When introducing version control into a web development environment, the process often happens gradually. Generally, one developer will set up a repository, throw all the code in there, and maintain two working copies: one to develop with, and one to act as the production codebase. On occasion, these will even be the same working copy.

At this point, it's simple to work the version control: the developer commits from his local working copy, and updates the "live" working copy to the new revision. There are, of course, a couple of drawbacks to this approach:

No testing phase:
Unless the developer has a testing environment set up on his local machine, there's no way to test code that's being written, until it's sitting on the production server. If the code is bug-ridden this presents a problem, since the live dataset is being corrupted.

Single-developer testing:
Even if the developer in question has a test environment set up, a new developer assigned to the product will not be able to test his work without a replica of that setup on his own machine. Ideally, a testing machine would be made available so that any developer could test their work without recourse to a specialised setup on their own computer.

Manual deployment:
Once a developer is satisfied that their work will stand up in production, whether this be tested with a common testbed or on their own workstation, the repository now has to be `export`ed, and copied to the live server. Depending on the convolutions required to connect to the live server, this can be a tedious and/or complicated process.

This article discusses the next phase of repository setup: separation of the testing and production environments, and automated updating of both.

Subversion Hooks

If a common testing server is introduced into the process, and a working copy of the codebase is placed into testing, it's quite simple for a developer to test code: simply commit their working copy to the repository, and update the copy on the testing server.

As with a standard production server, however, a manual step remains: the testing copy has to be updated before it can be used. The commit process itself must somehow automatically update the testing copy if the test server is to be of any real use. Fortunately, Subversion provides a facility to perform actions as part of a commit, with the *hook* scripting system.

A Subversion hook is a script that is run by the Subversion server whenever a particular thing happens. There are a few actions which can trigger a hook, but we're only interested in the `commit` hook scripts:

- **start-commit**: Run before Subversion opens a database transaction to store the updated work. If this script returns an error, the commit fails before it even began.
- **pre-commit**: Run just before Subversion writes the commit into the database. If this errors, the commit is reversed, and not saved.
- **post-commit**: Run just after a revision has been committed and has a revision number. Any errors produced by this script are ignored by the Subversion server.

For the automated deployment process, the `post-commit` hook will be used to ensure that only successful commits are replicated to the testing and/or live servers.

Updating the testing environment

The `post-commit` hook script is given two parameters by the Subversion server: the path to the repository that's just been updated, and the revision number of the update. Since the hook script is specific to a repository, it can be customised:

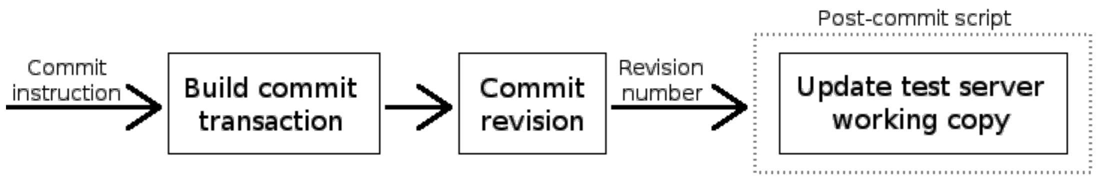

Figure 1: Control flow for a commit with automatic testing environment update

The hook scripts can be in any language usable by the server, including Bash, Python or Perl; I've used Bash for the purposes of this article, so the above control flow would translate into the following `post-commit` hook:

Automated update of a test-server working copy

```
#!/bin/bash
REPO="$1"
REV="$2"

TEST_SERVER="192.168.1.55"

# Update the working copy on the test server
ssh -l root $TEST_SERVER -t "cd /var/www && svn up"
```

Note in the above example that the testing environment is accessed by `ssh`, which means that the `root` user on the testing server must know the public key of the Subversion server's user account. For example, if the Subversion repositories are accessed by WebDAV through Apache, and the Apache process is running as user `nobody`, the user calling the `post-commit` hook is `nobody@SVN_SERVER`, and a public/private `ssh` key pair must be prepared for this user and copied to the testing server.

Deployment notification

The control flow in the above example updates the testing environment every time an update is committed to the repo. What's needed next is a method of automatically pushing committed updates to the production system, using some part of the commit action as a trigger. The ideal vector for this is the commit message: the description entered by the developer as the reason for this update.

A command is available as part of the Subversion distribution to examine the properties of a repository: `svnlook`. This can be used to check the commit message for the latest revision, and look for a signal inserted by the developer to indicate a request for deployment.

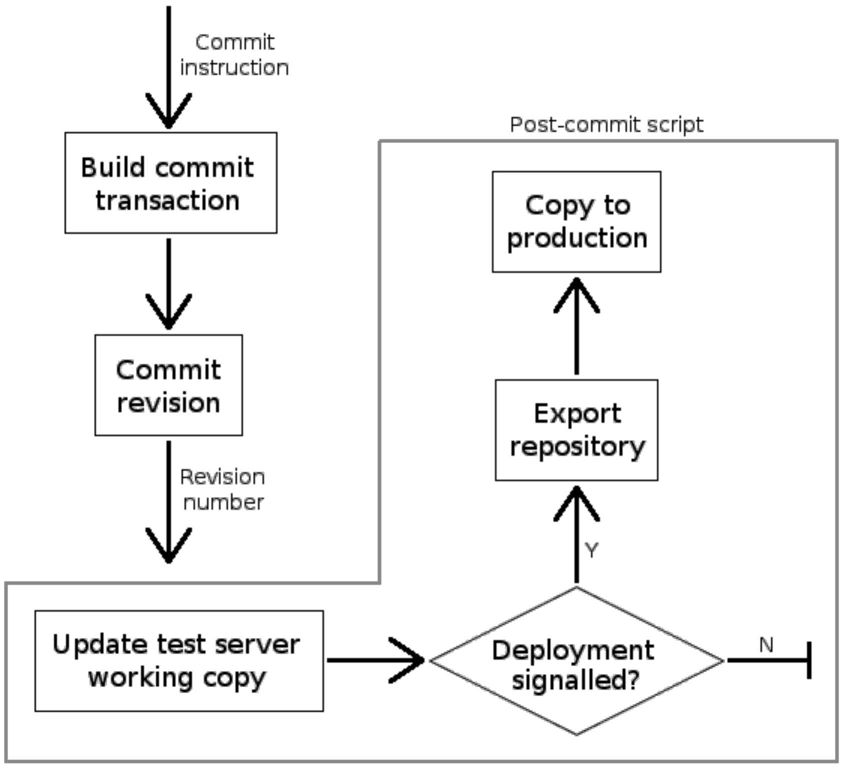

Figure 2: Control flow for a commit with deployment request

The deployment signal can be as simple as a block of text inside the commit message: if the `post-commit` hook detects this block of text in the message, it will perform the deployment. As stated above, `svnlook` can be used to look at the commit message:

Checking the commit message: `svnlook`

```
PROD_SERVER="172.16.16.1"

if ( svnlook log -r $REV $REPO | grep "~~DEPLOY~~" )
then
    /usr/local/bin/svn-deploy $REPO $REV "root@${PROD_SERVER}:/var/www"
fi
```

By asking for a specific revision using the `-r` flag, we can ensure that the revision number passed into the hook script is the one that gets checked. Even though this number should be the latest revision in the repo, it's best to make use of the revision number when it's given.

Performing a deployment

One way to deploy a Subversion repo is to simply keep a working copy as the production environment: in this situation, deployment is as easy as updating the production working copy. The disadvantage of this is that the Subversion control files and directories will be available in production; since these control files include the text base of the working copy, this exposes the backend code and database interfaces in plain text files.

Subversion provides a command targeted to producing a "clean" copy of the repository: a dump of the contents, without `.svn` directories littering the structure. That command is `svn export`:

Exporting the contents of a repo: `svn export`

```
svn export -r $REV "file://$REPO" /destination/path
```

By asking for a specific revision, as with `svnlook`, we make sure that the revision passed to the hook script is the one exported. Once the export has occurred, this can be uploaded to the production server, or synchronised with `rsync` in whatever way is required.

Putting things together

With these components, we can put together the hook and deployment scripts:

`post-commit`: Hook script

```
#!/bin/bash
REPO="$1"
REV="$2"

TEST_SERVER="192.168.1.55"
PROD_SERVER="172.16.16.1"

# Update the working copy on the test server
ssh -l root $TEST_SERVER -t "cd /var/www && svn up"

# Check for a deployment signal
if ( svnlook log -r $REV $REPO | grep "~~DEPLOY~~" )
then
    /usr/local/bin/svn-deploy $REPO $REV "root@${PROD_SERVER}:/var/www"
fi
```

`svn-deploy`: Example deployment script

```
#!/bin/bash
REPO="$1"
REV="$2"
TARGET="$3"

# Connect to datacentre VPN
sudo pppd call datacentre nodetach
sudo route add -net 172.16.16.0/24 dev ppp0

# Export the repo
rm -rf /tmp/export
svn export -r $REV "file://$REPO" /tmp/export
```

```
# Synchronise with production
rsync -az -e ssh /tmp/export/* $TARGET
```

In this particular case, the production server is behind a VPN at the datacentre, which must be tunneled through for the deployment to occur.

Running a deployment: The developer's view

Once the post-commit hook has been put into place by a repository administrator, any developer with a checked-out copy of the repo is free to commit updates; any update will cause the testing environment copy to be updated, allowing for a common testing point.

Deployment is signalled as part of the commit message for a revision, as below:

Sample commit message with deployment

```
- Frontend: Checkout process: CC payment handling added
- Admin: Orders: Status dropdown now autosaves on change
~~DEPLOY~~
```

As can be seen above, the deployment code "`~~DEPLOY~~`" must be present in the commit message for deployment to be signalled. Any files changed as part of the commit will be saved in the new revision, before deployment; the copy to production will include all files in the repository that have changed since the last deployment.

Nice-to-haves: possible advancements

There are a few ways in which the above simple scripts could be enhanced.

Branch handling:
The scripts assume that the codebase is stored in its entirety in the repository, and only the trunk of the codebase is in the repo. If the repo is structured in a trunk-and-branch fashion, everything will be exported and deployed. By using a commit-message signal similar to the deployment signal, it should be possible to test a particular branch, by updating the trunk on the testing server and then copying the contents of the signalled branch over the top.

Changelog production:
By checking the commit messages using `svnlook`, it's possible to generate a Changelog for the repository: a list of revisions ordered by date, showing what changes were made to the codebase at each point. It's also possible to email the Changelog to the developers, if this is desired.

Database structure updates:
`svnlook` also allows the hook script to look at which files were modified with the commit. If a pre-determined SQL file is modified, this can be used to signal a change in database structure, and the changes can be applied to the production database through an `ssh` connection.

Each of these possible changes would introduce complexity into the automated deployment system; for now, the scripts presented here are a simple way to speed up the testing and deployment process.

Article dated: 21st Oct 2008