

OSSIE Subversion Usage Guidelines

Revision 1.0



Purpose

This document should serve as the authoritative set of rules for using the OSSIE subversion repository (hereon called 'repo'). All persons intending to contribute code to the OSSIE project should read and abide by the protocol explained in the following sections.

Note that this document does not in any way replace or cover the material in full-fledged subversion manuals, guides, and tutorials. This document merely explains what is needed to contribute to the OSSIE repo.

Introduction to Version Control Systems

Version Control Systems (VCS) provide a way for a group of people to work simultaneously on software projects; 'Subversion', or 'svn', is just one example of a VCS. Using a VCS provides many benefits, including:

- Backups
- Reverting broken code
- Synchronization between developers
- Organization
- Development history
- Safe way to test new code

Basic Explanation of Subversion

As you may have already gathered, Subversion consists of two parts: the server, or repo, and the client computers. The entire code-base, and its history of changes, is stored in the repo.

Let's pretend that a developer named Joe wants to add a file to code-base. After he creates the file, he will then 'add' the file to the VCS. Another developer, named Matt, now wants to contribute something to this file. Matt will 'check out' the file from the repository, edit the file as he wishes, and then 'commit' his changes. The repository now stores the newest version of the file, including Matt's changes. Joe will obviously want to work on the newest version of the file when he makes changes, so he will 'update' the file on his computer from the repository, thereby pulling all of Matt's changes from the repo. Both developers now have the most up-to-date version of the file on their computers, and can work on it simultaneously.

What follows is an in-depth explanation of each of the important Subversion commands.

Adding a File to the VCS

Before you can commit changes to a file to the repository, you must first tell the repository that it should monitor the file. To do this, use the 'add' command:

```
$ svn add <file>
```

Note that a file only needs to be 'added' to the repo once!

If the file argument passed to the 'add' command is a directory of files, then each file within the directory will also be added to the VCS - i.e. 'add' works recursively on directories.

Basic Check-ins

Now that the VCS is monitoring the file, you can commit the changes you have made to the file by using the 'commit' command:

```
$ svn ci <file> -m "Initial check-in."
```

Note that each time you commit changes to a file, you should submit a comment explaining what you changed. This is done by using the '-m' option, and putting the comment in quotes.

Each time the file is edited, these changes should be submitted to the repo by using the 'commit' command. Remember that you will never need to 'add' the file to the VCS again.

If no <file> argument is passed to the 'commit' command, then it will commit all of the changes made to every file in the current directory. This is an easy way to commit changes to multiple files at once.

Importing

Subversion provides a way to add files to the VCS and commit them recursively in one fell-swoop. This is done using the svn 'import' command.

```
$ svn import <file/directory> <URL>
```

The 'import' command will recursively add and commit an entire directory, or a single file, to the VCS at the provided URL. Furthermore, if the directory structure passed into the URL doesn't exist, 'import' will create the folders necessary to create the path.

Importing a directory of files is the best way to add a new directory or folder hierarchy to the VCS.

Basic Check-outs

Let's say that there is a file in the repository that you want to edit. First, you need to 'check-out' the file to get the most recent copy of it on your computer.

```
$ svn co <https://path/to/repo/and/file>
```

This will pull the file from the repo into the current directory on your computer. Again, note that if the passed path actually points to a directory rather than a single file, the whole directory will be downloaded to your machine.

After you have made your changes to the file, you should commit your changes using the process explained in the 'Basic Check-ins' section above.

Updating

'Updating' a file simply means pulling the most recent version of the file from the repo. This should be done on a regular basis to ensure that you are always working on the most up-to-date version of the file. Updating is quite simple:

```
$ svn update <file>
```

If no file argument is passed to the command, then it will update every file in the current directory. This is the most common usage of the command.

It is also possible to 'update' a file to a specific revision. This is done using the '-r' option:

```
$ svn update -r<revision> <file>
```

If the passed revision number is lower than your current revision, then you are effectively downgrading that file.

Moving

There is a subversion command to do exactly what the standard 'mv' UNIX command does - i.e. move a file from one name/location to another:

```
$ svn mv <file1> <file2>
```

Just like the traditional UNIX command, svn 'mv' can be used to rename a file or change its directory location. Any changes made will need to be committed, using the process discussed previously.

Basic Diffing

So you've been editing a bunch of files, and you've forgotten which files you've edited and exactly what changes you have made (Note: If you are in this position, then you have probably violated a basic principle of using VCS, which we will discuss later). In order to see what files you have changed, you can use the 'status' command:

```
$ svn status
```

This command will list every file in the current directory that has been modified in some way. These files are listed with keys, indicating how they have been changed. Now, let's say you want to look at the specific changes made in one file. This can be done with the 'diff' command:

```
$ svn diff <file>
```

This command will list all of the differences between your local copy of the file, and the version currently stored in the repo.

Reverting

Argh! You accidentally just wrote a bunch of changes to a file that broke everything! Time to revert!

```
$ svn revert <file>
```

Reverting a file will throw away all of the changes you made to the file, and pull a pristine copy of the file from the repository.

However, let's pretend that you actually checked in the changes that broke the file, and now you want to revert to a previous version of the file before the bad changes were made. This can be done by checking out a previous revision:

```
$ svn co -r<revision> <file>
```

In this way, the VCS provides a backup system that stores each individual version of the file!

Branching

By now you should have a pretty good idea of how to check out files, edit them, commit your changes, and diff files. Now we will cover development that might break the software for other users.

Since every developer on a project is using the same repository, if one developer is working on experimental code and checks it in to the primary development line (called trunk), it could break the code-base for everyone else. This is where 'branching' comes in.

Creating a 'branch' is really very easy. You merely copy the current trunk development line into another directory:

```
$ svn copy https://path/to/repo/trunk https://path/to/repo/branch
```

All of your experimental work should be committed to your branch to prevent breaking the trunk code-base for everyone else.

Merging

So you branched the code, developed an experimental new feature, fully tested it, and have deemed it ready for the primary development trunk. To add your new feature back to trunk, you ‘merge’ it back to the trunk line.

Let’s say that when you originally branched, you were at revision 100. Your branch line at the end of development was at revision 150. To commit the changes occurring in your branch from revisions 101 to 150, from the trunk directory, you would do:

```
$ svn merge -r101:150 https://path/to/repo/branch
```

This tells the VCS to grab all of the changes made to the passed branch directory from revisions 101 to 150, and apply them to the current directory, which should be trunk if you run it from the trunk directory.

Sometimes, the VCS will be able to merge all of the changes successfully without help. Other times, there might be conflicts. If there is a conflict, the repo will refuse the merge, and tell you where the conflicts are. It is up to you, the merging developer, to resolve the conflict. This can either be done by changing your code to work with the trunk line, or overriding trunk with your own code.

Tags

Tags provide a way of storing what the development line looked like at a certain point in time. They should be treated as read-only directories, created solely for check-outs. They are primarily used for indicating a release. To create a tag, merely copy the current development line into another directory:

```
$ svn copy https://path/to/repo/trunk https://path/to/repo/tag
```

Note that calling something a tag does not make it read-only in the repo. It is up to the developers to honor something called a ‘tag’, and not commit changes to it.

VCS Usage Principles

There are some basic rules to using a VCS that should always be followed. If all of the developers on a project follow these protocols, then problems should rarely surface.

“Commit early and commit often!”

This is a very common phrase among experienced software developers, and is a common response to a developer whining about losing a bunch of work. If you are creating a new file, don’t wait until you’ve written the entire 500-line file from scratch to check it in. Get it into the repo early, and commit your changes often. This provides better backups and a better revision history (which comes in handy if you realize you broke something down the road). Imagine committing 100 changes with a single commit, and then later realizing that you want to undo just 5 of them. There is no way to revert just those 5 changes. However, if you are committing your changes often, you will likely lose much less work when reverting the changes.

Do Not Write to Tags

This is self-explanatory. Tags are meant to be a sort of time-capsule of the code. A developer should be able to look at a tag and see exactly how the code looked at that point in time - changing the tag defeats this entirely. It is up to each developer to honor tags as read-only directories.

Update Before Committing

Imagine that two developers are editing the same version of a file at the same time. The first developer finishes his changes and commits them. Ten minutes later, the second developer does the same thing.

But wait, the second developer didn't have the first developer's changes in his version of the file! When he committed, he just undid all of the first developer's changes. For this reason, before committing a file, you should *always* first run an update on the file(s) you are committing. This way, the file you commit has all of the changes committed to the repo as well as your changes.

Subversion will actually not commit your files if they are not up-to-date, but not all VCS solutions do this. It is generally good practice to update before committing regardless of what VCS you use.

Do All Experimental Work in Branches

Committing changes to the trunk development line that breaks the code-base is not only rude, but poor software engineering. The trunk development line is a common code-base, and this needs to be remembered during all code commits.

If You Create a Branch, It Is Your Responsibility

If you create a branch to do experimental work in, and don't plan on being able to merge the changes back into trunk for another 2 months, it is your responsibility to ensure that when that time comes, your branch is still compatible with trunk. This can be done by periodically merging changes from trunk into your branch. See the discussion on 'Merging' for more information.

Do Not Commit Binaries

Do not commit binary files to the repo. This includes anything that is generated from other files, such as *.o files generated during compilation, PDF files generated from LaTeX source, RPMs generated from spec files, etc. Binaries bloat the repository.

Committing images is somewhat of a gray area. If the image will remain static for the most part, then committing images is generally okay. However, if the image will be changing regularly, then committing them continually will bloat the repository.

Naming Conventions

As a general rule, file and directory names should follow the naming conditions laid out in the "OSSIE Coding Style Guidelines" paper.

To make paths more succinct, use well-known abbreviations for paths - e.g. 'doc' for 'documentation', 'bin' for 'binaries', 'src' for 'source', etc. In addition, OSSIE naming conventions prefer the use of CamelCase names to Under_Scored names.

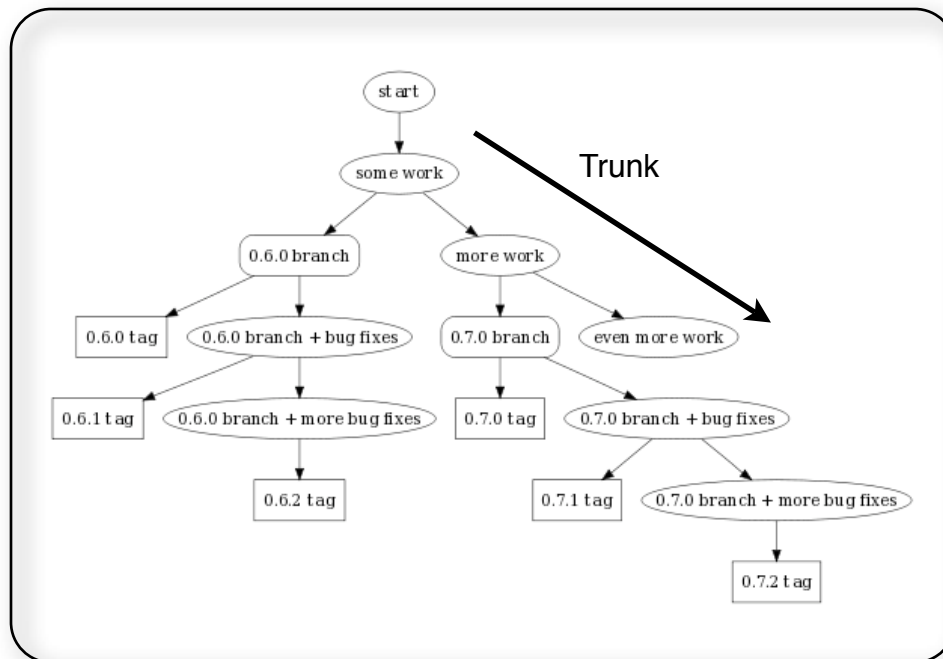
Use upper-case letters only when CamelCasing a word. Do not start folder names with upper-case letters unless it is necessary.

When In Doubt, Ask!

Remember that the repo is a shared resource. If you aren't sure what the protocol for a certain task is, or where something should go, ask! Once something is done in the repo, that history is there forever.

The OSSIE Repository

This is a basic diagram of how the repository should operate (image modified from original image provided by Philip Balister).



The 'trunk' development line is represented by the top-right path. When releases are made, a branch is created, followed by a tag for that branch.

It is now important to distinguish between maintenance and development. Maintenance consists of bug fixes made on a previously released version of the software. The patched version of the branch can then be released again as a new version of the same primary release - e.g. 0.6.1 + patches = 0.6.2. Keeping the previous release stable with bug fixes is extremely important to maintain a user base.

New development, however, should go into the trunk line. This is where the next release will come from (hence, a new version number: 0.6.0 + development = 0.7.0).

Recommended Reading and References

- “A Visual Guide to Version Control”, <http://betterexplained.com/articles/a-visual-guide-to-version-control/>
- Subversion Manual, <http://svnbook.red-bean.com/>
- “Hacker’s Guide to Subversion”, <http://subversion.tigris.org/hacking.html>