

KVS Tutorial

Kyle Ross
<kyle@osl.iu.edu>

1 Introduction

KVS is a simple versioning system I designed to track changes to my web-site. The main criterion I was looking for in the version-control system was simplicity. So, I decided, like any good geek, why not write my own version control system?! “KVS” is the name I used for the joke versioning system I used as an undergrad, so I decided to call my new system this. It is basically implemented as an O’Caml library that I’ve linked into a custom top-level interpreter. This has the advantage that I (and anyone else who decides to use KVS!) can use the normal O’Caml syntax and libraries for “free”.

1.1 Terms

To describe KVS in further detail requires defining a number of terms—to define the model that KVS uses for the domain of version control.

A user has some *working files* that e is editing. Suppose the user has an initial working file f_0 : E edits this file repeatedly, changing it from f_0 to f_1 , from f_1 to f_2 ... from f_{n-1} to f_n . Then, the sequence $\langle f_0..f_n \rangle$ is the version *history* of w . This history is meaningful if and only if the user chooses the *revisions* f_i to represent significant events in this history.

KVS presumes that one has a directory structure, rooted at a *working path* in which selected working files are in a coherent state at various points in time. A *version* v of files $\{f_1..f_{n_v}\}$ is a set of files in such a coherent state.

A *revision history*, then, is a series of versions $\langle v_1..v_n \rangle$. An *archive* contains a revision history and *metadata* that describe this history. This revision history is stored in a *bookkeeping path*.

Each version has an associated *log file*. These are plaintext files, created by the user in the working path while editing; this must be stored in a file named “log”.

After the user *creates* a new archive, e must *add* files to be revision controlled. Adding a file inserts its name into the initially-empty *inventory*. To create a

new archived version, the user *commits* a particular state of files named in the inventory. Committing copies any changed files into the *file pool* directory and updates some metadata.

When the user wishes to retrieve a particular version of a file or of the archive, *e restores* the desired files from the archive. These files are then copied into the working path, overwriting whatever version might be there currently.

2 Tutorial

This tutorial will describe a fictional use scenario for KVS, highlighting the commands available and describing their functions.

2.1 register: register a new archive

Suppose we have some files in the directory “project” for which we would like to maintain a versioning history. First, we create a directory where KVS can store its bookkeeping files; we’ll call this “archive”. We must also create a directory where the version files will be stored; we’ll call this “pool”:

```
$> cd project
$> mkdir archive
$> mkdir pool
```

Now, we start KVS and register our new archive. The command `register arch bpath ppath wpath` registers a new archive `arch` with bookkeeping path `bpath`, pool path `ppath`, and working path `wpath`:

```
$> kvs
Welcome to KVS; the following archives are registered:
# register "project" "project/archive" "project/pool" "project" ;;
[creating versions file]
[creating inventory file]
[saving settings]
[activating archive]
[REGISTER COMPLETE]
```

KVS informs us that it has successfully registered the archive.

2.2 add: bring a file under revision control

Now, suppose we’ve created some files in the working directory—files for which we’d like to maintain version history. Suppose we create a file “foo” with the contents:

```
Hello, I'm a file!
```

and a file “bar” with the contents:

```
I'm another file.
```

We must tell KVS to track changes for these files. The command `add f` adds file `f` to the inventory of files managed by the archive:

```
# add "foo" ;;
[added foo to the inventory]
# add "bar" ;;
[added bar to the inventory]
```

KVS informs us each time that it has successfully added each file to the inventory.

2.3 commit: create a version

Next, we commit our first version. This means that we tell KVS to “remember” the present state of the files currently in the inventory. We can refer to this state in the future to restore files, compare files, etc.

2.3.1 make_log: write a log file

First, though, we must create a log file. This contains a descriptive message that we can use in the future to help remember the significance of this version to our project’s progress. The command `make_log` writes the lines to the log file, overwriting any previous contents:

```
# make_log ["our first version"] ;;
[writing log]
[LOG WRITE COMPLETE]
```

We could also edit this file directly; it is called “log” and must be located in the working path.

Finally, we can commit our first version, which we’ll call “version-1”. The choice of name is entirely arbitrary, so we could just as well call it “fun version”. The command `commit vers` commits the present log file and files’ states as version `vers`:

```
# commit "version-1" ;;
[computing changes]
[writing version file]
[writing log file]
[setting current version]
[COMMIT COMPLETE]
```

KVS informs us that it has committed successfully.

2.4 `rm`: remove a file from revision control

We realise that we don't really need the "bar" file any more for the project, but we'd like to keep it in the working directory. We can remove "bar" from revision management with `rm f`, which removes the file `f` from the inventory:

```
# rm "bar" ;;  
[removed bar from the inventory]
```

We can see what's in the inventory by examining the variable `!c.s.inventory`:

```
# !c.s.inventory ;;  
- : string list = ["foo"]
```

As we can see, "bar" is not in the inventory any more.

2.5 `del`: delete a file and remove it from revision control

If we would like to perform the functionality of `rm` *and* delete the actual file from the working path, we can use `del`. Let's add "bar" back to the inventory and then use `del` to remove it from the inventory and delete "bar":

```
# add "bar" ;;  
[added bar to the inventory]  
# del "bar" ;;  
[removed bar from the inventory]  
[DELETE COMPLETE]
```

2.6 `restore_file`: restore the archives version of a file

Let's edit "foo" so it now reads:

```
Hello!
```

Oh, no! We've made a mistake and would like the old version of "foo" back. We can use `restore_file f` to restore the version of `f` from the current version of the archive to the working path:

```
# restore_file "foo" ;;  
[restoring foo]
```

Several similar functions exist: `restore ()` restores *all* the files from the current version. `restore_file_for vers f` restores the version of file `f` from version `vers`. `restore_for vers` restores *all* the files from version `vers`.

Now's a good time to mention: If we ever want to know what is the current version, we can examine the `!c.s.current_version` variable:

```
# !c.s.current_version ;;  
- : string = "version-1"
```

2.7 `diffs`: show what's changed

Let's make a couple changes. We'll edit "foo" to read:

```
Hello, I'm a fun file!
```

and create a new file "moo" with the following contents:

```
Hey, I'm fun, too!
```

and add it to the inventory:

```
# add "moo" ;;  
[added moo to the inventory]
```

Wow, all the changes! We can ask KVS what's new by using the `diffs ()` command:

```
# diffs () ;;  
R bar  
C foo  
A moo
```

KVS tells us that "bar" was [R]emoved, "foo" was [C]hanged, and "moo" was [A]dded.

Several similar functions exist: `diffs_for vers` calculates differences between version `vers` and the current working path. `diffs_between v1 v2` calculates differences between versions `v1` and `v2`.

2.8 `cat_log_for`: show the log file for a version

What was "version-1" all about again? We can use the command `cat_log_for vers` to show the log for version `vers`:

```
# cat_log_for "version-1" ;;  
our first version
```

A similar function exists: `cat_log ()` shows the current log.

2.9 `mv`: rename a file

Now, we realise, "foo" should really be called "baz". We can rename the file, remove the old name from revision control, and add the new name to the inventory with `mv f1 f2`, which renames file `f1` to file `f2`:

```
# mv "foo" "baz" ;;  
[removed foo from the inventory]  
[added baz to the inventory]  
[MOVE COMPLETE]
```

KVS notifies us that the move has been completed successfully.

2.10 shutdown: shutdown an archive

We're done with work for now. We'd like to close down the archive, saving all relevant data about the current state. The command `shutdown ()` accomplishes this:

```
# shutdown () ;;
[saving versions]
[saving inventory]
[saving settings]
[SAVE COMPLETE]
[deactivating archive]
[CLOSE COMPLETE]
[SHUTDOWN COMPLETE]
```

KVS tells us the archive has been shut down.

Several similar functions exist: `save ()` saves the current state while keeping the current archive active. `close ()` closes down the current archive without saving its state (This is useful if we've made many mistakes, like erroneously removing numerous files from the archive.).

2.11 quit: exit KVS

Well, now we're done working. We can exit KVS with the command `quit ()`:

```
# quit () ;;
[EXITING KVS]
```

It's important to keep in mind that we can only use `quit ()` after the current archive has been closed.

Several similar functions exist: `force_quit ()` exits even if an archive is active; it does not save the archive. `save_quit` saves the current archive and quits.

2.12 startup: load up an archive

Let's restart KVS. Now we want to continue working with "project". We can tell KVS to use the "project" archive with the `startup arch` command, which starts up archive arch:

```
# startup "project" ;;
[loading bookkeeping path]
[loading versions]
[loading settings]
[loading inventory]
[activating archive]
[STARTUP COMPLETE]
```

KVS tells us that we've successfully started up an archive. If we want to check which archive is currently the active one, we can examine the `!c.s_archive` variable:

```
# !c.s_archive ;;
- : string = "project"
```

2.13 `archives`: show registered archives

If we forget which archives are registered, we can use the command `archives ()` to show a list:

```
# archives () ;;
project
web
```

2.14 `unregister`: unregister an archive

Let's suppose we're done with the "project" archive. To avoid having an unmanageable number of archives cluttering things up, we can use the command `unregister arch` to unregister the archive `arch`. However, we must first shutdown "project"—we can't unregister the current archive:

```
# close () ;;
[deactivating archive]
[CLOSE COMPLETE]
# unregister "project" ;;
[UNREGISTER COMPLETE]
```

KVS tells us that the archive has been successfully unregistered. It's important to remember that unregistering an archive will *never* delete any of the revision history; it only removes the archive from the list available to commands like `startup`.

2.15 `status`: what's happening?

Finally, let's learn one last command. To find out some basic information, we can use the command `status ()` (We'll use another archive of mine, "web" since we've unregistered "project".):

```
# startup "web" ;;
[loading bookkeeping path]
[loading versions]
[loading settings]
[loading inventory]
[activating archive]
[STARTUP COMPLETE]
- : unit = ()
# status () ;;
current archive = web
bpath           = /home/royal/projects/kvs2b/web-test/archive
ppath           = /home/royal/projects/kvs2b/web-test/pool
wpath           = /home/royal/projects/kvs2b/web-test/web
inventory       = {639 files}
versions        = {2 versions}
current version = version-1
```

It tells us the current archive, path information, the number of files under revision control, the number of versions for the current archive, and the current version.

2.16 help: how do I do ...

If we need a reminder of how to use a particular function, we can type `help cmd` to learn what `cmd` does. Let's try looking up `add`:

```
# help "add" ;;
add f
  adds /f/ to the inventory
```

We can, of course, look up any other command, too.