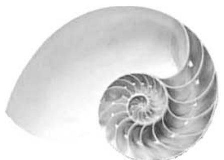
Learning the WEB SHELL



by Pablo Collins

AESTIVA PRESS

Learning the Web Shell

Aestiva Press
24430 Hawthorne Blvd.
Torrance, CA 90505

Sales information: sales@aestiva.com
Publishing website: htmlos.com
The HTML/OS companion Web site: http://dev.aestiva.com/advanced/

ISBN: 0-9749081-1-8

Dedication

This book is dedicated to my wife, Shoshana, and our daughter, Chava, who were graciously supportive of my efforts to write this book, and to Max Annavedder, whose friendship is sorely missed.

About the Author

Pablo Collins was born in Montevideo, Uruguay and now lives in Westwood, CA with his wife and their one-year-old daughter. He bikes a few blocks to and from work, where he writes code for a commercial Web site.

Acknowledgments

I would like to acknowledge the accomplishments of David Silverberg, Paul Forsyth, and Matthew Esquivel in their efforts to make Web tools worthy of artistic recognition. Every professionally prepared book is a team effort. Aestiva's David Silverberg served as this book's publisher, J.W. (Jerry) Olsen served as project manager, Sydney Jones served as copyeditor and more, and Joann Woy served as proofreader and indexer.

Table of Contents

Introduction

This book describes the Web-based shell, a new kind of user interface (UI), that until now has not been available to Web developers. Like all computing environments, the UI determines what a user can do with a system, how a user perceives its capabilities, and how effectively a user can perform tasks.

The nature of a UI drives the way a computer is used. For example, a bank's ATM terminal guides its users through performing simple account management tasks. These tasks are made accessible through a UI that presents users with simple buttons and dialog boxes to encourage ease of use and quick transactions. On the other hand, the same banking system provides very different UIs to its employees. A bank teller, for example, must be given access to multiple accounts with the ability to perform advanced tasks. Still, a bank system administrator would be given a more complex UI. Conceivably, all these UIs would run on the same system and would have access to the same underlying capabilities. But they are tailored to provide only the capability necessary to make the user more effective, depending on the user's role.

Since the advent of the Web and Web-based applications, only a few new tools have been tailored for Web development. Therefore, Web developers typically do their work in either a UNIX shell, or a client-side GUI such as Visual Studio. Both of these development environments are quite good for their designed purposes, but they weren't designed for the Web. Shells are good for developing command-line applications, and GUI tools are good for creating GUI applications. But each fails to provide a true integrated development environment (IDE), for the Web developer.

If GUI applications are best developed with GUI IDEs, and command-line applications are best and most often developed in shell environments, why not use Web-based applications to develop other Web-based applications? Web developers can benefit from working with

Web-based IDEs in practical ways that make their common tasks simpler. This book discusses the advantages of developing Web applications with Web IDEs in general, and with the Web shell in particular.

Why the Web Shell?

The most common UI today is the windowing environment (sometimes called a GUI or graphical user interface). Most computers sold to the general public have a commercial implementation of this kind of UI. GUIs are mouse-based graphical environments that use windows to encapsulate programs and interact with the user through graphical cues and dialog boxes. This kind of UI is good for many tasks, but it doesn't work well across networks (if you've ever logged into your Windows computer remotely, you know how true that is), and they don't provide convenient access to the professional user. Although client GUI applications are popular, their Web-based counterparts have shown themselves to be spectacularly more effective when you are performing tasks that require network content.

Before the wide adoption of GUI environments, the most common interface was the shell. A shell is a text-based interface in which users issue commands to the computer by typing them on a command line. A shell user, instead of double-clicking a folder icon to see its contents, would type a command, such as `ls` to list the file contents of a directory to the screen. This simple example doesn't demonstrate an advantage with either the shell or the windowing environment. But what if the task were to require getting the directory contents and determining whether the output appeared in the contents of a file? Such a task would be simple in a shell environment, but might require several steps in a GUI if it even supported such unusual functionality. Shells also work well across networks. It is usually just as convenient to run a shell on a remote computer as on the local host.

All major operating systems still supply shell interfaces alongside their polished graphical environments, because a command-line environment provides quick, networked access to a library of powerful commands, many of which can be combined to perform a complex set of related instructions. To a skilled user, a shell can be more powerful than a GUI. If the user knows which commands to use, it is just a matter of will. The user need not even lift his hands from the keyboard. The primary disadvantage of a shell is that its users need to learn the command set before it becomes useful. Until then, to the inexperienced user, most shells behave rudely.

The wide adoption of the graphical user interface has made computing accessible to the inexperienced users and has helped make personal computing a mass-market phenomenon, because GUI designers have worked to make computers as simple to use as the breadbasket at the dinner table. To move a file, grab it, move its pictorial representation (its icon) to where you want it, and let go. Similarly, ATMs have had great success partly because of their simplicity. Whereas banks previously wouldn't have allowed access to their computer systems from a publicly available terminal on a sidewalk, the ATM protects the integrity of the bank's computing environment and makes the user experience simple and effective.

You can bet, though, that the people who maintain banking systems use shell interfaces for the bulk of their tasks. As in so many corporate and industrial arenas, the workhorse UI for professional users — even though windowing environments have become very mature and sophisticated — is the ugly-looking but powerful shell interface.

The Web shell uses both of these concepts to its advantage. It provides a command-line interface to a Web-based system, but it also limits a user's access to a system. That means that a Web host can safely provide Web shell access to a system, just as a bank can place an ATM on a sidewalk, without risking the file areas that should be off limits to a remote user, and the user has the power of a Web-enabled shell environment.

The Web Shell

The Web shell is a Web-based implementation of the shell concept. Crudely speaking, it can be considered a series of Web pages that look and behave like a traditional shell and act upon the server hosting the application. The Web shell is written in H_2O, a secure, cross-platform scripting language. It parses the commands the user submits and responds to the commands accordingly. When a user submits a command, it is interpreted and executed through this scripting language. The result is a platform-independent product that doesn't rely on any shell functionality native to the host.

The Web shell is unique in that it enables users to operate a remote computer by way of a Web browser that acts like a command-line interface: A user can log onto a remote computer using a Web browser on any type of system, whether it's a desktop computer or a Web-enabled cell phone, and perform tasks on the remote system. No command-line environment is required on either the host or the client. And because the Web shell can be hosted by all the major platforms, the user could perform such functions on any type of system that is running Web-shell software. This means that a copy of the Web shell running on Windows will behave in exactly the same way as a copy running on, for example, BSD. A developer writing in a cross-platform language using the Web shell therefore generally doesn't have to account for operating system issues.

Web Computing

From a Web perspective, the scope of a file system extends to those files that are directly accessible from a browser. This file area is known as the sandbox. From the perspective of the Web shell, the sandbox is the entire file system. One advantage to this kind of file system is that because the Web is mostly platform independent, the file system can be platform independent from the perspective of the user. Additionally, a Web sandbox contributes to system security because sandboxed users see only files relevant to Web functionality and not those files used to manage the server.

For the majority of configurations, the sandbox area comprises all files that are accessible from a browser. On a typical UNIX system, this corresponds to everything in the `htdocs` and `cgi-bin` directories. Because, from the perspective of the Web shell, the sandbox is the same across all supported types of Web servers, Web shell users by default are forced to write cross-platform Web applications. This will be true as long as the script interpreter and/or database functionality is supported on the other platform as well. Perl, PHP, H_2O, and HTML/OS work well across major platforms. HTML/OS conveniently provides an integrated database, too. A Web shell user writing a Web application in H_2O, HTML/OS, PHP, or PERL, for example, won't stray from the sandbox area where file systems can vary across different operating systems. This book explains how to use the Web shell commands `pack` and `unpack` to package and deploy cross-platform applications. It also covers how to encrypt your source-code when you deliver an H_2O or HTML/OS-based Web application to a client.

On the browser side, Web files can be rendered in many ways from within the Web browser. For example, whereas a traditional shell can provide a text-based list of the image files (gif, jpg, bmp, and so on) in any given directory, the Web shell, as any Web-based application, could also actually display the images, referenced by HTML image tags, from within its Web-based interface. The Web shell was written with these distinctions in mind. The Web content should be manageable as it is from within a traditional shell environment, but it should also be viewable as the application user sees it. The Web shell maintains in its various capacities the perspective of the browser-based Web. Therefore, although you use a typical command-line interface when working within the Web shell, you can also view files as though you were working within a browser.

It is because of its Web-based perspective on the server side, as well as on the client (browser) side that the Web shell can be thought of as a Web IDE.

Practical Uses

The Web shell was designed for use as a file-management system as well as a Web development tool. As a file-management system, the Web shell has file manipulation capabilities similar to those of a traditional shell. These basic shell operations, include adding, deleting, moving, and renaming files and directories within the Web sandbox, and more advanced capabilities, such as text file manipulation, searching, and file processing. This book covers the commands available in the Web shell as well as how to use these commands to create Web applications and to manage Web systems.

Because most Web development consists of editing text files and testing them in a Web browser, the Web shell also ships with a Web-based file-editing tool supplied as an alternative to traditional command-line text editing tools, such as vi. The shell's editor is a browser-based interface that naturally allows for development in any Web scripting language. Some of the more powerful of the languages supported are H_2O, HTML/OS, PHP, C#.NET, Perl, and Java Server Pages (JSP). The only caveat is that files need to be in the Web sandbox to be visible to the Web shell. A file outside of the sandbox (outside of the `cgi-bin` or `htdocs` folder) will not be accessible from the Web shell. This seeming restriction ends up being an advantage to developers and hosting providers, as you'll see below.

Obviously, there are several ways of developing a Web application. A developer using a traditional shell logs into a system via a remote connection, such as Telnet or SSH (if such access is provided by the host) and then edits a file using a text-based program, such as vi. This process requires knowledge of vi's arcane command set as well as how to use an uncorrelated browser interface to view the resulting application.

In contrast, a Web shell user logs into the Web shell through a browser and edits a text file using the Web-based text editor provided. The Web shell user edits the file in a text area in his browser. No text editing application is required by the host, and the user needs no special knowledge or application. And because the development environment is the same as the deployment environment, the developer has the ability to run the script from within the Web shell platform. No longer does the developer have to coordinate the development process with application testing. The editing environment of the Web shell provides testing functionality that is an integrated part of the development environment.

Hosting Providers

The Web shell's file system sandbox provides added usefulness to hosting providers. By supplying Web shell access, hosting providers achieve a balance between providing access to clients and keeping inexperienced or malicious users from interfering with the system and other users. The Web shell keeps individual Web shell users in their file system sandboxes: Users can manage the files needed to run a Web site but don't see files that should only be of interest to a system administrator. In this way, a hosting provider of shared accounts on a single machine can provide full Web-enabled shell access to power users, and keep distinct users from hindering each other.

About This Book

This book will show you how to use the Web shell to manage server files and to develop Web applications. As you proceed, you will be able to log in to your own copy of the Web shell and test the concepts and examples that we will cover. The only resources you will need are a Web shell account (which we provide for you), a Web browser, and an Internet connection. You won't need any technical knowledge either, although an understanding of HTML, structured programming, and basic Web concepts would help.

Initially, this book covers basic concepts relating to how to get started with a copy of the Web shell. Subsequent chapters show you how to use more advanced features relating to application development and deployment. As you progress, you build various small Web applications to learn how Web development is accomplished using the Web shell's editor. By the end of the book you should have a good understanding of how to use the Web shell to manage a Web site, create a Web application, deploy a project, protect your source code, and even add your own customized commands to the Web shell's command set.

Chapter 1, "Getting Started," explains how to gain access to a copy of the Web shell that you can use as you progress through this book. It covers how to access the copy of the Web shell provided with the purchase of this book. It also shows you how to install and use your own copy of the Web shell. You also get a tour of the Web shell's functionality and the environment in which it runs, H_2O or HTML/OS.

Chapter 2, "File Management," shows you how to use the Web shell as you would use a traditional shell. It covers basic commands and methodologies you can use to administer a Web file system with the Web shell. It also covers more advanced topics relating to the HTML/OS virtual file system including sandboxing and mirroring.

Chapter 3, "The Shell Environment," shows you how to develop Web applications with the Web shell. It explains how to use the Web shell as an IDE to create and deploy Web applications and introduces you to the primary language used throughout this book, HTML/OS. Finally, it walks you through developing and testing a simple Web application using the Web shell's editor.

Chapter 4, "Redirection," covers how to combine shell commands to achieve more powerful functionality and how to write the output of a command to a file. It also shows you how to use some of the Web-specific commands you can use to transfer Web-based files from a remote server to your host, as well as between your host and your local machine.

Chapter 5, "Useful Web Shell Commands," provides examples of how to use some of the Web shell's more commonly used commands to perform your daily development and administrative tasks, how to use the shell's Web-based nature to make Web management easy, and how to package, deploy, and install a Web application using the shell's deployment commands.

Chapter 6, "Creating Custom Commands," shows you how to write your own commands for the Web shell. This chapter covers the H_2O language in more detail and shows you how to use H_2O to program the Web shell's command API.

About Web Shell Culture

The first iteration of the Web shell was developed in California at the offices of the Web language company, Aestiva. It is written in Aestiva's own language called H_2O. H_2O is a cross-platform Web development environment. HTML/OS is an extension of H_2O that includes an advanced database. Both H_2O and HTML/OS include the Web shell. Although the Web shell is most commonly used to develop H_2O and HTML/OS applications, it is not restricted. This book does not assume the reader will necessarily be developing in H_2O or HTML/OS.

The Web shell can be used when developing ASP, PHP, PeRL, and JSP-based applications.

This book does cover H_2O basics but is not a comprehensive resource on the subject. For detailed information on using H_2O visit h2o.aestiva.com or look into the book, *Advanced Web Sites Made Easy*, by D.M. Silverberg.

The Web shell continues to be updated and maintained by its original authors as well as by new recruits, so you can expect exciting new features on a regular basis. In addition to adding more commands, the Web shell is evolving to support interface enhancements that reduce both typing and mouse usage.

As the first book on the subject, *Learning the Web Shell* provides an important resource for developers of browser-based applications. Most important, since the Web shell is free and available for all the major hardware platforms, its popularity is likely to explode, making this book a must-read for everyone interested in gaining an appreciation for this exciting new technology.

1

Chapter

Getting Started

Before you can begin working with the Web shell, you will need to log in to a copy of Aestiva H_2O or Aestiva HTML/OS. Both environments are based on the H_2O language. Both include a Web shell. If you aren't familiar with H_2O or HTML/OS, they are integrated development and deployment environments for Web development. At its foundation, H_2O is an interpreted scripting language; HTML/OS is the same as H_2O but it also includes an integrated database engine. Both include a suite of system-management and Web-development tools built on the H_2O language.

One of these tools is the Web shell, a Web-based application developed with and for H_2O and HTML/OS as part of their suites of Web development tools. The Web shell is a versatile application that transcends the H_2O environment. You can use it to manage and develop a Web-based system even if you aren't going to be using H_2O applications.

This chapter introduces you to H_2O. After that, you will be introduced to the basic elements of the Web shell environment. If you already have a hosting account that includes H_2O, then accessing the Web shell is a matter of clicking the Web shell icon provided with H_2O. Ask your hosting provider for details.

Your Copy of Aestiva HTML/OS

To use the Web shell, you'll need a copy of H_2O or HTML/OS. Free downloads are available at h2o.aestiva.com. Hosting accounts may include free H_2O. It is also available by purchasing HTML/OS. The Web shell is available from the login screen from H_2O or HTML/OS.

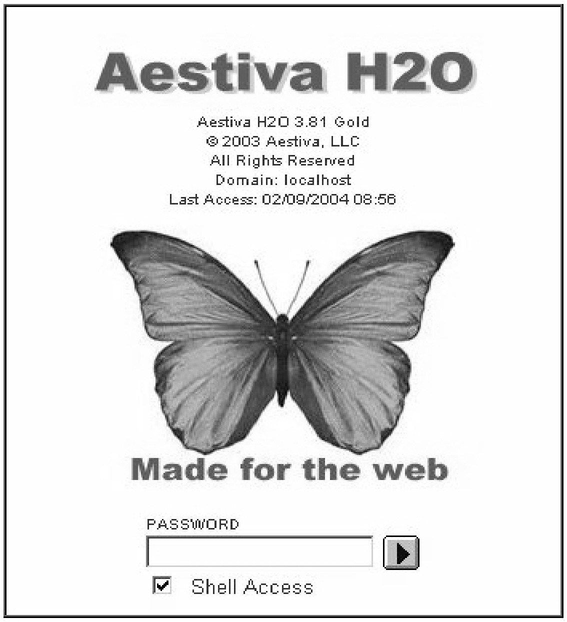


Figure 1.1 Shows the login screen of H_2O.

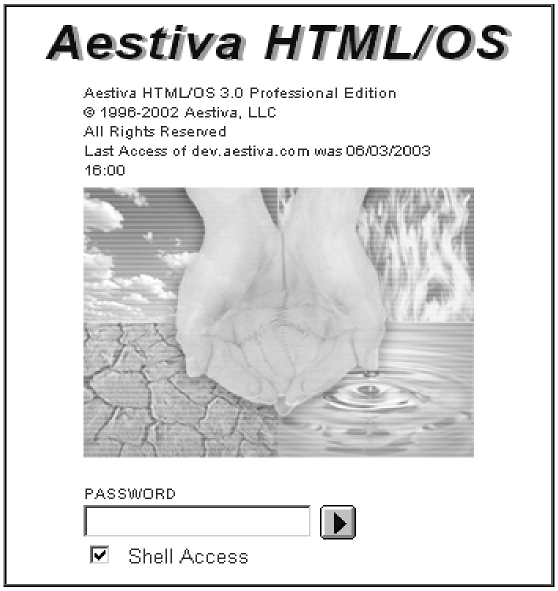
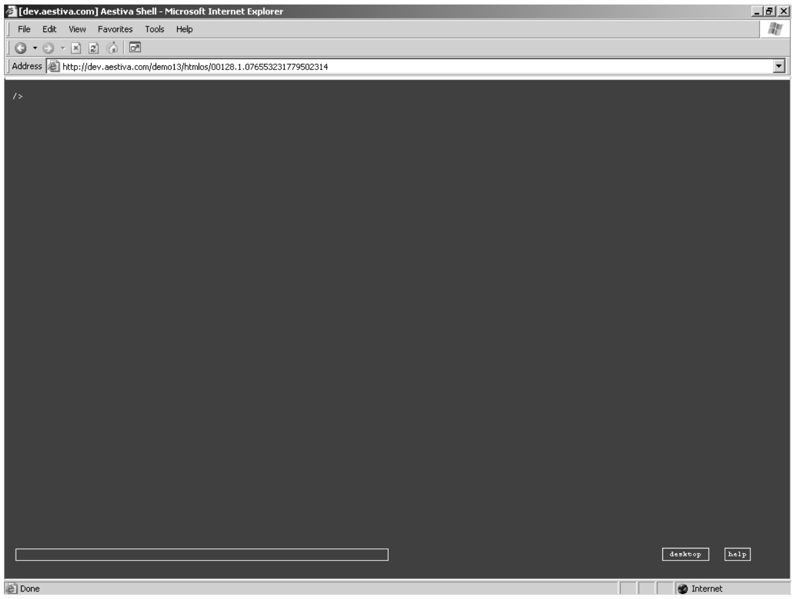

Aestiva HTML/OS
Aestiva HTML/OS 3.0 Professional Edition
© 1996-2002 Aestiva, LLC
All Rights Reserved
Last Access of dev.aestiva.com was 06/03/2003 16:00
PASSWORD
Shell Access

Figure 1.2 Shows the login screen of HTML/OS.

At the bottom of the login page, you'll see a Shell Access checkbox. This checkbox tells the Aestiva HTML/OS whether you want to log into the Web shell or into the Aestiva desktop. The Aestiva desktop is a graphical user interface that has many of the same capabilities of the Web shell but in a graphical layout.

For the purposes of this book, you will want to log into the shell, so make sure that the Shell Access checkbox is checked when you log in. If your login was successful, your first run of the Web shell should look like a relatively blank, dark window, as shown in Figure 1.3.


[dev.aestiva.com] Aestiva Shell - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://dev.aestiva.com/demo13/htmlos/00128.1.076553231779502314
desktop
help
Done
Internet

```
/>
```

Figure 1.3 When you log in, the Web shell consists of a black screen with a text area at the bottom.

After you successfully log in to your copy of the Web shell, you have full file permissions to work with the file system and run any commands. Because each copy of the Web shell is a single-user application, you have no file permission issues to consider, as you would with a traditional shell.

Introducing the Web Shell

The Web shell is a file management tool for Web-enabled servers. It allows you to remotely manage files on a server via a command line, create and edit files in a Web-based text editor, view remote image files, transfer files across the network, and encapsulate and deploy Web applications, as well as several other features covered throughout this book. In essence, the Web shell is a command-line driven Web-based integrated development environment. You can use it to create, manage, and deploy a Web application without needing to resort to third-party tools.

This section introduces you to the Web shell, explains some of the basic features of the Web shell interface, and covers some of the terms that are used throughout this book.

The Basic Interface Elements

Most of what you see when you first log into the Web shell is an empty screen. This area is where the Web shell outputs information when you run commands. This book refers to this area as the output area.

The small text box at the bottom of the screen is the command-line input box. This is where you type shell commands. The cursor is active in this area when the Web shell loads, and you can type in this text box without finding it and clicking it first. If you type a command in this text box and press Enter, the command itself as well as the results of the command are displayed in the output area.

To the right of the command line are two buttons labeled Desktop and Help. Clicking these buttons produces the same results as running the commands of the same name does. Clicking the Desktop button exits the Web shell and takes you to the Aestiva desktop, while clicking the Help button displays the basic shell Help file. When you become more experienced, you can type these commands on the command line instead of clicking the buttons provided. They are there for the first-time user and can be turned off later. (Turning off these buttons is covered in Chapter 3, "The Shell Environment.")

Web Shell Commands

All commands you issue to the Web shell have a few things in common. They are all combinations of words and symbols that are meaningful to the Web shell and tell it to perform tasks. Some commands are only one word long; others are several words long, with letters and symbols interspersed among them. For any single shell command, the first word is the most important. This is the word that indicates the name of the command itself. The rest of the words and symbols in a command statement are known as arguments and give the command specific instructions about how to accomplish its tasks. The general form of a command is the following:

```
/>command [ arguments]
```

Most often, the arguments you give to a command are simply filenames. Sometimes, though, the arguments indicate to a command how it should behave. This special type of argument is called a switch. The switches supported by Web shell commands vary widely. Some Web shell commands are complicated and support multiple switches, whereas others are simple and support no switches at all. (Refer to Chapter 2, "File Management," for more information on switches.)

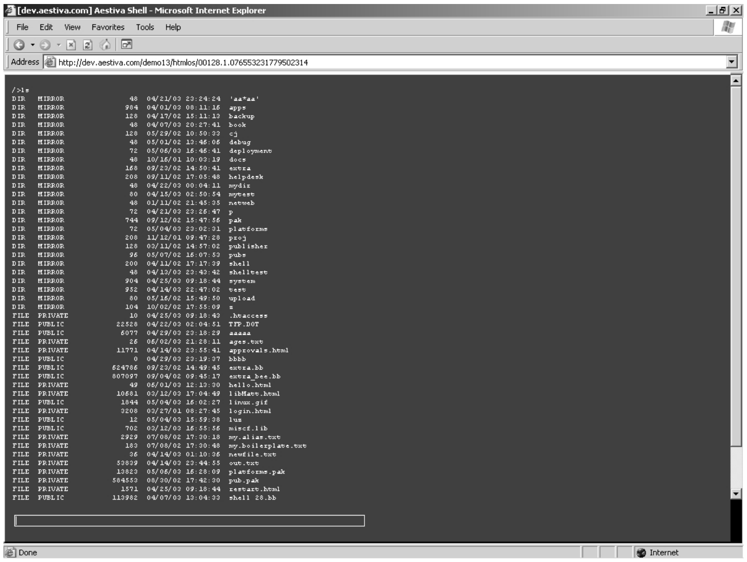
Common Commands

Although the Web shell supports several commands, some of which accomplish very specific tasks, there are two commands that most users use more than any others. These two commands list files in a directory, and change the current working directory, and are called `ls` and `cd`, respectively.

The ls Command

If you are logged in to a copy of the Web shell, type `ls` in the command input box at the bottom of the Web shell window and then press your Enter key. The Enter key on your keyboard submits your command to the Web shell and causes it to act and possibly to display information to the output area.

After you issue the command `ls`, the Web shell prints the contents of the current working directory to the output area. If you see something like the results in Figure 1.4, congratulations, you have just issued your first Web shell command!


[dev.aestiva.com] Aestiva Shell - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://dev.aestiva.com/demo13/htmlos/00128.1.076553231779502314
Done
Internet

```
/>ls
DIR   MIRROR          48   04/21/03 23:24:24  'aa*aa'
DIR   MIRROR         984   04/01/03 08:11:16  apps
DIR   MIRROR         128   04/17/02 15:11:13  backup
DIR   MIRROR          48   04/07/03 20:27:41  book
DIR   MIRROR         128   05/29/02 10:50:33  cj
DIR   MIRROR          48   05/01/02 13:46:06  debug
DIR   MIRROR          72   05/06/03 16:46:41  deployment
DIR   MIRROR          48   10/16/01 10:03:19  docs
DIR   MIRROR         168   09/23/02 14:50:41  extra
DIR   MIRROR         208   09/11/02 17:05:48  helpdesk
DIR   MIRROR          48   04/22/03 00:04:11  mydir
DIR   MIRROR          80   04/15/03 02:50:54  mytest
DIR   MIRROR          48   01/11/02 21:45:35  netweb
DIR   MIRROR          72   04/21/03 23:26:47  p
DIR   MIRROR         744   09/12/02 15:47:56  pak
DIR   MIRROR          72   05/04/03 23:02:31  platforms
DIR   MIRROR         208   11/12/01 09:47:28  proj
DIR   MIRROR         128   03/11/02 14:57:02  publisher
DIR   MIRROR          96   05/07/02 16:07:53  pubs
DIR   MIRROR         200   04/11/02 17:17:39  shell
DIR   MIRROR          48   04/13/03 23:43:42  shelltest
DIR   MIRROR         904   04/25/03 09:18:44  system
DIR   MIRROR         952   04/14/03 22:47:02  test
DIR   MIRROR          80   05/16/02 15:49:50  upload
DIR   MIRROR         104   10/02/02 17:55:09  z
FILE  PRIVATE         10   04/25/03 09:18:43  .htaccess
FILE  PUBLIC       22528   04/22/03 02:04:51  TFP.DOT
FILE  PUBLIC        6077   04/29/03 23:18:29  aaaaa
FILE  PRIVATE         26   06/02/03 21:28:11  ages.txt
FILE  PRIVATE      11771   04/14/03 23:55:41  approvals.html
FILE  PUBLIC           0   04/29/03 23:19:37  bbbb
FILE  PUBLIC      624786   09/23/02 14:49:45  extra.bb
FILE  PUBLIC      807097   09/04/02 09:45:17  extra_bee.bb
FILE  PRIVATE         49   06/01/03 12:13:30  hello.html
FILE  PRIVATE      10681   03/12/03 17:04:49  libMatt.html
FILE  PUBLIC        1844   05/04/03 16:02:27  linux.gif
FILE  PRIVATE       3208   03/27/01 08:27:45  login.html
FILE  PUBLIC          12   05/04/03 15:59:38  luz
FILE  PUBLIC         702   03/12/03 16:55:56  miscf.lib
FILE  PRIVATE       2929   07/08/02 17:30:18  my.alias.txt
FILE  PRIVATE        183   07/08/02 17:30:48  my.boilerplate.txt
FILE  PRIVATE         36   04/14/03 01:10:36  newfile.txt
FILE  PRIVATE      53839   04/14/03 23:44:55  out.txt
FILE  PRIVATE      13823   05/06/03 16:28:09  platforms.pak
FILE  PRIVATE     584553   08/30/02 17:42:30  pub.pak
FILE  PRIVATE       1571   04/25/03 09:18:44  restart.html
FILE  PUBLIC      113982   04/07/03 13:04:33  shell 28.bb
```

Figure 1.4 The ls command displays a directory's file contents.

If you successfully issued the `ls` command, you will see at the top of the window a sequence of characters: `/>ls`. This is what the command line would look like in a traditional shell, but in the Web shell this is a pseudo command-line, because this is not where your cursor writes as you type. When you issue commands to the Web shell, you type into the text box at the bottom of the screen and press Enter to submit your command. Your commands then are displayed in the output area after you submit them. The Web shell prints your commands and their output to the output area in the order in which you submitted them, so the context of every command's output is available to you.

The command you just issued, `ls`, prints a list of the files in the current directory as well as other more detailed information about the files `ls` found. By default, `ls` displays a list of output five columns wide. The most important column is the one farthest to the right; this is the name of the file. The name of the file is placed farthest to the right because its length can vary widely. Putting the filename on the right ensures that long filenames don't displace other data on your screen.

```
1     2                 3     4                      5
DIR   MIRROR            984   04/01/03 08:11:16      apps
```

The columns printed by `ls`, as shown in the previous excerpt, are the following:

```
   1. File type
   2. Aestiva file area
   3. File size in bytes
   4. Date stamp
   5. File name
```

The first column indicates the type of file, either a `FILE` or `DIR` (directory). Files hold data and directories hold other files and directories.

The second column indicates the file area in which the file resides. This tells you whether the file resides in the static or dynamic area of a Web server's file system. The Web shell marks files as `PRIVATE` when they reside in the directory that handles dynamic Web scripts (on a UNIX system, the `cgi-bin`); `PUBLIC` when they reside in the file area used for static documents (on a UNIX system, `htdocs`); and `MIRROR` when they reside in the `PRIVATE` and `PUBLIC` file areas. The Web shell's file system is a superposition of these two file system areas. The internal location of a file, therefore, is indicated by both the location of the file in the Web shell's virtual file system as well as by the public/private attribute of the file. Superimposing files in the public and private areas makes file management easier. The details of public and private files are discussed in Chapter 2, "File Management."

The third column of the output of the `ls` command displays the size of the file in bytes.

The fourth column displays the date stamp of the file. The date stamp tells you when a file was created or last modified, but not necessarily when it was last moved from one file location to another.

The fifth column displays the name of the file. A file's name is distinct from the file itself. It is a sequence of characters that you and the system use to identify the file. To stay out of trouble, a filename in the Web shell must not contain any nonalphanumeric characters. Because Web servers and browsers generally don't behave well with filenames that contain spaces or other nonalphanumeric characters, the Web shell has similar restrictions.

The cd Command

The other commonly used command is the `cd` command. The `cd` command changes the current working directory.

Introducing the Current Working Directory

The current working directory is the directory the Web shell assumes you mean when you don't specify any particular directory. When you first log in to the Web shell, the current working directory is set to the root directory of the Web shell's file system, specifically referred to with a single forward slash (`/`). As you work in the Web shell, the current working directory remains the same unless you specifically change it with the `cd` command.

When, for example, you perform an operation on a file or ask for a list of the files in a directory using `ls` without specifying a directory, the Web shell assumes that you mean the current working directory and looks there for your files. To take another example, if you need to delete a file in the current working directory, you would have to specify only the name of the file and not its containing folder. If the file were outside your current working directory, you would have to precede the filename with the directory in which the file resides.

Files don't have to reside in the current working directory for the current working directory to have an effect on your work. You can also specify filenames that are relative to the current working directory instead of typing their full names. The Web shell will assume, if you don't precede a folder name or a file name with a slash, that the file you specified starts with the name of the current working directory. Navigating the file system with the `cd` command and relative versus full paths is discussed in detail in Chapter 2, "File Management."

Changing the Current Working Directory

The `cd` command stands for *change directory* and changes the current working directory—the directory you are in—to a value that you specify as an argument. For example, if the current working directory is `/web/` and you want to go into a directory called `notes` located at `/web/notes`, you would type the following:

```
/web/>cd notes
```

As a result, the bottom-most line in the output area will read as follows. It indicates the new current working directory, `/web/notes`.

```
/web/notes/>
```

If you want to move to a directory within a directory, you may specify its path directly instead of performing intermediate steps. The following is an example of moving into a directory two directories deeper than the current working directory by issuing only one command:

```
/web/>cd notes/myproject
```

Using the `cd` command to navigate the file system is covered in more detail in Chapter 2.

Getting Help

The shell provides its own Help system to guide you through learning its features and to provide a reference for more advanced functionality. The command that you use to get basic help is `help`, or its equivalent, `man` (which stands for manual). These `man`/`help` commands can be run by themselves or with an argument containing the names of the commands you would like to read about in the manual. The following example retrieves the manual page for the `ls` command:

```
/>man ls
```

If you want a list of all the shell commands, type `help shell` at the command line. When you invoke the `help` command without arguments, you get a list of all the Web shell commands for which you can get help. Otherwise, you get a help file, or manual, for the command you entered as an argument. The exception to this is getting help about the shell. For this enter `help shell` to see a synopsis of the Web shell's features.

Exiting the Shell

It is a good idea to run the `logout` command when you exit the shell. The `logout` command invalidates the shell session and makes it impossible for other users to log into your shell account by simply running a URL that you used during your Web shell session. If you don't do this, a malicious user could log in to your shell session, if it hasn't yet expired, and have his or her way with your Web shell system. If you are using a public computer, don't forget to log out!

Summary

Aestiva makes available to you a copy of the Web shell that you can use to run the examples in this book. Once you register on the Aestiva Web site, you will have access to this copy and be able to run test commands. To begin, you will want to run the most commonly used commands, `ls` and `cd`, to get accustomed to the way the Web shell behaves. Along the way, use the integrated Help system to expand your knowledge of particular commands.

2

Chapter

File Management

The Web shell comes with a set of commands you can use to manage the files and directories. These commands list, move, delete, and create files and directories within your Web environment.

In addition to moving files and directories, you can also use the file-management commands to search for files, set their Aestiva file attributes, edit their contents, and search for text within files. In addition, the Web shell supports wildcard file expansion, so that you can perform tasks on multiple files by supplying a filename pattern instead of an explicit list of filenames.

Managing files already on your server is easy when you use the Web shell, but at times you will need to upload or download files. The Web shell provides seamless functionality to accomplish these tasks as well. You won't have to go looking for third-party tools to transfer files across the network. The Web shell provides networked file transfer functionality as part of its file-management command set.

Working in the File System

The basic unit of data storage on any computing system is the file. Files hold the data that computers use to run programs and that users utilize to store information. Any given system can handle and store data in other areas, such as in RAM, but any data that has to persist when the computer is turned off must be stored in the file system.

The three primary components of a file are its name, its location, and its contents. A file's name must uniquely identify it in its directory and must have a proper form. (For more information on filenames, see the accompanying note, "Filename Pitfalls.") Often, the filename, in addition to identifying the file, reveals the type of its contents as well. The part of a file’s name that indicates its type is called its extension and consists of a short sequence of letters preceded by a dot and is placed at the end of the file name

Filename Pitfalls

*Typically filenames don’t contain spaces or funny characters, such as ^ or *. These characters can cause problems on the Web and within the shell interface, so you should avoid them. Use the underscore character to signify a space; with the exception of the dot (.) and dash (-), avoid nonalphanumeric characters altogether.*

Often, filenames also indicate by their extension the type of data they contain. A file’s extension, or the code after the last dot in its filename, indicates the type of data contained in the file. A few typical file extensions used on the Web are `.html` for Web page files, and `.gif` and `.jpg` for image files. Extensions are not always necessary, but this depends on the operating system and what the file will be used for. On UNIX systems, file extensions have been traditionally avoided; whereas on Windows systems, file extensions are generally required. On the Web, file extensions are also generally required. They can tell servers and clients what to do with a file. The Web shell also uses file extensions to determine what to do with a file in many cases. How the Web shell deals with extensions is discussed later in this chapter.

Another important attribute of a file is its location. For convenience, modern operating systems allow for the organization of files in a system of folders or directories. All files on a system reside in folders, most of which themselves reside in other folders. The way to specify the location of a file in the Web shell, as well as on most other systems, is by naming the succession of folders leading up to the target location, where folders are separated by a forward slash (`/`). This list of the folders containing a file or folder is known as a path.

All paths begin with the base directory on the system, the root directory. The root directory is specified by a forward slash (`/`) and is the parent of all files and directories. Child directories reside within these directories and may themselves contain other directories. The resulting arrangement is a hierarchical tree of files and folders.

Specifying a file in this tree involves listing the various directories that contain the files in their hierarchical order, each separated by a forward slash. This enumeration of directories, when starting from the root directory, is known as a full or absolute path. Later, this chapter talks about a similar concept, the relative path.

Navigating File Paths with the cd Command

The Web shell comes installed as part of a suite of Web-based tools and their related files and directories. For illustration, this chapter references these files, because they are common to every copy of the Web shell.

When you log into the shell, the current working directory is the root directory. (For more information on the current working directory, refer to the note, "The Current Working Directory, in this section.) Running the command `ls` as your first command yields a list of all of the files in the root directory. All files and folders in the root directory have the path `/<filename>`. For example, a file named `test.txt` in the root directory has the path `/test.txt`.

The Current Working Directory

The current working directory is the primary directory in which you are presumed to be working. When you log in, the current working directory is set to `/`, the root directory. You change its value by using the `cd` command and utilize it when you refer to files without explicitly naming their directories. Doing so saves you from having to type directory names.

From the root directory, you can change the current working directory to the directory `/apps` by using the `cd` command:

```
/>cd /apps
```

After you run this command, the command prompt should indicate the new value of the current working directory:

```
/apps/>
```

At this point, when you type the command `ls` with no arguments, all files and directories listed have the general path `/apps/<filename>`.

You could refer to these files by their full paths, as in the following example, which changes the working directory to `/apps/shell/`:

```
/apps/>cd /apps/shell
```

Because it could quickly become tedious to type the full paths of files and directories, especially if they reside deep inside the file hierarchy, most users omit the current working directory part of a file's path. The Web shell assumes that you mean the current working directory when you omit the full path to a file, and you can indicate whether this is your intention by supplying or omitting a leading slash to a file argument.

In the previous example, because the working directory was `/apps/`, you could have more easily issued the following command:

```
/apps/>cd shell
```

Because in this case the file argument, `shell`, doesn't start with a slash, the Web shell prepends it with the value of the working directory. These kinds of file paths, those that don't begin with a slash and therefore imply the working directory, are called relative paths.

Most of the time, you refer to a file in the shell by its relative path, that is, by its path relative to the current working directory. In a sense, it is the sole purpose of the current working directory to free you from having to refer the full paths of files and folders that you want to reference. It is often easier to change the working directory to the one closest to the file area you want to manipulate and use relative paths than it is to not change the working directory and use full paths to files you want to reference, even if you have only one simple operation.

Using the dot and dot-dot Directories

The dot (`.`) and dot-dot (`..`) directories are pseudo directories present in every directory on your system. They are not real directories in that you don't have the ability to add or delete them as you would normal directories. They are instead references to existing directories: The dot directory is a reference to the indicated directory, and the dot-dot directory is a reference to the directory above the indicated directory.

To move up in the directory structure, use the dot-dot directory with the `cd` command. If, for example, the current working directory is `/apps/shell/` and you want to move into the parent directory, `/shell/`, issue the following command:

```
/apps/shell/>cd ..
```

The dot-dot directory represents the parent directory in any given directory. When you issue the command `cd ..`, you are telling the Web shell to change the working directory to the parent of the current working directory.

Use the dot-dot directory with any valid file path. For example, you could use the `cd` command to switch into a sibling directory, if it exists, by issuing the following command:

```
/apps/shell/>cd ../calendar
```

This command tells the Web shell to change the working directory to a directory in the parent directory called `calendar`. Doing so is equivalent to changing from `/apps/shell` up one directory to `/apps/` and then down to `/apps/calendar`. You can use the dot-dot (`..`) directory to traverse up the directory tree as far as you need to go by invoking it multiple times. For illustration, the following command uses the dot-dot directory in multiple places to change the working directory to the current working directory; in other words, it changes nothing:

```
/apple/banana/cherry/>cd ../../banana/cherry/date/..
```

In the preceding command, the current working directory is changed to two directories above the current working directory, `/apple`, then down to `/apple/banana/cherry/date` and back up to where you started, `/apple/banana/cherry`.

Another pseudo directory at your disposal when you use the Web shell is the current directory, represented by one dot (`.`). You can use the single-dot directory when a command needs an explicit directory and you don't want to type the current working directory. For example, if you want to delete the current working directory, you can issue the command `rmdir` with the argument (`.`). The `rmdir` command is discussed later in this chapter. All you need to know about `rmdir` for now is that it deletes the directory whose name you pass to it as an argument. If you issue the following `rmdir` command, the date directory is deleted.

```
/apple/banana/cherry/date/>rmdir .
```

It would have had the equivalent effect if you had issued the following command using a full path instead of a relative path.

```
/apple/banana/cherry/date/>rmdir /apple/banana/cherry/date
```

As is evident in the preceding example, using relative paths at the command line almost always saves you typing.

You can also use the dot directory as you would the dot-dot directory in a file path, although it is difficult to conceive of a use for it in this context. The following example changes the current working directory to `/apple/banana/` and utilizes the dot-dot directory as well as the dot directory.

```
/apple/banana/cherry/date/>cd .././../.
/apple/banana/>
```

In the preceding example, you can safely ignore any dot slashes (`./`) in the file path because they refer to the current directory and are therefore redundant. The preceding command, stripped of its dot slashes (`./`), is equivalent to the following command, which moved the current working directory to a directory two directories above its current value:

```
/apple/banana/cherry/date/>cd ../..
/apple/banana/>
```

The dot slashes were not needed in the preceding example, but you will often need to use them. The dot directory is more often used to perform a command like the following:

```
/apple/banana/cherry/date/>mv /index.html .
```

The preceding command tells the shell to move the file called `index.html` located in the root directory (`/`) into the current working directory, the directory represented by the single dot. The first argument (`/index.html`) is the filename, and the second argument (`.`) is the destination directory. (The `mv` command moves files and is covered in more detail later in this chapter.) In general, the dot directory doesn't signify the current working directory but rather the directory indicated by the file path.

Since the second argument in the preceding command (`.`) doesn't begin with a slash and therefore signifies a relative path, it is equivalent to the following command:

```
/apple/banana/cherry/date/>mv /index.html /apple/banana/cherry/date/
```

The dot directory invokes the value of the current working directory when used by itself; but it is not limited to this usage.

The cdl Command

The `cdl` command combines `cd` and `ls` into one command to save typing. Instead of having to type `ls` to see the files in a directory after moving into it, the list is displayed automatically when you type `cdl`. On many Web copies of the Web shell, the `cd` command is aliased to a command called `cdl`. That means that when you type `cd`, internally, the Web shell runs the `cdl` command. To find out whether this is the case on your system, use `cd` to change into a directory and see if the list of files in that directory is displayed. Chapter 3, "The Shell Environment," covers how to manage aliases.

Working with Directories

Your directory structure is not static. You often need to make changes, such as move your directories, or create new ones to meet your needs. You also need to perform basic housekeeping; for example, you might need to remove a directory that is no longer used. The Web shell makes such administrative tasks easy to perform. In this section, you learn about the commands, such as `mkdir` and `cpdir`, you can use to create, remove, copy, and move your directories. This section also presents a detailed discussion of `rmdir`, which was mentioned previously.

Creating Directories

To make new directories on your Web server, use the command `mkdir`. The arguments you supply to `mkdir` are the names of the directories you want to create. If, for example, you are in your test directory (in this example, `mytest`), to create a new directory inside this folder, issue the following command:

```
/mytest/>mkdir newdir
```

In this case, `mkdir` makes a new directory called `newdir` and places it inside `mytest`. The full path to the `newdir` directory is, therefore, `/mytest/newdir`. Alternatively, you could have called `mkdir` with the full path to the new directory as its file argument, in the following way:

```
/mytest/>mkdir /mytest/newdir
```

But using the full path requires additional typing. Because you started in the `/mytest` directory already (meaning that `/mytest` was the current working directory), you can omit that part of the path to the new directory and imply it by using a relative path.

If you want to create a new directory outside of the working directory, issue the `mkdir` command with the full path of the new directory or use the dot-dot directory to refer to it. It is up to you to determine when it is more convenient to use the `cd` command to move to a new directory and work from there, or when to issue commands with more complicated paths.

To create a new directory outside of the current directory, execute a command similar to the following:

```
/mytest/>mkdir /mytest2/newdir
```

The preceding command creates a new directory in `/mytest2` with a full path. You could also have issued the following command that uses a relative path and the dot-dot parent directory as follows:

```
/mytest/>mkdir ../newdir
```

Using relative pathnames with the dot-dot parent directory is convenient in simple cases. But when you would otherwise have to use multiple dot-dot directories, you are less prone to make an error if you use the full path.

Regardless of how you refer to a directory argument when you use the `mkdir` command, make sure that the parent directories — the directories that will contain the new directory — already exist. If the parent directories don't exist, the `mkdir` command fails to create the new directory.

In the preceding example, if the parent directory to the new directory — `/mytest2` — doesn't exist, `mkdir` quietly fails to make the new directory. If you want to make a directory inside of several other directories that don't exist, you have to do so incrementally. For example, if you want to create the directory `/first/second/third`, and none of the parent directories exist, you have to create each directory first, starting from the root directory, with separate commands, as shown here:

```
/>mkdir first
/>mkdir first/second
/>mkdir first/second/third
```

The result of the preceding commands is a new directory path, `/first/second/third`.

Although you can't create directories if their parent directories don't exist, you can create multiple directories with one instance of `mkdir` by passing multiple directories as arguments. To do this, separate additional directories with spaces:

```
/>mkdir aaa bbb ccc
```

In the preceding example, all three directories, `aaa`, `bbb`, and `ccc`, were created at the same time (well, almost). You can see the result of this command by calling `ls`:

```
/>ls
DIR   MIRROR                  48  04/20/03 20:49:15  aaa
DIR   MIRROR                  48  04/20/03 20:49:15  bbb
DIR   MIRROR                  48  04/20/03 20:49:15  ccc
```

The result of the `ls` command tells you that `mkdir` was successful in creating the directories `aaa`, `bbb`, and `ccc`. You'll probably want to call `ls` after running `mkdir`, just to make sure that the directories were created properly.

Removing Directories

Because directories can contain multiple files as well as other directories, the Web shell treats directories differently from the way in which it treats files. Directories often have their own commands. To delete a directory, use the command `rmdir`. The `rmdir` command deletes an empty directory if you supply it as an argument. In the following example, `rmdir` deletes the empty directory `/emptydir`.

```
/>rmdir emptydir
```

If you want to delete a directory that contains files or other directories, you have to tell `rmdir` explicitly to delete the contents of the directory you want to delete before it deletes the directory itself. You can tell `rmdir` to do this by putting a `-r` by itself after the command. Most users put special indications, such as `-r`, immediately after the command and before any file arguments. (See the accompanying note, "Command Switches.") The following example deletes the `/junk` directory and all of its contents:

```
/>rmdir -r /junk
```

If you try to delete a directory that has file contents without including the special `-r` switch, `rmdir` won't delete anything. Instead you see a message that tells you that the directory you are trying to delete is not empty.

Command Switches

The `-r` argument to the `rmdir` command is an example of a switch. A switch comprises a dash followed immediately by a letter and is a special argument to a given Web shell command. Many Web shell commands support switches to give you added control over how the command behaves.

In the case of the `rmdir` command, the `-r` switch is required for nonempty directories to ensure that you know that the directory you want to delete is not empty.

It goes without saying that the `rmdir -r` combination is a powerful command. You could potentially destroy a tremendous number of files, and you wouldn't be able to revive them afterwards, so be careful when you use it. Thankfully, although you could delete any number of files and directories in your file system, the Web shell won't let you delete the root directory (`/`), partially because that would mean suicide for the Web shell itself.

Copying and Moving Directories

One simple way to back up all your work is to copy the base directory that contains it. To copy an entire directory, use `cpdir` the command. The `cpdir` command copies a directory and its contents to another directory. The general form of the command is as follows:

```
/>cpdir SOURCE DESTINATION
```

The source and destination have to be directories. If the destination directory already exists, the clone of the source directory is placed inside the destination directory and given the same name as the source.

If you use `cpdir` to back up your work, you would probably want to devise a naming convention for your backup directories. If, for example, your application were in a directory `/calendar`, you might store the backup copies of your calendar application in a `/backup` directory and give the backup a name that indicates the date on which the backup took place:

```
/calendar/>cpdir /calendar /backup/calendar/2003_NOV_3
```

Assuming the `2003_NOV_3` directory doesn't already exist, the contents of `/calendar` would be copied into that directory and could serve as a snapshot of the current state of development of your `/calendar` application.

To move or rename a directory, use `mvdir` in a similar manner. The `mvdir` command works just like the `cpdir` command; but it deletes the source directory after copying it to the specified location.

Working with Files

After your directory structure is established, you probably won't have to change it very often. Files, however always need to be moved around and managed as part of the basic task of managing a Web-based system. The Web shell supplies commands, such as `cp`, `mv`, and `rm`, to enable you to copy, move, and delete files.

Copying Files

You'll often need to create a copy of a file for various reasons, for example, to back up an existing file, or to create a new file using an existing file as a starting point. To copy files in the Web shell, use the `cp` command. The `cp` command creates a copy of an existing file and gives it the name you specify. Generally, the `cp` command takes the following form:

```
/>cp SOURCE DESTINATION
```

In this command, `SOURCE` is the path to the existing file and `DESTINATION` is the path to the copy of the file. If the destination filename specifies a folder in addition to a filename, make sure that the folder already exists. Otherwise, you'll have to create it with `mkdir`.

To create a backup copy of a file in your working directory, you might use the `cp` command in the following manner:

```
/mydir/>cp index.html index.html.bak
```

The preceding command creates a copy of `index.html` and names that copy `index.html.bak`.

If the destination file already exists, using the `cp` command will overwrite the existing file. Be careful when calling `cp` if you aren't sure whether the destination file exists. You can easily delete a file if you choose a destination filename that already exists.

If you want to copy the source file to a different directory while keeping the source file's name, you need to specify only the destination directory as an argument. In general, if the destination filename is a directory, `cp` creates a copy of the source file in that directory. Here is an example that creates a copy of `index.html`:

```
/>cp index.html mydir
```

The preceding command creates a copy of `/index.html` and places it in `/mydir`. If you move into `mydir`, you would then see it by calling `ls` as follows:

```
/>cd mydir
/mydir/>ls
FILE  PRIVATE              143  04/20/03 22:06:52  index.html
```

The presence of the newly created `index.html` file indicates that the file copy was successful.

To copy multiple files into an existing directory, name the source files as arguments before the final argument, the destination directory. In general, if the last argument you issue to the `cp` command is a directory, the Web shell tries to make copies of all the files specified as arguments and places those copies in the destination directory, as in the following example:

```
/>cp img1.gif img2.gif img3.gif mydir
```

The preceding command creates copies of `img1.gif`, `img2.gif`, and `img3.gif` and places those copies in `mydir`. If you move into `mydir`, you see these new files by issuing a call to `ls`:

```
/>cd mydir
/mydir/>ls
FILE  PRIVATE              623  04/20/03 22:07:22  img1.gif
FILE  PRIVATE              611  04/20/03 22:07:22  img2.gif
FILE  PRIVATE              607  04/20/03 22:07:22  img3.gif
```

Again, keep in mind that `cp` overwrites any destination files if they already exist. This is particularly important when you call `cp` with a directory destination, because you aren't specifying explicit target filenames. The files whose names are inferred by the Web shell will be overwritten.

Removing Files

You'll often want to remove files that you don't need, either to free disk space, or to tidy your file system. To delete files in the Web shell, use the `rm` command. The `rm` command quietly and permanently removes any file whose name you supply as an argument. The `rm` command is easy to use; but it's also powerful, so be careful when you use it. It won't ask you whether you are sure about deleting a file, and you won't be able to undelete any deleted files if you happen to change your mind.

After you make sure that you want to delete a file, you can do so by supplying it as an argument to `rm`, as in the following example:

```
/mydir/>rm index.html.bak
```

The preceding example removes the file named `index.html.bak` in the working directory, `/mydir`. The `rm` command tells you if it was unable to remove the file for any reason; but if it was successful, it says nothing. You run an `ls` command to get an updated view of the file system.

To remove multiple files, supply those filenames as additional arguments. The following example removes all the files in the `/images` directory:

```
/images/>rm armadillo.jpg gazelle.jpg llama.jpg zebra.jpg
```

If you were to run `ls` against the `images` directory, you should see that the files are gone.

Take note that the `rm` command works against files, but doesn't work against directories. If you want to remove directories, use `rmdir` instead.

Moving Files

When you need to move a file to a different directory, or rename a file, or both, use the `mv` command. The `mv` command works in a manner similar to `cp`. It changes the path of a file from its current value to another value. In general `mv` requires a source and a destination filename as arguments.

```
/>mv SOURCE DESTINATION
```

As with the `cp` command, the source filename must be an existing file, and the destination filename must be another valid filename, whose parent directory must exist, or it must be an existing directory. If the destination is a valid filename, the source file is renamed and effectively moved to the destination filename if the new directory is different. If the destination filename is a directory, the `mv` command moves the indicated file to the destination directory.

To move a file called `gazelle.jpg` to a directory called `images`, you issue the following command:

```
/>mv gazelle.jpg images
```

You check to see that the file moved by changing into the `images` directory and getting a file list, as follows:

```
/>cd images
/images/>ls
FILE  PRIVATE            21991  04/20/03 23:09:44  gazelle.jpg
```

The `ls` command tells you that the move was successful, because its results indicate that the file `gazelle.jpg` is now in the `/images` directory.

As with the `cp` command, you can move multiple files at one time by supplying them as arguments, with the final argument being the destination directory. The following example moves three `.jpg` files into the `images` directory:

```
/>mv zebra.jpg armadillo.jpg llama.jpg images
/>cd images
/images/>ls
```

```
FILE   PUBLIC               11661  04/20/03 23:27:36  armadillo.jpg
FILE   PRIVATE              21991  04/20/03 23:09:44  gazelle.jpg
FILE   PUBLIC               19878  04/20/03 23:27:47  llama.jpg
FILE   PUBLIC               41563  04/20/03 23:27:26  zebra.jpg
```

Check your work with the `ls` command to make sure that the `mv` command was successful.

Using Wildcards

It can be tedious to type the full name of files you pass as arguments to a command. The Web shell supports the use of wildcards (sometimes called file globs) to make it easier to indicate one or more files with one filename pattern. Instead of typing the full name of a file, you can specify a filename pattern and let the Web shell fill in the blanks and draw the appropriate file matches. The character you use to specify to the Web shell that it can fill in the blanks for filenames is the asterisk character (`*`).

For example, to delete all the `.jpg` files in the `/images` directory, you issue the following command:

```
/images/>rm *.jpg
```

The `*` signifies any sequence of characters of any length and draws matches from the names of files in the Web shell's file system. In the preceding example, all files with names ending with `.jpg` are matched. The Web shell supplies those matching files to the command—in this case `rm`—as file arguments. Because the argument (`*.jpg`) doesn't begin with a slash and doesn't indicate any folder names, the Web shell matches files in only the current working directory.

Because the `/images` directory contains no other files, as evidenced by the output of `ls`, you could have issued the following command:

```
/images/>rm *
```

The command `rm *` removes all files in a directory.

Before issuing the `rm` command, the `images` directory contained four files:

```
/images/>ls
FILE   PUBLIC               11661  04/20/03 23:27:36  armadillo.jpg
FILE   PRIVATE              21991  04/20/03 23:09:44  gazelle.jpg
FILE   PUBLIC              19878  04/20/03 23:27:47  llama.jpg
FILE   PUBLIC              41563  04/20/03 23:27:26  zebra.jpg
```

To remove only one of the files, you don't have to supply the full name to the `rm` command. Instead, you could supply a portion of the filename and use the asterisk character to imply the rest of the name. If, for example, you want to remove the `armadillo.jpg` file, you issue the following command:

```
/images/>rm a*
```

The `a*` argument matches all files in the current directory whose names begin with an `a`. Because `armadillo.jpg` is the only file whose name begins with an `a` in the `images` directory, it is the only one that matches the pattern and is the only filename argument supplied to the `rm` command.

More complicated patterns are also possible. For example, to move files in the working directory that begins with `a` and end with `.gif` to a folder called `/trash`, you issue the following command:

```
/graphics/>mv a*.gif /trash
```

The preceding command moves all matching files, those that begin with `a` and end with `.gif`, in the `/graphics` directory and places them in the `/trash` directory. If thousands of files matched that pattern, using the wildcard character would save you a tremendous amount of typing.

But even when only one file or directory produces a match, the wildcard character almost always saves typing. After you get accustomed to wildcards, you will likely start using them most often with commonly used commands, such as `cd`. Instead of, for example, moving into a new directory by naming it explicitly, you need to specify only enough to identify it uniquely and let the Web shell do the pattern matching. If the root directory contains only one file starting with `b`, `/business_applications`, you could issue the following command to move into that directory:

```
/>cd b*
```

Remember though, that the wildcard characters work only for files and directories. They won't match anything else, such as commands or shell environment values.

In rare cases, you'll want to turn off pattern matching at the command line. If, for example, a filename contains an asterisk in its name and you want to supply it as an argument, you have to tell the Web shell that you mean a literal asterisk, not a wildcard character. To turn off wildcard matching, wrap the file argument in double quotes:

```
/mytest/>rm "a*b.txt"
```

The preceding command deletes the file `a*b.txt`. If you omit the double quotes, the `rm` command matches all files beginning with `a` and ending with `b.txt`, including, but not limited to `a*b.txt`.

In general, the Web shell always assumes that any non-switch argument that you pass to a command is a filename. If you want to supply an argument that is not a file or directory, a good rule of thumb is to wrap that argument in double quotes so that the Web shell doesn't go looking for matching files.

In addition to matching filenames, the Web shell also matches partial or entire paths of files and directories. For example, say you have a directory with the following subdirectories:

```
/mydir/>ls
DIR   MIRROR                  48  04/21/03 23:36:06  aaa
DIR   MIRROR                  48  04/21/03 23:36:06  bbb
DIR   MIRROR                  48  04/21/03 23:36:06  ccc
```

If each of the directories contains multiple files, you could remove all of them at once, and keep their parent directories intact, with the following command:

```
/mydir/>rm */*
```

The Web shell allows for any number of wildcard characters in a pathname. The following moves all matching files in the `/myapplications` directory, into the `/trash` directory:

```
/>mv /myapplications/*/documents/*.html.old /trash
```

The preceding path name matches the following files. The variable part of the pathname is printed in caps:

```
/myapplications/AAA/documents/XXX.html.old
/myapplications/BBB/documents/YYY.html.old
/myapplications/CCC/documents/DDD.html.old
```

Using wildcards at the folder level in your pathnames can be a powerful way to reference a large number of files in your file system, but it can be expensive for the Web shell to look in all of these directories for matches, so it isn't a good means of saving time typing.

Finding Files

To find a file when you know part or all of its name, but not its location, use the `find` command. The `find` command looks in multiple subdirectories for a file whose name you specify as an argument. If, for example you know you have a file called `checkout.php` somewhere on your system, but you don't know where it is, you could issue the following command:

```
/>find checkout.php
/myapps/pages/checkout.php
/work/test/checkout.php
```

The lines after the `find` command statement are the output of the command. These are the files the `find` command has found that match your argument.

By default, the `find` command traverses the entire directory tree beneath your current working directory as it looks for matches. To tell `find` to refrain from going too deep into the directory structure (and make you wait while it does so), you can use the `-d` switch to tell it how deep to go. If you use the `-d` switch, you must supply a number after the switch indicating directory depth. Some switches require arguments. See the accompanying note, "Switches That Require Arguments," for more information.

Switches that Require Arguments

Some switches are special in that they require an argument themselves. The `-d` switch for the `find` command is an example of a special switch. It requires an integer argument that tells the `find` command how many directories deep to look for matching filenames. The argument `1` corresponds to the working directory, `2` to the level below the working directory, and so on. For example, to find the file `checkout.php` and to avoid looking deeper than four directories, use `find` in the following way:

```
/>find -d 4 checkout.php
/myapps/pages/checkout.php
/work/test/checkout.php
```

The Web shell is smart about these special switches. If a switch requires an argument, the Web shell won't confuse the switch argument for a filename argument; a switch argument is tightly bound to its switch.

By default, the `find` command looks in the current working directory for the file with a name you specify. If you want to find a file in a directory other than the current working directory, you must supply it as the first argument, as in the following example:

```
/>find /myapps checkout.php
/myapps/pages/checkout.php
```

In general, excluding any switches and their arguments, if you give the `find` command one argument, it interprets that argument to be the filename or file pattern for which it should look; if you give it two arguments, the `find` command interprets the first argument to be the base directory in which it should look and the second argument to be the filename.

The preceding `find` command looks only inside of the `/myapps` directory for possible instances of files named `checkout.php`. That is why `find` won't find the file of the same name that was found previously, `/work/test/checkout.php`.

The `find` command also accepts wildcard filenames. When you supply a wildcard filename to `find`, make sure to wrap the file argument in double quotes to prevent the Web shell from expanding the file pattern and replacing it with file matches. Remember that filenames containing wildcards are expanded into and replaced by a list of files whose names match the filename's pattern. To look in multiple subdirectories for a filename pattern, you have to turn off wildcard expansion by wrapping any file pattern in double quotes. The following example looks for all files in the current directory with names containing `aaa`:

```
/>find "*aaa*"
```

If you had omitted the quotes and there had been a file named `aaa.txt` in the current working directory, the find command would have been interpolated into the following statement, which is not what you wanted:

```
/>find aaa.txt
```

The preceding erroneous command, lacking double quotes around the filename pattern, would have therefore only found files on your system with the name `aaa.txt`, not files containing the text `aaa`, as was the initial intent.

Managing Text Files

Because most of your Web files will likely be text-based, the Web shell comes with basic commands to display and search for content within text files. You can view the content of one or more text files in the Web shell output area with the `cat` command, or look for text-based content anywhere on your system with the `findtxt` command.

Viewing Text Files

One way to view the contents of one or more text files quickly is with the command `cat`. The `cat` command was designed to concatenate multiple files and display them on the screen, but it is also typically used to display the contents of a single file. To print the contents of a file to the screen, pass the filename value to the `cat` command as an argument as in the following example:

```
/apps/shell/conf/>cat settings.conf
"workingdircolor","#cfcf99"
"outputcolor","#999999"
"inputcolor","#6699cc"
"runheight","500"
"runwidth","900"
"scrollbuffer","40"
"autoscroll","on"
"inputsize","80"
"showbuttons","on"
"backgroundcolor","#000000"
"fontface","courier new"
"fontsize","10"
```

If the file is a text file (for example, it's not an image or other binary file), `cat` displays its contents to the output area of the Web shell. If you accidentally use `cat` on the contents of a binary file such as an image, however, you might get some funny characters. This won't cause you problems (as it could in legacy shells). But keep in mind that if the output isn't what you expected, you might not be looking at a text-based file.

To print multiple files, pass the files as arguments to the `cat` command. The following statement displays the contents of all files two directories deep that end with `.html`:

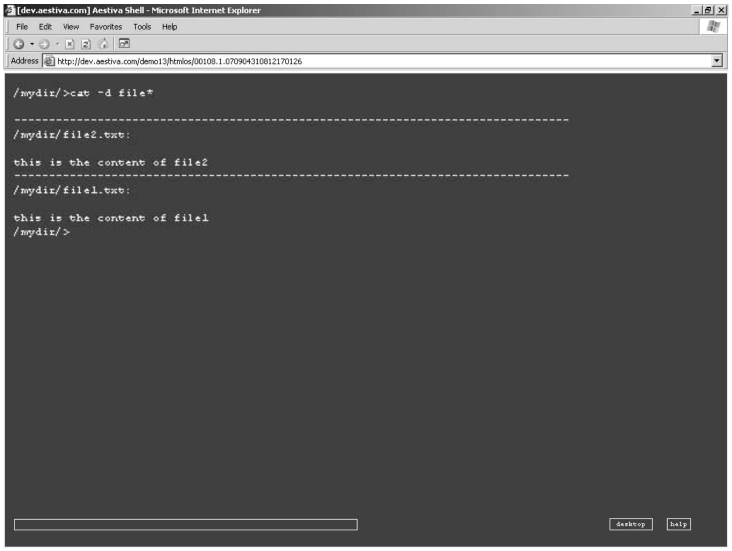
```
/>cat */*/*.html
```

The contents of the files that match the file arguments will be concatenated and displayed to the screen.

The cat command also accepts the following switches:

- `-d` separates files with a text divider.
- `-l` outputs line numbers.

The `-d` option tells `cat` to separate multiple files' contents with a text divider as well as the a filename header.


[dev.aestiva.com] Aestiva Shell - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://dev.aestiva.com/demo13/htmlos/00108.1.070904310812170126
desktop
help

```
/mydir/>cat -d file*

--------------------------------------------------------------------------------
/mydir/file2.txt:

this is the content of file2
--------------------------------------------------------------------------------
/mydir/file1.txt:

this is the content of file1
/mydir/>
```

Figure. 2.1 Running the `cat` command against multiple files with the `-d` option makes it easy to view the contents of multiple files at a time, while keeping the files' contents distinct.

When it is necessary to print the line numbers of a file's contents, you can use the `-l` option:

```
/>cat -l file1.txt file2.txt
/apps/shell/conf/>cat -l settings.conf
1      "workingdircolor","#cfcf99"
2      "outputcolor","#999999"
3      "inputcolor","#6699cc"
4      "runheight","500"
5      "runwidth","900"
6      "scrollbuffer","40"
7      "autoscroll","on"
8      "inputsize","80"
9      "showbuttons","on"
10     "backgroundcolor","#000000"
11     "fontface","courier new"
12     "fontsize","10"
```

The `cat` command resets the line numbers at the beginning of every file when you use `cat -l` on multiple files.

Searching for Text in a File

If you want to find an instance of a word or phrase in a set of files, use the command `findtxt`. The `findtxt` command is similar to the `find` command in that it searches all files in a directory tree for a match; but whereas `find` looks at the names of files for matches, `findtxt` looks at the contents of files.

If you supply one argument to `findtxt`, it looks in all files in subdirectories of the current working directory for files with contents that match the argument you supply. In the following example, the `findtxt` command looks in all directories in the Web shell's file system for files containing the word `aestiva`:

```
/>findtxt aestiva
/apps/readme.txt
```

The `findtxt` command prints all the matching files' pathnames to the output area. In the preceding case, the `findtxt` command found one file with the word `aestiva` in it, `/apps/readme.txt`.

As with the `find` command, you can limit the maximum depth to which `findtxt` traverses the file system with the `-d` switch. After the `-d` switch, remember to supply a number indicating the maximum traversal depth:

```
/>findtxt -d 3 aestiva
```

Because of the `–d` switch, the preceding call to `findtxt` won't go more than three levels down any directory path.

The `findtxt` command also supports a starting directory other than the current directory, if one is supplied as the first argument, as in the following example:

```
/>findtxt /apps aestiva
```

If you supply a `-d` switch to `findtxt`, remember that it signifies a directory depth relative to the starting directory.

Also be careful to wrap any arguments that contain spaces or odd characters in double quotes. The space, in particular, sets off different arguments unless it is contained in double quotes as in the following example:

```
/>findtxt "peter rabbit"
```

If you had omitted the quotes, the Web shell would have responded as though `peter` and `rabbit` were two distinct arguments instead of one and would have looked for instances of the word `rabbit` in the directory `peter`.

The result of `findtxt` is a list of paths to files that made a match. To make this list clickable, such that clicking any file path opens an editor window for that file, use the `-c` option. You might want to try using the `-c` option and clicking a resulting file to see this happen. After you click a filename (it won't appear clickable to you, but it will be), the Web shell opens an editor window with the file you clicked.

Managing H_2O File Attributes

Lying underneath the Web shell's file system is the HTML/OS file system. The file system is a bit unusual in that it maps two different server directory trees onto the same HTML/OS directory. That means that when you look at the file and directory tree from within the Web shell, you are actually looking at a superposition of two directory structures—the document directory and a folder in the scripts directory—on the Web server. On a UNIX system, the document directory is usually called `htdocs`, and the scripts directory is called `cgi-bin`. The HTML/OS file system—and by extension the Web shell file system—is the superposition of these two directories and their contents, `cgi-bin/private` and `htdocs`.

The `htdocs` side of the file system is called the public side, and the `cgi-bin/private` side is called the private side. This naming convention is used because Web servers give unrestricted access to files in the `htdocs` side of the file system unless restricted. On the other hand, files outside of the `htdocs` area not accessible from the Web. Whether a file or directory resides on the public side or the private side or both is considered by the Web shell to be an attribute of the file. The possible attributes of an HTML/OS file are `public`, `private`, or `mirror`.

When you make a new directory in the Web shell with the command `mkdir`, the Web shell creates two copies of the new directory: one on the public side and one on the private side. The resulting attribute of the directory is `mirror`. If for some reason a parent directory doesn't exist on one side, issuing `mkdir` creates the directory on the side that has the parent directory.

When you create a file, HTML/OS either makes it public or private, depending on the file's extension and depending on HTML/OS' configuration. HTML/OS handles the public/private issue for you, so you generally don't have to worry about managing this aspect of your files, but in those cases when you do, the Web shell provides commands to alter this attribute of a file. Managing file associations is covered in Chapter 3, "The Shell Environment."

Even though the HTML/OS system running underneath the Web shell takes care of putting files in the private area or the public area, sometimes it is necessary to override a file's placement or to apply HTML/OS' file area determination to an existing file. To move files between file areas, use the `fixfile`, `fixpublic`, and `fixprivate` commands. When you use any of these special commands, you don't change the location of files in the Web shell's file system. You change only the location of files on the server, between the public and private side, thereby changing the `public/private` attribute of a file as viewed from within the Web shell.

The fixfile Command

To move a file to its appropriate location, either public or private, use the `fixfile` command. The `fixfile` command takes any number of file arguments and puts the files where they belong according to their extension and HTML/OS' configuration. The following command applied the `fixfile` command to `.gif` files in the current working directory:

```
/>fixfile *.gif
```

The preceding command by default moves all `.gif` files in the working directory to the public area, because HTML/OS considers all image files to be public files (unless you specify otherwise in the HTML/OS control panel). Any file that is in its correct location will go untouched. The `fixfile` command displays a list of the files it moved and a success or failure message next to each file name.

The fixpublic Command

The `fixpublic` command moves files into the public (`htdocs`) area. The `fixpublic` command takes the following form:

```
/>fixpublic FILE
```

In this command, `FILE` can be one or more files. To make all `.gif` files in this directory public, issue the following command:

```
/>fixpublic *.gif
```

This moves all `.gif` files in the working directory into the public file area.

The fixprivate Command

The `fixprivate` command moves files into the private area (`cgi-bin/private`). The `fixprivate` command takes the following form:

```
/>fixprivate FILE
```

In this command, `FILE` can be one or more files. To make all `.html` files in this directory private, issue the following command:

```
/>fixprivate *.html
```

You generally won't have to worry about changing the private/public attributes of a file because HTML/OS takes care of this for you. When there is a problem with a file attribute, it often means that the HTML/OS control panel needs to be tweaked. The next chapter, "The Shell Environment," explains how to access and manage this aspect of the HTML/OS control panel.

Uploading and Downloading Files

Although most file-management work consists of creating, moving, and deleting files on a server, it is often necessary to retrieve a file from the server or add a file to the server from your local machine. Although you could use a third-party tool, such as FTP, to transfer files between your local computer and the Web server, the Web shell provides integrated tools for this kind of file transfer. The tools are easier to use than FTP and require no extra software.

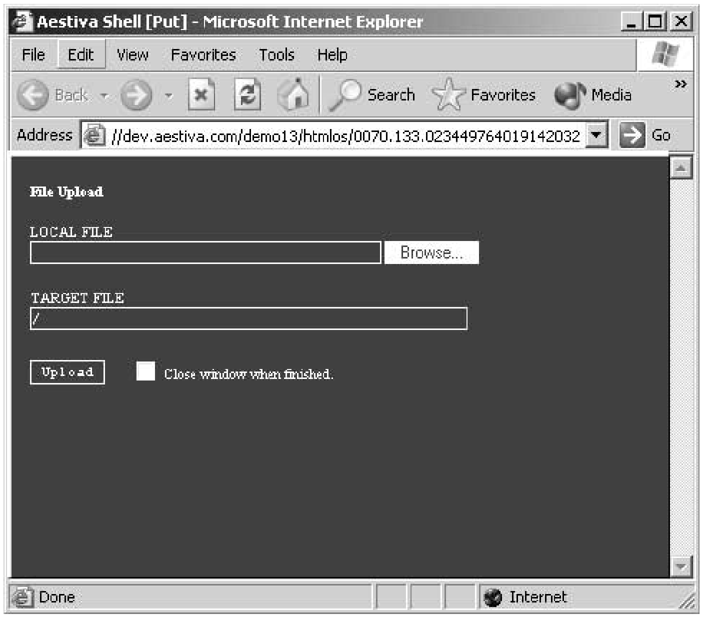

Aestiva Shell [Put] - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address //dev.aestiva.com/demo13/htmlos/0070.133.023449764019142032 Go
File Upload
LOCAL FILE
Browse...
TARGET FILE
/
Upload
Close window when finished.
Done
Internet

Figure 2.2 The put file upload window prompts you for input.

Uploading Files to the Server with the put Command

To upload files to the Web server, use the command `put`. The `put` command launches a small window that prompts you for your local file, as shown in Figure 2.2.

Click the Browse button to browse your local file system for the file you want to upload. After you choose a file, click the Upload button in the upload window. The file you uploaded is saved to the Web server in the working directory with the same name it had on your local computer. Alternatively, you can override this and specify the name manually.

Check the Close window check box when finished if you have only one file to upload. Otherwise, leave it unchecked and upload as many files as necessary before closing the upload window. You might want to run `ls` after an upload just to make sure that the upload worked as you expected.

Downloading Files from the Server with the get Command

To transfer a file from the server hosting the Web shell to your client computer, use the `get` command. The `get` command opens a window with a list of all of the files supplied to it as arguments. Click a filename in the pop-up window to download it to your local computer.

If, for example, you want to download all the `.bb` files in the working directory, you would issue the following command:

```
/>get *.bb
```

If there are matches, a window opens with a list of the matching files. Each file is clickable and invokes a download of the file to your local computer.

Summary

Moving files and directories around is an important part of managing a computer system. The Web shell's file-management tools are modeled closely on those of other shells and have similar advantages arising from their pattern matching capabilities and English-like syntax. The Web shell, however, in addition to providing these basic tools, has Web-based extensions, such as `put` and `get`, which make Web-based file management easy and obviate the need for the third-party tools to which so many shell users resort. It also provides tools to search and view the content of text files. For HTML/OS file attribute management, it provides a foolproof means of moving files between the public and private file areas.

The next chapter, "The Shell Environment," details more of the Web-centric features that distinguish the Web shell from its legacy counterparts.

3

Chapter

The Shell Environment

The Web shell is an ideal tool for developing and managing Web-based applications because it allows you to create and edit files, view documents and images, and run applications using the same interface that application users implement, the Web-enabled browser. This important characteristic distinguishes the Web shell from its legacy counterparts. The Web shell supports standard file-management commands, as do legacy shells; but it does so in a Web-centric environment. This chapter discusses some of the features of the Web shell that are not command specific, but rather relate to the working environment of the Web shell.

The first part of this chapter explains how to read the output area of the Web shell and how to retrieve and run prior commands. The second part explains how to create files with the shell's integrated editor and run them from within the editor. Then, it shows you how to run applications and view images on your server. Finally, it explains how to use the Web shell's customization options to make these environmental features look and behave in a manner that suits your needs.

The Command Prompt

When you first log in to the Web shell, you see a black screen with a text box at the bottom, two buttons to the right of the command line, and a funny symbol, the slash forward arrow (`/>`) at the top of the screen. The symbol at the top of the screen is the command prompt.

In traditional shells, the command prompt prompts you for a command. Next to it the cursor moves along and displays your commands as you type them. The Web shell, because it is Web based, doesn't provide a real-time display of your command in the output area. Instead, you type into a text box at the bottom of the screen. After you submit a command, the command you typed appears next to the command prompt just as you typed it.

In the Web shell, one of the purposes of the command prompt is to tell you what the current working directory is. The area before the `>` character, delimiting the end of the command prompt from your shell commands, contains the value of the current directory when you issued the command. Because the shell output displayed on the screen before the command you most recently issued is historical, only the most recent command prompt on the screen tells you the current value of the working directory. All other instances of the command prompt tell you what the working directory was when you issued the command printed immediately to the right of it.

The Command History

As you work in the Web shell, the commands you type and the results of those commands are displayed to the screen. Depending on how your copy of the Web shell is configured, multiple commands and their output can be displayed at once in the order you submitted them.

If, for example, you type `ls` to get a directory listing and then change to the parent directory and get another directory listing, you should notice several command prompts on the screen, as the following example shows:

```
/apple/banana/cherry/>ls
FILE  PRIVATE              19  04/13/03 23:44:11  pitted_fruits.txt
/apple/banana/cherry/>cd ..
/apple/banana/>ls
DIR   MIRROR              144  04/12/03 11:16:43  cherry
FILE  PRIVATE              12  04/13/03 23:45:31  tropical_fruit.xml
/apple/banana/>
```

Most of the command prompts are old, meaning they relate to the commands right next to them, not the command that you are about to issue. The only command prompt that tells you the current working directory is the one at the bottom. In the previous example, it is `/apple/banana/>`.

Accessing Prior Commands

Because it is so common to have to issue a command that you have already submitted, the Web shell provides a few ways to recall old commands. The old commands available to you for recall are those that you issued through the course of the current Web shell session. (For more information on Web shell sessions, see the note, “Web Shell Sessions,” in this section.) The Web shell forgets all commands that you issued in previous sessions.

There are two ways to recall old commands: by using the up arrow in the text box, and by using the exclamation operator with an argument.

Web Shell Sessions

Web shell sessions begin when you log in to the Web shell and end either when you log out, or after a certain amount of idle time has elapsed. Depending on how your Aestiva copy is configured, sessions become invalid after about 100 minutes of inactivity. When a new session is started, your command history is reset. All other data and configuration information persists.

Viewing the Command History with the history Command

The `history` command displays all the commands that you entered into the Web shell during the current session. When you run it, it displays the command history with the older commands at the top and newer commands at the bottom. In the left column, it displays the command number, which is useful when you use the exclamation operator to run a prior command.

The following is an example of a brief command history. If after you log in, you run `ls`, run `cd /test`, then run the `date` command, the `history` command would produce the following result:

```
/>history
1     ls
2     cd /test
3     date
/>
```

The `history` command produces a list of all of the commands that you issued to the Web shell in the current session. On the left side of its output, it displays the command number that you can use to reference the command as discussed in the following sections.

Using the Exclamation Operator to Recall Commands

To run a prior Web shell command without retyping it, use the exclamation operator. You can give two classes of arguments to the exclamation operator: an absolute command number, or a relative command number.

Absolute Command Numbers

An absolute command number is one that is referenced as displayed by the history command. The first command of the session is associated with the number 1, the second with 2, and so on. To access one of these commands, type the exclamation mark with the command number. For example, to run the second command `cd /test` using the command history in the preceding example, you type `!2`. The Web shell replaces this command with the value of the original command. When you run a command in this manner, you don't see the exclamation operator you just typed in the output area, but rather you see the original command, in this case `cd /test`.

Relative Command Numbers

Sometimes it is easier to reference a command by its relative command number. To use relative command numbers, precede the command number with a minus sign. The number following the minus sign signifies how many commands ago the desired command was executed. For example, to recall the prior command, run the statement `!-1`. This tells the Web shell to run the previous command. To run the command issued two commands ago, enter `!-2`.

Using the Arrow Keys to Recall Commands

The arrow key method to recall old commands is probably the most convenient, because it consolidates browsing for commands and recalling them. Unfortunately, this feature is available only to users of Internet Explorer. (If you use another browser, you'll have to settle for using the exclamation operator to recall old commands.)

If you are an Internet Explorer user, use the arrow keys while the cursor is in the command input box to scroll through your command history. When you press the up arrow, you recall commands that get progressively older; when you press the down arrow, you recall commands that get progressively newer. The oldest command available to you when you use the arrow keys is 20 commands old. If you want to access older commands, get a view onto your command history with the `history` command and recall a command with the exclamation operator.

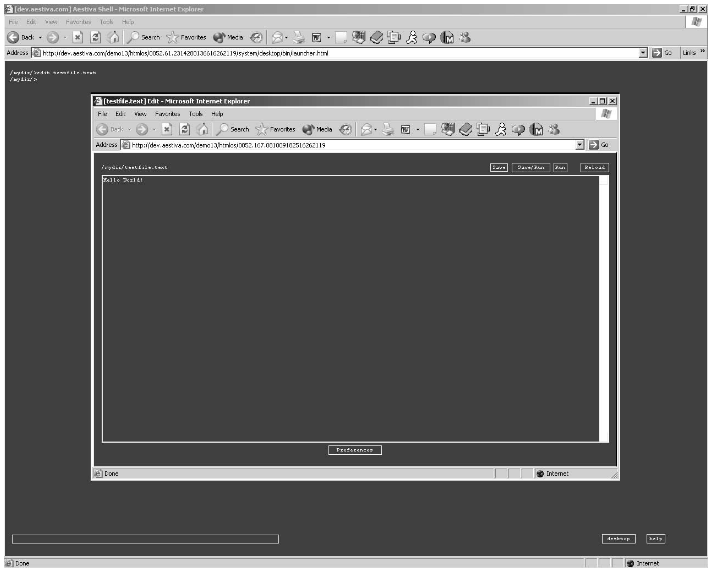
Editing Files

Creating and editing text files are some of the more important tasks most users perform on a Web server. The Web shell provides a simple but powerful tool for editing and creating text files without having to leave the shell environment.

The command you use to edit files is `edit`, followed by the filename you want to create or modify. To test the `edit` command, type the following command in the directory you want to create your test file:

```
/mydir/>edit testfile.txt
```

The Web shell creates a new window with a large text box and a few buttons. This window is the Web shell's text editor, as shown in Figure 3.1. You can use this editor to alter or


[dev.aestiva.com] Aestiva Shell - Microsoft Internet Explorer
http://dev.aestiva.com/demo13/htmlos/0052.61.2314280136616262119/system/desktop/bin/launcher.html
[testfile.text] Edit - Microsoft Internet Explorer
http://dev.aestiva.com/demo13/htmlos/0052.167.081009182516262119
Save
Save/Run
Run
Reload
Preferences
desktop
help

```
/mydir/>edit testfile.text
/mydir/>
```

```
/mydir/testfile.text
Hello World!
```

Figure 3.1 The Web shell editor window.

create any text file in the shell's file system, as well as display the file as a Web browser would see it when you are ready to run the page.

When you edit `testfile.txt`, if the file does not exist, the shell launches a blank editor window. At the top left of the screen, you will notice the full path of the file you supplied as an argument as well as a little note about the state of the file, in this case, `New File`.

The `New File` message tells you that the editor window was launched with a file that doesn't yet exist. This file is created only when you save your work. Until then, the filename you have chosen represents an unrealized plan.

At this point, you may type something into the text area in the editor window. When you are done, click the Save button at the right of the window. Clicking the Save button submits the text in the text box to the server and saves the file under the name and location that you specified when you issued the `edit` command. At that point you may continue to work in the editor window and save the file as necessary. Every time you save, the file is overwritten with the content you have in the text box.

Before you save, your work exists only on your local computer in your browser. You can manipulate it any way you want without causing any network transmissions; but if you fail to save your work and then close the browser, your work will be lost. It's a good idea to save your work often to minimize the possibility of losing your data.

Alternatively, if you make changes to your file and want to revert to the previously saved version, you can click the Reload button in the editor window to populate the editor's text box with the previously saved version of the file.

When working in the Web shell's text editor, you reap the benefits of working in a networked environment but with the interface speed and flexibility of a client-side application. When you are working with files in the editor, you can invoke the GUI features of your browser's text box, including text drag and drop, mouse-based text selection, find, and find/replace. The client-side features of the Web shell's text editor are limited only by what your browser can do. For the majority of text editing and code development, your browser's text editing features are adequate. Though you have these features available to you, you are still saving your files on the remote server, thereby giving you networked access to your file, and freeing you from having to transfer the fizle to or from the server manually.

After you've saved `testfile.txt`, close the editor window and place your cursor in the command-input box in the Web shell. When you issue the command `ls` this time, you should see the results of your save:

```
/mydir/>ls
FILE  PRIVATE                12  04/14/03 01:12:47  testfile.txt
```

You have just created a file and saved it to the Web shell's file system. Now when you edit `testfile.txt`, the editor window shows the file's contents. You can try to edit this file as follows to see what happens:

```
/mydir/>edit testfile.txt
```

The editor window that opens contains the contents of `testfile.txt`. From now on, you will be editing this existing file, and any changes you make via the Web shell editor will be effective and permanent. Be careful when working on important files.

Running Files from within the Editor

Typically, you use the shell's editor to create and edit Web-related, text-based files. Web files are those that are run from a Web server and are displayed by a Web browser. Depending on your environment and the type of the Web file, your Web server might present the file to the client in a raw form, or it might recognize commands in the Web file to run and upload the output to the Web client. The details of how servers and clients handle different types of files are largely unimportant to you. What matters is the result when a browser requests a file from your server. That is why the Web shell runs files just as the Web server and browser will process them.

The Web shell provides a simple and effective tool with which to run your Web-based files. It opens a new browser window with the URL of the file you want to run. The server and browser take care of the rest. For simple, free-standing HTML or scripting files, this process works fine. For more complex files that depend on form variables, you can launch the entry page to your application with this process and leave it open while you do your development work.

For now, you create a simple text file with the editor and run it from within the Web shell. If you don't have a Web shell session available to you, log into one now:

1. If you don't have a test directory, create one with the following command, and move into it:

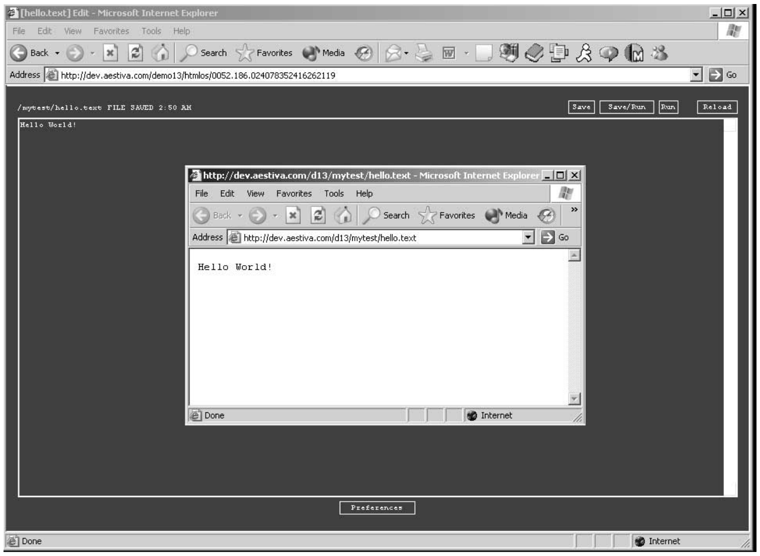
```
/>mkdir mytest
/>cd mytest
/mytest/>
```

2. Next, launch the editor and create your test file with the following command:

```
/mytest/>edit hello.text
```

3. You can name the test file whatever you want; but make sure that your file has the `.text` extension, because a different extension could cause the server and browser to behave in unexpected ways.

   The example file used in this demonstration, `hello.text`, is a simple text file. The `.text` extension will likely be unknown to the server and browser and will therefore be treated as a simple text file. Most default configurations don't perform processing on text files as they do on files created in a specific scripting language, so this will yield a simple example.
4. When the editor window opens, type anything you like in the text area, although a simple statement like `Hello World!` will do. When you are finished, click the Save button to save your file to the server.
5. Then click the Run button at the top right of the screen; a new browser window launches with appropriate the URL filled in, and the contents of your file in the browser window as in Figure 3.2. Read the note "Editor URLs" for more information on the value of this URL.


[hello.text] Edit - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address
Go
Save
Save/Run
Run
Reload
Preferences
Done
Internet
http://dev.aestiva.com/d13/mytest/hello.text - Microsoft Internet Explorer
Done
Internet

http://dev.aestiva.com/demo13/htmlos/0052.186.024078352416262119

```
/mytest/hello.text FILE SAVED 2:50 AM
Hello World!
```

http://dev.aestiva.com/d13/mytest/hello.text

```
Hello World!
```

Figure 3.2 Running a file from within the Web shell editor opens a new window.

6. The contents of your browser window should now display the text you entered into your `.text` file. At this point, if your editor window is still open, edit the content of the file, and click Save to commit your changes to the server. Then go back to the browser window launched by the Run command and reload that page by clicking the Refresh or Reload button, or by clicking the URL and pressing Enter.

The changes you made in the shell editor should be reflected in the newly refreshed page in your browser when the page finishes loading.

Editor URLs

You might notice the value of the URL when you reload the page. The path to the file in the URL should be the same as that of the file in the Web shell file system. This is because the Web shell's file system has the same paths as your Web server. Underneath, at the operating system level, the paths to your Web files also contain the values of the paths to your document root. On the Web, as with the Web shell, file paths are relative to the path to the document root of your server.

Editing an HTML File

In this section, you create and run a very simple HTML file. If you don't know any HTML, that's okay because this section uses only the most rudimentary HTML.

1. In your test directory, create a new file with an `.html` extension.

```
/mytest/>edit hello.html
```

2. Because the browser will expect this to be an HTML file, create a simple HTML document. If you don't know HTML, you can try this:

```
<html>
Hello World!
</html>
```

3. Save and run this file by clicking the Save and Run buttons in the editor. If you followed the example in the previous section, the result in your browser should look somewhat familiar, reading simply `Hello World!` with no special formatting. This time though, the URL in your `hello.html` window should be different from that of `hello.text`. It should contain an auto-generated URL that doesn't explicitly reference `hello.txt`. This encoded URL is the result of the Aestiva engine running the page and looking for Aestiva code to run before it gets sent to the browser.

Creating and running HTML and other text-based files comprises the majority of your development tasks and has been made easy with the Web shell. Because the Web shell creates a new window for each editor session, you can have multiple editor windows open. After you become accustomed to using this method of Web development, you'll likely find it convenient. A great deal of Web development has been accomplished using the simplicity and versatility of the Web shell's editor. For many users, the Web shell's editor is the primary motivation for using the Web shell.

Running Applications

Now that you know the basics of creating and running a Web-based file using the Web shell's editor, you can run a file from the command line. In this section, you run the file that you created in the previous section from outside the editor. Later, you learn how to set up your environment to run general dynamic files.

Running hello.html from the Command Line

Close your editor windows and if you need to, change to the directory where you saved the file you created by typing the following command:

```
/mytest/>run hello.html
```

The Web shell opens a new browser window displaying the file `hello.html`. You might not consider this to be running a file because you are viewing it in your browser, but HTML/OS makes the assumption that your text-based Web files (as opposed to, for example, images and PDFs) are either script-based files, or are components in a Web application or Web site. In the Web shell, the `run` command encompasses all text-based files that you might want to view in your Web browser.

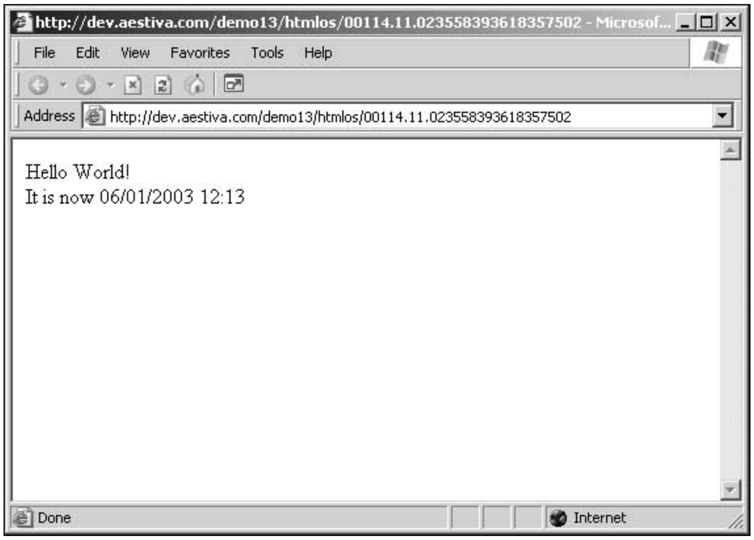
Be careful when you use the `run` command, because many files, and especially many of the dynamic `.html` files that ship with Aestiva, won't work when you try to run them by themselves. Typically, in a Web application, there are only a few points of entry to an application. If you know what these entry points are, you can run those files to make the application work.

If, for example, you were creating a shopping cart application, running the cart page directly wouldn't work as well as running through the process of adding an item to your cart and then viewing it. In this case, if you were editing the cart page, you would have to have the process leading up to the cart page in place while you did your testing. Running the cart page as a free-standing page wouldn't produce the desired results. In these cases, use the Run button in the shell editor or issue the `run` command on the command line on the start page of the application and keep that window open for testing as you work.

Running Dynamic Pages

In the Web shell environment, when you run an HTML file, the Web shell runs it through the HTML/OS engine. Later versions of the Web shell might have the ability to run HTML files normally, but for now, if you want to launch an HTML file from the Web shell, it has to run through the HTML/OS engine. This means that you can add HTML/OS code to your file and it will render properly without configuration headaches. For example, alter `hello.html` to look like the following:

```
<html>
Hello World!<br>
It is now <<now>>
</html>
```


http://dev.aestiva.com/demo13/htmlos/00114.11.023558393618357502 - Microsof...
File Edit View Favorites Tools Help
Address http://dev.aestiva.com/demo13/htmlos/00114.11.023558393618357502
Done
Internet

Hello World!
It is now 06/01/2003 12:13

Figure 3.3 When HTML/OS code is added to the `hello.html` file, it is parsed and displayed automatically.

The newly added line contains the text, `It is now`, followed by an embedded HTML/OS statement, `now`, contained within double angle brackets. The statement within the double angle brackets is HTML/OS code that gets expanded by the Aestiva HTML/OS engine. If you want to write HMTL/OS code, use the double angle brackets to encapsulate HTML/OS code and distinguish it from HTML. When you run the preceding HTML file, you should get a result like that in Figure 3.3.

If you are familiar with HTML/OS, you can create HTML pages and add HTML/OS code as you need to.

On the other hand, if you want to run files in another scripting language, such as PHP, you will have to do two things:

1. Give your files an extension other than `.html` or `.htm`. The extensions `.php` or `.phtml` are good choices for PHP files.
2. Set up your Web server to be able to parse your scripting files when they are run on the public side (outside of the `cgi-bin` or other script directories).

After you do this, the Web shell treats your scripting files as public text documents and won't try to run them through the HTML/OS engine, and your Web server treats them as dynamic documents because of their file extension.

Viewing Images

To view an image on your server from within the Web shell, issue the `view` command with the images you want to view as file arguments, as in the following command:

```
/images/>view img1.gif
```

As with the `edit` command, the `view` command opens a new window with the file presented as the Web site would display it to a common client. In this case, the image you specify is embedded into a page in a new Web browser window. This window displays the image as well as path to the image on your server. Image paths are displayed, in part, because the `view` command is capable of displaying multiple images in a folder at one time. If, for example, you want to view all the `.gif` images in your current working directory, you could issue the following command:

```
/images/>view *.gif
```

The previous command launches a new browser window containing all of the `.gif` images in the `/images` directory. The file argument in this example, `*.gif`, matches all files in the working directory that end with `.gif`. (For more information on using the asterisk character at the command line, refer to Chapter 2, "File Management.") You can try this now on a system directory in the Aestiva environment, `/apps/bundlebee/`, by issuing the following commands:

```
/>cd /apps/bundlebee
/apps/bundlebee/>view *.gif
```

After you run these commands, the `view` command creates a new window that displays all the `.gif` files in this directory. Above each image is the path to the image in the Web shell's file system. The `view` command, because it runs in a browser, supports any image file type that your browser supports. Typically, this means files that end in `.gif`, `.jpg`, and `.bmp`. You can view any combination of image files in a view window.

Setting Shell Preferences

Now that you know how to edit files and view images, you will learn how to configure the Web shell to your liking by editing the shell's preferences.

The command that edits the shell's preferences is called `set`. When you run `set` by itself, it displays the current preference values:

```
/>set
autoscroll          on
backgroundcolor     #000000
fontface            courier new
fontsize            10
inputcolor          #6699cc
inputsize           80
outputcolor         #999999
runheight           500
runwidth            900
scrollbuffer        40
showbuttons         on
workingdircolor     #cfcf99
/>
```

When you edit one of these values, it has an immediate and lasting effect. The next time you or someone else logs in to your copy of the Web shell, the preferences as saved in the previous session are recalled.

To set a shell environment value, run the `set` command with the name and its new value as arguments, as in the following example, which changes the background color of the Web shell:

```
/>set backgroundcolor #000033
```

Then issue the `set` command again to display the current environment values in the Web shell. The results should look similar to the following:

```
/>set
autoscroll          on
backgroundcolor     #000033
fontface            courier new
fontsize            10
inputcolor          #6699cc
inputsize           80
outputcolor         #999999
runheight           500
runwidth            900
scrollbuffer        40
showbuttons         on
workingdircolor     #cfcf99
/>
```

The command you issued to change the background color of the shell had an immediate effect. It also changed the values of Web shell's preferences. Unless you are fond of the blue background, run the following command to revert to a black background color:

```
/>set backgroundcolor #000000
```

Generally, the `set` command, when not run alone, requires a parameter and a value after the `set` command, in the following form:

```
/>set [parameter] [value]
```

The parameter can be any of the setting parameters displayed by running the `set` command, such as `backgroundcolor` in the previous example. The following are the environment parameters that you can manipulate with the `set` command:

- The `autoscroll` parameter tells the shell whether to scroll the window to the bottom when the output of shell commands goes off the page. By default this parameter is set to off, because it makes some Macintosh browsers misbehave. Because it is easier to work with this parameter set to on, turn it on when you get a chance. If you are one of the unlucky few whose browsers fail, turn it back off, although you'll have to do this blindly, because your browser won't display anything until it is turned off.
- The `backgroundcolor` parameter sets the background color for all Web shell windows, including the image viewer, the editor, and the main window. Alter this parameter with caution, because while setting it you could make text invisible and have to fix the problem in the dark. Browsers have a limited color vocabulary that includes words like `red`, `orange`, `lightgreen`, and so on. If you want more control, you can use an HTML color value in the form of `#xxxxxx`, where `x` is a hexadecimal character (between 0–9 or A–F). The first two characters correspond to red, the second two to green, and the third to blue. All zeroes signify black and all F's signifies white.
- The `fontface` parameter accepts any value that your browser will accept. Some examples are `courier`, `verdana`, `sans`, `serif`, and `times`. If you want to set the font face to a value that contains spaces, use quotes around the font name, as shown in the following example. (Issuing command arguments with spaces is covered in the next chapter.) If you choose a font face that your browser doesn't recognize, the browser chooses a font for you.

  ```
  />set fontface "courier new"
  ```
- The `fontsize` parameter sets the font size, measured in pixels, throughout the Web shell. Set it to a value above 8 to avoid painfully miniscule fonts. If you want a slightly larger font than the one the shell ships with, run this command:

  ```
  />set fontsize 12
  ```

  This command bumps the font size up to 12 pixels and should be easier on your eyes than the default 10 pixels. Keep in mind though that less data will fit on your screen and in the editor window as you increase the font size. Also, note that when you change the font size, you alter the size of the Web shell editor's main text area proportionately. You will probably have to edit the editor's preferences in this case to make it fit in the browser window.

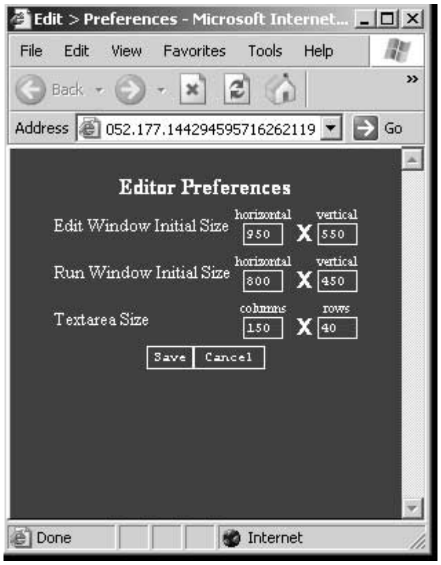
- The `inputcolor` parameter sets the color of text that you type and buttons that appear throughout the Web shell, including command-line and pseudo command-line text, as well the text in the editor text area. Be careful when setting this color to maintain good contrast against the background color, because your text might disappear altogether, which would give you trouble when trying to revert.
- The `inputsize` parameter sets the size of the command-line input box at the bottom of the screen in character units. Larger input sizes show you more of what you type if you are issuing long or multiple commands, and smaller input sizes accommodate smaller shell windows better.
- The `outputcolor` parameter sets the color of any text generated by the Web shell as the result of a command. The difference in color makes it easier to distinguish your own commands from the command prompt and from the system output.
- The `runheight` parameter tells the Web shell how high to make pop-up windows when you issue a `run` command. You specify this value in pixels. (Keep in mind that your screen height is on the order of 1,000 pixels.)
- The `runwidth` parameter tells the Web shell how wide to make pop-up windows when you issue a `run` command.
- The `scrollbuffer` parameter tells the Web shell how much content to keep on one screen before letting it drop off and be displaced by new content. A `scrollbuffer` value of 40 allows up to 40 lines of output to be displayed on the screen, consisting of any number of commands and their output, before it stops displaying old output after subsequent commands are run. A large value of the `scrollbuffer` keeps a lot of content on the screen. You might find this convenient if you need easy access to the output of commands you issued several commands ago; but a large `scrollbuffer` means that your browser has to download that much more information every time you issue a command. Don’t forget to turn on the `autoscroll` parameter if you choose a `scrollbuffer` value that is larger than your shell window.
- The `workingdircolor` is the color of the command prompt in the output area of the Web shell. You can set it to any valid HTML color value, as with the other color parameters.

Setting Editor Preferences

The Web shell editor has its own preference editor. Because it is GUI based, and not command-line based, the preferences menu is a GUI as well. To access editor preferences, open an instance of the editor window by issuing any `edit` command.

```
/>edit foo
```

Click the Preferences button at the bottom of the editor window, shown in Figure 3.4. A new, smaller window opens to let you edit the attributes of the editor environment, namely the size of the editor window, the size of the run window, and the size of the editor's text area.


Edit > Preferences - Microsoft Internet...
File Edit View Favorites Tools Help
Back
Address 052.177.144294595716262119
Go
Save
Cancel
Done
Internet

Editor Preferences

Edit Window Initial Size horizontal 950 X vertical 550

Run Window Initial Size horizontal 800 X vertical 450

Textarea Size columns 150 X rows 40

Figure 3.4 The editor preferences window changes how the editor behaves.

The editor preferences window enables you to alter three pairs of values: the initial size of the edit window, the initial size of the run window, and the size of the text area in the editor window.

- The edit window size determines the size of the main edit window when it pops up after you issue an `edit` command. Set this value to a large size that still fits on your screen to be able to display the greatest amount of text data.
- The initial size of the run window determines the size of the window that launches when you click the Run button in the editor. It is probably best to set

this to a value that is smaller than the edit window's initial size so as not to cover and hide the editor window when running an application.

- The text area size is the width and height of the editor text area in character units. Set these values as high as they can go while keeping the text area within the boundaries of the edit window.

Click the Save button when you want to save any changes you might have made. Clicking Cancel closes the window without saving your changes.

When you make a change to the editor preferences, it won't be reflected until you reload the editor window. To reload the window, click the Save button if you have text that you want to save, or click the Reload button if you want to reload an existing file.

Mouse Features

Although one of the most important benefits of the Web shell is its ability to accomodate users interested in keeping their fingers on the keyboard, the Web shell also provides a means of navigating the file system and performing basic tasks using a mouse. Even though it doesn't look like it, the output of the `ls` command and the command line are actually clickable. When you click a clickable element in the Web shell environment, it issues a Web shell command that appears in the command history as if you had typed it manually.

Move into a directory that has multiple files and subdirectories with the command `cd`, and then issue the command `ls`. The following code illustrates the results of issuing this command:

```
/mydir/>cd /apps/shell
/apps/shell/>ls
DIR   MIRROR            72  04/07/03 13:05:27  bin
DIR   MIRROR            80  04/07/03 13:05:27  conf
DIR   MIRROR            48  12/17/01 16:15:23  etc
DIR   MIRROR            48  01/18/02 15:42:42  help
FILE  PUBLIC          2148  04/07/03 13:05:27  app_icon_shell.gif
FILE  PRIVATE           68  04/07/03 13:05:27  app_install.txt
FILE  PRIVATE        30757  04/07/03 13:05:27  fcns.lib
FILE  PRIVATE          613  04/07/03 13:05:27  index.html
FILE  PRIVATE         3191  04/07/03 13:05:27  inputframe.html
FILE  PRIVATE            3136  04/07/03 13:05:27  outputframe.html
FILE  PRIVATE             499  04/07/03 13:05:27  parent.html
FILE  PRIVATE            2148  04/07/03 13:05:27  shell.gif
```

When you move the mouse pointer across the filenames, the directories and files listed by `ls` are clickable. When you click a directory in the Web shell, you are moved into that directory as if you had typed the command `cd`. Try it on the `bin` directory. After you click `bin`, the command appears on the pseudo command line:

```
/apps/shell/>cd /apps/shell/bin
/apps/shell/bin>
```

When you click a file, the shell runs a particular command with that file as its argument. The Web shell determines which command to run against what file types by comparing the file's extension with values in a customizable file.

The file that tells the Web shell what to do when you click a filename generated by the `ls` command is called `associations.conf`, and it lives in the `/apps/shell/conf` directory. This file consists of two columns: The left column lists file extensions, and the right column lists shell commands. To associate a command with an extension, edit the right-hand column value, or add it to the list if it's not already present. When you click a file, the Web shell runs the command you supply in the right-hand column with the filename you click as its argument. If for example, you want to associate the `cat` command with all files that end in `txt`, edit `associations.conf` so that the line with `txt` in the left column has `cat` in the right column. After you do this, clicking a `.txt` file causes the `cat` command to run with the file you clicked as its argument, as if you had issued the command manually.

Summary

The shell environment is unique in that it allows you to run and edit Web-based files in a command-line environment, but with the flexibility and ease of use more commonly found in GUI-based environments. Although the Web shell contains the features that make it an integrated development environment for the Web, it also maintains its command-line heritage by supporting the standard commands covered in Chapter 2, as well as some of the more arcane but powerful command line features, such as data redirection, a concept covered in the next chapter.

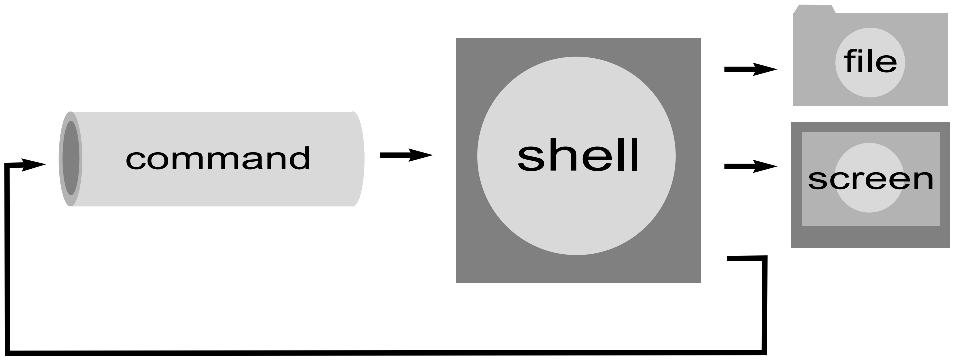
4

Chapter

Redirection

Web shell commands in general read data from an input channel, perform tasks, and print data to the screen. This is true for all commands even though some commands don't appear to require any input, and other commands don't appear to produce any output.

If you were to draw a picture representing a command, you might draw a box with a tube representing data in, and a tube representing data out. You pass data to a command from the command line through the use of arguments; the command reads those arguments, performs its tasks, and displays any results on the screen.



By default, a command gets its input from the arguments you send to it and sends its output to the screen. In the Web shell, you can divert — or redirect — data from the output stream of a command into a file or into another command. When you divert data into a file, it is called "redirection." When you divert data into another command, it is called "piping." This chapter covers these two basic concepts, as well as two common commands for which you would use data piping, the `sort` command and the `grep` command.

Introducing Redirection

Redirection is the diverting of data from the output stream into a file. When you redirect data into a file, the data that would otherwise be displayed to the screen is written to a file. This is useful when you want to archive the output of a command for future reference.

The Web shell support two kinds of redirection, one that uses the `>` operator and creates new files or overwrites existing ones, and another that uses the `>>` operator and appends to files.

Redirection with the > Operator

When you want to write the output of a command to a file, use the `>` operator. The `>` operator redirects the data that would otherwise be displayed on the screen and writes it to the file you specified after the `>` character. If, for example, you want to record the names of the files in the working directory, you issue the following command:

```
/>ls > filelist.txt
```

This command runs the `ls` command, but writes its output to a file called `filelist.txt`. Nothing is displayed on the screen. Instead, the results of the `ls` command are written to `filelist.txt`. If `filelist.txt` already exists, the Web shell overwrites it. Be careful when redirecting output, because the Web shell deletes and replaces your data if you redirect output to an existing file.

To see the contents of the `ls` command, you have to open the file and view its contents. Use the `cat` command to verify the results:

```
/>cat filelist.txt
DIR   MIRROR             984  04/01/03 08:11:16  apps
DIR   MIRROR             128  04/17/02 15:11:13  backup
DIR   MIRROR              48  10/16/01 10:03:19  docs
DIR   MIRROR              48  04/22/03 00:04:11  mydir
DIR   MIRROR              80  04/15/03 02:50:54  mytest
DIR   MIRROR             904  04/25/03 09:18:44  system
DIR   MIRROR             952  04/14/03 22:47:02  test
DIR   MIRROR              80  05/16/02 15:49:50  upload
FILE  PRIVATE             10  04/25/03 09:18:43  .htaccess
FILE  PRIVATE           3208  03/27/01 08:27:45  login.html
FILE  PRIVATE                36  04/14/03 01:10:36  newfile.txt
FILE  PRIVATE              1571  04/25/03 09:18:44  restart.html
FILE  PRIVATE                85  04/25/03 16:21:51  test.html
FILE  PRIVATE                49  03/12/03 17:00:32  unscramble.html
```

Here you see the results of the `ls` command, not as the output of a direct call to `ls`, but by reading the contents of a file created with redirection from the `ls` command.

It might help to know that the `>` operator is not a command attribute. It is not like a switch or an argument to a command — something handled by the command itself — and possibly subject to the interpretation of the given command. The `>` operator works independently of the command as a part of the Web shell environment. Therefore, the `>` operator behaves in exactly the same way for all commands and can be used for all commands. You could, for illustration, redirect the output of `cd` into a file, as in the following example:

```
/>cd mydir > empty.txt
```

The `cd` command would change the working directory; but because `cd` produces no output, the Web shell creates an empty file with the name provided.

More typically, you might want to save the output of an expensive command (one that takes a while to run, or one that produces a great deal of output) to a file. If, for example, you want to keep a record of all `.html` files in your file system, you could issue the following command:

```
/>find "*.html" > myhtmldocs.txt
```

This would generate a list of paths found by the `find` command and save them in a file for future reference. You wouldn't have to run the `find` command again to retrieve your list of `.html` files, and you could keep a log of your file system for future use.

The >> Operator

The `>>` operator works exactly like the `>` operator; but it appends to an existing file, instead of creating a new one or overwriting an existing one. You probably won't need to use this functionality very much, if ever, but it offers a convenient shortcut to manual file manipulations if you need it.

In the preceding example, if you wanted to put a date stamp on the `myhtmldocs.txt` document, you could use the `>>` operator to append the date at the bottom of the file:

```
/>date >> myhtmldocs.txt
```

The file `myhtmldocs.txt` might look like this after appending the date:

```
/system/desktop/bin/new_folder.html
/system/desktop/bin/library.html
/system/desktop/bin/add_font.html
/system/desktop/bin/custom.html
/system/desktop/bin/colortable.html
/system/desktop/bin/new_icon.html
/system/desktop/bin/file_select.html
/system/desktop/bin/start_link.html
/system/desktop/bin/overwrite.html
/system/desktop/bin/settings.html
/system/desktop/bin/function.html
/system/desktop/bin/launcher2.html
/system/desktop/bin/advanced.html
/system/desktop/bin/close.html
/system/desktop/bin/upload.html
/system/desktop/bin/icon_select.html
/system/desktop/index.html
04/28/2003 21:28
```

More likely, you would want to put the date at the top of the file, in which case you would reverse the order of the commands:

```
/>date > myhtmldocs.txt
/>find "*html" >> myhtmldocs.txt
```

You can see the results of the preceding commands by running `cat` against the file:

```
/>cat myhtmldocs.txt
04/28/2003 21:28
/system/desktop/bin/new_folder.html
/system/desktop/bin/library.html
/system/desktop/bin/add_font.html
/system/desktop/bin/custom.html
/system/desktop/bin/colortable.html
```

```
/system/desktop/bin/new_icon.html
/system/desktop/bin/file_select.html
/system/desktop/bin/start_link.html
/system/desktop/bin/overwrite.html
/system/desktop/bin/settings.html
/system/desktop/bin/function.html
/system/desktop/bin/launcher2.html
/system/desktop/bin/advanced.html
/system/desktop/bin/close.html
/system/desktop/bin/upload.html
/system/desktop/bin/icon_select.html
/system/desktop/index.html
```

Because in this example `myhtmldocs.txt` was created with the date command, the date appears at the top of the file. The results of the `find` command were appended to produce the desired result.

The usefulness of the `>>` operator is limited when you consider the stock commands supplied by the Web shell. It is not until you create your own Web shell commands (a topic covered in Chapter 6, "Creating Custom Commands,") that the `>>` operator becomes particularly useful. If, for example, you created your own Web shell command that analyzes a text-based server log file and displays the results to the screen, you could run it periodically and append the results to a log file when you needed to. This way, you could keep a history of your server activity but you would have the flexibility to run the custom command without necessarily writing to a file.

Piping Data into Commands with the | Operator

Piping is a lot like file redirection in that it redirects the output of a command. Piping is different, though, because instead of redirecting output to a file, it redirects output to another command. You use piping to make the output of one command the input of another command in a compound command statement. The result is that you don't see the intermediate data stream but rather the final output after all the commands have processed their data.

This section explains how to use piping by using some commands that are typically used to process piped data. These command are explained here in subsequent sections that digress slightly from the topic of redirection, but which will help you use piping with these commands to process data at the command line.

The way you redirect output to a command is with the pipe operator (`|`). In general you put the pipe operator between commands to redirect data from one command to the other. The general form of the pipe operator is the following:

```
/>command1 | command2
```

The preceding statement tells the Web shell to run `command1`, redirect the output of `command1` to the input of `command2`, and then to run `command2`. In this example, `command1` has to produce some output, and `command2` has to require input and must be able to read the format of `command1`. If this is the case, `command2` will display its output to the screen.

Because in this example, `command2` produces output, you can pipe this output into yet another command, as in the following example:

```
/>command1 | command2 | command3
```

The preceding example runs `command1`, pipes its output into `command2`, which runs and pipes its output into `command3`, which runs and displays its output to the screen.

Before demonstrating how piping can be useful in the Web shell, a few commands that are commonly utilized when using piping have to be covered. These commands are the `sort` command, the `grep` command, and the `wc` command. The following sections explain how to use these commands in general, and then explain how to use them with piping.

The Sort Command

The `sort` command sorts the lines of a text document and returns the result. For example, suppose you had a file called `ages.txt` on your system with the following content as viewed by the `cat` command (create the file with the `edit` command to test this example):

```
/>cat ages.txt
John 28
Alice 26
David 31
```

You could view this file in sorted order by issuing the `sort` command with the filename as its argument, as in the following example:

```
/>sort ages.txt
Alice 26
David 31
John 28
```

By default the `sort` command sorts in ascending alphabetical order on the first column, where columns are delimited by any number of spaces. If you want to sort a text document on a column other than the first column, you could specify the column number with the `-k` switch. Put the column number after the `-k` switch to achieve this. If, for example, you want to sort `ages.txt` on the second column, you would use the `-k` switch; but you would include the `-n` switch to indicate that the second column is a numerical column, as in the following example:

```
/>sort -nk 2 ages.txt
David 31
John 28
Alice 26
```

By default, numerical sorts are performed in descending order, so to sort `ages.txt` in ascending order on the age column, add the `-r` switch to reverse the sort order, as in the following example:

```
/>sort -rnk 2 ages.txt
Alice 26
John 28
David 31
```

The `-k` switch tells the `sort` command that the argument that immediately follows it is the column number on which to sort.

If you try the preceding command without the `-n` switch, you might notice that it doesn't work correctly. That's because `sort` sorts alphabetically by default, and you want to sort on a numeric column. An alphabetical sort treats words, even "words" that contain only numbers, such that smaller values have higher precedence regardless of their size. For example, alphabetically, *aab* has a lower precedence than *ab*; in a similar manner, 112 has a lower precedence than 12. To fix this, use the `-n` switch to sort numerically.

Additionally, if you try the preceding command and put the `-k` switch somewhere other than at the end of the list of switches, the command fails. The `-k` switch is a special switch that requires an argument and must therefore appear immediately before that argument.

Now that you know how the `sort` command works, you can use it to demonstrate data piping. In the preceding examples, the data on which the `sort` command was operating was the content of the file whose name was supplied as an argument. If you wanted to act on data that was not in a file, like, for example, the output of another command, you would pipe that data into the `sort` command and omit any filename argument.

For example, to sort the output of the `ls` command on file size, you would issue the following command:

```
/>ls | sort -rnk 3
FILE  PRIVATE            10  04/25/03 09:18:43  .htaccess
FILE  PRIVATE            36  04/14/03 01:10:36  newfile.txt
DIR   MIRROR             48  10/16/01 10:03:19  docs
DIR   MIRROR             48  04/22/03 00:04:11  mydir
DIR   MIRROR             48  10/16/01 10:03:19  docs
DIR   MIRROR             48  04/22/03 00:04:11  mydir
FILE  PRIVATE            49  03/12/03 17:00:32  unscramble.html
DIR   MIRROR             80  04/15/03 02:50:54  mytest
DIR   MIRROR             80  05/16/02 15:49:50  upload
FILE  PRIVATE            85  04/25/03 16:21:51  test.html
DIR   MIRROR            128  04/17/02 15:11:13  backup
DIR   MIRROR            904  04/25/03 09:18:44  system
DIR   MIRROR            952  04/14/03 22:47:02  test
DIR   MIRROR            984  04/01/03 08:11:16  apps
FILE  PRIVATE          1571  04/25/03 09:18:44  restart.html
FILE  PRIVATE          3208  03/27/01 08:27:45  login.html
```

There is a lot going on in the preceding command. First, the `ls` command is run, and its output is piped into the `sort` command. The `sort` command processes this data and sorts it numerically (as specified by the `-n` switch), on the third column (as specified by `-k 3`) in reverse/ascending order (as specified by the `-r` switch). Because the `ls` command displays file size data in the third column (for details on how the `sort` command distinguished columns see the note, "How the `sort` Command Recognizes Columns"), this command effectively sorts files and directories by file size.

How the `sort` *Command Recognizes Columns*

When you use the `sort` *command, you sort data on one of its columns. The* `sort` *command delimits columns by spaces. Any number of spaces constitutes a column delimiter. Columns, therefore, are the text areas between clusters of spaces in a given file or data stream.*

The wc Command

The wc command counts the number of words and lines in a document. If you give the wc command a file argument, it counts the number of words and lines in that file. If, for example, you had a server log that contained one line for each page request, you could count the number of requests by running the wc command to determine the number of lines in the file. For example, to count the number of lines and words in mylog.txt, you issue the following command:

```
/>wc mylog.txt
words: 4011
lines: 313
```

The wc command in the preceding example found 4,011 word and 313 lines in the file mylog.txt.

To count the number of files returned by the ls command, you could pipe the results of the ls command into wc and look at the line count. Because ls prints a line for each file and directory, the number of lines ls returns is the number of files it finds, as shown in the following command, which tells us that there are 39 .gif files in the working directory:

```
/images/>ls *.gif | wc
words: 142
lines: 39
```

The grep Command

The grep command is the command that is most commonly used in conjunction with piping. The grep command searches its input for lines containing a match to a pattern you supply as an argument. By default, grep prints the lines that contain a match of the given pattern. You use grep by giving it a file argument, such as an explicit filename like "aztec.html", or a file glob, such as a*, and a pattern consisting of a clusters of letters and wildcards (asterisks), and it finds lines containing the pattern you specified in the files you specified. Alternatively, you can pipe data into grep, and it finds lines in the piped data stream that match the pattern you supplied as an argument. If, for example, you wanted to print only the directories in the working directory, you would issue the following command:

```
/>ls | grep dir
DIR   MIRROR                    984   04/01/03 08:11:16   apps
DIR   MIRROR                    128   04/17/02 15:11:13   backup
DIR   MIRROR                 48  05/01/02 13:46:06  debug
DIR   MIRROR                 48  10/16/01 10:03:19  docs
DIR   MIRROR                 48  04/22/03 00:04:11  mydir
DIR   MIRROR                 80  04/15/03 02:50:54  mytest
DIR   MIRROR                744  09/12/02 15:47:56  pak
DIR   MIRROR                904  04/25/03 09:18:44  system
DIR   MIRROR                952  04/14/03 22:47:02  test
DIR   MIRROR                 80  05/16/02 15:49:50  upload
```

In this example, the `ls` command first generates data relating to files in the working directory. This data is then piped into `grep`, which throws out lines not containing the word `dir` and returns the rest. It also means that, in this case, a file with `dir` in its filename will remain in the list. The result printed to the screen is a list of directories in the current working directory.

Multiple Pipes

There is no artificial limit to the number of pipes you can use in a single command. If, for example, you want to view only the directories in the current working directory and sort the output by file size in ascending order, you could issue the following command, using two pipes.

```
/>ls | grep dir | sort -rnk 3
```

This compound command generates a list of files in the working directory with the command `ls`, pipes those results into `grep`, which strips out all lines without the value `dir`, then pipes those results into the `sort` command, which sorts the results in numerically reverse order on the third column.

If you want to save the results, you redirect to a file at the end of the statement as follows:

```
/>ls | grep dir | sort -rnk 3 > mydirs.txt
```

Summary

Using piping and redirection is a powerful way of managing data at the command line. The piping and redirection operators allow you to preserve data that you generate at the command line, and to channel data dynamically from one command to another without having to use intermediate files.

5

Chapter

Useful Web Shell Commands

In this chapter, you create a simple Web page from the ground up and use some of the commands that distinguish the Web shell from a traditional shell. These commands are covered during the course of a tutorial consisting of creating a Web page, retrieving image files from a remote source, and packaging the page and its related files into a single installation file for deployment on any other Web shell system. The Web shell, being a development and deployment environment, makes these aspects of Web development simple.

This chapter begins by walking you through the creation of a Web page modeled on an existing Web page at `www.aestiva.com`, which lists the operating systems on which you can run the Web shell. After this Web page is completed, you learn how to add image files to your Web server across the network using the Web shell's file transfer commands and then add these images to your Web page. When the page is completed, you package all of these files into a single archive file that can be used to redeploy the Web page and all of its images.

Creating a Simple Web Page that Lists Supported Platforms

In this chapter, you create a Web page modeled on an existing Web page on the Aestiva Web site. The Web page will list all of the platforms on which the Web shell can be deployed. There are three steps to completing this Web page. The first step is to create the HTML document. The second step is to retrieve the image files from the Aestiva server and put them on your Web server. The last step is to add the image links to the HTML page and verify your work. After the page is completed, you pack the files you created into an installation file for deployment on other servers that use the Web shell's application packing tool.

Creating the HTML Document

First, you will need to create a directory where you will add your files. Call this directory `platforms` to indicate that the document will relate to the platforms (operating systems) on which the Web shell can run.

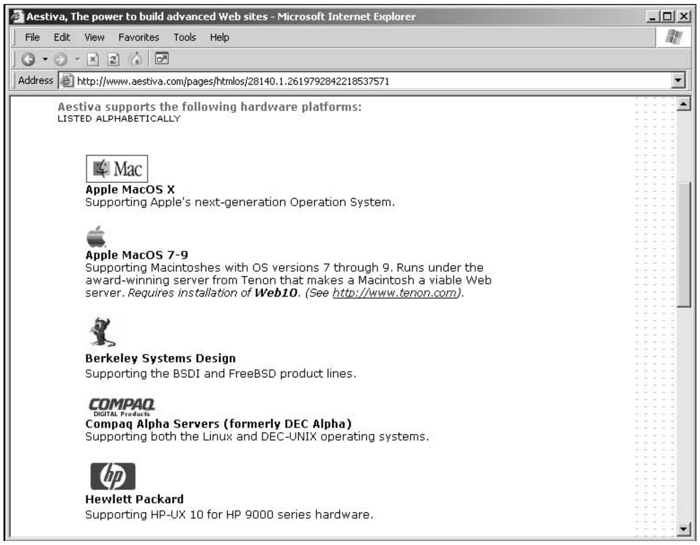
To create the directory, run the `mkdir` command with `platforms` as an argument. Remember that `mkdir`, like many shell commands, requires a filename argument with either an absolute or a relative path. If you are in the directory in which the new directory will reside, you need to supply only a relative path, as in the following example:

```
/>mkdir platforms
/>cd platforms
/platforms/>
```

Now you have a directory to hold the data. You put the document and image files in this directory. Use the `edit` command to create the new document:

```
/platforms/>edit index.html
```

When you run this command, the Web shell creates a new edit window in which you will put the contents of the `index.html` file.


Aestiva, The power to build advanced Web sites - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address
Mac
COMPAQ
DIGITAL Products

http://www.aestiva.com/pages/htmlos/28140.1.2619792842218537571

Aestiva supports the following hardware platforms:
LISTED ALPHABETICALLY

Apple MacOS X
Supporting Apple's next-generation Operation System.

Apple MacOS 7-9
Supporting Macintoshes with OS versions 7 through 9. Runs under the award-winning server from Tenon that makes a Macintosh a viable Web server. *Requires installation of* ***Web10****. (See http://www.tenon.com).*

Berkeley Systems Design
Supporting the BSDI and FreeBSD product lines.

Compaq Alpha Servers (formerly DEC Alpha)
Supporting both the Linux and DEC-UNIX operating systems.

Hewlett Packard
Supporting HP-UX 10 for HP 9000 series hardware.

Figure 5.1 The original Web page at www.aestiva.com.

After issuing the preceding command, you should have an editor window in front of you with a blank text area. You will put the HTML for the `index.html` page in this text area.

The page that you're going to model yours on can be found on the Aestiva Web site. To see the original page, go to `www.aestiva.com` and click any of the operating system icons at the bottom of the page. See the page in Figure 5.1, which shows a part of this page. The Web page you create lists the platforms on which you can run Aestiva, and by extension, the Web shell.

The following is the primary HTML content retrieved from a page posted on the Aestiva site in February 2004. Add it to the text area in the editor window:

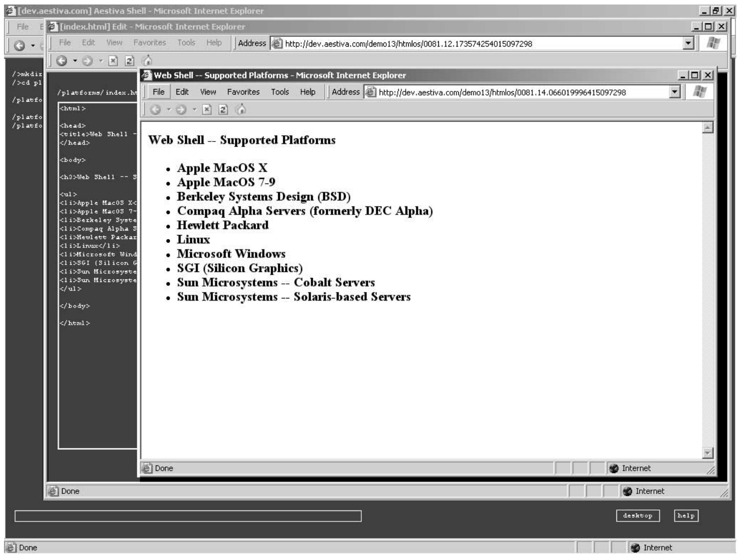
```
<html>
<head>
<title>Web Shell — Supported Platforms</title>
</head>
<body>
<h3>Web Shell — Supported Platforms<h3>
<ul>
  <li>Apple MacOS X</li>
  <li>Berkeley Systems Design (BSD)</li>
  <li>Hewlett Packard</li>
  <li>Linux</li>
  <li>Microsoft Windows</li>
  <li>Sun Microsystems — Cobalt Servers</li>
  <li>Sun Microsystems — Solaris-based Servers</li>
</ul>
</body>
</html>
```

After you finish typing the HTML into the text area, save the file by clicking the Save button. The page reloads and at the top of the editor, the status message should read something like the following, depending on the Web server's time when you saved it:

```
/platforms/index.html FILE SAVED 9:41 AM
```

The next step is to check your work. Click the Run button to launch a new window with your Web page as it will appear when rendered by your Web browser.

When you click the Run button, you should see a newly opened page like that in Figure 5.2. If you see a page like this, you have finished the first step in creating your example page.


[dev.aestiva.com] Aestiva Shell - Microsoft Internet Explorer
[index.html] Edit - Microsoft Internet Explorer
Address http://dev.aestiva.com/demo13/htmlos/0081.12.173574254015097298
Web Shell -- Supported Platforms - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://dev.aestiva.com/demo13/htmlos/0081.14.066019996415097298
Done
Internet
desktop
help

```
/>mkdir
/>cd pl
/platfo
/platfo
/platfo
```

```
/platforms/index.ht
<html>

<head>
<title>Web Shell -
</head>

<body>

<h3>Web Shell -- S

<ul>
<li>Apple MacOS X<
<li>Apple MacOS 7-
<li>Berkeley Syste
<li>Compaq Alpha S
<li>Hewlett Packar
<li>Linux</li>
<li>Microsoft Wind
<li>SGI (Silicon G
<li>Sun Microsyste
<li>Sun Microsyste
</ul>

</body>

</html>
```

Web Shell -- Supported Platforms

- **Apple MacOS X**
- **Apple MacOS 7-9**
- **Berkeley Systems Design (BSD)**
- **Compaq Alpha Servers (formerly DEC Alpha)**
- **Hewlett Packard**
- **Linux**
- **Microsoft Windows**
- **SGI (Silicon Graphics)**
- **Sun Microsystems -- Cobalt Servers**
- **Sun Microsystems -- Solaris-based Servers**

Figure 5.2 The platforms Web page lists the supported platforms.

The process that you just completed involved writing HTML code and saving it to a server across the network, then viewing the HTML page on that server from your local machine. The process of creating content on a remote server and then testing that content can be deceptively easy in the Web shell because you don't have to do any explicit networking activity. The network is an integral part of the application environment.

In the next section, you explore some of the more useful aspects of the Web shell's networking features by explicitly transferring image files across the Internet and onto your Web server. You will eventually add these image files to your platforms Web page.

Adding Images to the Web Server

HTML/OS' version of this page at `www.aestiva.com` includes an image next to each operating system. The next step is to add these images to your page. You could easily reference their addresses on Aestiva's Web site directly in the HTML document, thereby causing any visitors to the page to retrieve the image files from Aestiva, but that would be undesirable because you would be "stealing" bandwidth from somebody else's server, and you wouldn't have control of the files you are linking to. Instead, you'll have to add these files to your server and reference them from the HTML document you just created, `index.html`.

You could put the images directly inside of the platform's directory, but it's generally neater to put images in their own directory. Create an images directory called `images` inside of the `platforms` directory:

```
/platforms/>mkdir images
```

Now that you have a place to put your image files, you can transfer them from Aestiva's Web server to your Web server by using the `put` command.

Problems with Accessing Files via FTP

Historically, when developers wanted to move files around on the Web, they used File Transfer Protocol (FTP), which is the most widely used way of transferring files on the Internet. As the name indicates, FTP is not just an application; it's an entire protocol, distinct and independent of the protocol used by the Web, called Hyper Text Transfer Protocol (HTTP). Part of what that means is that when you want to move a file with FTP, your computer (the file destination computer) has to have an FTP client application running (often confusingly called FTP), and the server (the file source computer) has to have a FTP server running as well. A Web server won't necessarily have an FTP server installed, and if it is installed, it may not be configured to suit your needs. If you want to get a file from the server, you have to log in with a username and password. After you are logged in, you have to have permission to go into the directory that has the relevant files.

Using the put Command to Upload Web Files

The Web shell uses the `put` command as a substitute for FTP. It won't always be able to take the place of FTP, because the files you want may be in a directory not served by the remote computer's Web server. In that case, you can still use FTP to download the remote file to your local computer and use the `put` command from there to upload the downloaded files to the Web server. Otherwise, when it does serve your needs, the Web shell's `put` command is much more convenient and better integrated than FTP.

In this example, you copy the image files that you will reference from the `index.html` page from the Aestiva's Web server to your Web server. You do this by transferring the files from the source server to the destination server without using your personal computer as an intermediary. You could, on the other hand, download files to your personal computer and upload them, also using the `put` command, but that would require an undue amount of work and network traffic, because you would be effectively transferring files from Aestiva's Web server to your local machine and then transferring the file again to your Web server, as illustrated in Figure 5.3.

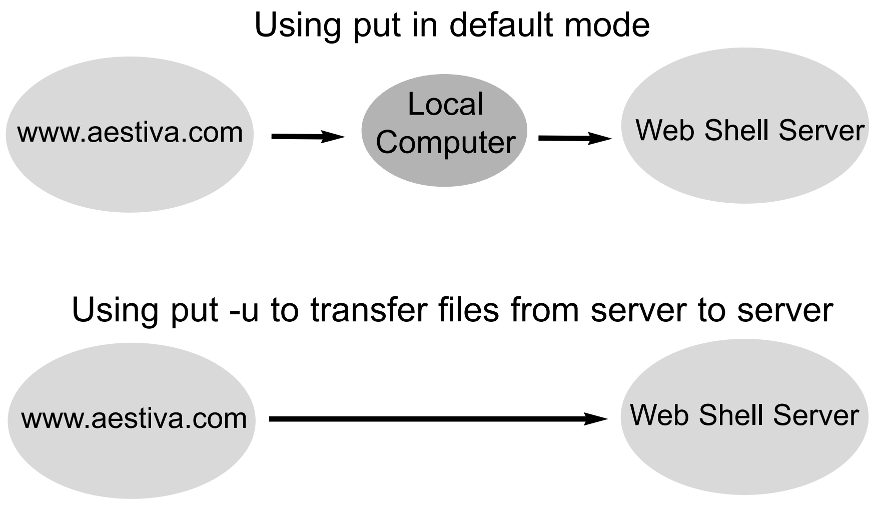

Using put in default mode
www.aestiva.com
Local Computer
Web Shell Server
Using put -u to transfer files from server to server
www.aestiva.com
Web Shell Server

Figure 5.3 The `put` command can either upload files from your personal computer or transfer files from a Web server.

Conveniently, the `put` command supports copying files from one Web server to another, without requiring that you use your client as an intermediary. To copy a file from a Web location to your Web server, use the `-u` switch (you can think of the "u" as in URL).

The general form of the `put` command is the following:

```
put [-u URL] FILENAME
```

When you don't use the `-u` option, the target filename is optional because it is inferred from the source filename. When you do use the `-u` option, the target filename is required.
For example, to copy the image file at:

```
http://www.aestiva.com/aestiva/images/partners/mac_logo.gif
```

to a local file called `/platforms/images/mac_logo.gif`, run the following command:

```
/platforms/images/>put -u
http://www.aestiva.com/aestiva/images/partners/mac_logo.gif
mac_logo.gif
```

This fetches the file located at the specified URL and creates a copy on your Web server named `mac_logo.gif`. See the accompanying note, "The `put` Command and Connection Speed."

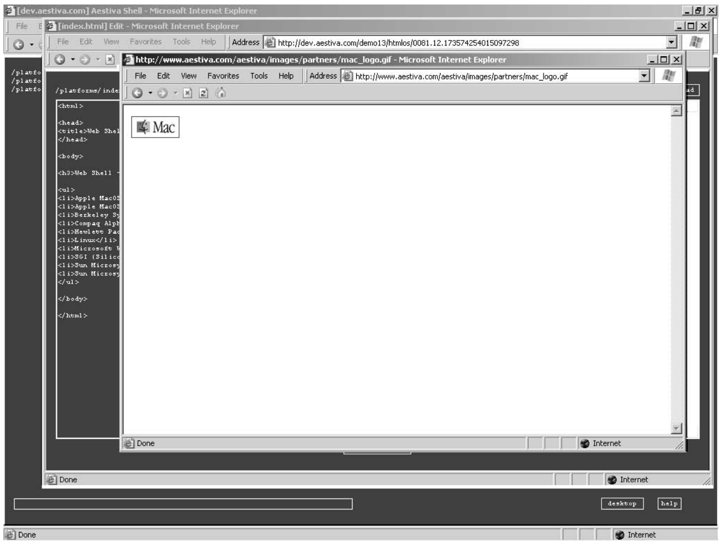
The `put` *Command and Connection Speed*

When you run the `put` *command with the* `-u` *option, the Web shell makes an HTTP request from the Web server and writes the contents of the resulting file to the server's file system. Your client computer requests only that the transaction take place; it has no role in the network transaction. That means that if the file were large, and if you were on a computer with a slow Internet connection, the transfer could be fast, depending on the connection between the shell Web server and the source Web server. Typically, the connection between one Web server and another is quite fast, so these transactions are efficient, regardless of the speed of the Web connection of your personal computer.*

To view the newly created image file, run the `view` command with the filename as its argument:

```
/platforms/images/>view mac_logo.gif
```

You should see the image appear in a new Web browser window, as shown in Figure 5.4.


[dev.aestiva.com] Aestiva Shell - Microsoft Internet Explorer
[index.html] Edit - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://dev.aestiva.com/demo13/htmlos/0081.12.173574254015097298
http://www.aestiva.com/aestiva/images/partners/mac_logo.gif - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www.aestiva.com/aestiva/images/partners/mac_logo.gif
Mac
Done
Internet
desktop
help

```
/platfo
/platfo
/platfo

/platforms/inde

<html>

<head>
<title>Web Shel
</head>

<body>

<h3>Web Shell -

<ul>
<li>Apple MacOS
<li>Apple MacOS
<li>Berkeley Sy
<li>Compaq Alph
<li>Hewlett Pac
<li>Linux</li>
<li>Microsoft W
<li>SGI (Silico
<li>Sun Microsy
<li>Sun Microsy
</ul>

</body>

</html>
```

Figure 5.4 `mac_logo.gif` in a view window.

Now that you have the Macintosh logo on your Web server, you have to add the rest of the images. Run the following commands to transfer the rest of the files:

```
/platforms/images/>put -u
http://www.aestiva.com/aestiva/images/partners/apple_logo.gif
apple_logo.gif
/platforms/images/>put -u
http://www.aestiva.com/aestiva/images/partners/bsd.gif bsd.gif
/platforms/images/>put -u
http://www.aestiva.com/aestiva/images/partners/digital_logo.gif
digital_logo.gif
/platforms/images/>put -u
http://www.aestiva.com/aestiva/images/partners/ht_logo.gif
ht_logo.gif
/platforms/images/>put -u
http://www.aestiva.com/aestiva/images/partners/linux.gif linux.gif
/platforms/images/>put -u
http://www.aestiva.com/aestiva/images/partners/sun_logo.gif
sun_logo.gif
```

Because typing all of the preceding commands would be tedious and subject to costly typos, this is a good time to use the command history feature of the Web shell. The Web shell remembers your previous commands and lets you recall these commands in several ways. (For more details on accessing the command history, refer to Chapter 3, “The Shell Environment.”) The most convenient way to recall a command is by using the up arrow on your keyboard while the cursor is in the command input box. (This feature only works in Internet Explorer, so if you are running another browser, you have to cut and paste your commands into the input box to save typing.)

If you are running IE, press the up arrow to recall the first `put` command that you ran:

```
/platforms/images/>put -u
http://www.aestiva.com/aestiva/images/partners/mac_logo.gif logo.gif
```

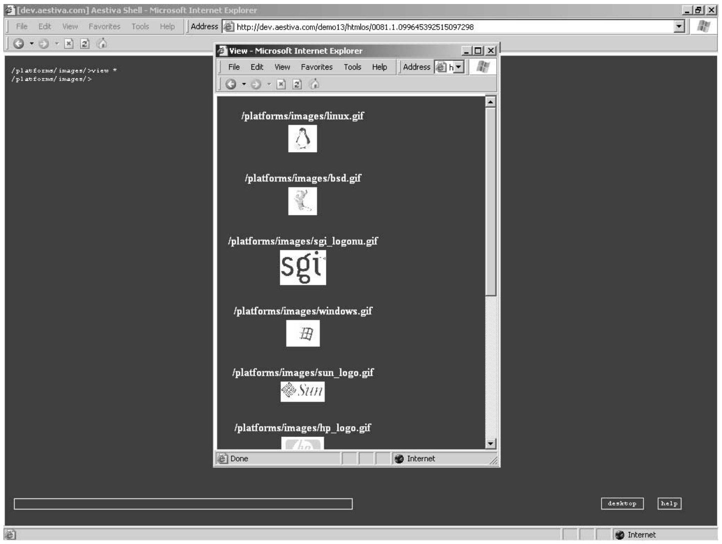
Then you edit that command so it refers to the second file in the example, `apple_logo.gif`. The only text values you have to change are `mac` to `apple` in the URL and the filename. You can use the same technique to generate all the remaining `put` commands.

Checking Your Work

When you are done putting all the image files onto the server, view the results by running the `view` command with a wildcard file argument:

```
/platforms/images/>view *
```

Because the `*` character matches all files in the current working directory, the `view` command runs against all of the filenames of the images you just uploaded. You should see a window that displays all the matching images and their filenames, as shown in Figure 5.5.


[dev.aestiva.com] Aestiva Shell - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://dev.aestiva.com/demo13/htmlos/0081.1.099645392515097298
View - Microsoft Internet Explorer
Done
Internet
desktop
help

```
/platforms/images/>view *
/platforms/images/>
```

/platforms/images/linux.gif

/platforms/images/bsd.gif

/platforms/images/sgi_logonu.gif

/platforms/images/windows.gif

/platforms/images/sun_logo.gif

/platforms/images/hp_logo.gif

Figure 5.5 Viewing all the platform images.

Adding the Image Tags to Your Web Page

Now that you have the necessary images on the Web server and have demonstrated with the `view` command that they are accessible from within a Web page, you have to add them to the platform's Web page. If you don't have the editor window open for `index.html`, open one now with the `edit` command:

```
/platforms/>edit index.html
```

Then add the image tags so that the HTML looks something like this:

```
<html>
<head>
```

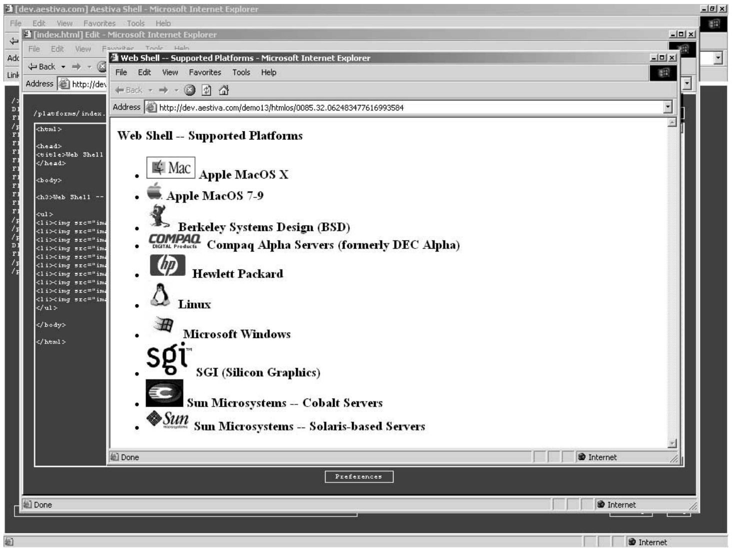
```
<title>Web Shell – Supported Platforms</title>
</head>
<body>
<h3>Web Shell – Supported Platforms<h3>
<ul>
<li><img src="images/mac_logo.gif"> Apple MacOS X</li>
<li><img src="images/apple_logo.gif"> Apple MacOS 7-9</li>
<li><img src="images/bsd.gif"> Berkeley Systems Design (BSD)</li>
<li><img src="images/digital_logo.gif"> Compaq Alpha Servers
(formerly DEC Alpha)</li>
<li><img src="images/hp_logo.gif"> Hewlett Packard</li>
<li><img src="images/linux.gif"> Linux</li>
<li><img src="images/windows.gif"> Microsoft Windows</li>
<li><img src="images/sgi_logonu.gif"> SGI (Silicon Graphics)</li>
<li><img src="images/cobalt_logo.gif"> Sun Microsystems – Cobalt
Servers</li>
<li><img src="images/sun_logo.gif"> Sun Microsystems – Solaris-based
Servers</li>
</ul>
</body>
</html>
```

Then run the page to verify your work. Your page should look something like the one in Figure 5.6.

The next step is to package this Web page with its image files into a single archive file that can be deployed on any Web shell system.

Encapsulating and Deploying Web Applications

Often, when you do Web development, you create and verify a Web-based system in a development environment, either on the same server as the deployment environment but in a different directory, or on a different server altogether. In any case, you will often have to move an entire Web application, with source code, HTML documents, code libraries, images, and other files, as well as the directory structure that contains and organizes these


Web Shell -- Supported Platforms

- Apple MacOS X
- Apple MacOS 7-9
- Berkeley Systems Design (BSD)
- Compaq Alpha Servers (formerly DEC Alpha)
- Hewlett Packard
- Linux
- Microsoft Windows
- SGI (Silicon Graphics)
- Sun Microsystems -- Cobalt Servers
- Sun Microsystems -- Solaris-based Servers

```
<html>

<head>
<title>Web Shell
</head>

<body>

<h3>Web Shell --

<ul>
<li><img src="im
<li><img src="im
<li><img src="im
<li><img src="im
<li><img src="im
<li><img src="im
<li><img src="im
<li><img src="im
<li><img src="im
<li><img src="im
</ul>

</body>

</html>
```

[dev.aestiva.com] Aestiva Shell - Microsoft Internet Explorer
[index.html] Edit - Microsoft Internet Explorer
Web Shell -- Supported Platforms - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://dev.aestiva.com/demo13/htmlos/0085.32.062483477616993584
/platforms/index.
Preferences
Done
Internet

Figure 5.6 The completed platforms Web page now contains all the image files you inserted.

files, to the environment that will host the deployed application. In the next section, you package the platforms page and its image files into one file and deploy this mini-application in a different directory on your server.

Creating an Installation File with the pack Command

The `pack` command bundles the files in a Web application, compresses and archives the files into one installation file, and embeds directory structure information so that the application can be unpacked to create a clone of the original.

To pack files, invoke the `pack` command with the name of the archive as its first argument. The following arguments are considered the file sources from which the `pack` command draws to create the archive file.

- To create a pack file, you must specify the `TARGET` pack file. If the file exists, use the `-c` option to overwrite it. The arguments following the `TARGET` file are the files and directories to be packed; any directories given will be recursively packed. Full or relative pathnames are allowed.
- By default the `BASE_DIRECTORY` is the current working directory unless otherwise specified by the `-b` option. All files to be packed must exist somewhere within the `BASE_DIRECTORY`.

- To specify protected files, use the `-p` option followed by a space-delimited list of the files to be protected. If the list contains more than one file, the entire list must be wrapped in quotes. All pathnames to the protected files must be relative to the given base directory (`cwd` by default).

  To extract (unpack) a pack file, use the `-x` option followed by the pack filename. By default, the pack file will be extracted to the current working directory unless otherwise specified by the `-b` option. To extract only specified files, use the `-p` option followed by a list of files to be extracted. The list of files must be space-delimited and wrapped in quotes. To view the contents of an existing pack file, use the `-l` option.

  Table 5.1 gives a list of the available options.

Option	Function
`-c c TARGET`	Creates a pack file and forces the overwriting of `TARGET` if it already exists
`-p "FILE..."`	Protects the specified files (for packing only)
`-b BASE_DIRECTORY`	Set the base directory (`cwd` by default) for extraction or packing
`-n`	Sets no compression (for packing only)
`-l`	Lists files embedded in given pack file
`-x`	Extracts the given pack file `-f "FILE..."` Extracts only the specified files

Table 5.1 Available Options for the `pack` Command.

The `pack` command is one of the more feature-rich commands supported by the Web shell, but for typical use, it is quite simple.

Packing the Platforms Page and Images

This section shows you how to use the `pack` command to create an installation file that can be used to redeploy the platforms page you just created. You pack up the HTML file as well as all of the images into one file called `platforms.pak`. The following files are associated with the platforms page and are going to make up the contents of the pack archive file:

```
/platforms/>ls *
FILE  PRIVATE             803  05/04/03 15:10:24  index.html
```

```
images:
FILE  PUBLIC               1040  05/04/03 23:26:19  apple_logo.gif
FILE  PUBLIC               1690  05/04/03 23:26:23  bsd.gif
FILE  PUBLIC               1520  05/04/03 23:26:38  cobalt_logo.gif
FILE  PUBLIC               1874  05/04/03 23:26:42  digital_logo.gif
FILE  PUBLIC               1194  05/04/03 23:26:59  hp_logo.gif
FILE  PUBLIC               1844  05/04/03 23:26:50  linux.gif
FILE  PUBLIC               1289  05/04/03 23:26:55  mac_logo.gif
FILE  PUBLIC                698  05/04/03 23:27:05  sgi_logonu.gif
FILE  PUBLIC                481  05/04/03 23:27:09  sun_logo.gif
FILE  PUBLIC               1768  05/04/03 23:27:13  windows.gif
```

You could pack these files by specifying them individually as arguments supplied to the `pack` command; but it would be much easier to specify their directory. The following command packs all of the files in the previous list (take note of the current working directory, `/`):

```
/>pack platforms.pak platforms
target file: /platforms.pak
compression: on
archived files:
platforms/images/linux.gif
platforms/images/bsd.gif
platforms/images/sgi_logonu.gif
platforms/images/windows.gif
platforms/images/sun_logo.gif
platforms/images/hp_logo.gif
platforms/images/cobalt_logo.gif
platforms/images/apple_logo.gif
platforms/images/digital_logo.gif
platforms/images/mac_logo.gif
platforms/index.html
```

The `pack` command, when used in this way, requires two arguments. The first argument, in the preceding example, `platforms.pak`, is the name of the archive file you are about to create. The following arguments, in the preceding case one argument, `platforms`, are the names of the files and directories that you want to include in your archive file. When you run the `pack` command, it lists the individual files it added to the archive file. In the preceding example, the files added to the `platforms.pak` archive file are the files required to run the platforms page.

In the previous example, when the `pack` command runs, you move out of the directory that contains the files to pack and reference the base directory of the Web application from there. This is the simplest method for packing an application and should serve most of your needs. In general, move into the directory immediately above the directory that contains the Web application code (if you consider the root directory to be at the top of the file hierarchy) and give `pack` the name of that container directory as its only source file argument. The `pack` command adds all files and folders as deep as the file tree goes from that container directory.

Listing Embedded Pack Files with pack -l

At this point if you were to give `platforms.pak` to another developer, he or she could examine the contents of the `.pak` file by running the `pack` command with the `-l` option. Run the `pack` command with the `-l` option to view the contents of the `.pak` file you just created, `platforms.pak`:

```
/>pack -l platforms.pak
platforms.pak:
platforms/images/linux.gif          CPUBLIC     1850
platforms/images/bsd.gif            CPUBLIC     1696
platforms/images/sgi_logonu.gif     CPUBLIC     704
platforms/images/windows.gif        CPUBLIC     1774
platforms/images/sun_logo.gif       CPUBLIC     487
platforms/images/hp_logo.gif        CPUBLIC     1200
platforms/images/cobalt_logo.gif    CPUBLIC     1259
platforms/images/apple_logo.gif     CPUBLIC     695
platforms/images/digital_logo.gif   CPUBLIC     1880
platforms/images/mac_logo.gif       CPUBLIC     1254
platforms/index.html                CPRIVATE    668
```

The resulting list displays the contents of the embedded files. The first column lists the file name, the second column lists the attribute of the file, and the third column lists the uncompressed size of the file.

These details tell you a lot about which files are contained in the `.pak` file, and how the archive will be installed. The first column, for example tells you that when the installation is performed, a `platforms` directory will be created with an `images` directory underneath.

Note that the file paths in the first column are relative to the root directory. In general, when you pack files, they will be assigned a path relative to the base directory at the time of the creation of the pack file. By default, the base directory is the current working directory at the time of creation. You can override this by using the `-b` switch and specifying a base directory other than the current working directory.

The second column tells you whether the files are public or private. (Refer to Chapter 2, "File Management." for more information about private versus public files.) When the archive is extracted, the files listed will be created on the public or private side based on the attributes listed when you run `pack -l`. These are the same attributes the files had when the archive was created. The `C` character before the public/private indication, tells you that the file is compressed in the archive. The `pack` command compresses files by default. Use the `-n` switch to turn compression off.

If, for example, you want the archive to appear like the following version, you specify that the base directory be the `platforms` directory in the following manner:

```
/>pack platforms.pak -b platforms platforms
target file: /platforms.pak
compression: on
archived files:
images/linux.gif
images/bsd.gif
images/sgi_logonu.gif
images/windows.gif
images/sun_logo.gif
images/hp_logo.gif
images/cobalt_logo.gif
images/apple_logo.gif
images/digital_logo.gif
images/mac_logo.gif
index.html
```

In the preceding example, the argument immediately after the `-b` switch overrides the default base directory, the current working directory. It might look funny to see `platforms platforms` on the command line, but in this case, the first `platforms` binds to the `-b` switch and the second `platforms` specifies the source directory. When you check the contents of the pack file, it is evident the paths are relative to `platforms`:

```
/>pack -l platforms.pak
platforms.pak:
```

```
images/linux.gif          CPUBLIC    1850
images/bsd.gif            CPUBLIC    1696
images/sgi_logonu.gif     CPUBLIC    704
images/windows.gif        CPUBLIC    1774
images/sun_logo.gif       CPUBLIC    487
images/hp_logo.gif        CPUBLIC    1200
images/cobalt_logo.gif    CPUBLIC    1259
images/apple_logo.gif     CPUBLIC    695
images/digital_logo.gif   CPUBLIC    1880
images/mac_logo.gif       CPUBLIC    1254
index.html                CPRIVATE   668
```

The importance of the paths of the file contents in a pack file arises at installation time. When you unpack a pack file, it gets unpacked into the paths specified by the pack archive. In the preceding case, when you unpack the `platforms.pak` file it will, by default, create the `index.html` file in the current working directory, create an `images` directory relative to the current working directory, and then unpack the remaining image files into that directory.

Unpacking a Pack File

To test unpacking `platforms.pak`, create a new directory called `/deployment` with the `mkdir` command then move into that directory with the `cd` command:

```
/>mkdir deployment
/>cd deployment
```

Run the `pack` command with the `-x` argument to tell `pack` to unpack the file at the given file path, as in the following example:

```
/deployment/>pack -x /platforms.pak
/platforms.pak - OK
```

Keep in mind that the `platforms.pak` file is not in the current working directory anymore because you are in the deployment directory. Use a full path to specify the location of the `platforms.pak` file.

The output of the unpacking process is a status message like the OK message in the preceding example. This tells you whether the unpacking process was successful. You can tell the `pack` command to give you more information by using the `-v` switch to indicate verbose mode. The `pack` command in verbose mode will display the names of the files it unpacked.

Now that the `platforms pack` file has successfully unpacked, list the contents of the current working directory to verify the success of the installation:

```
/deployment/>ls
DIR   MIRROR               360  05/06/03 16:46:41  images
FILE  PRIVATE              803  05/06/03 16:46:41  index.html
/deployment/>
```

Now you can run the `index.html` file with the `run` command.

```
/deployment/>run index.html
```

You should get a Web page exactly like the version you created by hand in the `platforms` directory. Now you have an exact copy of the platforms page as well as all of its images.

If you create an entire Web site with thousands of files, the deployment process is just as simple, provided that the entire application respects the rules of portability. When writing HTML files, the primary rule for portability is referring to embedded content, such as images with relative paths when you write HTML. You should avoid, when possible, using full paths in your `HREF`, `IMG`, and other tags that refer to files on your server. This ensures that a pack file like the one you created in this chapter will be portable across different directories on the same system or on different domain names.

Writing Web Applications for Portability

In the content of the `platforms` page, the image tags referred to files with relative paths, which allowed you to redeploy the page in another directory without any problems. If `index.html` contains references to its image files with full paths, the page references images in the source code area, `/platforms`. Any alteration to the development file system results in an alteration of the production file system.

The following code is an example of a less desirable version of `index.html`. Note that the image files are referenced absolutely:

```
<html>
<head>
<title>Web Shell – Supported Platforms</title>
</head>
<body>
<h3>Web Shell – Supported Platforms<h3>
<ul>
<li><img src="/platforms/images/mac_logo.gif"> Apple MacOS X</li>
<li><img src="/platforms/images/apple_logo.gif"> Apple MacOS 7-9</li>
```

```
<li><img src="/platforms/images/bsd.gif"> Berkeley Systems Design
(BSD)</li>
<li><img src="/platforms/images/digital_logo.gif"> Compaq Alpha
Servers (formerly DEC Alpha)</li>
<li><img src="/platforms/images/hp_logo.gif"> Hewlett Packard</li>
<li><img src="/platforms/images/linux.gif"> Linux</li>
<li><img src="/platforms/images/windows.gif"> Microsoft Windows</li>
<li><img src="/platforms/images/sgi_logonu.gif"> SGI (Silicon
Graphics)</li>
<li><img src="/platforms/images/cobalt_logo.gif"> Sun Microsystems —
Cobalt Servers</li>
<li><img src="/platforms/images/sun_logo.gif"> Sun Microsystems —
Solaris-based Servers</li>
</ul>
</body>
</html>
```

If you try to redeploy the preceding file and delete the original `images` directory, the page will not work. In general, keep all paths relative in your application to avoid problems when you deploy the application.

Summary

The functionality outlined in this chapter sets the Web shell apart from other shells, because the Web shell supports the ability to move files across the Web from server to server and create and deploy Web-based applications using a powerful file archiving and unarchiving functionality. You can create, manage, and deploy Web-based applications with relative ease and without having to resort to third-party tools.

6

Chapter

Creating Custom Commands

The Web shell's stock functionality is rich enough to allow you to develop, manage, and deploy Web applications. You will most likely find that your needs are met by the functionality supplied by the Web shell without much customization. However, it might be reassuring to know that the Web shell is extendable. In addition to being customizable, it is designed to allow you to create your own commands to supplement the stock command set. In this way, it can serve as a platform from which you can call and control custom scripts.

This chapter is more advanced than previous chapters in that it goes under the hood of the Web shell and looks at how commands fit within the Web shell API. It walks you through writing and testing an example command in much the same way that the designers of the Web shell created its stock functionality. By the end of the chapter, you will know everything you need to know about how to create a new command.

The language of the Web shell is Aestiva HTML/OS. HTML/OS is a simple but powerful language, and this chapter implements only enough of it to show you how to interface with the Web shell when you create your own commands. You don't need to be an expert programmer to understand what will be presented. The example command that you will create won't require the use of any complicated algorithms or functions. But it will implement the various means by which the Web shell and its commands interact, so you'll have all of the Web shell's command API tools at your disposal.

Because HTML/OS is a high-level, English-like language, you should be able to glean some of the logic in the code in this chapter, even if you don't have any coding experience. If you do have coding experience, you won't be bored by the details of how to write scripts and Web applications in HTML/OS in this book. Those details are covered in *Advanced Web Sites Made Easy* by D.M. Silverberg.

Designing a Web Shell Command

This section walks you through creating a new command called `calc`. The `calc` command will do the work of a simple calculator at the command line. More important, it will showcase the various ways in which you can pass data to a command, via arguments, switches, bound switches (see the note "Bound Switches" for more information), and piped input. You will build this command incrementally, from a very basic form into a more complicated command.

Bound Switches

A bound switch is a switch that requires an argument. An example of a bound switch is the `-k` switch in the `sort` command covered in Chapter 4, "Redirection."

Creating the calc Command File

All Web shell commands live in the same directory. If you are using the copy of the Web shell that was supplied to you by the default installation of HTML/OS, that directory is `/apps/shell/bin`. If you move into this directory with the `cd` command and list its contents, you see files for each of the Web shell commands:

```
/apps/shell/bin/>ls
FILE   PRIVATE            1009   05/12/03  21:53:16   alias.cmd
FILE   PRIVATE            1047   05/12/03  21:53:16   cat.cmd
FILE   PRIVATE             344   05/12/03  21:53:16   cd.cmd
FILE   PRIVATE             582   04/25/03  16:19:53   cdl.cmd
FILE   PRIVATE              49   05/12/03  21:53:16   clear.cmd
FILE   PRIVATE             659   04/25/03  16:19:53   cleardb.cmd
FILE   PRIVATE            1785   05/12/03  21:53:16   cp.cmd
FILE   PRIVATE             547   05/12/03  21:53:16   cpdir.cmd
FILE   PRIVATE               9   05/12/03  21:53:16   date.cmd
FILE   PRIVATE              82   05/12/03  21:53:16   desktop.cmd
FILE   PRIVATE            2476   05/12/03  21:53:16   diff.cmd
FILE   PRIVATE             498   05/12/03  21:53:16   do.cmd
FILE   PRIVATE            1635   05/12/03  21:53:16   edit.cmd
FILE   PRIVATE             980   05/12/03  21:53:16   env.cmd
FILE   PRIVATE             357   05/12/03  21:53:16   exit.cmd
FILE   PRIVATE              1320  05/12/03  21:53:16   find.cmd
FILE   PRIVATE              1279  05/12/03  21:53:16   findtxt.cmd
FILE   PRIVATE               327  05/12/03  21:53:16   fixfile.cmd
FILE   PRIVATE               331  05/12/03  21:53:16   fixprivate.cmd
FILE   PRIVATE               329  05/12/03  21:53:16   fixpublic.cmd
FILE   PRIVATE               538  05/12/03  21:53:16   get.cmd
FILE   PRIVATE              1224  05/12/03  21:53:16   grep.cmd
FILE   PRIVATE               358  05/12/03  21:53:16   help.cmd
FILE   PRIVATE               141  05/12/03  21:53:16   history.cmd
FILE   PRIVATE              5755  04/25/03  16:19:53   ld.cmd
FILE   PRIVATE               118  05/12/03  21:53:16   logout.cmd
FILE   PRIVATE              5355  05/12/03  21:53:16   ls.cmd
FILE   PRIVATE              3748  04/25/03  16:19:53   ls.lib
FILE   PRIVATE               261  05/12/03  21:53:16   mkdir.cmd
FILE   PRIVATE              1134  05/12/03  21:53:16   mv.cmd
FILE   PRIVATE              1037  05/12/03  21:53:16   mvdir.cmd
FILE   PRIVATE              3042  05/12/03  21:53:16   pack.cmd
FILE   PRIVATE               271  05/12/03  21:53:16   password.cmd
FILE   PRIVATE               460  05/12/03  21:53:16   put.cmd
FILE   PRIVATE                 9  05/12/03  21:53:16   pwd.cmd
FILE   PRIVATE               224  05/12/03  21:53:16   rm.cmd
FILE   PRIVATE               922  05/12/03  21:53:16   rmdir.cmd
FILE   PRIVATE              1474  05/12/03  21:53:16   run.cmd
FILE   PRIVATE               679  05/12/03  21:53:16   set.cmd
FILE   PRIVATE               531  05/12/03  21:53:16   sort.cmd
FILE   PRIVATE               127  05/12/03  22:08:05   t.cmd
FILE   PRIVATE               101  05/12/03  21:53:16   version.cmd
FILE   PRIVATE              1652  05/12/03  21:53:16   view.cmd
FILE   PRIVATE               660  05/12/03  21:53:16   wc.cmd
```

The preceding list shows all the commands in the Web shell. All Web shell commands must live in this directory and must have a `.cmd` extension. Because, as you might have guessed, the name of the `cmd` file corresponds to the name of the command, the command file for the `calc` command must be called `calc.cmd`.

Testing the Command API

If you are in the `/apps/shell/bin` directory already (if you aren't move into it now with the `cd` command), open an editor window for `calc.cmd` with the `edit` command:

```
/apps/shell/bin/>edit calc.cmd
```

Now that you have an edit window open, you can add text to the document. Add the following line to the editor and save the document when you are finished:

```
io = "Currently, the calc command is under development." + lf
```

As soon as you save the file, the `calc` command will be live. You can then run it at the command line to test your work:

```
/apps/shell/bin/>calc
Currently, the calc command is under development.
/apps/shell/bin/>
```

As you can see, the `calc` command in its current state displays the preceding status message when you run it. This is true even though you didn't say explicitly in the contents of `calc.cmd` that anything should be displayed. Instead you set the value of the `io` variable to the previous message. The Web shell always displays the `io` variable when it runs a command. If you want to display anything, you have to put the text that you want to display in the `io` variable. (For more information on variable in HTML/OS, read the note "Variables in HTML/OS.")

Variables in HTML/OS

A variable is a container of sorts, consisting of a name and contents. Variables in HTML/OS can be called anything, as long as the name doesn't conflict with a word that has special meaning in HTML/OS code, and as long as the variable name doesn't have any nonalphanumeric characters. The contents of a variable can consist of anything as well, but HTML/OS treats variables differently according to whether their contents contain numbers or text.

The `calc.cmd` line of code assigns the sentence on the right side of the equals sign to the variable called `io`. The sentence is wrapped in quotes to tell HTML/OS that you aren't referring to variables, but text. The quotes also delimit the beginning and ending of the sentence. The sentence contains a plus (+) symbol, which tells HTML/OS to concatenate the value on the left of it (the sentence) with the symbol to the right the `lf` keyword. (For more information on concatenation versus arithmetic, read the note "The = and + Symbols in HTML/OS.") The `lf` keyword signifies a line feed. If you omit the line feed keyword at the end of this sentence, the Web shell won't display the command prompt on the next line after calc's output.

The = and + Symbols in HTML/OS

The = symbol can signify two things in HTML/OS: assignment and comparison. Assignment tells HTML/OS to put a value into a variable. Comparison determines whether the values in two variables are the same. When you use the = symbol, HTML/OS knows which of these meanings to apply by looking at the context of the statement.
The + symbol can signify two things in HTML/OS. One is an arithmetic addition, and the other is text concatenation. HTML/OS either adds numbers arithmetically, or concatenates text, depending on the values of the variables on either side of the symbol. If both values look like numbers, it adds the two values arithmetically. If either value looks like text, it concatenates the two text values. In version 3 or above of HTML/OS the & character may be used to unambigiously concatenate two strings.

The variable you set in your one-line version of the `calc` command is the `io` variable. The `io` variable stands for input/output. It contains any data that was piped into the command from another command, and it holds output data to be piped into another command, redirected to the file system, or displayed to the screen. After a command is called, the Web shell determines what to do with the resulting `io` data based on whether you told it to redirect to the file system, pipe to another command, or display to the screen. (Refer to Chapter 4, “Redirection,” for more information on piping and redirection.)

The `io` variable is the most important variable in the Web shell’s command `API`. It is the means by which commands talk to each other and to the Web shell.

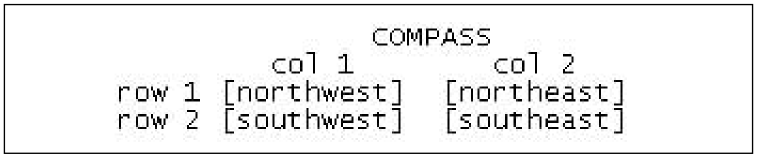
Creating Basic Functionality for the calc Command

The `calc` command will serve as a rudimentary arithmetic calculator and have the ability to add and multiply. The initial amount of functionality you will code is the ability to add two numerical arguments. Later, you add code so that you can multiply the result using shell switches.

You already know how to return information to be displayed to the screen using the `io` variable. What remains for this command is reading the two numerical arguments and adding them. The Web shell provides a simple way of accessing the argument values passed to a command by setting the values of an array called `arg_array`. The array `arg_array` is a one-dimensional array (a column), which contains the arguments you pass to the `calc` command. (For more information on arrays, refer to the note, “Arrays in HTML/OS.”)

Arrays in HTML/OS

An array is a type of variable that contains multiple distinct values. These values are organized into a grid or table. To access an individual cell in an array, use the array name *and the indexes corresponding to the column and row position of the cell. The general form of accessing an array cell is* `arrayname[column, row]`*. If a two-by-two array called* `compass` *contains the values* `northwest`, `northeast`, `southwest`*, and* `southeast`*, as in Figure 6.1, you would access the value* `northwest` *by referring to* `compass[1,1]`, `northeast` *by* `compass[2,1]`, `southwest` *by* `compass[1,2]`*, and* `southeast` *by* `compass[2,2]`*. If you omit the row index, it will be assumed to be* `1`*. The value of* `compass[2]`*, therefore, is* `northeast`*. If you omit all indexes, the row and column values are assumed to be* `1` *and* `1`*. The value of compass (no indexes specified), therefore, is* `northwest`*.*



```
                 COMPASS
           col 1          col 2
row 1 [northwest]  [northeast]
row 2 [southwest]  [southeast]
```

Figure 6.1 The compass array consists of a two-by-two matrix of data values.

Now open an editor window on the `calc.cmd` command file and replace its contents with the following line:

```
io = "argument1: " + arg_array[1,1] + lf + "argument2: " +
arg_array[2,1] + lf
```

This line returns the values of the arguments you passed to the `calc` commands to the Web shell. It serves to demonstrate that you can retrieve the arguments passed to the `calc` command. After you save the command code and run `calc` at the command line, you should see something like the following results:

```
/apps/shell/bin/>calc aaa bbb
argument1: aaa
argument2: bbb
/apps/shell/bin/>
```

Because there are `lf` keywords (line feeds) concatenated with the plus (+) operator between the two arguments, the two argument statements are separated with a line break in the output area as well.

Now that you can access the command arguments, you can add their values and output the sum. Replace the contents of `calc.cmd` with the following line:

```
io = (arg_array[1,1] + arg_array[1,2]) + lf
```

When you run this command with numerical arguments, it displays the sum to the output area as in the following example:

```
/apps/shell/bin/>calc 4 6
10
/apps/shell/bin/>
```

If you give it decimal arguments, it works just as well:

```
/apps/shell/bin/>calc 1.3 0.1
1.4
/apps/shell/bin/>
```

On the other hand, if one or both arguments are non-numerical, it concatenates the two values.

```
/apps/shell/bin/>calc Apollo 11
Apollo11
/apps/shell/bin/>
```

This is what you would expect since the addition operator adds arithmetically when its arguments are both numbers, and concatenates otherwise.

Using a Switch to Add a User Control

Now that you have a command that reads from the argument list, you will add a control to the `calc` command by adding support for a switch. The switch that you implement tells the `calc` command to print in verbose mode when a `-v` switch is present. When a user passes the switch `-v` to the `calc` command, the `calc` command displays a message about what it's doing instead of simply displaying the result of its calculation.

A few steps are involved in reading a switch from within a command. The first step is to run a function called `getflags` in the following manner.

```
void = getflags("v")
```

The `getflags` function tells the Web shell to look for the flags that you specify in the text submitted by the Web shell user. (For more information on functions, refer to the note "HTML/OS Functions.") When you run the `getflags` function, the environment is prepared so that you can retrieve the flags that the user supplied along with the command.

Clear the contents of the `calc.cmd` file, and replace them with this `getflags` statement. It tells the command to look in its argument list for a flag. The `getflags` command doesn't retrieve flag values; but it prepares the environment so that you can retrieve flag values later. In general, the `getflags` function accepts multiple switch indications. Call `getflags` in the following manner:

```
void = getflags("a,b,c,d,e")
```

The getflags function looks for switches `a`, `b`, `c`, `d`, and `e` from the command line.

HTML/OS Functions

HTML/OS functions encapsulate blocks of code that can be called with the name of the encapsulating function. You can pass data to the function from within the parentheses required when you call the function. Data is then optionally returned to the caller and assigned to a variable or used for some other purpose. Just as it is necessary to use parentheses when calling functions that require no arguments, it is also required that a variable be ready to receive any returned data, even if no data is returned. If an HTML/OS function returns no data, place a dummy variable on the left side of the equation as in `void = getflags(…)`*. In this case* `void` *is not a keyword, but the name of a dummy variable.*

After the `getflags` statement, you can assign the value that corresponds to the presence or absence of the `-v` flag to a variable. In this example, call that variable `isverbose` to approximate its meaning in English. To set the value of `isverbose` after the `getflags` call, add the following line of code to the `calc.cmd` file:

```
isverbose = getswitch("v")
```

This line sets the variable `isverbose` to a one or a zero depending on whether this switch is present. A one indicates that the switch is present, and a zero indicates that it is not present. For more information on the meaning of zero in HTML/OS refer to the note "Truth in HTML/OS."

Truth in HTML/OS

When you invoke an `if` *statement in HTML/OS code, HTML/OS checks to see whether the value of its argument is* `true` *or* `false`*. The only values that are considered to be* `false` *are the number* `zero` *and the word* `false`*. All other values evaluate to* `true`*.*

The next task that has to be accomplished is testing the value of `isverbose` and acting accordingly. To test a value in HTML/OS, as in many other languages, use an `if` statement, as in the following test for `isverbose`:

```
if (isverbose) then
```

```
  io = "The sum of " + arg_array[1,1] + " and " + arg_array[1,2] +
" is "
/if
```

Add the preceding statement to the `calc.cmd` after the `isverbose` assignment. Then add the following statement to append to `io` the value of the sum:

```
io = io + ( arg_array[1,1] + arg_array[1,2] ) + lf
```

Notice that `io` is being assigned to the value of itself plus the value of the sum. You wouldn't want to lose the verbose content by overwriting it.

The entirety of the `calc` command as developed thus far should look like the following:

```
void = getflags("v")
isverbose = getswitch("v")
if (isverbose) then
  io = "The sum of " + arg_array[1,1] + " and " + arg_array[1,2] +
" is "
/if
io = io + ( arg_array[1,1] + arg_array[1,2] ) + lf
```

After you save this file, run the command to see its effect. If you run it with the `-v` switch, it displays the explanatory sentence; otherwise, it displays only the value of the sum of the two arguments.

Here is the command as run without the `-v` switch:

```
/apps/shell/bin/>calc 111 222
333
/apps/shell/bin>
With the -v switch, it prints the descriptive sentence:
/apps/shell/bin/>calc -v 111 222
The sum of 111 and 222 is 333
/apps/shell/bin/>
```

In general, switches are placed at the beginning of the list of command arguments for the sake of clarity. In the Web shell, however, you can place these switches wherever you like. All of the following calls to the `calc` command in verbose mode are equivalent:

```
/apps/shell/bin/>calc -v 111 222
The sum of 111 and 222 is 333
/apps/shell/bin/>calc 111 -v 222
The sum of 111 and 222 is 333
/apps/shell/bin/>calc 111 222 -v
The sum of 111 and 222 is 333
/apps/shell/bin/>calc 111 222-v
The sum of 111 and 222 is 333
/apps/shell/bin/>calc "-v" "111" "222"
The sum of 111 and 222 is 333
```

The functionality that makes the preceding commands identical from the perspective of a command is taken care of by the Web shell layer. When you write a new command, you don't have to concern yourself with the details of parsing the command line and accounting for the various possibilities of human input variations and errors, because those matters are handled by the Web shell. The switch and argument data retrieval tools are robust enough for you to be able to code without having to perform error checking at the command level.

Adding a User Control with a Bound Switch

The second type of switch is one that is bound to a piece of data followed immediately after the switch. This kind of switch itself requires an argument.

You add multiplicative functionality to the `calc` command through the use of a bound switch. When the `calc` command is called with an `-m` switch and a switch argument, the sum of the arguments will be multiplied by the switch argument. For example, the following command produces a result of `6`, the sum of `4` and `2`.

```
/apps/shell/bin/>calc 4 2
6
/apps/shell/bin/>
```

Adding the `-m` switch with a multiplier, would produce the following output:

```
/apps/shell/bin/>calc 4 2 -m 2
12
```

The `-m` switch specifies that its argument multiplies the sum of `calc`'s arguments.

To add the `-m` bound switch to the `calc` command, modify the `calc.cmd` text file to look like the following:

```
void = getflags(”v,m=true”)
isverbose = getswitch(”v”)
multiplier = getswitch(”m”)
sum = arg_array[1,1] + arg_array[1,2]
if (multiplier != ‘’) then
  sum = sum * multiplier
/if
if (isverbose) then
  io = ”The sum of ” + arg_array[1,1] + ” and ” + arg_array[1,2]
  if (multiplier != ‘’) then
    io = io + ” multiplied by ” + multiplier
  /if
  io = io + ” is ”
/if
io = io + sum + lf
```

The first line calls the `getflags`, telling it to look for the switches `-v` and `-m` in the command line and telling it that the `-m` switch will be a bound switch. When a switch is bound (meaning, when it has an argument) the convention is to put an `=true` next to the switch in the `getflags` argument string. This tells `getflags` to look for an argument next to the switch and remove that argument from the general argument list, `arg_array`. It is also important to avoid putting spaces between the flag arguments in the argument string. Spaces cause the `getflags` function to fail.

The next line calls the `getswitch` function with an argument of `-v`, telling it to return a zero or one, depending on whether the switch is present among the command arguments.

The following line calls `getswitch` again, this time looking for the existence of the switch `-m` and its argument. When a switch requires an argument, the `getswitch` command behaves differently from when the switch does not. In particular, `getswitch` will return nothing (the empty string, `“”`) when the switch is not present or when the switch has no argument. When the switch and an argument are both present, `getswitch` returns the value of the switch argument.

After the multiplier is set by the call to `getswitch`, the sum of the two arguments is taken and applied to a variable called `sum`. It is often more convenient to use a container variable like `sum`, instead of making the calculation inline, because it allows you to manipulate the sum value if you need to without adding additional logical statements in the display routine. In this example, you want to multiply the value of `sum` if a multiplier has been specified.

On the following line, the code checks to see whether the multiplier variable is not equal to the empty string (the `!=`, or not equal to symbol is the logical negation of the `=`, or equal to symbol, because the exclamation mark, or negation operator, generally signifies logical negation). If the multiplier variable is not equal to the empty string, the switch and an argument are present. This switch argument, having been returned by `getswitch`, is now contained in the multiplier variable. In this case, the sum is multiplied by the multiplier value.

The following line executes the multiplication. The asterisk (`*`) signifies multiplication. Therefore, the following statement is one of assignment in which the value of the contents of the sum variable is multiplied by the value of the multiplier variable and the result is assigned to the contents of the sum variable:

```
sum = sum * multiplier
```

At this point in the code, the value of `sum` contains the resulting value, regardless of whether the multiplier switch was provided.

The next line prints the verbose message if the verbose flag was set. Now that there is a multiplier, the code that constructs the verbose message has to account for the possibility of a multiplier. If one exists, the output message is appended to indicate that the result was multiplied. These lines append text to the `io` message accordingly.

The final line of code appends to the `io` variable the value of the calculation. This line causes the correct result to be printed, regardless of whether the verbose flag was set. If it is set, the result is concatenated onto the message. Otherwise, the result is concatenated to the value of `io`. Because `io` is often empty when there is no piped input to a command, this doesn't present a problem when the `calc` command is run by itself.

Because you will eventually want to add support for piping, and because you shouldn't count on users not piping commands anyway, this deficiency will have to be addressed before the `calc` command is completed. (For more information on how piping is used by Web shell users, refer to the Chapter 4, "Redirection.")

Supporting Piped I/O

To demonstrate how to use piping, you add support for piping to the `calc` command so that you can evaluate more complicated statements like the following:

```
    (2+2)*3 + (4+4)*5
```

To do so, you use the following command:

```
/>calc 2 2 –m 3 | calc 4 4 –m 5
```

Managing data via the `io` variable is the only matter you have to consider when supporting piping in your commands. The value of the `io` variable at the beginning of a command is the value of the data that was piped to the command. On the other hand, if the command is called without any piped input, the `io` variable starts with the value of the empty string.

To be able to support arithmetic statements similar to the preceding one, you have to save the value of the `io` variable before you write to it and add that initial value to the output of `calc` command.

Alter your code to look like the following code block to allow for the support of piping in the manner prescribed:

```
void = getflags("v,m=true")
isverbose = getswitch("v")
multiplier = getswitch("m")
sum = arg_array[1,1] + arg_array[1,2]
if (multiplier != '') then
  sum = sum * multiplier
/if
input = io
if (input != '') then
  sum = sum + input
/if
if (isverbose) then
  io = "The sum of " + arg_array[1,1] + " and " + arg_array[1,2]
  if (multiplier != '') then
    io = io + " multiplied by " + multiplier
  /if
  if (input != '') then
    io = io + " and added to " + input
  /if
  io = io + " is " + sum + lf
else
  io = sum
/if
```

The `calc` command code is altered to save the value of `io` and assign it to another variable called input. After the sum is calculated, the value of `input` is added to it. The addition of the `input` value also required that the verbose text be altered to indicate the input value added. Importantly, whereas previous iterations of the `calc` command were appending a line feed to all output, verbose or not, any version that supports piping should not.

Presumably, it is obvious that the verbose version of the `calc` command could not be piped into another `calc` command, because the second `calc` command would try to add the verbose statement to its calculation instead of the value arrived at by the previous command. Because the `io` variable is the only channel of communication between commands, you have to pass the exact value you want read, without extraneous text, spaces, or line feeds to the target command.

After saving this final version of the `calc` command, run it in the following way:

```
/>calc 2 2 -m 3 | calc 4 4 -m 5 -v
The sum of 4 and 4 multiplied by 5 and added to 12 is 52
```

The preceding command is equivalent to `( ( 2 + 2 ) * 3 ) + ( ( 4 + 4 ) * 5 ).` Notice that the `-v` flag isn't set until the final instance of the `calc` command. If it had been set in the first instance of the `calc` command, the second `calc` command would have added its calculation to the sentence, `"The sum …"`, and not to the numerical value its command produced. This is an example of a command that has two modes of communication: one for other commands and one for humans. Make sure that you program your Web shell commands with these considerations in mind. A command might generate output that is functional for an end user, but isn't easily interpreted by computer code. It is often better to implement two different modes of output for this reason.

Summary

The Web shell can be a powerful tool for running Web-based scripts. After you learn the HTML/OS language, you can create custom commands that can interface with other commands and manage your Webbased system according to your particular needs. The Web shell, in addition to being a file-management tool and Web-development and deployment tool, can serve as a tool for custom system administration through the use of its extensible command set.

a Appendix

HTML/OS Resources

The Web and elsewhere offers multiple resources where you can join communities of Web shell and HTML/OS users, read about UNIX shells, and learn how to create Web applications with Aestiva HTML/OS. This appendix provides many of these resources that may be of interest to you.

The Web Shell Web Site

Access your account and other resources related to the Web shell provided by Aestiva at the following address:

http://h2o.aestiva.com

Books on UNIX Shells

There are many good books on UNIX shells. The following two books are among the most popular. The first introduces you to basic shell-related concepts and the second is a thorough resource for a large number of UNIX commands.

Learning the UNIX Operating System, by Jerry Peek, John Strang, and Grace Todino-Gonguet, O'Reilly, October 2001.

UNIX in a Nutshell: System V Edition, by Arnold Robbins, O'Reilly, September 1999.

A Book on HTML/OS

The following is a good book on HTML/OS

Advanced Web Sites Made Easy, by D.M. Silverberg, Top Floor, 2001.

The Aestiva Web Site

The Aestiva Web site is a good place to find information on the latest Web-based products offered by Aestiva. It contains code samples and other resources you can use to learn

HTML/OS and packaged applications you can install on your copy of Aestiva Web shell. Visit www.aestiva.com.

The H2O Web Site

The Aestiva H_2O Web site is a good place to find information on Aestiva H_2O, a freely available version of HTML/OS. Visit h2o.aestiva.com.

A Book on HTML

The following is a must-read for Web developers. It teaches you how to use HTML as it was intended to be used.

HTML & XHTML: The Definitive Guide, 5th Edition, by Bill Kennedy and Chuck Musciano, O'Reilly, August 2002.

b Appendix

Shell Man Pages

This appendix provides a reference to the manual pages provided by the Web shell for each of its commands. To access these manual pages, use the `man` command with the name of the command as an argument. For example, to retrieve a manual page for the `sort` command, issue the following command: `/>man sort`.

The previous example command will display the manual page for the `sort` command—the same manual page contained in this appendix. This appendix begins, under the heading "Basic Supported Shell Features," with the manual page for the Web shell itself, and a synopsis of the most important features supported by the Web shell. Then all of the Web shell commands are listed along with the manual or `man` page for each command.

Basic Supported Shell Features

The Web shell supports the standard features common to most shell environments. To perform tasks, you issue commands at the command line and control those commands via arguments and switches. The Web shell features, including data piping and redirection, wildcard expansion, command aliasing, and command history access, make running commands easy. This section of this appendix provides an outline of these fundamental features.

Command Syntax

Commands often require or allow arguments. Arguments are symbols that appear after a command: `command arg1 arg2 arg3`.

Commands also often allow switches, which are letters after a hyphen (`-`), and give the user control over a particular feature in a command. For example, `ls -t` sorts the output of `ls` by the time stamp on the file.

Some switches require an argument immediately following the switch. For example, `sort -nk 5 myfile.txt` sorts `myfile.txt` numerically on column 5. If the order were reversed this way, `sort -kn 5 myfile.txt`, the command would fail because the switch `k` requires an argument immediately following it. Alternatively, each switch can be specified individually: `sort -n -k 5 myfile.txt`.

Executing Multiple Commands

To execute multiple commands in succession without waiting for the command prompt, separate the commands with semicolons: `cp index.htm index.html; rm "*.htm"; ls`.

Piping

Piping allows multiple commands to operate in conjunction with each other by causing the output of one command to be the input of another without printing the intermediate output to the screen. By default the output of any given command will be printed to the screen. This output can instead be piped, using the pipe (`|`) symbol on the command line, into another command as its input, given that the second command accepts piped input. A typical use of piping might be to use `grep` to print only matching lines to the screen and suppressing all other lines. For example, `ls | grep dir` outputs only the directories listed in the current directory. First, `ls` outputs a file list that is piped into the `grep` command; then, `grep` outputs only the lines of its input — the file list — that contain a match with the given argument, `dir`. In effect, only the directories are printed to the screen, and all other lines output by `ls` are suppressed. Additionally, another command can be added to this command set. For example, `ls | grep dir | sort -nk 3` sorts the output of `grep` numerically on column 3.

Redirection

Redirection is similar to piping in that it channels the output of a command. Redirection, however, channels that output into a file and it uses the `>` symbol. For example, `ls > files.txt` redirects the output of `ls` from the screen and writes it to a file named `files.txt`. Any output can be redirected this way, and the file does not have to exist already; but if the file does exist, it will be overwritten without prompting. The burden is on the user to make sure that data isn't overwritten.

Similarly, output can be redirected and appended to an existing file using the `>>` symbol. For example, `cat myfile >> file.txt` appends the contents of `myfile` to the existing file `file.txt`.

Wildcards

The Web shell environment will expand any command argument containing the asterisk symbol (`*`) into a list of the appropriate matching files. For example, `a*` will be expanded into a list of files in the current directory with names that begin with the letter `a`. Using `*a` expands a list of files in the current directory with names ending with the letter `a`. `q*txt` matches all files with names that begin with `q` and end with `txt` and contain any number of characters in between.

You can use `*conf*` to match all files containing the word `conf` anywhere in their filenames.

This concept can be used to find files in multiple directories. For example, `*/*.html` matches all files in any directory of the current directory that end with `.html`. `/*/*b*/a*.txt` matches all files in any directory two levels deep containing the letter `b` with filenames that begin with the letter `a` and end with `.txt`.

An example of the use of wildcards might be viewing all `.gif` files in a particular directory: `view *.gif`. You might also need to remove a set of files: `rm *_old*`.

To disable wildcard expansion, wrap the argument in double quotes: `cp "my*file" newname`.

Aliasing

Commonly used commands can be tedious to write if they are somewhat long, so they can be "aliased" with a convenient name that will invoke the associated long command. For example, you can alias `home` in this way:

```
alias home "cd /dir1/dir2/dir3/dir4"
```

Then, typing `home` on the command line changes the directory accordingly. To unset or delete this alias type `alias home`.

Sometimes it is convenient to alias a command with an already-existing command name, such as alias `ls "ls -d"`. In this case, typing `ls` always invokes the `ls -d` command. To turn this off, wrap the command in double quotes: `"ls"`. This way `ls` invokes the raw, unaliased command, `ls` and not `ls -d`. Type `help alias` for more detailed information.

History

Use the exclamation mark at the command line to invoke previously executed commands, as shown in the following examples:

- `!12` invokes the twelfth command executed since the beginning of the user session.
- `!-3` invokes the command executed three commands ago.
- `!e` invokes the most recent command starting with `e`, such as `edit myfile`.

Type `help history` for more detailed information.

Command History Scrolling

At the command prompt use the arrow keys to scroll through the recent command history and automatically write them to the command-line input box. The up arrow scrolls back in the command history, and the down arrow scrolls forward in the command history. This feature works in Internet Explorer, but is not supported by other browsers at the time of writing.

Shell Commands

The following is a list of the Aestiva Shell commands. Type `help shell` for details concerning the Aestiva Shell. Type `help` and the command details relating to a command.

- `alias` associates a command with an alias.
- `cat` concatenates and displays files.
- `cd` changes the working directory.
- `cdl` changes the working directory and then lists files.
- `clear` clears the screen.
- `cleardb` clears a database.
- `cp` copies files.
- `cpdir` copies a directory.
- `date` outputs the date and time.
- `desktop` launches the HTML/OS desktop.
- `diff` displays differences between two text files.

- `do` runs HTML/OS code from the command line.
- `edit` creates or edits a document.
- `exit` exits the shell.
- `find` finds a file.
- `findtxt` finds files with a given text occurrence.
- `fixfile` moves a file to its appropriate file area.
- `fixprivate` moves a file to the private area.
- `fixpublic` moves a file to the public area.
- `get` downloads files from the server.
- `grep` finds lines with a string match.
- `help` accesses the command documentation.
- `history` outputs the command history.
- `ls` outputs a file list.
- `mkdir` creates a new directory.
- `mv` moves files.
- `mvdir` moves a directory.
- `pack` creates, extracts, or displays an Aestiva pack file.
- `password` redefines the Aestiva system password.
- `put` transfers files to the server.
- `pwd` prints the working directory.
- `rm` removes files.
- `rmdir` removes a directory.
- `run` runs an HTML document.
- `set` changes/displays shell settings.
- `sort` sorts text.
- `version` outputs the shell version information.
- `view` displays an image file.
- `wc` counts the words and lines in a `documentNAME`.

alias

Name `alias` aliases a shell command.

Syntax `alias [ NAME [ STRING] ]`

Description Create a convenient name for a command. The new name then invokes the associated command at the shell prompt. Nesting aliases is okay. If two arguments are given, the first argument is the alias, and the second argument is the associated command. If the command contains a space, it must be encased in double quotes. If one argument is given, then if such an alias exists, it is deleted. If no arguments are given, a list of the current aliases is displayed.

Examples

```
alias home "cd /mydir1/mydir2/mydir3"
```

```
alias my_unwanted_alias
alias
```

cat

Name `cat` concatenate files.

Syntax `cat [ -d] [ FILE...]`

Options `-d` separates files with a text divider.

`-l` outputs line numbers.

`-c` makes output clickable.

Description Concatenate files to standard output. Input can be piped in and/or be read from file arguments. Use option `-d` to print dividers between files.

Examples

```
ls | cat zzz.txt
```

```
cat -l a*
cat myfile1 myfile2 myfile3 > all.txt
```

clear

Name `clear` clears the screen.

Description Removes all content from the screen and puts the command prompt at the top of the frame.

Example

```
Clear
```

cleardb

Name	`cleardb` erases all records in an Aestiva database file.
Synopsis	`cleardb [ -n] FILE...`
Options	`-n` creates no database (`.db`) backups.
Description	Clears all records in a database. By default, the existing `.db` file is moved to `.db.bak`. A new, recordless `.db` file is created in its place. The `FILE` argument may contain the `.db` extension, or no extension. Requires the existence of a `.conf` file.
Examples	`cleardb mydb`
	`cleardb mydb.db`
	`cleardb mydb.db -n` doesn't backup existing database (.db) files.

cd

Name	`cd` changes the directory.
Syntax	`cd DIRECTORY`
Description	Changes the current working directory to the specified relative or absolute path.
Examples	`cd mydir`
	`cd /apps/myfolder`

cdl

Name	`cdl` changes the directory and then outputs a file list of the new directory.
Syntax	`cdl DIRECTORY`
Description	Changes the current working directory to the specified relative or absolute path. Lists the files in the new directory. Typically invoked via a `cd` alias.
Examples	`cdl mydir`
	`cdl /apps/myfolder`

cp

Name	`cp` copies files.

Syntax	`cp SOURCE DESTINATION`
	`cp FILES... FOLDER.`

Description	Copies a file to a destination or several files to a directory.

Examples	`cp myfile new_filename`
	`cp *.txt myfolder`
	`cp file1 file2 file3 myfolder`

cpdir

Name	`cpdir` copies a directory.
Syntax	`cpdir SOURCE DESTINATION`
Description	Copies a directory to a target directory. If the `DESTINATION` directory exists, the `SOURCE` directory will be placed inside `DESTINATION`.

Examples	`cpdir source_dir target_dir`
	`cpdir /aaa /aaa/bbb`

date

Name	`date` outputs the date and time.
Description	Outputs the current date and time as set on the server.
Example	`date`

desktop

Name	`desktop` exits to the desktop.
Description	Exits the Web shell and opens the Aestiva HTML/OS desktop.
Example	`desktop`

diff

Name	`diff` outputs the differences between two text files.
Syntax	`diff FILE1 FILE2`
Description	`diff` finds the differences between two text files and outputs the unique blocks of text in `FILE1` followed by the unique blocks of text in `FILE2`.

Example	`diff /mydir/lib.txt /backup/mydir/lib.txt`

do

Name	`do` runs HTML/OS code from the command line.
Syntax	`Do [ HTMLOS_CODE]`
Description	Runs HTML/OS code from the command line. Results are displayed in a pop-up window. Wrap code containing spaces in double quotes.

Examples	`do now`
	`do fibonacci(12)`
	`do "for name=x value=1 to 100 do display x*x+'<br>' /display /for"`

edit

Name	`edit` creates or edits a text file.
Syntax	`edit [[ -b] or [ -f]] FILE...`
Description	Launches an editor window for each file argument given. If the file exists, it is editable in its window. If the file does not exist, a blank unsaved document window is opened with the appropriate filename. To prevent the opening of a new window for nonexistent filenames, use the `-f` option. To start one or more new files with a predefined boilerplate use the `-b` option. `edit` accepts piped input in the format of a column of paths. Each path spawns its own editor window.
Options	`-f` edits existing file. `-b` opens new, nonexistent file with HTML boilerplate as defined in `conf/boilerplate.txt`.

Examples	`edit myfile`
	`edit *.html`
	`edit -f existingfile` opens edit window only if file exists
	`edit -b newfile` adds boilerplate to nonexisting file
	`edit -b a.html b.html c.html` opens list of new files adding boilerplate to each
	`findtxt "my string" | edit`

exit

Name	`exit` exits to prior desktop or log out.
Description	Exit the shell environment. If there is no prior desktop, this command kills the current session and returns to the login page; otherwise it returns to the desktop.
Example	`exit`

find

Name	`find` finds files in a directory tree.
Syntax	`find [ BASEDIR] [ -d DEPTH]  STRING`
Options	`-d` sets the number of directories deep.
	`-c` outputs lines that link to an editor.
Description	Searches for an exact filename match in a directory tree. Wrap argument in double quotes to use wildcard matching. Specify a `DEPTH` to limit the relative depth of the file traversal.
Examples	`find -d 1 myfile` returns the paths of files whose names are `myfile`. Searches only in the current working directory.
	`find /x "*.txt"` traverses the directory tree starting at `/x` and returns the paths of all files whose names end with `.txt`.
	`find / "a*z"` traverses the entire directory tree starting at the root directory returning the paths of all files whose names begin with `a` and end with `z`.

findtxt

Name	`findtxt` finds text in a directory tree.
Syntax	`findtxt [ BASEDIR] [ -d DEPTH]  STRING`
Options	`-d` set the number of directories deep.
	`-c` outputs lines that link to an editor.
Description	Searches for a string in the contents of files contained in a particular directory and its subdirectories. By default if `BASEDIR` is omitted, it is set to the current working directory. Specify a `DEPTH` to limit the relative depth of the file traversal. Returned is a

	list of the files that contain a string match. The string match will not necessarily be word based. Search strings containing spaces must be wrapped in double quotes.
Examples	`findtxt abc` traverses the directory tree starting at the current working directory. Returns the names of files containing the text `abc`.
	`findtxt “Edward V”` returns the names of files containing the text `Edward V`.
	`findtxt / “ zz”` traverses the entire directory tree, returning the names of files containing a word beginning with `zz`.
	`findtxt -d 1 myfnc` looks only in the current working directory returning the names of files containing the word `myfunc`.

fixfile

Name	`fixfile` system fix file.
Syntax	`fixfile FILE...`
Description	Places a file in the correct area (public or private) according to the file extension and HTML/OS settings. If the file is already correct, it does nothing.
Example	`fixfile *.html`

fixprivate

Name	`fixprivate` moves a file to the private HTML area.
Syntax	`fixprivate FILE...`
Description	Moves a file to the private area.
Examples	`fixprivate a.txt`
	`fixprivate *.html`

fixpublic

Name	`fixpublic` moves a file to the public HTML area.
Syntax	`fixpublic FILE...`
Description	Moves a file to the public area.

Examples

```
fixpublic a.txt
fixpublic *.html
```

get

Name: `get` downloads files from the server.

Syntax: `get FILENAME...`

Description: Downloads one or more specified files. A pop-up window opens with a list of links which, when clicked, initiate a download of the file.

Examples

```
get myfile.txt
get *.gif
```

grep

Name: `grep` outputs lines matching a pattern.

Syntax

```
grep [options] PATTERN
grep [options] PATTERN FILE...
```

Description: `grep` searches the named input files (or standard input if no files are named) for lines containing a match to the given pattern. By default `grep` prints the matching lines. `grep` accepts piped input.

Options

`-v` outputs only nonmatching lines.

`-c` outputs only the number of matching lines.

`-n` outputs line numbers and suppresses all other output.

Examples

```
grep -n mystring some_file
grep "this string has spaces" *.txt
ls | grep private
```

history

Name: `history` displays shell command history with associated line numbers and invokes a previous shell command.

Description: The `history` command displays shell command history of a user session. The associated line numbers allow the convenient calling of a previous command using the exclamation operator (`!`). Such commands can be invoked absolutely or relatively.

Examples

`history !101` invokes command number 101 in the shell history.

`!-12` invokes the command twelve commands ago in the shell.

`history !gr` invokes the most recent command beginning with `gr`.

ls

Name

`ls` lists directory contents.

Syntax

```
ls [ FILE...] [ DIR...]
```

Description

If no arguments are given the files in the current directory are listed. Otherwise the file contents are listed of any directory given as an argument, and the file details are listed of any file given as an argument. Output is by default sorted alphabetically and in the following format:

```
TYPE  PUBLIC/PRIVATE/MIRROR  SIZE(BYTES)  TIMESTAMP
```

Column 2 indicates the area in which the file is located. Refer to HTML/OS documentation for details about file areas. If the `-x` switch is used, the format is the following:

```
TYPE  PUBLIC/PRIVATE/MIRROR  SCRAMBLED  SIZE(BYTES)
TIMESTAMP  NAME
```

Options

`-t` sorts by date.

`-d` sorts by type (dir/file).

`-s` sorts by size.

`-x` indicates whether scrambled in column 3 of output (S=scrambled, N=normal).

Examples

`ls *.html myfiles.*`

`ls /*/*.xyz` No trailing A `/` returns files and folder contents. The leading `/` specifies full path.

`ls */*aaa*/` The trailing. A `/` returns only folder contents. No leading `/` specifies relative path.

`ls -t`

mkdir

Name	`mkdir` creates a new directory.
Syntax	`mkdir [ DIRNAME...]`
Description	Creates an empty directory.
Example	`mkdir dir1 dir2 dir3/dir3`

mv

Name	`mv` moves files.

Syntax	`mv SOURCE DESTINATION`
	`mv SOURCE... FOLDER`

Description	Moves a file to another location or moves multiple files to a directory. If two arguments are given, a file is moved to its destination. If more than two arguments are given, multiple files are moved to a destination folder; in this case, the final argument must be an existing folder.

Examples	`mv oldname.txt newname.txt`
	`mv *.txt myfolder`

mvdir

Name	`mvdir` moves (renames) a directory.
Syntax	`mvdir SOURCE DESTINATION`
Description	Moves a directory. If the destination directory already exists, the source directory is placed inside the destination directory.
Example	`mvdir source_dir target_dir`

pack

Name	`pack` creates, extracts, or displays an Aestiva pack file.

Syntax	`pack [ -p "FILE..."] [ -b BASE_DIRECTORY] [ -n] [ -c] TARGET FILE/DIR...`
	`pack -x FILE... [ -f "FILE..."] [ -b BASE_DIRECTORY]`
	`ack -l FILE...`

Description

To create a pack file, the target pack file must be specified. If the file exists, use the `-c` option to overwrite. The arguments following the target file are the files and directories to be packed; any directories given will be recursively packed. Full or relative pathnames are allowed. By default the `BASE_DIRECTORY` is the current working directory unless otherwise specified by the `-b` option. All files to be packed must exist somewhere within the `BASE_DIRECTORY`. To specify protected files use the `-p` option followed by a space-delimited list of the files to be protected. If the list contains more than one file, the entire list must be wrapped in quotes. All pathnames to the protected files must be relative to the given base directory (`cwd` by default).

To extract (unpack) a pack file use the `-x` option followed by the pack filename. By default, the pack file is extracted to the current working directory unless otherwise specified by the `-b` option. To extract only specified files, use the `-p` option followed by a list of files to be extracted. The list of files must be space-delimited and wrapped in quotes.

To view the contents of an existing pack file, use the `-l` option.

Options

`-c TARGET` creates a pack file and forces the overwrite of `TARGET` if it already exists.

`-p "FILE..."` protects the specified files (for packing only).

`-b BASE_DIRECTORY` sets the base directory (`cwd` by default) for extraction or packing.

`-n` provides no compression (for packing only).

`-l` lists files embedded in a given pack file.

`-x` extracts.

`-f "FILE..."` extracts only the specified files.

Examples

`pack myfile.pak *` packs all files and folders (recursively) in the current working directory into `myfile.pak`.

`pack -c existingfile.pak *.html somedirectory` packs all `.html` files in the `cwd` and all files in `somedirectory` and overwrites `existingfile.pak`.

```
pack /pak/new.pak mydir -p "mydir/settings.txt
```

`mydir/config"` packs `mydir` into `/pak/new.pak`, protects `settings.txt` and `config`; the protected files are given pathnames relative to the current working directory (`basedir=cwd` by default).

`pack -n calc.pak -b /mycalc /mycalc/* -p "protectme.txt metoo.txt"` turns compression off. The base directory is set to `/mycalc`. This packs all files within `mycalc` and protects `protectme.txt` and `metoo.txt` (note pathnames relative to the base dir).

`pack -x myfile.pak` extracts `myfile.pak` to the current working directory.

`pack -x myfile.pak -b mydir` extracts `myfile.pak` to `mydir`.

`pack -l myfile.pak` lists the embedded files in the existing pack file, `myfile.pak`.

password

Name — `password` redefines the Aestiva system password.

Description — Sets the password used at the Aestiva HTML/OS login page. If the new password contains spaces, it must be wrapped in quotes. The password must be entered twice on the command line for confirmation.

Examples

```
password newpass newpass
```

```
password "my newpass" "my newpass"
```

put

Name — `put` uploads a file to the server.

Syntax — `put [ -u URL] DESTINATION`

Options — `-u URL` in lieu of the pop-up window, a file is transferred to the server from the given URL.

`-l` converts filename to lowercase.

Description — By default, a pop-up window appears which prompts for a local file. After the upload has been initiated, the window must be left open until the file has been uploaded. If the `-u` option is used, a file is transferred directly to the server destination file from the given URL.

Examples

```
put my_filename
put -u www.latimes.com latcopy.html
put -u www.anyware.com/mydownloads/asetup.cgi
  asetup.cgi
```

pwd

Name: `pwd` prints the working directory.

Description: The `pwd` command prints the current working directory to the screen.

Example

```
pwd
```

rm

Name: `rm` removes files or directories.

Syntax: `rm FILE...`

Description: Removes files or directories. Nonempty directories may not be removed (use `rmdir -r` for nonempty directories).

Examples

```
rm myfile.txt
rm */*.html
```

rmdir

Name: `rmdir` deletes a directory.

Syntax: `rmdir [-r] DIRECTORY...`

Options: `-r` recursively deletes a nonempty directory.

Warning: Using `rmdir -r` permanently and completely deletes all data in the given directories. No prompts are given. Use with care.

Description: The `rmdir` command deletes the directories you supply as arguments.

Examples

```
rmdir dir1 dir2 dir3/dir4
rmdir -r my_unwanted_dir
```

run

Name: `run` runs/displays an HTML document.

Syntax `run [ -h] [ FILENAME...] or [ -u URL...]`

Options `-h` forces a local public file to run Aestiva tags.

`-u` runs URLs,

Description Displays an HTML document in a pop-up window. The document may be on the Web or on the local server. By default, local public documents that are not scrambled will not run HTML/OS tags. Local private documents that are scrambled will run HTML/OS tags. If the document is in the public folder, use the `-h` option to force it to run HTML/OS tags.

Examples

```
run start.html index.html
```

```
run -u www.yahoo.com www.aestiva.com
run -h mypublicfile.html
```

set

Name `set` changes Aestiva shell system settings.

Syntax `set [ PARAMETER  VALUE]`

Description If no arguments are given, then the current settings are displayed. Otherwise the first argument should be the parameter name and the second its new value. Parameter (values) descriptions include the following:

`autoscroll` (off/on) toggles scrolling of top frame. Note: this causes some Macintosh browsers to fail.

`backgroundcolor` (color-name) sets the background color. Use a word like `"blue"` or use a hex value like `#ccddee`.

`fontface` (font-name) sets the font face. It is recommended that a fixed-width font face, such as Courier, be used. Make sure that you enclose in double quotes any font names containing spaces.

`fontsize` (1-7) Sets the font size.

`inputcolor` (color-name) sets the color of the shell prompt and user input.

`inputsize` (integer) sets the size of the shell input box.

`outputcolor` (color-name) sets the color of shell output.

`scrollbuffer` (number) sets the number of lines at which the

	shell truncates old output. If a number larger than the number of lines that fit on the screen is used, `autoscroll` should be set to `on`.
	`showbuttons (off/on)` toggles the display of the Help and Desktop buttons.
Examples	`set`

	`set autoscroll on`
	`set inputcolor red`

sort

Name	`sort` sorts lines of text.
Syntax	`sort [OPTIONS] [FILE]`
Options	`-k COLUMN` sorts on column number `COLUMN` as split on spaces.
	`-n` sorts numerically.
	`-r` reverses the sort.
Description	Sorts lines of text. Default sort is on column 1, alphabetically. Columns are split on spaces. `sort` accepts piped input.
Examples	`ls	sort -k 2` sorts output of `ls` alphabetically on column 2.
	`sort myfile -nrk 4` sorts `myfile` in reverse numerical order on column 4.

version

Name	`version` displays the Aestiva Shell version information.
Description	Execute this command to view the version number of the Shell, the credits, and copyright information.
Example	`version`

view

Name	`view` displays an image file in a pop-up window.
Syntax	`view FILE...`
Description	Opens a new pop-up window with an image file (`.gif`, `.jpg`, and so on) for each file argument.

Examples

```
view *.gif
```

```
view myimage.jpg
```

wc

Name
`wc` counts the words and lines in a document.

Syntax
`wc FILE...`

Description
Counts the number of words (spaces) and lines (line breaks) in a document.

Examples

```
wc myfile
```

```
wc *.txt
cat * | wc
```

Index