

CVS HOWTO

\$Revision: 1.6 \$

Blaine Simpson. blaine.simpson@admc.com

Table of Contents

Important CVS Resources	2
Using CVS via the Eclipse or WSAD CVS plugin	
.....	3
Setup	3
Recommended Practices	3
Making a new CVS module for a WSAD project	4
Checking out an existing CVS module	4
Working with Branches	5
Using CVS with a traditional CVS client	6
SSH	6
The Bare Basics	6
Repository Root	7
Tags	7
Module Alias Conventions	7
Use of Module Aliases	9
Creating and Updating Tags	9
Binary files	9
Sticky tags	10
My CVS Scripts	11
Server-side CVS Administration	12
Security	12
How to create a new CVS Module	12

Author: Blaine Simpson. blaine.simpson@admc.com

\$Revision: 1.6 \$

The software which is referred to in this document and which is hosted on the admc.com web site, is distributed for free according to the Gnu Public License.

All-in-one HTML

<http://admc.com/blaine/howtos/cvs/cvs-all.html> (`cvs-all.html`)

Split-up HTML

<http://admc.com/blaine/howtos/cvs/index.html> (`index.html`)

Info

<http://admc.com/blaine/howtos/cvs/cvs.info> (`cvs.info`) (useful if you are on UNIX and want to view this in a terminal window instead of with a web browser, and without losing the linking features).

Important CVS Resources

http://linuxcommand.org/man_pages/cvs1.html

This is the UNIX man page for *cvs*. Even if the MANPATH was not all messed up on our Solaris servers, the CVS man page is way too long to be convenient at the command-line.

http://www.linuxselfhelp.com/gnu/cvs/html_chapter/cvs_toc.html

This site has excellent examples of most things you will want to do with CVS.

Using CVS via the Eclipse or WSAD CVS plugin

IMPORTANT! Before adding any WSAD projects to CVS, make sure that you have named the directories correctly. See the porting howto at <http://admc.com/blaine/howtos>. Pay particular attention that the source directories should be named *source* for Java projects, and *JavaSource* for Web projects.

Setup

BEFORE you check out or commit any project/module, go to *Windows / Preferences / Team* and make the following settings.

I recommend that you click the box *Use Incoming/Outgoing mode when synchronizing*. This is just a suggestion. IMO, the default separate Incoming and Outgoing modes are confusing.

Under *CVS*, set *Default keyword substitution* to *-kkv*. FYI, this set the *default substitution mode* for the individual server-side files of new files that you you **add** from this workspace. This does not effect retrievals because the WSAD CVS Plugin always uses the default substitution mode that the CVS server has for the file.

Under *Ignored Resources* repeatedly hit the ADD button to add the following patterns

- classes
- bin
- *.class
- org
- ibm

These prevent storage of a bunch of unnecessary crap under SCM, and results in significant speed increase for WSAD synchronizations.

These changes have to be made for *every* WSAD Workspace that you use. (This is because these settings are saved in the workspace, and therefore apply only to the current workspace).

Recommended Practices

The WSAD/Eclipse Team plugins follow the CVS convention that

commit means upload changes *TO* repository
update means download changes *FROM* repository.

committing and updating

In your J2EE navigator, select a super-set of all of the items that you wish to commit and/or update. These could be individual files and/or folders and/or WSAD projects containing the files that you wish to commit/update. Right-click and *Team/Synchronize* (either one). Your synchronization window will

list all suggested commits and updates. Look it over carefully with particular attention to the icons which clearly show the type of change. Arrows to the right mean commits, arrows to the left mean updates. In most cases, you will want to make all of the suggested changes. In that case, select all of the top-level items in the syncro view, right-click and select *update*, then the same but select *commit*. If you add new resources, for some reason WSAD insists that you confirm you are adding new files (even though you just confirmed the icons in the synchro window which show file additions).

Keyword expansion

It is often very useful to code in keywords that will dynamically expand to the appropriate value. To see the CVS revision number right in your source file (like a .java or a .xml file), type `$Revision$` in the source file. When you commit it, and upon all future checkouts and updates, `$Revision$` will be replaced with the appropriate revision number.

In Javadoc comments, keywords can save a **lot** of work, and increase accuracy and consistency. For your class Javadoc header, instead of typing in your name for Author, the data and revision number, you cut and paste a boilerplate with the appropriate keywords.

The available keywords are listed at [http://www.linuxselfhelp.com/gnu/cvs/html_17.html#SEC78](http://www.linuxselfhelp.com/gnu/cvs/html_17.html#chapter/cvs_17.html#SEC78).

Making a new CVS module for a WSAD project

To add a new WSAD project to CVS, you will add a new CVS *module*. Be aware that you can not just right-click on the PVCS module and do *Team/Share....* We have restrictions in place so that you can only upload a project if a CVS Administrator has created and set up the module directory on the CVS server.

Tell your CVS Administrator the name(s) of the project(s) to add.

If you haven't already done so, set up the repository by opening the CVS Repositories view (in the Repository perspective in WSAD), right clicking in the left pane, and adding a New Repository Location. Choose Connection Type of "extssh".

Now in your main navigator, right click on an existing project and select *Team/Share*. Use all defaults. Confirm that you want to add even though the module appears to exist already. Confirm that you want to add to the HEAD branch.

Checking out an existing CVS module

If you already have a WSAD project with the name of the CVS module, then remove it.

In the CVS Repositories view, expand the HEAD for the repository and select all of the desired CVS module(s) (the top level nodes under HEAD).

Right click on any of them and choose *Check Out As Project*.

After checking out all of the projects that you need, you should go to the main navigator, select every *connected* module (i.e., every module using CVS), synchronize then *Override and Update* everything. This is often necessary because during the *Check Out As Project* process, WSAD often makes undesirable changes to your workspace (we definitely don't want this, since what we want is to load exactly what is stored in CVS).

Working with Branches

Load up your repository like usual (see about New Repository Location above). Your goal is to list the desired branch identifier and CVS modules under the *Branches* node in the view.

In the *CVS Repositories* view, select any module (top-level folder) under HEAD, which has the desired branch. Right-click and *Configure Branches and Versions*.

In the left top window, select any file containing the branch.

In the right top window, click on the branch identifier.

Click *Add Checked Tags*.

Now the added branch(es) will have their own branches in the CVS Repositories view under *Branches*. (ALL modules will be added to the new branch. The plug-in is not smart enough to add just those modules containing the given tag).

Right click on the desired project in the CVS Repositories view, but select the project from under the desired branch identifier under *Branches*, not under *HEAD*. Select *Check Out As Project* as usual.

Using CVS with a traditional CVS client

I give instructions on how to accomplish these tasks using the command-line interface to CVS. If you use some graphical CVS tool, it is up to you to figure out if and how to accomplish the same CVS operations using your tool.

SSH

Make sure to export the env variable `CVS_RSH` and set it to the value `ssh`. This tells CVS to use `ssh` instead of `rsh` to communicate with the CVS server host. Besides encryption (which I don't think we need), `ssh` allows us to authenticate in a secure manner without having to type in a password each time (this is not possible in a secure manner with `rsh`).

If you don't enjoy entering your password time after time after time, then set up your `ssh` keys so that you can `ssh` into your CVS repository host without supplying a password. If that works, the commands in this section and the *My CVS Scripts* section should not prompt you for a password.

If you get error messages about `ssh` and libraries, check if you have a crazy setting for `LD_LIBRARY_PATH`. In most cases, the only thing you should have in your `LD_LIBRARY_PATH` is `$ORACLE_HOME/lib`. (The Iplanet build environment files are known to do nasty things to the `LD_LIBRARY_PATH`).

The Bare Basics

If you are going to be using CVS at the command-line, **memorize** these idioms:

cvs -H *command*

 Gives a syntax message for use of the command *cv*s... *command*....

cvs -help-commands

 Lists all of the available *cv*s commands (i.e., *cv*s... *X*...).

cvs -help-commands

 Lists all global options (i.e., options that you put after *cv*s but before the *command*).

If you are familiar with RCS, be aware that, unlike RCS, CVS switches that take arguments usually require whitespace (i.e., an additional argument) after the switch itself (see the use of the `-m` switch a few lines down). Exceptions to this rule are the `-k` and `-r` switches, which usually work better for CVS without whitespace.

Commands that every developer must know.

cvs -n -q update

 lists all changes made both up in the repository and in your local files, since your local files were last synchronized. I give `-q` with almost all CVS commands because the default behavior of nearly all CVS commands is too verbose for my taste. If you have derived files in your local area, there are several methods to filter them out. This is best explained at http://www.linuxselfhelp.com/gnu/cvs/html_chapter/cvs_21.html#SEC148.

cv_s -q update

means download changes *FROM* repository. Note that this is exactly the same as the previous command without the global -n switch (which means to Not modify any files).

cv_s -q commit -m 'my comment' resource...

means upload changes *TO* repository. You should always run an update command (see immediately above) before committing changes. I always use the -m switch to supply a comment when committing because otherwise I would be confronted with an editor session just to type in a 1 or 2 line comment. You use the -r switch to specify a tag, but you should generally **not** use *cv_s... commit -r* with a static tag, but only with branch tags (as explained latter in this section). If you use no -r switch, the commit defaults to the HEAD branch (unless you checked out or previously updated with another tag... but that's also covered latter in this section).

Repository Root

Be aware that the CVS commands need to know where the repository root is. This is accomplished in a few different ways. Most *cv_s* commands use a file in the CVS subdirectory of the current directory to find the location. The *checkout* and Remote commands (like *rtag* and *rlog*) will use the same method as above if you have a CVS subdirectory, otherwise you must use the -d switch or the CVSROOT env variable. (Beware that the precedences explained in the CVS documentation are not what I observe in practice). (My CVS scripts (explained in a section below) which do remote CVS commands use the env variable SCRIPT_CVSROOT.)

Tags

Tags are completely analogous to PVCS labels.

Do not use

```
cvs... commit... -r TAGNAME...
```

to check in file(s) and apply a static (non-floating) tag TAGNAME to the checked-in version. That works ok for floating tags (branches), but otherwise, TAGNAME must be an existing tag, and the data you check in will assume that revision— i.e. it will not move the tag to a different version. (Note that HEAD **is** a floating tag, and the default behavior, if you give no -r switch, is to make a new revision and apply HEAD to that revision). If you want to check in something to head and apply a label to it, then just do that. Check it in to head (with no -r switch), then use "cv_s tag" or "cv_s rtag" to apply the desired tag to it. (Of course, if you want to tag a file in some other branch, then use the -r switch to commit to that branch, then run a tag command on that branch).

Example checking in a file and a directory branch, and tagging these resources.

```
cvs commit -m 'fixed a bug' file.txt subdir
cvs tag TAGNAME file.txt subdir
```

Module Alias Conventions

Many of the remote CVS commands take CVS module name(s) as argument(s). Often, we want to apply some command to all of the modules for a specific development project. Most critically, when working with tags, we need a way to specify the target resources more reliably (and *easier*) than typing in long lists of CVS modules and depended-upon jar files. We use *module aliases* to achieve this goal. In this section, I use the phrase *traditional module* to mean the normal CVS modules which have a physical directory branch rooted in the CVS repository directory, as opposed to a module alias.

Note that *module aliases* are named so because after a module alias is defined, it is used in remote CVS commands exactly as if it were a traditional module, but the contents of a module alias are not necessarily a module or modules. Module aliases often are a short-hand for just one (or more) artifact file(s) (artifact files = normal files under CVS control).

At any time, you can see the list of all the usable module aliases by catting the file `CVSROOT/modules` under the CVS repository on the repository host.

Since it is extremely important for users to understand exactly which resources are encompassed by a given module alias, we use the following naming convention to differentiate between the different types of aliases that we use.

Justification (skip this paragraph unless you want to know how we decided on these conventions). The normal convention for CVS is to use all-caps for tag names. Since we do not want to be confusing tag names with module aliases, we use lower-case for module aliases. We use the hyphen and dot characters, with specific suffixes, to indicate the type of module alias. We do not use slashes because we don't want our aliases to look like paths (you can often append a path to a module specification, so that would get really ugly). Since we do not use dots in traditional module names, we decided to use the dot-specifier to differentiate an alias for a specific file. That is intuitive since if you ran a `diror ls` and saw a node with a dot, you would assume that it is a file. Since our traditional modules all contain the hyphen character, we use the hyphen character for aliases which contain modules. To differentiate module aliases from traditional module names, we use the specific suffixes *-ali* and *-all*, which can never collide with our base CVS modules (which are all named x-ear, x-ejb, x-web, x-java and x-jop).

- all-all** This is one module alias with the constant name given. It is equivalent to every traditional module in the repository. You could use this, for example, if you needed to check out all resources with tag TAGX, but didn't know for sure all of the modules which have TAGX. The checkout will be inefficient because every file in every traditional module will be checked for the tag, but you will be sure to achieve your goal.

- *-ali** This is equivalent to a set of traditional modules. There should be an alias of this form for every application, and for each group of non-application traditional modules. For example, there should be an *emt-ali* and a *stoshared-ali*. *emt-ali* contains the modules *emt-ear*, *emt-ejb*, *emt-web*, *emt-jop*.

- *-all** This is equivalent to all of the resources needed to build a particular application. An *-all* alias typically is defined to contain the main application *-ali* alias, plus

other depended-upon -ali's, plus any external file resources needed to build the app. *emt-ali* contains emt-ali and storeshared-ali and coreEJB.jar

****** These are just aliases to an individual file in a module. These are defined to make it easier to specify (and remember) depended-upon binary jar files that live in projects that we don't develop in. For example, coreEJB.jar is an alias for core-ejb/bin/coreEJB.jar.

See the *Server-side CVS Administration* section for instructions on how to add module aliases.

Use of Module Aliases

You could definitely do everything using only traditional module names, and skip module aliases altogether. I'll give an example to show why you probably do want to use module aliases.

A common task is to *update* a tag to the HEAD. Our example is to update the tag 'ATAG' for the EMT application. You need to make sure that every single artifact needed by your application build gets updated. If you depend upon coreEJB.jar, for example— even though coreEJB.jar is not in any of your development modules, you still need to tag it so that the build procedure knows which version to check out when you do your UNIX build (which is why we say that it is *depended upon*).

Using traditional module names, the update command looks like this

```
cvs -d /cvs/cvsrepos -q rtag -F -a ATAG aisleCore-ejb catalog-ejb \
  core-ejb/bin/coreEJB.jar coreFrameWork-jar corePresentation-jar \
  emt-ear emt-ejb emt-jar emt-web ratePlan-ejb storeAccount-ejb emt-jop
```

Using module aliases, the update command looks like this

```
cvs -d /cvs/cvsrepos -q rtag -F -a ATAG emt-all
```

Creating and Updating Tags

You create a new tag like this.

```
cvs -d /cvs/cvsrepos -q rtag NEWTAG emt-all
```

You update a tag like this.

```
cvs -d /cvs/cvsrepos -q rtag -F -a EXISTINGTAG emt-all
```

You have to use the -F and -a switches to update tags. See the man page if you want to know why.

Binary files

When committing adds of binary files, you must flag them as binary so that the files don't get corrupted when transferring them between different OSes. The *-kb* option to *commit* (or to *cvs admin*) flags a file as binary. Example.

```
cvsc commit -kb -m 'A new jar file' file.jar
```

You don't have to worry about this from Eclipse or WSAD, since the plugin sets the flag based on the filename extension.

Sticky tags

Many CVS commands that use the `-r` switch are *sticky*. I.e., once you use that command, future CVS commands in that directory (or upon that file, etc.) will use that tag specification by default. For example, if you run `cvsc co -r TAGX modulename`, then any future "updates" done within the checkedout "modulename" branch will apply to tag TAGX.

You need to understand this because when `cvsc.tag*` property(s) are set in `env.properties` (the UNIX build environment file), the "co" target will checkout with the specified tags, and will therefore set sticky tags.

You can always view the sticky tags for a resource by using the "cvsc... status" command. As explained elsewhere in this document, you should not attempt to commit changes using a static (non-floating) tag. Therefore, if you use the build script to check out your modules and later wish to commit changes you make to that source, you will need to clear the sticky tag with `cvsc -q update -A` (if you wish to apply a non-HEAD branch, you'll also have to specify the branch tag).

My CVS Scripts

These scripts reside at `/global_tmp/bin` on our developer servers at this time. It's likely that these will be moved to a more suitable location at some point (when that happens, I'll update this document).

Some of these scripts use the environmental variable `SCRIPT_CVSROOT`, so export this variable and set its value to the location of your CVS repository, for example `:ext:nhqsnpr1:/cvs/cvsrepos`. (I suggest that you set `SCRIPT_CVSROOT` to avoid any possible conflict with `CVSROOT`, but if you know what you are doing you can set `CVSROOT` instead).

Change in behavior. These is no longer a default `CVSROOT`, you must set and export the `SCRIPT_CVSROOT` (or `CVSROOT`) env variable.

Read the part in the *Traditional VCS client* section about ssh.

allmods directory1 directory2...

This is run against local CVS directory branches. It is basically a wrapper for *cvs update* that eliminates a lot of stuff that you will never want to see, and it formats the output by main directory branch. The arguments are top-level checked-out CVS modules. Most often, I use it from the main UNIX build directory like this.

```
allmods *-ear *-ejb *-jar *-web [*-jop]
```

nontips [-r] TAG module1 module2...

Lists all files that have the specified tag, but which have been removed or modified since the tag was applied.

If you give the `-r` switch, it will also list all files which do not have the tag (this includes files added after you added the tag).

It is usually most convenient to use this command with module aliases, like

```
nontips -r TAGNAME emt-all
```

This will list all differences between `TAGNAME` and the `HEAD` of all artifacts used to build EMT.

rmed PRE-TAG POST-TAG module1 module2...

Lists all files with the specified `PRE-TAG` but which don't have the specified `POST-TAG`. If the `PRE-TAG` and `POST-TAG` are tags for one release and some later release, the output list will contain those files which were in the earlier release but not in the latter release (i.e., these files were either removed or did not have the `POST-TAG` applied to them).

This is the command to use to see what's been removed between two tags. If you want to see what's been removed between a `TAG` and the `HEAD`, then use the `nontips` script (explained immediately above).

It is usually most convenient to use this command with module aliases, like

```
rmed WAS_R2B WAS_R2C emt-all
```

This will list all EMT artifacts removed between `WAS_R2B` and `WAS_R2C`.

Server-side CVS Administration

This section only describes my recommended OS-level Server-side Administration strategy. It does not describe user-level CVS Administration commands such as creating branches. Those items are covered elsewhere in this document. The work covered herein must be performed on the host which houses the repository you want to work with.

Whenever you edit files under the repository root, always use RCS (i.e., *co* and *ci*). See the man pages for *co* and *ci* at http://linuxcommand.org/man_pages/co1.html and http://linuxcommand.org/man_pages/ci1.html. Remember that RCS switches take trailing space in a manner exactly opposite to that of CVS. Whereas CVS takes a space after switches, like

```
cvs... commit -m 'blah blah...
```

RCS does not

```
ci... -m'blah blah...
```

Security

Due to time constraints, I am describing only our normal, typical setup.

Any UNIX user in the OS group *cvs* may read the resources in any CVS module.

For a UNIX user to be able to update (add, remove, modify) resources in a CVS module, some (any) OS group to which they belong must be listed in the file `scripts/modulegroup.list` under the CVS repository root.

For a UNIX user to be able to perform OS-level CVS Administration (i.e., add new modules, grant and revoke privileges), they must be in the OS group "cvsadm". To perform this work, the user runs the command

```
sudo.alt su - cvs
```

All of the OS commands below should be performed as the user *cvs*.

How to create a new CVS Module

1. Create the directory of the module name under the CVS Repository root, and set the ownerships and permissions to `cvs/cvs 02775`.

```
cd /the/repos
mkdir new-ejb new-web
chown cvs:cvs new-ejb new-web
chmod 0775 new-ejb new-web
chmod g+s new-ejb new-web
```

(Our version of Solaris can't handle `chmod 02775`).

2. Add an OS group if necessary (in the next step you will grant write access to the module with OS groups).
3. Check out (with *co*) and edit the file `scripts/modulegroup.list` according to the comments in that file (see the description about `modulegroup.list` above). Remember to check it back in with *ci*.

4. Check out (with *co*) and edit the file `CVSROOT/modules`. This step has no effect on WSAD whatsoever, since WSAD doesn't use module aliases. This sets up aliases which make it very easy to manage tags from the CVS command line.

1. If other development teams work with binaries that you create (like a `.jar` or an `-ejb.jar` file), then create an alias for each of them, and add this alias to all existing `-all` aliases which depend upon it.

```
newAccounting.jar -a new-ejb/bin/accounting.jar
newAdding.jar -a new-ejb/bin/newAdding.jar
# Adding newAccounting.jar onto dependencies for the EMT project:
emt-all -a emt-ali storeshared-ali coreEJB.jar newAccounting.jar
```

The names for aliases for files should contain a dot so users can easily see that it is an alias for a file and not a module. The name must make it obvious what CVS module the file comes from, in order to avoid confusion and future name collisions. The value of these aliases is a relative path from the repository root directory.

2. If this is a new application, define a new alias for this module(s), otherwise, add the new module(s) to the alias which should contain this module(s).

```
# Adding alias for application "new".
new-ali -a new-ejb new-web
```

The name should end with `-ali` so users can easily see that it is a simple (non-nested) alias for module(s). Note that we list only CVS modules, not file aliases nor other aliases, in `-ali` aliases.

3. If this is a new application, define a new `-all` alias, and also add our new `-ali` alias to all existing `-all` aliases which depend upon it. For example, if we were adding the `storeshared` modules, we would have added `storeshared-ali` in the previous step, and we would now add `storeshared-ali` to the `store-all`, `emt-all` and `ast-all` aliases. For our current example, we are adding a new application that depends on `storeshared-ali` and the `core ejb jar` file (but nothing else depends upon our new stuff).

```
new-all -a new-ali storeshared-ali coreEJB.jar
```

The name should end with `-all` so users can easily see that it is a nesting alias for all the resources needed for our application.

4. Check the definition for `all-all`. If it does not somehow include the items you added, then add them. (E.g., if you defined a new `*-ali`, you will probably need to add that `*-ali` to `all-all`). This is important so that users can build your code using a tag even if your application-specific aliases get messed up.
5. Remember to check the `modules` file back in with *ci*.