

# Websh: User Manual and Developer Documentation

Kevin Waldron  
<http://www.zazzybob.com>

April 12, 2004

Version 0.0.8

## Abstract

This document serves as a reference manual for `websh`, the interactive web-based shell interface, hosted at <http://www.zazzybob.com/websh.html>. The version number of this document, indicated above, should always match the current version of `websh` hosted on the aforementioned site.

The manual will contain information relevant to both users of the `websh`, and developers alike, who might find some of the code commentry and architecture notes of interest.

The source code to `websh` will always be available, in the tradition of open-source software, and comments are *always* welcome.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Current Implementation Overview</b>	<b>2</b>
2.1	The Shell . . . . .	2
2.2	Commands . . . . .	3
2.2.1	help . . . . .	3
2.2.2	pwd . . . . .	3
2.2.3	ls . . . . .	3
2.2.4	cd . . . . .	3
2.2.5	variable=value . . . . .	3
2.2.6	echo . . . . .	4
2.2.7	let . . . . .	4
2.2.8	printenv . . . . .	4
2.2.9	unset . . . . .	5
2.2.10	cat . . . . .	5
2.2.11	edit . . . . .	5
<b>3</b>	<b>Editing Files using edit</b>	<b>5</b>
3.1	Introduction . . . . .	5
3.2	Starting the editor . . . . .	6
3.3	Viewing the contents of the buffer . . . . .	7
3.3.1	Using the p command . . . . .	7
3.3.2	Using the n command . . . . .	8
3.4	Editing a file . . . . .	8
3.4.1	Appending data . . . . .	8
3.4.2	Inserting data . . . . .	9
3.4.3	Changing data . . . . .	10
3.4.4	Deleting data . . . . .	11
3.5	Comitting your changes . . . . .	12
3.6	When you're done . . . . .	12
<b>4</b>	<b>Conclusion</b>	<b>13</b>

# 1 Introduction

I am designing `websh` as an interactive web-based shell-like interface, which (it is hoped) will retain a lot of Bourne-shell type syntax. I am implementing the interface using the Java programming language, and the shell itself is presented as a Java applet.

The first implementation of `websh` will implement only a two-level deep filesystem, allowing me to develop other functionality of the shell.

At present, the two-tier filesystem works as hoped, and you can `cd` between subdirectories of the root (`/`) filesystem, and the root filesystem itself successfully. You can also get `ls` and `ls -l` listings of the contents of these directories.

Files and directories are coded as instances of `VirtualFile` and `VirtualDirectory`, which implement \*almost\* all of the attributes of a standard Unix directory entry.

I will elaborate more on all of these points, and others, throughout this manual, and as I develop and extend the existing `websh` implementation.

I will also discuss the various commands and their syntax, as each command is coded into the shell. You should ensure that your downloaded documentation matches the current version of `websh`. The version number of this documentation is printed on the cover page. `websh` displays its version number when the shell is first launched.

If you want to contribute to the project, or would like to comment, criticise, help, moan or otherwise contact me, please do not hesitate to do so by dropping me an e-mail at [kevin@zazzybob.com](mailto:kevin@zazzybob.com).

The source code to the project will always be available and freely distributable under the terms of the GNU General Public Licence. A copy of this licence can be found on the GNU website at <http://www.gnu.org>.

By downloading and viewing this document, it shows me that you are interested in the work that I am doing, which is very important to me. It makes all the coffee drinking and swearing at the computer worthwhile!

## 2 Current Implementation Overview

### 2.1 The Shell

The shell GUI has been implemented, as previously stated, as a Java applet. It contains only one AWT component, which is a subclassed `TextArea` component. I have designed this subclass, `WebshTextArea`, to disable most mouse input, which would otherwise allow users to reposition the cursor and upset my parsing algorithms! If a mouse is clicked anywhere on the GUI, the cursor is automatically repositioned after the last prompt, which is where you want it to be!

If, however (and through fairly thorough testing I've not found a case where this should happen), you find your cursor somewhere in the middle of the text, just press RETURN to print another prompt and reposition the cursor where you'd want it. Just like a normal shell.

The main applet class (the one that runs `init()`) is `websh_concept` and this extends `java.awt.Applet` as you'd expect. It implements `KeyListener` and contains all of the keystroke handling code. I have disabled some "special" keys (ESC, F1-F12, etc) for now, but these will be reinstated as the shell is developed and their functionality becomes required.

The text area is parsed by tokenizing the text by the prompt delimiter, which is in this case the DOLLAR (\$) character. Therefore, **YOU SHOULD NOT USE THE DOLLAR CHARACTER ANYWHERE!** I have implemented a handling routine, so that if you type a \$, the shell will return a message informing you that you shouldn't do so! This is a shortcoming of the `*current*` implementation, but when we reference variables I have worked around this problem by using `::` instead of `$` to reference the variable. And it works very well indeed.

A few of the commands are built into the main class, such as `ls` and `cd`. Other commands are called from external classes, such as `echo` which is contained within its own `cmd_echo` class. Containing the individual command code within the main class has a very slight running performance benefit, however the initial class will take longer to download as it will be larger to accommodate the extra code. When commands are contained within individual classes, the initial invocation of the command is slightly slower than subsequent invocations. So both methods have their pros and cons. I use both methods throughout the shell, as this is truly a "hackers" shell. By hacker I mean somebody who enjoys fiddling with computers and code, not the irresponsible foolish ill-defined type that maliciously harms and intrudes upon other systems.

At present, only two levels of a file system hierarchy are implemented. This has allowed me to concentrate on other shell functionality. After all, we're only at Version 0.0.2! There is the root directory (`/`), and this can contain other files and directories. Each of these subdirectories in the second layer can contain other files, but *not* directories.

Internally, the files and directories are implemented as instances of `VirtualFile` and `VirtualDirectory` respectively. Files are stored within `Hashmaps` within the `Directories`, and those `Directories` as well as any root files are stored within a top-level `VirtualDirectory` which represents the root (`/`) directory. Browse the source code for class `CreateBaseStructure` to see how the filesystem is generated.

Each `VirtualFile` and `VirtualDirectory` has \*most\* of the attributes of a standard Unix directory entry, with the exception of an inode number. All of the `get` and `set` methods are implemented to allow the `VirtualFile` and `VirtualDirectory` instances to be manipulated as the shell expands.

## 2.2 Commands

This section will be added to as and when I implement more commands.

### 2.2.1 help

Not the most interesting of commands, nor the most functional. Just displays a very terse listing of the available commands to the terminal window.

### 2.2.2 pwd

This command takes no arguments, and prints the present working directory to the terminal window.

### 2.2.3 ls

Prints a listing of the contents of the *current* directory. Listing the contents of *other* directories is not yet supported. Will also take the `-l` option, i.e. `ls -l` to print a “long” listing of the files in the current directory.

### 2.2.4 cd

Change directory. Obviously, will only change to a directory (slightly deceiving error message of `file not found` printed if you try to `cd` into a file). Also understands `cd ..` to go up a level in the heirarchy.

### 2.2.5 variable=value

Set the value of `variable` to `value`. Ensure that you **do not** leave spaces on either side of the assignment operator, as with Bourne shell assignments. You can also assign the value of an existing variable to a new variable, using an assignment of the form:

```
$ a=10
```

```
$ b=:a
$ echo b is ::b and a is still ::a
b is 10 and a is still 10
```

### 2.2.6 echo

Echo specified text to the terminal. e.g.

```
$ echo My name is Kevin
My name is Kevin
```

You can reference the values of variables using the :: operator. For example, a sample shell dialog:

```
$ myname=Kevin
$ echo My name is ::myname
My name is Kevin
```

### 2.2.7 let

Provides a means to manipulate the value of variables. The variable on the left side of the assignment **must** be initialised (to zero if necessary). Only integer arithmetic is currently supported. The four "common" assignment operators are supported, +, -, \* and /. You can put numeric values, or variable references, on the right hand side of the assignment. Some examples to illustrate this:

```
$ a=0
$ b=10
$ c=2
$ result=0
$ let ::a = 1 + 2
$ echo ::a
3
$ let ::result = ::b + 15
$ echo ::result
25
$ let ::result = ::b / ::c
$ echo ::result
5
```

### 2.2.8 printenv

Prints all current variable assignments to the terminal:

```
$ a=1
$ foo=bar
$ printenv
a=1
foo=bar
$
```

### 2.2.9 unset

Provides a mechanism for unsetting variables. Pseudo-variable `__all__` can be used to unset all variables at once:

```
$ a=1
$ foo=bar
$ myvar=myvalue
$ printenv
a=1
foo=bar
myvar=myvalue
$ unset foo
$ printenv
a=1
myvar=myvalue
$ unset __all__
$ printenv
$
```

### 2.2.10 cat

Use `cat filename` to view the contents of the file `filename` in the terminal window.

### 2.2.11 edit

A nicely featured line-editor based upon `ed` which allows for the editing of existing files, as well as the creation of new files in the filesystem. See the section **Editing Files using edit** for further information.

## 3 Editing Files using edit

### 3.1 Introduction

`edit` is a line-editor for `websh` based upon the UNIX line editors `ed` and `ex`. Currently, `edit` allows us to edit existing files, and also create and edit new files in the filesystem.

`edit` operates in two modes. Command mode allows you to input commands, and can be identified by `edit` outputting a prompt, which is colon-space, i.e. `:` . Whilst editing a file (either using append or insert - discussed later), there is no prompt, and you are editing the file until you enter a full-stop `.` on a line by itself. At this point, you will re-enter command mode. This will be elaborated on during this section.

Commands come in three syntactical flavours:

1. `x,y c [a]`
2. `x c [a]`
3. `c [a]`

Where `x,y` indicates a range of lines and `x` indicates a single line for `edit` to operate on. `c` represents the command itself, and `[a]` is an optional argument, which may be used with certain commands. All will become clear during the discussion, honest!

## 3.2 Starting the editor

`edit` can be invoked in a number of ways

```
$ edit
```

The above invocation will start editing from the command line using a completely empty buffer, which can then be saved to the filesystem using an appropriate `w` command (more on this later).

```
$ edit filename
```

If `filename` exists, the contents of the file will be loaded into the buffer ready for editing. If the file does not already exist, then `edit` will respond with

```
?filename
```

This indicates that no file was found by the specified filename. If you type a `h` command whenever you see a `?` error message, a brief explanation of error the will be displayed, e.g.

```
$ edit nofile
?nofile
: h
no such file - w to create
:
```

In this example, `edit` informs you that the file doesn't exist, and you must at some point

type a `w` command to create the file, and if appropriate, write the contents of the buffer to that file.

### 3.3 Viewing the contents of the buffer

The buffer can be viewed using one of two commands, each of which behaves similarly. The `p` command prints the contents of the buffer to the terminal, and the `n` command does exactly the same thing, with the inclusion of line numbers. This is particularly useful when editing a file and needing to discern at which position the edit needs to be made.

#### 3.3.1 Using the `p` command

Let's look at an example, suppose we would like to load a file into the buffer, and view it's contents using the `p` command.

```
$ edit existingfile
: p
This is a short
file to demonstrate
some of edits capabilities
:
```

As we can see, the contents of `edit`'s internal buffer is displayed. The `p` command, entered on its own, display the **ENTIRE** contents of the buffer, i.e. from the first line, right through to the last line. But, suppose we just want to view the first line. We can specify a single line number, which for the first line is (no surprises here), 1. So, we type

```
: 1 p
This is a short
:
```

and as we can see, only the first line is output. If we wanted to view lines four to six of a larger file, we can specify a range of addresses for `edit` to display. For example (supposing that a large file has been loaded into the buffer)

```
: 4,6 p
line four of a large file
line five of a large file
line six of a large file
:
```

If you enter invalid addresses, `edit` will respond with a `?`.

```
: p
This file only has
two lines
: 10 p
?
: h
invalid address
:
```

### 3.3.2 Using the n command

The **n** command works in EXACTLY the same way as the **p** command, but with the addition of printing line numbers. For example:

```
: 5,8 n
5: line five of a large file
6: line six of a large file
7: line seven of a large file
8: line eight of a large file
:
```

I won't dwell any more on the **n** command, however, I will state that this highlights one of **edit**'s features, and that is that all addresses are **inclusive**. In the above example, all lines including line 5 and line 8 are output.

## 3.4 Editing a file

Whether the file exists or not, editing a file acts upon **edit**'s internal buffer, and nothing is written to any actual file until a **w** command is issued.

There are three main methods of editing a file. Appending data, inserting data, and changing data. Let's start with appending data.

### 3.4.1 Appending data

Appending data is accomplished using the **a** command. Entering the **a** command on its own will start appending to the end of any existing data in the buffer. That is to say that it will begin appending after the last line. For example:

```
: p
We have a one-line text file here
: a
I am appending
some text.
```

```
.
: p
We have a one-line text file here
I am appending
some text.
:
```

As you can see, we appended the data after the last line in the buffer. We stopped the append by issuing a full-stop on a line by itself, thus exiting edit mode and returning to command mode.

We can also append after a given line. Suppose that we want to edit after the second line of a file. We do so by supplying the appropriate line number before the `a` command.

```
: n
1: this is line one
2: this is line two
3: this is line three
4: this is line four
5: this is line five
: 3 a
this should be appended after line three
and so should this!
```

```
.
: n
1: this is line one
2: this is line two
3: this is line three
4. this should be appended after line three
5. and so should this!
6: this is line four
7: this is line five
:
```

### 3.4.2 Inserting data

Whereas `append` either appends at the *end* of the buffer, or *after* a specified line, the insert command `i` inserts text either at the *beginning* of the buffer, or *before* a specified line. For example:

```
: p
```

```
We have a one-line text file here
: i
I am inserting
some text.
.
: p
I am inserting
some text.
We have a one-line text file here
:
```

The above has inserted the text at the beginning of the buffer. Again, input is terminated by issuing a full-stop on a line by itself. You can also specify a line number before the `i` command, indicating that you'd like to insert text before the specified line. For example:

```
: n
1: this is line one
2: this is line two
3: this is line three
4: this is line four
5: this is line five
: 3 i
this should be inserted before line three
and so should this!
.
: n
1: this is line one
2: this is line two
3. this should be inserted before line three
4. and so should this!
5: this is line three
6: this is line four
7: this is line five
:
```

### 3.4.3 Changing data

`edit` allows us to change a line in place. Suppose we wanted to change the text in line two of the buffer. An appropriate command session would be:

```
: n
```

```
1: line one here
2: line two here
3: line three here
: 2 c
line two here
: p
line one here
line two here
line three here
:
```

Two points are worthy of note. The `c` command **MUST** be preceded by a single line-address. Other combinations (or omissions of) addresses will cause an error. Also, note that a `.` does not need to be issued after the edit. Because you are changing a single line only, you are returned to command mode on completion of the edit.

#### 3.4.4 Deleting data

As you'd expect, `edit` allows us to easily delete data from the buffer. You can delete a single address, a range of addresses, or the entire buffer. The `d` command, when used without an address, will wipe out the **entire** buffer, so tread carefully! An example of all three uses of the `d` command follows:

```
: n
1: line one
2: line two
3: line three
4: line four
5: line five
6: line six
7: line seven
8: line eight
9: line nine
10: line ten
: 5 d
: n
1: line one
2: line two
3: line three
4: line four
5: line six
6: line seven
7: line eight
```

```
8: line nine
9: line ten
: 2,5 d
: n
1: line one
2: line seven
3: line eight
4: line nine
5: line ten
: d
: n
:
```

Three separate `d` commands were issued in the above example. The first command, `5 d` deleted line five from the buffer. The second command, `2,5 d` deleted lines two through five inclusive. The final command, a solitary `d`, wiped out the entire buffer.

### 3.5 Comitting your changes

The `w` command is supplied for comitting your change to a file in the filesystem, and writing the contents of the buffer out to "disk". The `w` command comes in two flavours. If you are editing an existing file, specified at the command line, the command `w` will write the contents of the buffer over that file, thus clobbering anything already in it, and saving your edits. Any argument specified to the `w` command is ignored.

If you are editing a non-existent file, specified at the command line, `w` will create the file using the filename you specified at the command line, and write the contents of the buffer to it. Any argument specified to the `w` command is ignored.

If you are editing a non-existent file, and you did not enter any filename at the command line, then you **must** pass an argument to the `w` command. The file will then be created (or, if it already exists, overwritten) and the data in the buffer will be written to the file. If you write to an already existing file, and that file happens to be a directory, you will overwrite the directory and thus remove the directory entries it contains from the filesystem. You have been warned!

Any new files are created within the directory that you invoked `edit` from originally.

### 3.6 When you're done

When you have finished your editing session, issue a `q` command, and `edit` will place you back in the shell where you can `cat` your newly created/edited files to the terminal. Note: if you forgot to write your changes, then those changes are lost. `edit` is strong but silent

and won't warn you if you forgot to issue your `w` command!

## 4 Conclusion

`websh` and its associated commands, tools and utilities are constantly evolving and maturing. Visit the website at <http://www.zazzybob.com> for the latest developments on this and other projects.

Thanks for reading this manual, I hope it has proved useful and informative.

Kevin Waldron

**`kevin@zazzybob.com`**