

Managing Software Projects Like a Boss with Subversion and Trac

Beau Adkins
CEO, Light Point Security
lightpointsecurity.com
beau.adkins@lightpointsecurity.com

- Introduction 4
- Setting Up Your Server 5
 - Install Subversion, Apache, and Supporting Libraries..... 5
 - Create a Directory to Contain Subversion Repositories 5
 - Create a Password File 6
 - Link Apache to Subversion 6
 - Set Up SSL 7
 - Disable the Default Site..... 7
 - Enable the Apache SSL Module 7
 - Enable the Default SSL Site..... 7
 - Disable Port 80 8
 - Install Trac..... 8
 - Install the Trac Package and Python Module 8
 - Create a Directory to Hold All Trac Projects 8
 - Link Apache to Trac 9
 - Restart Apache..... 9
- Creating a Subversion and Trac Project.....10
 - Create the Subversion Repository10
 - Give Apache Ownership of the Repository10
 - Create the Trac Project10
 - Give Apache Ownership of the Trac Project11
 - Enabling Extra Trac Features11
 - Enable the Trac Admin Interface.....11
 - Enable the Trac Post-Commit Hook.....12
 - Fix the Ownership of the Post-Commit Hook.....12
 - Make the Post-Commit Hook Executable12
 - Link to the Trac Post-Commit Hook.....13
- Giving Access to Additional Users.....14
 - Granting Repository Access14
 - Granting Trac Admin Rights14
- Finalizing Server Setup15
- Setting Up a Subversion Client16
 - Installing a Linux Subversion Client.....16
 - Installing a Windows Subversion Client16
 - Checking Out Repositories16
 - First Checkout Setup.....17
 - Root Directory Creation.....17
 - Trac Cleanup18
- Using Subversion Like A Boss19
 - The Basics19
 - Check-out a Repository (aka “co”)19
 - Adding Files to the Repository (aka “add”)20
 - Deleting Files from the Repository (aka “del”)20

- Checking In Your Changes (aka “ci”)20
- Updating Your Repository (aka “up”).....21
- Checking Your Repository’s Status (aka “st”).....21
- Checking File Differences (aka “diff”)22
- Reverting Your Changes (aka “revert”)22
- Advanced Topics22
 - When to Perform a Check-in22
 - The Trunk Directory23
 - Development Branches23
 - Managing Releases25
 - Resolving Conflicts27
- Using Trac29
 - The Trac Tabs29
 - The Admin Tab29
 - Wiki29
 - Timeline30
 - Roadmap30
 - Browse Source30
 - New Ticket30
 - View Tickets.....30
 - Search31
 - Suggested Ticket Workflow31
- Summary32

Introduction

If you create software for a living, your source code is your most valuable asset. If you are not taking adequate steps to protect your source code, disaster could be just around the corner. The best way to protect the integrity of your source code is to use a version control system such as Subversion. (<http://subversion.tigris.org/>)

Once you have your source code being managed by a version control system, you can make use of another tool to greatly enhance your efficiency, reliability and code quality. I am referring to a ticket tracking system, such as Trac. (<http://trac.edgewall.org/>)

The purpose of this talk is to help you set up a subversion and trac server. In addition, it will teach a set of best practices about how to use these two tools effectively and professionally.

The level of this talk is aimed at Noobs. I assume you don't have any previous experience with subversion, trac, or setting up web servers. I will try to explain not only the hows, but the whys behind each step.

Setting Up Your Server

My recommendation is to use a dedicated server in your own internal intranet. Because this server will only be used for version control, it does not need to be anything special. A \$100 used computer should be plenty.

Once you get your computer, load a quality Linux operating system on it. I recommend using the latest Ubuntu LTS (for Long Term Support) distribution. At the time of this writing, this is Ubuntu 10.04. The rest of this talk has instructions for this specific version. If you choose to use a different Operating System, you may need to make some changes to the following instructions.

You can download Ubuntu from here:

<http://www.ubuntu.com/download/ubuntu/download> for the desktop version, or <http://www.ubuntu.com/download/server/download> for the server version. The desktop version is a traditional graphical OS like Windows, while the server version is a text only operating system. For non-Linux aficionados, use the desktop version.

The rest of this section contains instructions you will need to perform only once for your new server. You will need to have administrator rights on this server to complete the rest of the setup.

Install Subversion, Apache, and Supporting Libraries

We have already talked about what subversion is. Apache is a very popular web server. A web server is used to serve web pages to web browsers. In our case, it will also be serving our subversion repository to the network.

To install, log into your server, and open a command prompt. Issue the following command.

```
sudo apt-get install subversion apache2 libapache2-svn
```

“sudo” is a Linux command that will execute the rest of the line as an administrator. Any command that makes changes to the operating system or protected parts of the file system will need to be prefixed with “sudo”.

Ubuntu uses a package management tool called “apt-get”. You can use this tool to download all sorts of great free software. This command says you want apt-get to install the packages named “subversion”, “apache2”, and “libapache2-svn”. As mentioned before, subversion is the version control system, apache2 is our web server, and libapache2-svn is a library to help these two work together.

Create a Directory to Contain Subversion Repositories

At the command prompt, issue the following command:

```
sudo mkdir /srv/svn
```

The “mkdir” command stands for “make directory”. This states we are making a directory called “svn” in the already existing directory called “/srv”. This will become the root directory for your subversion projects that we will create later.

Create a Password File

Next, we need to create a file that apache can use to authenticate users. This will allow you to grant access to your code to only people that should have it, and also to allow each check-in to be tied to a user. At the command prompt, issue the following command, replacing “username” in the command with the actual username you want to use for yourself.

```
sudo htpasswd -c /srv/svn/passwd username
```

htpasswd is a program that can create and modify authentication user files used by apache. The “-c” option means you are creating a new file. “/srv/svn/passwd” is the location of the file you are creating. Finally, the last parameter is the username you want to add to this file. It does not have to match your username that you use to log into the server itself.

After you issue this command, it will ask you to create a password for yourself.

Link Apache to Subversion

Next, you need to tell apache about subversion. To do this you will need to edit the apache configuration file. You will need to edit the file as an administrator using the sudo command.

Open the file “/etc/apache2/apache2.conf” in a text editor with administrator rights. At the very bottom, add the following block of text:

```
<Location /svn>
    DAV svn
    SVNParentPath /srv/svn
    AuthType Basic
    AuthName "Subversion Credentials"
    AuthUserFile /srv/svn/passwd
    Require valid-user
</Location>
```

A location block tells apache how to handle an incoming request for a specific location. In the first line, the “/svn” indicates that apache should use this block to handle requests to apache that look like “http://hostname/**svn**”. “DAV svn” gets the libapache2-svn package loaded to handle the rest of the block. “SVNParentPath” tells apache that “/srv/svn” is the parent directory for all the svn repositories. This means if a request comes in looking like “http://hostname/svn/bingo”, then apache will forward that to the svn repository located at “/srv/svn/bingo”. The “AuthType” directive says to use the “Basic” authentication scheme. “AuthName” is the prompt displayed in the case that a web browser displays a box to enter your

user name and password. “AuthUserFile” tells apache which file to use to find registered users and passwords. Finally, “Require valid-user” tells apache to ignore any requests from a non-authenticated user.

Set Up SSL

This step is not necessary. At this point, you could access your repository via the URL “http://<server ip address>/svn/<project name>”. However, this method has some drawbacks. For example, when you enter your username and password, they will be transmitted to the server in plain text. This means anyone on your network could steal your credentials. As another example, an attacker could intercept and modify data being sent to and from your server, potentially compromising the integrity of your code. If these risks are acceptable to you, skip this step, but considering how easy it is, I recommend continuing.

Disable the Default Site

Apache is capable of serving multiple different websites from a single machine. By default it comes with a default sample site. We should start by turning it off. To do this, issue the following command at the command prompt.

```
sudo a2dissite default
```

“a2dissite” is an apache shortcut for disabling a site. “default” is the name of the site you wish to disable.

Enable the Apache SSL Module

By default, SSL is not enabled in Apache. We need to enable it. To do this, issue the following command at the command prompt.

```
sudo a2enmod ssl
```

Similar to the previous step, “a2enmod” is an apache shortcut to enable a module. “ssl” is the name of the module you are enabling.

Enable the Default SSL Site

Now that we have enabled the ssl module, there is a default SSL capable site configuration available to us. All we have to do is enable it. Issue the following at the command prompt.

```
sudo a2ensite default-ssl
```

As you can probably guess by now, “a2ensite” is an apache shortcut to enable a site. “default-ssl” is the name of the site you are enabling. The default-ssl site configuration contains links to SSL certificates and keys generated by Ubuntu when you installed it. If you want to generate your own keys, you can do that, but I will not be discussing how to do that here.

Disable Port 80

Port 80 is the port used by default for HTTP traffic. HTTP over SSL, also known as HTTPS, uses port 443 by default. The default apache configuration already supports both of these, but we should disable port 80 to force everyone to use SSL. To do that, open `/etc/apache2/ports.conf` in an editor with administrator rights. Look for the following two lines:

```
NameVirtualHost *:80
Listen 80
```

Add a “#” character to the beginning of each of these lines, so they look like this:

```
#NameVirtualHost *:80
#Listen 80
```

This means these lines are now “commented out”. Apache will ignore any line that starts with a “#” character. We could have just deleted them, but commenting out is a better solution if you decide you might want to put things back in the future.

Install Trac

Trac is a very powerful (and very free) web-based ticketing system. A ticketing system is one of those things that people don’t realize they need until they try it. Trac has many useful features to help build great software, such as a wiki, ticket list, timeline, roadmaps, web-based source code navigation and changeset diffs to name a few.

Install the Trac Package and Python Module

To install the Trac packages, issue the following command from the command prompt:

```
sudo apt-get install trac libapache2-mod-python
```

We have already talked about the `apt-get` command. To summarize, we are asking it to install the “trac” package and the “libapache2-mod-python” package. “libapache2-mod-python” is the apache python module, which gives apache the ability to execute python scripts.

Create a Directory to Hold All Trac Projects

When we installed subversion, we created a single root directory to hold all the subversion projects we ever create in the future. We must do the same thing for the Trac projects. Issue the following command at the command prompt:

```
sudo mkdir /srv/trac
```

As before, we are making a directory named “trac” in the already existing directory “/srv”.

Link Apache to Trac

Next, we need to tell apache how to handle requests for our Trac pages. This is very similar to what we did when installing subversion. To do this, you again need to edit the apache configuration file. Open a text editor with administrator rights and edit the file “/etc/apache2/apache2.conf”. At the very bottom of the file, add the following block:

```
<Location /trac>
    SetHandler mod_python
    PythonHandler trac.web.modpython_frontend
    PythonOption TracEnvParentDir /srv/trac
    PythonOption TracUriRoot /trac
    AuthType Basic
    AuthName "Subversion Credentials"
    AuthUserFile /srv/svn/passwd
    Require valid-user
</Location>
```

Just like when installing subversion, this is a block that apache will use to decide how to handle any incoming page request that starts with “https://<ip address>/trac”.

The “SetHandler” line tells apache to use the python module to handle the request. “PythonHandler” tells apache which python code should handle the request. The two “PythonOption” lines define extra information that will be passed to the python code. We have already discussed the other lines when we installed subversion.

With this configuration in place, if you use a web browser to view “https://<ip address>/trac/bingo”, apache will use the Trac python code to open the Trac project at “/srv/trac/bingo”.

Restart Apache

Any time we make changes to apache’s configuration, we need to restart it for the changes to take effect. Since we have now finished the configuration steps, this would be a great time to restart it. To restart apache, issue the following command from the command prompt:

```
sudo /etc/init.d/apache2 restart
```

“/etc/init.d/apache2” is a script that is responsible for starting and stopping the apache2 server. “restart” is the parameter we are passing that script. Other valid options are “stop”, which will turn off the service, and “start” which will start the service. Issuing a “restart” is the same as issuing a “stop” followed by a “start”.

Creating a Subversion and Trac Project

Now that you have your server set up, you can use it to host as many Subversion/Trac projects as you want. The next step is to actually create a Subversion/Trac project. You should complete this section for each project you want to host on your server.

To complete this section, you need to come up with a name for your project. Keep it simple. Use only lowercase letters and numbers. For the rest of this section, the instructions will assume your project name is “bingo”. So anywhere you see “bingo” in the following instructions, replace it with your own project name.

Create the Subversion Repository

The first thing we need is a subversion repository. To create one, issue the following command at the command prompt.

```
sudo svnadmin create /srv/svn/bingo
```

The “svnadmin” tool is part of subversion. We are using it here to create a repository in the directory “/srv/svn/bingo”. Again, replace “bingo” with your actual project name.

Give Apache Ownership of the Repository

Because we are using apache to host our repository, apache needs to own it. This will let apache, and only apache have permission to change the repository. To do this, issue the following command at the command prompt.

```
sudo chown -R www-data:www-data /srv/svn/bingo
```

“chown” is a Linux tool to change ownership of a file. Because we have the option “-R”, we are telling it to perform this change recursively, which means it will apply to every file and directory under the location we pass it. Apache runs as the user “www-data”. Because we are passing “www-data:www-data”, we are telling chown to change both the owner and the group of these files to “www-data”. Finally, “/srv/svn/bingo” is the location of the directory we want to change ownership of.

Create the Trac Project

Creating a Trac project is very similar to creating a subversion repository. To create the project, issue the following command at the command prompt.

```
sudo trac-admin /srv/trac/bingo initenv
```

“trac-admin” is a tool to administer Trac projects. “/srv/trac/bingo” tells “trac-admin” which Trac project you are working with. “initenv” tells “trac-admin” that you want to create a new Trac environment (or project) here.

This command will ask you for more info. First, it will ask for a name for the project. This can be anything you want.

Next, it will ask for a database string. Use the default by just pressing enter without entering anything else. The default database string will cause Trac to use a sqlite database.

Next, it will ask for a repository type. The default is subversion, so again, just press enter without entering anything else.

Finally, it will ask for the path to the subversion repository that this Trac project should be linked to. Enter your path, i.e. “/srv/svn/bingo”.

Give Apache Ownership of the Trac Project

Just like with the subversion repository, we want to give apache ownership of this new project. Enter the following on the command prompt.

```
sudo chown -R www-data:www-data /srv/trac/bingo
```

We have already discussed what this command does.

Enabling Extra Trac Features

While this section is not necessary to complete, I recommend doing so anyway because it is easy, and the benefits are worth it. There are two nice features in Trac that require a little work to get going, but will help you manage your software better. These are the Trac admin interface, and the Trac post-commit hooks.

Enable the Trac Admin Interface

While any user who has access to your Trac site can use the wiki, create/modify tickets, and browse the source, diffs, roadmaps and timelines, only a Trac admin can create/modify milestones, components, and versions. By default, no user can do this. Issue the following command at the prompt to give yourself Trac admin rights. Be sure to change “username” to the username you are using to log into subversion.

```
sudo trac-admin /srv/trac/bingo permission add username TRAC_ADMIN
```

Again, we are using “trac-admin” to administer the Trac project located at “/srv/trac/bingo”. In this case we are telling it to “add” the “permission” of “TRAC_ADMIN” to the user “username”.

When a user who has “TRAC_ADMIN” permissions views a Trac page, they will see an extra tab at the top-right called “Admin”. In here they have the ability to define components, versions and milestones (among others). We will talk more about these later.

Enable the Trac Post-Commit Hook

Trac has a built in post-commit hook which is very nice once you start using it. A post-commit hook is a script that runs right after anyone commits something to subversion. It can do whatever you want. A common one is a script that will email some people a summary about what was just committed. The Trac post-commit hook will allow the person who performed the commit to attach that commit to an existing Trac ticket. For example, if there is a ticket #3, and you are getting ready to commit some code that is related to the issue in ticket #3, you can attach the commit to the ticket. All you have to do is somewhere in your commit message, have “References #3”. When you check it in, ticket #3 will contain an extra status update which contains your log message, and a link to the changeset of the check-in. You can also say “Fixed #3” instead, which will do the same as “References”, but it will also close the ticket.

To enable the Trac post-commit hook, issue the following command at the command prompt.

```
sudo cp /srv/svn/bingo/hooks/post-commit.tpl
/srv/svn/bingo/hooks/post-commit
```

“cp” is a Linux command to copy a file. Subversion repositories contain templates for the different types of hooks. “/srv/svn/bingo/hooks/post-commit.tpl” is the post-commit template. “/srv/svn/bingo/hooks/post-commit” is the name of the copy we are creating. Because the new file is named “post-commit”, subversion will attempt to execute this script immediately after each commit.

Fix the Ownership of the Post-Commit Hook

This new file is not owned by apache like the rest of the repository. We have to now fix that. Issue the following command at the command prompt.

```
sudo chown www-data:www-data /srv/svn/bingo/hooks/post-commit
```

We have already discussed the operation of the “chown” command.

Make the Post-Commit Hook Executable

In order for a file in Linux to be executed as a script or program, it must have executable attributes set. To do this, issue the following command at the prompt.

```
sudo chmod 744 /srv/svn/bingo/hooks/post-commit
```

“chmod” can change file attributes. “744” is the desired attributes we want. This attribute notation consists of 3 numbers between 0 and 7. The first number is the desired attributes for the owner of the file. The second number is the desired attributes for the group members of the file. The last number is the desired attributes for everyone else.

To calculate which number to use, just add up each attribute you want. Read rights are worth a 4. Write rights are worth a 2. And executable rights are worth a 1. So in the above example, we want $4 + 2 + 1 = 7$. The other two numbers mean that everyone but the owner only has the ability to read this script, not write to it, or execute it.

Link to the Trac Post-Commit Hook

The default subversion post-commit hook is just a template. We must edit it so it will call the Trac post-commit hook. To do that, open `"/srv/svn/bingo/hooks/post-commit"` in a text editor with administrator rights.

Near the very bottom you will see these two lines:

```
REPOS="$1"  
REV="$2"
```

Delete everything below these two lines. Modify the file so it ends like this:

```
REPOS="$1"  
REV="$2"  
TRAC_ENV="/srv/trac/bingo"  
/usr/bin/python /usr/share/doc/trac/contrib/trac-post-commit-hook -p  
"$TRAC_ENV" -r "$REV"
```

Giving Access to Additional Users

If your team consists of more than just yourself, you need to give other people access to the repository. This section will tell you how to do that. Complete this section for each additional user you wish to grant access to your repositories.

Granting Repository Access

The first thing you need to do is to add the new user to the password file. You can either create a password for the person, or have them near you when you execute the following command, so they can enter their own password. Enter the following command at the prompt.

```
sudo htpasswd /srv/svn/passwd newusername
```

We have discussed the `htpasswd` command before. Because we do not include the “-c” option this time, we edit the existing file, rather than creating a new one. Replace “newusername” with the username of the new user you are creating.

Granting Trac Admin Rights

If, and only if, you want to give this new user access to the Trac admin page, you need to complete this step. You can perform this step in two ways. You could issue the following command for every Trac project you want to grant Trac admin rights to.

```
sudo trac-admin /srv/trac/bingo permission add newusername TRAC_ADMIN
```

So if you have 3 different repositories, and you wish to give this new user Trac admin rights to all of them, you will have to execute the above command 3 times: once for each Trac project.

Alternatively, you can do the same thing by going into the Trac admin interface yourself to grant admin rights to other users.

Finalizing Server Setup

At this point you have a working subversion/trac server. However, the default Ubuntu (and most other Operating Systems) install uses DHCP for its IP address. This means that the server's IP address can change over time. This is not a good thing. You should assign this computer a static IP address. Better yet, you should assign it a hostname that the other computers in your network can see.

Assuming you assign a static IP address of 192.168.0.100 to your server, your subversion clients can use the URL of <https://192.168.0.100/svn/bingo> to access a hypothetical project named bingo. In a client's web browser you can browse to <https://192.168.0.100/trac> to see a list of available Trac projects, or <https://192.168.0.100/trac/bingo> to access a specific project named bingo.

If you assign a hostname of "svnserver" to your server, you can replace "192.168.0.100" in the URLs above with "svnserver". For example, <https://svnserver/svn/bingo>.

Setting Up a Subversion Client

Now that you have a subversion server, you need to set up your client from which you will actually perform your programming. The client is where you download the repository to, make changes, and check them in from. The subversion client setup is very different between Linux and Windows computers.

Installing a Linux Subversion Client

This is the easy one. Simply use the built in Linux package management tool to install the “subversion” package. For systems which use “apt-get”, issue the following command at the prompt.

```
sudo apt-get install subversion
```

For systems which use “yum” issue this command instead.

```
sudo yum install subversion
```

Installing a Windows Subversion Client

The easiest way to get Subversion running on Windows is to use a full installer. This will give you all of the subversion executables, but it will also fix your environment variables to make them easy to use from the command line. Download and execute an installer from <http://www.sliksvn.com/en/download> and you should be all set up.

Checking Out Repositories

The final step to setting up a subversion client is to check out a repository. I recommend having a single directory called “projects”, and then use it to hold all your repositories. On Linux computers, I put this directory in my home directory. On Windows, I place it on the root of a drive. So on Linux, my projects directory is at “/home/beau/projects”, and on Windows, it is at “C:\projects”. While these are my recommendations, it really doesn’t matter where you put them.

To get started, at the command prompt, enter the command to create a projects directory, and then move into that directory.

On Linux:

```
cd ~  
mkdir projects  
cd projects
```

On Windows:

```
cd C:\
mkdir projects
cd projects
```

“cd” is the command to change directories. On Linux, “~” is a shortcut for your home directory. “mkdir” is the command to make a directory. The effect of these commands is to create the “projects” directory in the appropriate location, and then move into that directory.

Then assuming you have a subversion repository called “bingo” on your server with the IP address of 192.168.0.100, execute the following command.

```
svn co https://192.168.0.100/svn/bingo bingo
```

“svn” is the subversion command line tool. “co” is the svn command to check out a repository. “https://192.168.0.100/svn/bingo” is the URL of the repository you wish to check out. Finally, the last parameter of “bingo” is the directory you wish to put this repository in. After executing this command, you will have a new directory called “bingo” under your projects directory. The new “bingo” directory will contain the entire repository.

First Checkout Setup

If this is the first checkout of a brand new repository, you should take this time to create some directories, and perform some other administrative work.

Root Directory Creation

By convention, all subversion repositories contain 4 directories at the root.

- trunk
- branches
- releases
- tags

You need to create these yourself. To do that, “cd” into the new repository (i.e. “projects/bingo”), and execute the following series of commands.

```
mkdir trunk
mkdir branches
mkdir releases
mkdir tags
```

You will now have these 4 directories under the repository directory, but they are still not under svn source control. To add these directories to source control, issue the following:

```
svn add *
```

At this point these 4 directories will now be scheduled for check-in. The final step, is to actually check them in.

```
svn ci -m "Initial check-in"
```

You now have these 4 directories checked in to your repository.

Trac Cleanup

The final thing I recommend doing for brand new repositories is to clean up Trac. By default, it contains some sample components, milestones and versions. Go into the Trac admin tab, and delete these.

Using Subversion Like A Boss

At this point, we will talk about how to actually use Subversion. Not only that, but how to use it professionally and efficiently. This talk will focus on using the actual subversion command line client, rather than any Windows or IDE plugins. While it may seem to be a steeper learning curve, I believe it is worth it.

There is a free open-source book about using subversion at <http://svnbook.red-bean.com/> and you probably have a copy that came with your subversion installation. This book is an easy read, and it is packed with great information.

All subversion interactions are performed through a command line program named “svn”. All commands you execute will have the following format:

```
svn <command> <options>
```

Additionally, the subversion command line tool has a built in help system. Executing the command “`svn help`” will give you the list of available commands. If you want to know more about a specific command, for example the “up” command, you can execute “`svn help up`” for full usage information.

For the rest of this section, I will make two assumptions to make my examples easier. First, I will assume that the name of your project is “bingo”. Second, I will assume your subversion server has the IP address “192.168.0.100”. When you see either of these strings in the examples, that is your cue to substitute the real values for your situation.

The Basics

There are a handful of subversion commands that will account for 99% of your interaction with subversion. If you understand how and when to use these commands, you will be in great shape.

Check-out a Repository (aka “co”)

The first thing you want to do is check-out a repository. This is a command you really only have to use once per computer. The subversion paradigm is a single check-out, followed by any number of check-ins. From your projects directory, execute

```
svn co https://192.168.0.100/svn/bingo bingo
```

This command will download the entire source tree of the bingo project into a new directory named bingo. The subversion terminology says that this new bingo directory is your “working copy”. Additionally, inside each directory of the source tree, you will see an extra hidden directory named “.svn”. Just ignore this extra directory. This is where subversion stores all its context information so it knows how to check things in, what files have been changed, etc.

Adding Files to the Repository (aka “add”)

A repository with no files is a useless repository. The “add” command will fix that. Assume you are in your project’s “trunk” directory, and you have just created a new file named “main.cpp”. Just because a file is in your working copy’s directory structure does not mean it is under source control. At this point, subversion will ignore this file. You must explicitly tell subversion that you want it under source control. Use the add command to do this.

```
svn add main.cpp
```

At this point, “main.cpp” is scheduled to be checked in, but it is not checked in yet. You need to perform a check-in command to do this (discussed later).

If you create a new directory and use the add command on it, it behaves recursively. For example, assume you have created a new directory called “source”, and you have added a few hundred more files and directories inside it, and you execute

```
svn add source
```

Then the “source” directory and everything inside it will be added to the repository, you do not have to execute it over and over for every file.

What Files Should Be Added to the Repository?

You should try to keep the repository as minimal as possible. If you have a script which generates another file, only check in the script. There is no need to also check in the generated file. Don’t check in anything that is created from something else.

Deleting Files from the Repository (aka “del”)

If you decide you no longer need a file or directory anymore, it is not enough to simply delete it from your working copy. You need to tell subversion to delete it. Use the “del” command to do this. Assume you no longer need a directory in your repository named “testfiles”. Simply issue the command

```
svn del testfiles
```

The “testfiles” directory is now scheduled for deletion. It will not take effect until the next check-in (discussed later). Like the “add” command, you can use it for single files, or entire directories.

Checking In Your Changes (aka “ci”)

If you have modified, added or deleted files in your working copy, you need to do a check-in to officially tell subversion that these changes are changes that you want to actually be a part of the repository.

To perform a check-in, you need to give subversion a log message that it can attach to the check-in. The log message should be a description of what you have just checked in, such as “Just fixed the URL parsing buffer overflow bug”. There are two ways you can do this. One is that you can put the log message right on the command line like this.

```
svn ci -m "Just fixed the URL parsing buffer overflow bug"
```

The other option is that you put your log message into a file, for example, named "log.txt". Then you would say

```
svn ci -F log.txt
```

Performing a check-in will officially check-in all changes **from your current directory downwards only**. Therefore, make sure you are at least as high as the highest file that you want to check-in.

Updating Your Repository (aka "up")

Assume you are part of a team of programmers, and all of you are working on the bingo project. If you check-in a change to a file, none of the other programmer's working copies will contain this change. Similarly, the changes checked in by the other programmers does nothing to your working copy. You must tell subversion to download the changes made by others. Use the up command for this.

```
svn up
```

This command will bring all files from the current directory and downward up to the latest version.

Checking Your Repository's Status (aka "st")

Before performing a check-in, among other times, it makes sense to check your status. This is a good sanity check to make sure you are actually checking in what you think you are. You might discover changes you didn't intend to keep, but had forgotten about, for instance. Simply issue:

```
svn st
```

This command will give you a list of all files that are not the same as what you have checked out. Each file listed will also have a single letter to indicate its status. Here are the most common statuses.

- **M:** This file has been modified
- **A:** This file is scheduled to be added to the repository
- **D:** This file is scheduled for deletion from the repository
- **?:** This is a file in your working copy, but is not in the repository
- **!:** This is a file in the repository, but missing from your working copy
- **C:** This is a file with conflicts. See "Resolving Conflicts" for more info.

If you add the "-q" parameter to the end of the status command, you will not see files with the "?" status.

Checking File Differences (aka “diff”)

If you want to see exactly what changes you have made to your files before a check-in, issue the diff command.

```
svn diff
```

For each file that has changed in your repository, this command will show you exactly what parts of the file have changed.

Reverting Your Changes (aka “revert”)

Imagine you are trying to optimize an algorithm, but after you have made your changes, you find out that the new optimized algorithm is slower than the old one. You certainly wouldn't want to check that in. Or imagine you are searching for a bug, and you add a lot of extra code simply to give you more information to find the bug. While you do want to check in the bug fix, you don't want to check in the extra code.

This is when you want to use the revert command. If you want to revert all the changes you have made, execute this:

```
svn revert -R .
```

The revert command will undo changes you have made in your working copy. The “-R” signifies you want the command to work recursively on every file and directory. The “.” at the end means you want it to start in the current directory.

Alternatively, if you want to revert only a single file, i.e. “main.cpp”, execute this:

```
svn revert main.cpp
```

Advanced Topics

Now that you know the basics, you may need to know some more advanced operations and concepts. This section will teach you best practice concepts, along with the subversion commands required to implement them.

When to Perform a Check-in

Surprisingly, a lot of people have trouble with this concept. Some people like to perform a check-in based on time, such as once a day, or once a week. This is WRONG.

You need to perform a check-in when you have an atomic logical change. You should be able to describe what you have just checked in with a single sentence. If you fix a bug, even if you only changed one character, CHECK IT IN. You should think about it like this: if you later decide that something that you changed in a check-in was a bad idea, you should be able to revert that change, without also removing some other change that you do want to keep.

Good Check-in Examples

- Fixed the URL Parsing Bug
- Added a skeleton class for the new client
- Removed some code that is no longer needed
- Implemented the Iterable interface in the Bingo class

Bad check-in Examples

- Done for the day, checking in all changes
- Added a new skeleton class for the client, implemented a few functions and fixed that URL parsing bug
- It's been a few weeks, figured I better check in

The Trunk Directory

As you may recall, the root of your project should contain 4 directories: trunk, branches, releases, and tags. All of your day-to-day development should take place in trunk, as long as you follow a simple rule: Don't Break Trunk.

In most cases you can make all your changes right into trunk. However, you shouldn't check anything into trunk that would interfere with any other developers. If you check something in that causes your program to not build, or crash, or no longer be testable, then the other programmers will have to sit around and wait for you to fix it.

There are times when you are working on something big. Something that should be done across several incremental check-ins. If one of these intermediate check-ins breaks something that you plan to fix in a later check-in, you should not be doing this work in trunk. This is a job for a Development Branch.

Development Branches

Let's imagine that the now infamous bingo project is some sort of image processing application. You feed it an image, and it analyzes it to search for text or faces. Now let's assume you think you have come up with a new image processing algorithm which may be faster and more accurate than the old algorithm.

This sounds like a big change. This is something that will probably require many intermediate check-ins to see if it works. Additionally, it also sounds like something that will break the code base, at least for a little while, so that it doesn't build or run correctly anymore. While nothing is stopping you from doing this in trunk, I recommend using a development branch.

Creating a Development Branch

Subversion doesn't really have branches. It just simulates it with copies. So in subversion, a branch is simply a copy. To make a development branch, first decide on a name for it. For this example, let's say we will name it "new_algorithm". To make our new branch, execute the following:

```
svn cp https://192.168.0.100/svn/bingo/trunk  
https://192.168.0.100/svn/bingo/branches/new\_algorithm -m "Making a  
new branch to try out a new algorithm."
```

The svn "cp" command is used to make a copy. In this usage, we are doing a URL to URL copy. This means both our source and destination parameters are using URL format. This means the server will do all the work. Doing it this way is the same as performing the copy just in your working copy, and then performing the check-in of the copy. Because this command actually performs a check-in, you need to specify the log message at the end, using the same rules as the check-in command discussed earlier.

Downloading a Development Branch

Because the copy we made happened only on the server, our working copy still doesn't know about it. To pull down the new branch, go to your branches directory, and execute:

```
svn up new_algorithm
```

If you leave off the last parameter, you will update all branches. Depending on how many you actually have on the server, this could take a long time. By specifying which we want, we can save a lot of time.

Working in a Development Branch

Once you have the branch in your working copy, do all of your work related to the new algorithm there instead of trunk. You can do anything you want now, knowing that nothing you do will interfere with anyone else who is using trunk.

Merging a Development Branch to Trunk

So now you have fully implemented the new algorithm in your branch. If your algorithm worked, you now need to merge it back to trunk so it can become part of the real code base. To do this, we first need to know the revision number of when the branch was created. The easiest way to find this is through Trac (discussed later). Let's assume the branch was created in revision 567.

You want to go to your trunk directory and execute this:

```
svn merge -r 567:HEAD  
https://192.168.0.100/svn/bingo/branches/new\_algorithm
```

The merge command will apply changesets previously made in other check-ins to other parts of the repository. In this case, we are using the "-r" option, which means the following parameter will specify a range of check-ins to apply. We specify "567:HEAD". The last parameter specifies the source. Because the source was created in revision 567, and HEAD is a synonym for the current revision, we are saying we want every check-in to the source to be applied to our current directory (which is trunk). If everything goes right, everything you did on the branch will happen on trunk as well. All you need to do is check it in.

```
svn ci -m "Merged the new_algorithm branch back into trunk."
```

Removing a Development Branch

At this point you have either merged your branch back to trunk, or you have decided that you don't want to keep any of these changes. In either case, you should delete the branch to keep the repository clean.

This is another advantage of using a development branch. Imagine if you had done this exploratory programming in trunk, making dozens of check-ins, only to decide at the end that you don't want them anymore. You would have to go back and undo all of those changes individually. If you don't want to keep a branch, you can just delete it in one operation.

To delete the branch, just go to the branches directory, and execute:

```
svn del --force new_algorithm
```

We have talked about the “del” command already. The “--force” option tells it to not worry about extra files which may be in the directory that are not part of the repository.

Finally, check it in:

```
svn ci -m "Removing the new_algorithm branch." new_algorithm
```

Just like when we updated our branches directory to get the branch downloaded, we are specifying a location to limit our check-in to. This can save a lot of time if you have lots of branches, however it is not necessary.

Managing Releases

Managing releases effectively means you should be able to reproduce any release you have ever had at any time. In addition, you should be able to fix a bug in any release to make a new release.

While this sounds like a lot of work, done correctly it can be quite easy and intuitive. I recommend using a “stabilization branch” paradigm to facilitate this. In this section I walk you through the cycle of making an official release, and then performing a bug fix patch release.

Creating a Stabilization Branch

At some point, you will have your product working as you desire in trunk. This means the code in trunk works and has all the features you wanted in your 1.0 release. The first step to issuing the release is to make a stabilization branch.

```
svn cp https://192.168.0.100/svn/bingo/trunk  
https://192.168.0.100/svn/bingo/branches/1.0.x -m "Creating a 1.0.x  
branch."
```

Then check it out. From the branches directory:

```
svn up 1.0.x
```

We have already talked about the format of this when discussing creating a development branch. For a stabilization branch, notice the branch will be called “1.0.x”. When you want to do a 1.1 release, your branch should be “1.1.x”, and so on. We put “.x” at the end because this branch will be the source of all the patch releases, i.e.: 1.0.0, 1.0.1, 1.0.2, etc.

Testing the Stabilization Branch

Now that you have your branch, you should test it before releasing it. Because we are testing in a branch, trunk is free for developers to continue adding the features for the future versions.

However, while doing formal testing, you may discover a bug that you decide must be fixed before the release. Assuming you know how to fix the bug code-wise, how do you do it subversion-wise? Do we do it in the branch, and then do it again in trunk?

Because this is a bug fix, you want to do it in trunk, so it will be fixed in all future versions. So let’s assume you fix the bug in trunk, and the revision of that check-in is 600. At this point, trunk is fixed, but the 1.0.x branch is not. You could just manually duplicate the fix in the branch, but this is tedious and error prone. Instead, you should merge this check-in from trunk, into the branch.

To merge this bug fix into the branch, go to the 1.0.x branch, and execute:

```
svn merge -c 600 https://192.168.0.100/svn/bingo/trunk
```

We have discussed the merge command before. However, this time we are using the “-c” option, which will merge a single check-in, in this case revision 600. Make sure everything still builds and works, and that the bug is indeed fixed, and check it in:

```
svn ci -m “Merged [600] from trunk.”
```

I put the revision number in the log message in square brackets for Trac integration (discussed later).

Finalizing Your Stabilization Branch

Once you have gotten all the code tested, and bugs fixed, you may need to make some changes in the branch that do not belong in trunk. Perhaps you need to update version numbers in code, or generate encryption keys or certificates. In this case just make your changes right in the branch, and check them in as usual.

Marking the Official Release

Once you have the code in the stabilization branch exactly how you want it, its time to mark the release. The way to do this is just like when you made the stabilization branch. Execute the following command:

```
svn cp https://192.168.0.100/svn/bingo/branches/1.0.x  
https://192.168.0.100/svn/bingo/releases/1.0.0 -m “Marking the  
Official 1.0.0 Release.”
```

The Patch Release Cycle

Now let's assume you have released 1.0.0 and a month later you find some bugs that have slipped by. Because so much time has elapsed, the current trunk is probably much different than 1.0.0. What you should do in this case is fix the bug in trunk, merge that fix back to the 1.0.x branch (discussed above in the "Testing the Stabilization Branch" section), and then mark a new official release (discussed above in the "Marking the Official Release") branch. However, this time, mark it as 1.0.1 instead of 1.0.0.

You can keep doing this cycle as you issue official releases 1.0.1, 1.0.2, 1.0.3. and so on as needed. Later, when you move to release 1.1, you make a new stabilization branch for it named 1.1.x, and start marking official releases at 1.1.0, 1.1.1, 1.1.2, and so on.

Resolving Conflicts

When doing updates or merges, you may get a conflict. To explain what this is, let's consider an example.

Imagine your project contains a file "main.cpp". Let's assume you are working on this file, and you add a new function to this file. Let's also assume that a second developer is working on this file at the same time, and that they also add another new function to the file. If the second developer checks his change in before you, subversion will not let you check yours in until you bring your "main.cpp" up to date. To bring your file up to date, you perform an "svn up". At this point subversion will merge the second developers changes into your file. Subversion will tell you that the changes were merged successfully by marking the file with a "G". Your file in your working copy will contain both new functions. You can then check your changes in.

Now let's consider a second scenario. Instead of two programmers adding two distinct functions, what if two programmers are working on the same function. Let's imagine you change the name of a function, and a second developer changes its parameters. Again, the second developer checks his changes in first, so you must do an "svn up" before you can check in. When you do the update, you will see the file marked with a "C", which means there is a conflict.

Subversion's merging algorithm is line based, which means if two developers modify the same line of code, it can not merge the changes. Subversion will mark the file as a conflict and then it becomes your responsibility to figure out how to merge these two changes.

If a file is in conflict, its contents will be modified to show a human what they need to fix it. Let's try an example. Imagine before any changes, there is a function prototype in the file that looks like this:

```
void foo(int x);
```

As we said before, a second developer checks in a change to the parameter list to look like this:

```
void foo(int x, int y);
```

In the meantime, you change the function name so your working copy looks like this:

```
void function(int x);
```

When you do the update, subversion will not know how to put these two changes together. The file will then look like this:

```
<<<<<< .mine  
void function(int x);  
=====  
void foo(int x, int y);  
>>>>>> .r<revision number>
```

This notation shows you both your version, plus the version that is checked in. You need to turn these two lines into one, and remove the other lines. After you have done it, you should have this:

```
void function(int x, int y);
```

You must now tell subversion that you have resolved the conflicts before you check in. Issue this:

```
svn resolved main.cpp
```

Now you can check your change in.

Using Trac

Trac is a great, free, open-source ticket tracking tool. It is quite intuitive, but here is a quick guide to get you started.

The Trac Tabs

When you view a Trac page, there are a number of tabs across the top. Each tab is responsible for a different feature. Here is a rundown of each tab.

The Admin Tab

If you have Trac Admin rights, you have access to the Admin Tab. The Admin Tab lets you do a lot of things, but here are the most important. Along the left of the page is a menu of things you can administer.

Components

You should use this page to define all the components of your project. For example: Client, Server, Client Installer, Server Installer, Web Site. I recommend also adding a component named "Various" in the case that you wish to refer to things that don't fit neatly into just one of the other components. You can add a description to each component you create. You can define one of these components as the default. Each new ticket you create will have this component as the default.

Milestones

Use this page to define milestones of your project. For example: Add SSL Encryption, Create Installer for Windows, Create Firefox Extension. Think of the milestones as large improvements or features. You can optionally add a due date and description to each of these. You can also select one of these as the default for any new tickets you create.

Versions

This page allows you to define the different versions of your project. This one is pretty self explanatory, but for the sake of completeness, here are some examples: 1.0, 1.1, 2.0. As you might expect, you can optionally add a description to each, and select one as the default for each new ticket you create.

Wiki

This is the home page of your Trac project. It is a wiki page as you may have guessed by now. There are some extra linking shortcuts that are useful. If in your wiki you write a number in the

form of “[100]”, it will become a link to the changeset for revision 100. If you write a number in the form of “#100”, it will become a link to ticket number 100. There is more help available built into Trac.

Also, you will often view your check-in log messages in Trac. If you use the Trac wiki formatting in the log messages, they will be displayed correctly in Trac.

Timeline

The timeline page will show you the history of your repository. Here you can see the history of your check-ins. Additionally, you can see all the changes you make to your wiki, all the tickets you create or modify, and changes to your milestones.

Roadmap

When you create a ticket, you can specify which milestone that ticket is part of. When you look at the roadmap page, it will show you graphically how much of each milestone you have finished based on how many of its tickets you have closed.

Browse Source

This tab lets you look at any file in your repository at any revision that has existed. It also gives you access to view changesets, which will show you what parts of each file you changed in a revision.

New Ticket

This tab lets you create a new ticket. For each ticket you can specify a title, type, description, component, version, milestone, priority, etc. Not all of these are necessary, but they are there for you if you want them. Any of these things can be changed later if you wish. All of the ticket attributes (like the milestone, component, version) will be pre-filled with your chosen default for each of these.

View Tickets

This tab will let you see a list of the tickets that have been created. However, it is a two step process. Clicking on the View Tickets tab will show you a list of available ways to view the tickets. There are a number of built in views, such as all active tickets, active tickets by version, active tickets by milestone, etc. This page also has a way for you to create your own custom view of the tickets.

If you click on one of the views, you will then see the list of tickets according to that view. You can then click on any of the tickets to see all the details about that ticket. In here you can change any of the original properties, or you can add notes to it. You can also assign the ticket to someone. Lastly, you can resolve the ticket. When you resolve a ticket, you select one of the available resolutions such as fixed, invalid, won't fix, etc. When you change a ticket to resolved, it is then considered closed, which means it will not show up in lists of active tickets anymore.

Search

As you may have guessed, the search tab will let you search your project. You can search your tickets, wiki, and log messages from your check-ins.

Suggested Ticket Workflow

You can use Trac and its ticketing system any way you want, but here is the workflow I suggest.

1. Create new tickets. Define the Milestones, Versions, etc, but do not assign them to anyone.
2. When a developer is looking for something to work on, they go to the View Tickets tab and choose an unassigned ticket. They go into the ticket page, and assign the ticket to themselves by clicking “accept ticket”.
3. As they work on the ticket, for each check-in related to completing the ticket, they place “References #N.” in the log message, where N is the number of the ticket they are working. So if the ticket is number 100, they place “References #100.” in the log message. By doing this, the log message will be added as a note to the ticket. For the check-in that finishes the ticket, they can place “Fixed #100.” instead, which also has the effect of closing the ticket as fixed.
4. Go to step 2.

By using the “References/Fixed” log message trick, each ticket will contain links to every check-in related to fixing that ticket. This can be very useful, and I highly recommend doing this.

Summary

Hopefully, this talk has helped improve your version control reliability and usefulness. It may seem difficult looking at it from the outside, but once you get started, you will see it is really quite easy and logical.