

## Subversion and Version Control

### Table of Contents

Introduction.....	2
Version Control.....	2
Repository.....	2
Repository Structures.....	2
Trunk.....	3
Branches.....	3
Tags.....	3
Merging.....	4
Check In/Check Out.....	4
Theory and Practice.....	5
Developing with SVN.....	6
Installation.....	6
Tools.....	6
SVN Administration.....	6
Create a repository.....	6
Create a repository structure.....	7
Importing Files.....	7
Creating branches and tags.....	8
SVN Usage.....	9
Accessing a repository.....	9
Checking out the Trunk.....	9
Updating a check out.....	10
Checking the status of a check out.....	10
Committing back to the repository.....	11
Comments.....	12
Adding and removing files with the local working copy.....	12
Conflicts.....	13
In Summary.....	14



## Introduction

Subversion is a widely used version control system for electronic documents. This document will investigate how it can be used to aid the development process and provide a central location for all changes to be tracked and the project to be managed.

## Version Control

'Version control' is the concept of tracking the changes that occur within documents. The primary principle of this is simply that if a document is updated, a record is kept that a new version of the document exists, whilst still retaining the old version.

This means that a document can be reverted to an earlier version should a mistake have been made, or if the changes are no longer relevant. It also means that a history exists of the changes to a particular document.

This can be very useful during document development. It allows for preliminary development changes to be made, without ever needing to worry about overwriting the current 'master' document.

Subversion, a product by CollabNet (<http://subversion.tigris.org/>), is a version control system that provides a set of features needed for a centralised Version Control System.

## Repository

A repository contains all the files that are under version control. From the point of view of an SVN user, it is a collection of files and directories, much like any other file repository. The difference is that the repository has a version number attached, so the view a user has of these files depends on the version of the repository that is being viewed. The count would start at 0 when the repository is created, and each subsequent change to the repository increments this number – whether it be a new file being added, or a current one amended or removed.

It is important to remember that a repository only stores a file. If something needs changing in a file, the repository must be 'checked out'; that is, a copy is taken from the repository for a user to change in their development environment. Once done, the files are pushed back to the repository. The repository takes care of the version numbering and storing; but files are not directly changed in the repository itself.

## Repository Structures

The structure of the repository can be whatever the user wants it to be, but an established convention





is to have three directories at the top of the repository:

```
Repository/  
| - trunk/  
| - tags/  
| - branches/
```

## **Trunk**

The trunk is used as the main store of files, and generally will store the latest 'live' version of the files. Figure 1 shows the basic principles of the trunk, with a collection of files being edited and the versions being recorded.

## **Branches**

The concept of branching is akin to copying a collection of files from one location to another. This is useful within Version Control. If we consider the scenario that we have our main project in the trunk, but require some experimental work on a potential new feature, we can 'branch' the contents of the trunk into a new branch.

```
Repository/  
| - trunk/my_project  
| - tags/  
| - branches/new_feature/my_project
```

Now the development can occur completely independently of the trunk. If the feature is deemed successful it can be pushed back into the trunk to replace the project's old state. Alternatively, if the feature didn't work out as planned, the branch can be destroyed (though it will always exist within the repository by accessing the repository version when this branch was present).

Branches themselves can be branched off into new branches as often as is required. It is up to the project manager to determine how and when this should happen, be it based on a branch per feature, or branch per developer, or combinations of the both.

## **Tags**

Tags are used to provide a user friendly way of identifying a snapshot of a project. For instance, if a project has had its bugs fixed to the point of being stable, the current branch could be 'tagged' as "version-1.0" for easy reference. Work can then resume on the project, but the tag provides a way of finding the most recent stable version.

```
Repository/  
| - trunk/my_project  
| - tags/my_project-1.0  
| - branches/new_feature/my_project
```

Tags are actually nothing more than branches, but it's their use that differentiates them. Branches stored in the tags directory provide a way to snapshot milestones in a project for easy reference.

## ***Merging***

Merging is an intelligent feature of SVN's version control.

Take the previous example of a new feature being developed. We branch the trunk and work would begin on the new feature. Once completed and accepted as a change, the next step would be to update the trunk with these changes. This could be done manually, taking a copy of the trunk, changing the files, then pushing them back to the repository; or we could 'merge' the new feature branch back into the trunk. Subversion will take care of all the details, providing a review of the changes that will occur and allowing the user to authorise the change.

However, what if the trunk has had a small update to some files since the new feature branch was taken? Subversion will take care of that too, making sure the final outcome contains both the most recent changes to the trunk, as well as the major change to the branch. If at any point Subversion can't resolve the various changes, it produces a review of what it wants to do and why it can't do it, allowing the user to be aware of the problem and make the relevant changes themselves. In short, Subversion does its best to make sure that nothing is ever lost from a merge. Of course, if something were to go wrong, you can simply revert the repository back a version to before the changes were made.

## ***Check In/Check Out***

The key to avoiding branch conflicts is to regularly 'check in' and 'check out' the code base.

A 'check out' occurs when a developer wants to make a change to a file. For instance, a developer can 'check out' the new feature branch to provide a local copy to work on. Once the change has been made, the developer can 'check in' his local copy to commit the changes back to the repository. As such, check outs and check ins are an essential part of working with Subversion.

This is fine for a lone developer, but what about a team of developers working on the same feature? There is no guarantee that the developers will have checked out the branch at the same time, and they'll all be making regular changes to their own local copies – what happens when they all try and commit their changes to the repository?



This is where effective project management comes in. Each developer should be regularly checking the same code out to update their local version. If each developer is regularly updating their local copy, it will always contain the updates made by the rest of the group. Therefore, instances of conflict can be reduced as developers will not be basing their work on the old version of the branch. If a developer alters the behaviour of one file, every other developer will be aware of it sooner, and will not continue further work with the impression that the file works as it did in the past.

Similarly, changes made by the developers should be committed to the repository as often as possible, and after as small a change as possible. This means that each update will have the least impact on the rest of the development team, but also that Subversion can track each small change as it happens, allowing easy rollback if a mistake was made at any point. If this doesn't happen, it could be that the rollback to the previous version removes all the work, instead of just the small change that was a mistake.

Essentially, regular committing and updating provides a fine grained control to the project manager and to the developers, allowing much more flexibility.

### ***Theory and Practice***

So far we've discussed the theory of effective version control with Subversion. These are recommended practices that have evolved out of constant use of the software; however, how it is ultimately used is up to you, and you may find far better uses than have been outlined here.

We will now look at applying this theory with some practical commands.



## Developing with SVN

Now we've defined some of the main terms and ideas used in version control with subversion, we can start to apply this to some real world examples. For this we'll follow development on a faux web project; though this is merely representative (the principles can be easily applied to most development model, regardless of the content).

Subversion can be configured to work with a multitude of network protocols, allowing it to serve from a remote network location for maximum availability. However, due to the potential number of options and increased complexity, we'll focus this practical guide with the repository and the working copy on the same machine.

### *Installation*

Installation of SVN can be easy, but approaches vary depending on need. As such, there's no one strict way to do this. Please consult your IT support to do this.

### *Tools*

There are two tools used when dealing with Subversion: `svnadmin` and `svn`. The former deals with administrating the Subversion repository, and the latter is the general tool used when interacting with Subversion. Despite the former's designation, you may often see the `svn` tool being used on the server-side repository.

## SVN Administration

Repositories can be administered with the 'svnadmin'. This will be the tool that is used to create repositories and manipulate them once they are being used. Essentially, the svnadmin tool can be thought of as the tool that you use from outside the svn environment.

### *Create a repository*

In this example, we shall create our repositories in the '/repos/' directory, which is our holding directory for repositories.

```
mkdir /repos
svnadmin create /repos/my_project
```

The directory `/repos/my\_project` is now a Subversion repository, ready for our initial structure.

This is the last time we'll need to use the 'svnadmin' command in this document, as we'll not be doing any advanced administration. Everything else we can do with the 'svn' command, as these will be actions 'within' the repository, instead of 'without' it.

## ***Create a repository structure***

Now we have our repository created we need to create the structure we outlined earlier. We can do this with the `svn mkdir` command, which adds directories to a repository.

```
svn mkdir file:///repos/my_project/trunk -m "Adding Trunk Dir"
svn mkdir file:///repos/my_project/tags -m "Adding Tag Dir"
svn mkdir file:///repos/my_project/branches -m "Adding Branches Dir"
```

The repository is now ready for us to use for development.

## ***Importing Files***

The next step would be to import our project as it is into the trunk. This is achieved with the 'svn import' command. The following example supposes that our files are stored in the directory 'my\_files'.

```
svn import my_files file:///repos/my_project/trunk -m "First import of
project files"
```

We now have the project files located in the trunk directory of our repository. We can remove all other copies of the project stored locally.

We can check what we've done so far by monitoring the change log with the 'log' command:

```
svn log file:///repos/my\_project/trunk
-----
r4 | lifesupport | 2010-01-26 10:35:27 +0000 (Tue, 26 Jan 2010) | 1 line
First import of project files
-----
r3 | lifesupport | 2010-01-26 10:23:20 +0000 (Tue, 26 Jan 2010) | 1 line
Adding Branches Dir
-----
```

```
r2 | lifesupport | 2010-01-26 10:23:15 +0000 (Tue, 26 Jan 2010) | 1 line
```

```
Adding Tag Dir
```

```
-----
```

```
r1 | lifesupport | 2010-01-26 10:23:07 +0000 (Tue, 26 Jan 2010) | 1 line
```

```
Adding Trunk Dir
```

## Creating branches and tags

Presuming all work is now complete on our application, we may wish to tag what we have as version 1.0 of our software. To do this, we use the 'svn copy' command.

```
svn copy file:///repos/my_project/trunk/ \  
file:///repos/my_project/tags/my_project-1.0 -m "Tagging version 1"
```

We now have a copy of the trunk as it stands at version 1.0 of the software. Tags can be created from the trunk or any branches at any point as the project manager sees fit.

It may now make sense to do all further bug fixing on release 1.0 within the trunk, with a tagged version of the original available for reference, and then to develop version 1.1 within a separate branch. This can be achieved with the 'svn copy' command again; tags and branches are merely copies of some other data, and it's their relationship within the directory structure that dictates their purpose.

```
svn copy file:///repos/my_project/tags/version-1.0 \  
file:///repos/my_project/branches/my_project-1.1 -m \  
"branching for 1.1"
```

Our repository structure now looks something like this:

```
Repository/  
|- trunk/  
|-- trunk/my_project/  
|- tags  
|-- tags/my_project-1.0  
|- branches/  
|- branches/my_project-1.1
```

Developers assigned to fixing bugs can now work in the trunk, and developers assigned to the new version of the software can check out the branch designated for this work:



```
svn co file:///repos/my_project/branches/my_project-1.1 \  
~/my_project-1.1/
```

The two interests can now work independently, as they are dealing with different areas of the repository. Once version 1.1 is complete, it too can be tagged.

## SVN Usage

A user's or developers view of SVN may be a little different. They will never need to use the 'svnadmin' tool, and the use of the 'svn' tool will be in a different context. From their point of view, they are using SVN to take copies of files from the repository, and to put back any changes.

## Accessing a repository

Accessing a repository varies depending on the protocol being used. As we are using a local repository in these examples, it can be accessed via the 'file' protocol; though there are four main variants on this.

Protocol	Prefix	Example
file	file://	file:///repos/my_project
svn	svn://	svn://server_name/my_project
http	http://	http://server_name/my_project
svn+ssh	svn+ssh://	svn+ssh://user@servername/my_project

The actual protocol you need to use will depend on how the server and environment was set up. Consult your administrator for advice on this.

## Checking out the Trunk

In order to get our files from the repository we have to check them out. This is done with the 'svn checkout' command, or its shorthand version 'svn co'.

In order to retrieve our files back from the repository we have to check them out. This is done with the 'svn checkout' command, or its shorthand version 'svn co'.

```
svn co file:///repos/my_project/trunk/ ~/myproject/
```

This command tells svn to check out the location specified in the third argument to the location specified in the fourth argument. If you want the files to appear in the current directory, and not to



create a new directory, simply replace the last argument with a “.”.

```
svn co file:///repos/my_project/trunk/ .
```

We now have a copy of the trunk in a local directory that we can work on directly. The SVN application shows what it has downloaded:

```
A myproject/libs
A myproject/libs/common.php
A myproject/index.php
```

## Updating a check out

If working in a team of developers all working on the repository, you can request updates on your check out. Change your current working directory to the base of your local check out, and run the following:

```
svn update
```

This instructs the local working copy to update itself to the latest version. In our example, another developer has added a second index file, so that is pulled down from the server to our local working copy.

```
$ svn update
A index2.php
Updated to revision 85.
```

The latest version of the repository is version 85, so that is what our repository updated to. The power of SVN shows itself here, as we can specify which version we wish to update to. For instance, we can actually 'update' our repository back a version:

```
$ svn update -r84
D index2.php
Updated to revision 84.
```

This has the effect of removing the new 'index2.php' file that was added, as well as any changes that have been made to the files, leaving your local copy in a state identical to that of revision 84 of the trunk. This can be used if the changes in revision 85 were considered a mistake, or if the older revision needed to be referenced.

## Checking the status of a check out

The method of tracking changes to the local working copy is done with the 'svn status' command.

This compares the changes made locally by the developer with the last checked out copy.

If we make a change to a file, then run the status command, SVN shows us the file is modified:

```
$ svn status
M      index.php
```

Here we see the file 'index.php' has been modified, dictated by the 'M' that precedes the file name. A full list of status codes are:

Code	Meaning
A	Marked for addition
D	Marked for deletion
M	Marked as modified
C	Conflict – needs resolving
I	Marked as ignore
?	Not marked as under version control
!	Missing – Checkout or checkin failed
	No Modification

## Committing back to the repository

Once we've made our changes to our files, we need to get them back to the repository. We do this by 'committing' these files back to the repository.

```
svn commit -m "added output fix to index.php"
```

This command recognises that the index.php file has been modified, and uploads that to the server:

```
$ svn commit -m "added output fix to index.php"
Sending      index.php
Transmitting file data .
Committed revision 6.
```

The output of the command shows what was done, and also returns the new repository revision number. Whenever another developer runs the 'svn update' command, they will receive a new copy of the index.php file.

## Comments

Each action on a repository should have a corresponding comment. Commenting changes allows the project administrator and other developers to see what changes have happened and why, creating a chronological changelog of the repository.

Comments are made when using the 'svn' command, and are done using the -m flag followed by a quoted comment:

```
svn actions -m "performing an action on the repository"
```

Providing the advice of lots of small incremental updates is being followed, the single line comments can be highly descriptive. You will see this commenting system used throughout the examples; as they provide important information to other members of the team, both present and future, on the status of individual commits to the repository.

These comments provide a chronological list of changes to the code base, allowing the project manager to keep up with all changes and events in the repository with the 'svn log' command.

## Adding and removing files with the local working copy

Earlier we used the import command to move an entire directory and file structure into our repository; but how do we make svn aware of file additions and removals from our working copy? For this we use the 'svn add' and 'svn remove' commands.

For instance, we create a new library file in our project:

```
$ svn status
?      libs/uncommon.php
```

Here we see svn knows that a file exists, but it is not marked as having any relevance inside our repository.

```
$ svn add libs/uncommon.php
A      libs/uncommon.php
```

We tell svn to 'add' the file. It now shows us that the file is marked for addition into the repository.

```
$ svn commit -m "added new library file"
Adding      libs/uncommon.php
Transmitting file data .
Committed revision 87.
```

Version 87 of the repository now exists, with our newly added file.

Next we decide we don't need the new index file.

```
$ svn remove index2.php
D      index2.php
$ svn commit -m "removed redundant index file"
Deleting      index2.php
Committed revision 88.
```

Version 88 of the repository now includes our latest changes.

## Conflicts

SVN also allows us to resolve conflicts between svn repositories. Imagine some more work was done on our repository, and we make a change to a file that has changed on the server.

```
$ svn ci -m "amended the index.php file"
Sending      index.php
svn: Commit failed (details follow):
svn: File '/trunk/index.php' is out of date
```

First we get an error telling us that our copy of this file is out of date, so the next step would be to update our local working copy. However, we can check the differences ourselves before committing them:

```
$ svn diff index.php
Index: index.php
=====
--- index.php      (revision 90)
+++ index.php      (working copy)
@@ -7,3 +7,6 @@
    echo ("sample index file");

    echo ("A new line.\n");
+
+echo ("end of file");
+?>
```

We can specify the version to compare against with the '-r' flag, otherwise it will compare against the latest version.

In this instance, SVN shows us the resulting differences between our version and the repository version. It even highlights the changes it will make to our local files. Note, however, that SVN will 'merge' our two files, keeping the changes from the latest repository version as well as the latest changes from our local copy.

```
$ svn update
G    index.php
Updated to revision 90.
```

We now do the update, being sure the changes will be what we expect. This time we see the list of updates attached to the revision number, but also a new flag, 'G'. This flag means that a merge has occurred. As this has happened without incident, we can commit the file back to the repository.

What happens when SVN can't intelligently merge two files? Simply, it presents the best options to you:

```
$ svn update
Conflict discovered in 'index.php'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (h) help for more options:
```

So, you can press 'h' for more help; 'p' to postpone this update and allow us to manually deal with this later; 'df' to get output which shows the differences between the two; or 'e' to edit a copy of the file with both changes present.

```
The output of the 'diff-full' looks similar to:
--- .svn/text-base/index.php.svn-base  Mon Jan  1 11:42:05 2000

+++ .svn/tmp/tempfile.2.tmp           Mon Jan  1 11:49:00 2000
@@ -9,4 +9,9 @@
    echo ("A new line.\n");

    echo ("end of file");
+<<<<<<< .mine
+echo ("this is the correct new end of file");
+=====
+echo ("a new of of file");
+>>>>>>> .r92
?>
```

This may look a little complicated at first, but this is showing us the difference between the two files – signified by the row of chevrons. If we then chose to 'edit' the file, we are presented with a text editor with these exact diff lines in it. We choose which of the diff lines we wish to keep, save the file, and then run the update again. As we've resolved the inconsistency, the update occurs without issue.

## ***In Summary***

Subversion is a powerful tool, and it sets few limits with regards to how it should be used. This documentation will have hopefully provided an overview of its most common commands and uses,



as well as a generalised overview of how to effectively manage Subversion Repositories.

ForLinux Limited, Innovation House, Beacon Hill Office Park, Cafferata Way, Newark, Nottinghamshire, NG24 2TN

T: 01636 881200    E: [info@forlinux.co.uk](mailto:info@forlinux.co.uk)    W: [www.forlinux.co.uk](http://www.forlinux.co.uk)

Registered in England & Wales Company No. 04227715



INVESTOR IN PEOPLE