drill into any recent build and see the specific actions that were taken. Using the **Console Output** link we see:



We can see Hudson detecting the changed repository (in this case because it is the first build we have attempted), checking out our build templates and performing the build. The return code of `0` from our script tells Hudson that it was successful; if the script return code was `> 0` then Hudson would report failure.

Automatic change tracking

There is one more thing we need to demonstrate in this introduction, in order to close the loop of our build cycle. That is, let us make a change to our `autobuild.sh` script so that it writes to the filesystem, and look at where Hudson writes the corresponding files.

This will enable us to write our controller scripts so that new successful builds can be approved in a given environment, and made available for host instances to boot from. It will also demonstrate Hudson automatically detecting changes.

We edit the `autobuild.sh` script so that it looks like:

```
#!/bin/bash
mkdir container
echo "Build occurs here." >> container/testout.txt
```

and check the new contents in to Subversion.

After 5 minutes, we should see a new build appear in the Hudson log as it picks up the change and builds from the new template.

If we look in the filesystem under `/root/.hudson/jobs/ClusterBuild/workspace`, we will see the results of the build:

```
[root@core workspace]# pwd
/root/.hudson/jobs/ClusterBuild/workspace
[root@core workspace]# ls -alh
total 16K
drwxr-xr-x 4 root root 4.0K May 28 12:59 .
drwxr-xr-x 4 root root 4.0K May 28 12:59 ..
drwxr-xr-x 3 root root 4.0K May 28 12:59 BuildTools
drwxr-xr-x 2 root root 4.0K May 28 12:59 container
[root@core workspace]# ls -alh container/
total 12K
drwxr-xr-x 2 root root 4.0K May 28 12:59 .
drwxr-xr-x 4 root root 4.0K May 28 12:59 ..
-rw-r--r-- 1 root root   19 May 28 12:59 testout.txt
[root@core workspace]# cat container/testout.txt
Build occurs here.
```

Hudson makes the `$WORKSPACE` environment variable available to our scripts giving the absolute path to the workspace. Clearly, we can now write our template scripts to publish the results of the build, and indeed Hudson provides a wealth of plugins to ease the process of distributing the build results.

References

Hudson home page
http://hudson-ci.org/

Hudson plugins
http://wiki.hudson-ci.org/display/HUDSON/Plugins

Alternative Continuous Integration Systems:
CruiseControl
(Java-based alternative, also has a .NET port)
http://cruisecontrol.sourceforge.net/

BuildBot
(Python-based, has some nice reporting tools)
http://buildbot.net/trac
http://buildbot.net/trac/wiki/ScreenShots

Team Foundation Server
(Microsoft's commercial offering, includes Team Build server based on the msbuild tool)
http://www.microsoft.com/visualstudio/
http://msdn.microsoft.com/en-US/library/ms364045(v=VS.80).aspx