SYBEX Sample Chapter

Linux Apache Web Server Administration (Craig Hunt Linux Library)

Charles Aulds

Chapter 4: A Basic Apache Server

ISBN: 0-7821-4137-4

Sybex Inc.
1151 Marina Village Parkway
Alameda, CA 94501
U.S.A.
Phone: 510-523-8233
www.sybex.com

4

A Basic Apache Server

When properly installed on a Linux server, Apache is completely configured and ready for use. In this chapter, I'll discuss the configuration, or customization, of a very basic Apache server, suitable for serving Web pages from directories of your choosing. While most administrators will want to go further in extending the functionality of their server, the configuration changes discussed in this chapter are certainly the most common. Nearly every Apache administrator will make at least a few of the changes discussed here.

Apache's behavior and configuration options are defined by using a collection of statements called *directives*. Apache directives are rigorously defined in how and where they can be used, and they have a specific syntax, very much like the commands of a programming language. Directives are not programming commands, though, and using directives is not like programming. Directives are instructions to Apache, telling it how to behave and where to find the resources it needs, but they do not directly control the actions of Apache. Rather, they are used to customize the Apache server for the requirements of a particular Web site (or multiple Web sites hosted on a single server).

Apache directives fall into two groups: those that are always available (the so-called *core directives*) and those supplied by optional add-on *modules*. The most important of these are the core directives. These are the directives that are compiled into the Apache executable, so they are always available and require no special configuration to be used. Other configuration directives become available to the administrator only when their

modules are added to the server. In fact, these directives are meaningless and unrecognized by Apache until their modules are enabled. You can enable these directives when compiling Apache by statically linking the module to the Apache kernel, or at runtime by using the `LoadModule` directive in `httpd.conf`. Chapter 5 is devoted to Apache modules and discusses the use of the `LoadModule` directive to load these as extensions to Apache. Many Apache add-on modules have been adopted by the Apache Software Foundation for inclusion with the Apache distribution, although their use is always optional.

Every directive is associated with a specific module; the largest module is the core module itself, which has special characteristics. This module cannot be unlinked from the Apache kernel and cannot be disabled; the directives it supplies are always available on every Apache server. Most of the directives presented in this chapter are from the core Apache module, and all of the most important directives from the core module are covered. The most important of the remaining modules and their directives are covered in relevant chapters throughout the book. (For example, `mod_proxy` and its directives are presented in Chapter 12's discussion of using Apache as a proxy server.) Apache's online documentation includes a comprehensive reference to all the modules and directives; Appendix D shows how to make effective use of this documentation.

The core module provides support for basic server operations, including options and commands that control the operation of other modules. The Apache server with *just* the core module isn't capable of much at all. It will serve documents to requesting clients (identifying all as having the content type defined by the `DefaultType` directive). While all of the other modules can be considered optional, a useful Apache server will always include at least a few of them. In fact, nearly all of the standard Apache modules are used on most production Apache servers; more than half are compiled by the default configuration and either linked into the server or made available as dynamically loadable modules.

In this chapter, we'll see how directives are usually located in a single startup file (`httpd.conf`). I'll show how the applicability of directives is often confined to a specific scope (by default, directives have a general server scope). Finally, I'll show how directives can be added or overridden on a directory-by-directory basis (using the `.htaccess` file).

Using Apache Directives

The emphasis its developers placed on a modular design has proven to be one of Apache's greatest strengths. From the start, Apache was designed with expandability and extensibility in mind. The hooks that were designed into the program allow devel-

opers to create modules to extend the functionality of Apache, and are an important reason for its rapid adoption and huge success. Apache modules add not only new functionality to the server, but also new directives.

In order to get the most out of Apache, you need to be familiar with all the standard modules. You may not need or use all of these modules, but knowing that they exist, and having a basic knowledge of what each does, is very valuable when needs or problems arise in the future.

Read this chapter in conjunction with the book's appendices. This chapter, like all of the others in this book, is a tutorial that explains when a directive should be used, what it does, and how you can use it in your server configuration. In addition, Appendix A is a tabular list of all directives enabled by the standard set of Apache modules. For each directive, the table includes the context(s) in which the directive is permitted, the Overrides statement that applies to it (if any), and the module required to implement the directive. Appendix D is a detailed guide to using the excellent Apache help system, which should be your first stop when you need to know exactly how a directive is used. In configuring Apache, you will need to make frequent reference to these appendices.

The All-Powerful *httpd.conf* File

In keeping with its NCSA `httpd` origin, Apache originally used three configuration files:

- The main server configuration file, `httpd.conf`
- The resource configuration file, `srm.conf`
- The access permissions configuration file, `access.conf`

The Apache Software Foundation decided to merge these into a single file, and in all current releases of Apache, the only configuration file required is `httpd.conf`. Not only does the use of a single configuration file greatly simplify creating backups and maintaining revision histories, it also makes it easy to describe your complete server configuration to a colleague—just e-mail them a copy of your `httpd.conf`!

> ***TIP*** To follow along with the descriptions in this chapter, you might find it useful to open or print the `httpd.conf` on your system to use for reference. On most systems, the file is stored as `/usr/local/apache2/conf/httpd.conf`. If you have Apache loaded from a Red Hat Linux distribution CD or a Red Hat Package Manager (RPM) distribution, you'll find the file as `/etc/apache2/conf/httpd.conf`. Nearly everything you do to change the Apache configuration requires some modification of this file.

For convenience, the `httpd.conf` file is divided into three sections. Although these divisions are arbitrary, if you try to maintain these groupings, your configuration file will be much easier to read. The three sections of the `httpd.conf` are:

Section 1: The *global environment* section contains directives that control the operation of the Apache server process as a whole. This is where you place directives that control the operation of the Apache server processes, as opposed to directives that control how those processes handle user requests.

Section 2: The *main*, or default, server section contains directives that define the parameters of the "main" or "default" server, which responds to requests that aren't handled by a virtual host. These directives also provide default values for the settings of all virtual hosts.

Section 3: The *virtual hosts* section contains settings for virtual hosts, which allow Web requests to be sent to different IP addresses or hostnames and have them handled by the same Apache server process. Virtual hosts are the subject of Chapter 6.

Directive Scope and Context

One of the important things to know about any directive is the context in which it operates. The context of a directive determines not only its scope—in other words, its range of applicability—but also where the directive can be placed. There are four contexts in which Apache directives can operate:

The General Server Context Directives that operate in the general server context apply to the entire server. Some of these directives are only valid in this context, and make no sense in any other. For example, the `StartServers` directive specifies the number of `httpd` listener processes that are spawned when Apache is first started, and it makes no sense to include this directive in any other context. Other directives (like `ServerName`, which is different for each virtual host) are equally valid in other contexts. When used in the general server context, most of these directives set default values that can be overridden when used in narrower contexts, just as a virtual host will override `ServerName` to set its own value for this directive.

The Container Context This group includes directives that are valid when enclosed in one of the three containers: `<Directory>`, `<Files>`, or `<Location>`. These directives are applicable only within the scope defined by the enclosing container. A good example is a `Deny` directive, which prohibits access to resources.

When used within any one of the three containers mentioned, it denies access to the resource or group of resources defined by the enclosing container.

The Virtual Host Context Although a virtual host is actually defined by the container directive `<VirtualHost>`, for the purpose of defining directive contexts, it is considered separately because many virtual host directives actually override general server directives or defaults. As discussed in Chapter 6, the virtual host attempts to be a second server in every respect, running on the same machine and, to a client that connects to the virtual host, appearing to be the only server running on the machine.

The `.htaccess` Context The directives in an `.htaccess` file are treated almost identically to directives appearing inside a `<Directory>` container in `httpd.conf`. The main difference is that directives appearing inside an `.htaccess` file can be disabled by using the `AllowOverrides` directive in `httpd.conf`. Apache configuration directives are often included in `.htaccess` files in order to decentralize administration. In other words, certain users who don't have access permissions that allow them to edit Apache's `httpd.conf` file might be allowed to edit `.htaccess` files in certain directories. As the name implies, `.htaccess` files are most commonly used to restrict access to resources on a directory basis. The use of security directives in an `.htaccess` context is fully covered in Chapter 13.

NOTE The name of the `.htaccess` file can be changed with the `AccessFileName` directive, though this is rarely done.

For each directive, Appendix A lists the context in which it can be used and the overrides that enable or disable it. For example, the information for the `Action` directive shows that it is valid in all four contexts, but can be overridden when used in an `.htaccess` file by a `FileInfo` override. If the `FileInfo` override is not specified for a directory, an `Action` directive in an `.htaccess` file in that directory has no effect. This is because the `Action` directive is controlled by the `FileInfo` override.

The Apache server is smart enough to recognize when a directive is being specified out of scope. You'll get the following error when you boot, for example, if you attempt to use the `Listen` directive in a `<Directory>` context:

```
# /usr/local/apache2/bin/httpd
Syntax error on line 925 of /usr/local/apache2/conf/httpd.conf:
Listen not allowed here
httpd could not be started
```

Defining the Main Server Environment

General server directives are those used to configure the server itself and its listening processes. General server directives are not allowed in the other contexts we'll discuss, except for virtual hosts. As you'll see in Chapter 6, general server directives also provide default values that are inherited by all virtual hosts, unless specifically overridden.

I changed four directives, all of which were modifications of lines found in the default `httpd.conf` file, to get my server up and running. Once you've installed Apache, you should be able to get the server running by making only these changes, and you probably won't require all four. The default Apache configuration that you installed in Chapter 3 is complete and you can usually start the server using this configuration. However, before doing that, you should understand the purpose of the four directives in this section. These directives, while simple to understand and use, all have a server-wide scope and affect the way many other directives operate. Because of the importance of these directives, you should take care to ensure that they are set properly.

The *ServerName* Directive

Apache must always be able to determine a hostname for the server on which it is run. This hostname is used by the server to create *self-referential URLs*—that is, URLs that refer to themselves. Later, we'll see that when more than one virtual host is run on the same system, each will be identified by a unique `ServerName` directive. For a system that hosts only a single Web server site, the `ServerName` directive is usually set to the hostname and domain of that server.

When I installed Apache and ran it for the first time, I was presented with the error `httpd: cannot determine local host name`. To correct this, I located the `ServerName` directive in my `httpd.conf` file, and discovered that the Apache distribution had created the directive using my fully qualified hostname as the Apache `ServerName`, but left the directive commented out. The directive was acceptable to me, so I uncommented the line:

```
ServerName jackal.hiwaay.net
```

The *ServerRoot* Directive

The `ServerRoot` directive specifies the directory in which the server lives and generally matches the value of the `--prefix` option that was set during the installation of Apache.

```
ServerRoot /usr/local/apache2
```

Typically this directory will contain the subdirectories `bin/`, `conf/`, and `logs/`. In lieu of defining the server root directory using the `ServerRoot` configuration directive, you can also specify the location with the `-d` option when invoking `httpd`:

```
# /usr/local/apache2/bin/httpd -d /etc/httpd
```

While there's nothing wrong with using this method of starting the server, it is usually best reserved for testing alternate configurations and for cases where you will run multiple versions of Apache on the same server simultaneously, each with its own configuration file.

Paths for all other configuration files are taken as relative to this directory. For example, the following directive causes Apache to write error messages into `/usr/local/apache2/logs/error.log`:

```
ErrorLog logs/error.log
```

The `ErrorLog` directive is covered in detail in Chapter 11.

The *DocumentRoot* Directive

The `DocumentRoot` directive is used to define the top-level directory from which Apache will serve files. The directory defined by `DocumentRoot` contains the files that Apache will serve when it receives requests with the URL `/`.

It's perfectly acceptable to use the Apache default, which is the directory `htdocs` under the Apache server root, but I usually prefer to change this to the `/home` filesystem, which is a much larger filesystem reserved for user home directories.

To change the value of `DocumentRoot` on my system, I commented out the Apache default, and added a new `DocumentRoot` directive of my own:

```
# DocumentRoot "/usr/local/apache2/htdocs"
DocumentRoot "/home/httpd/html"
```

Note that a full path to the directory must be used whenever the directory is outside the server root. Otherwise, a relative path can be given. (The double quotes are usually optional, but it's a good idea to always use them. If the string contains spaces, for example, it *must* be enclosed in double quotes.)

When you change the `DocumentRoot`, you must also alter the `<Directory>` container directive that groups all directives that apply to the `DocumentRoot` and subdirectories:

```
# <Directory "/usr/local/apache2/htdocs">
<Directory "/home/httpd/html">
```

The *ScriptAlias* Directive

The `ScriptAlias` directive specifies a directory that contains executable scripts; for example, CGI programs that can be invoked from a Web browser. By default, Apache creates a `ScriptAlias` for all URLs requesting a resource in `/cgi-bin/`.

I also changed the `ScriptAlias` directive for my server. I chose to comment out Apache's default location and add my own, which is located with the Web documents on my `/home` filesystem. There's nothing wrong with the Apache default location (under the

Apache server root directory) for the /cgi-bin directory, but you may want to change the location for ease of maintenance:

```
# ScriptAlias /cgi-bin/ "/usr/local/apache2/cgi-bin/"
ScriptAlias /cgi-bin/ "/home/httpd/cgi-bin/"
```

Make sure that only users specifically authorized to create executable scripts can write to the directory you name. I usually assign group ownership of my cgi-bin directory to a Web administrator's group:I usually assign user and group ownership of the cgi-bin directory (and its subdirectories, if they exist) to user and group accounts different from those under which Apache runs. Then I grant full privileges to this user and group, and grant only read and execute permissions to all other users. This ensures that Apache can read and execute the CGI scripts in this location, but the Apache processes cannot change the existing scripts or create new ones (no "write" permission). The following commands will accomplish this:

```
# chown -r www.webteam /home/httpd/cgi-bin
# chmod 775 /home/httpd/cgi-bin
```

The name of this group is arbitrary, but I use the command shown above to assign ownership of the cgi-bin directory (and all of its subdirectories and files) to a user named nobody and the group webteam. The default behavior of Apache on Linux is to run under the nobody user account. The group name is arbitrary, but it is to this group that I assign membership for those user accounts that are permitted to create or modify server scripts. The second line ensures that the file owner has full read, write, and execute permission, that members of the webteam group have read and write access, and that all other users have no access to the directory or the files it contains.

More General Server Directives

There are a number of other general server directives that you may want to modify before putting your server into operation. These directives are usually acceptable when left at their default values, but changing them rarely carries a significant risk. In all cases, if you feel that you understand the purpose of the directive well enough to add it to your httpd.conf file (or modify it if it's already there), don't be afraid to make such changes. These directives exist to make Apache as customizable as possible; they're for your use.

The *ErrorDocument* Directive

If Apache encounters an error while processing a user request, it is configured to display a standard error page, which gives the HTTP response code (see Chapter 1) and the URL that caused the problem. Use the ErrorDocument directive to define a custom error response to standard HTTP errors that are more user-friendly and understandable. Using ErrorDocument, you can configure Apache to respond to a particular HTTP error code in either of two ways:

1. By displaying custom error text. For example, the following would display a custom message for HTTP Error Code 403 (Forbidden). Note that the text begins with a double-quote that is not part of the message itself; it is not a quote-enclosed string. Do not end the message with a second quote.

```
ErrorDocument 403 "You are not authorized to view this info!
```

2. By issuing a redirect to another URL, which may be external or internal. A fully specified URL that begins with `http://` is assumed to be an external redirect. Apache will send a redirect to the client to tell it where to request the document, even if the redirect resolves to a resource on the same server. A relative URL is a local redirect, relative to the server's `DocumentRoot`, and Apache will serve the request directly, without sending a redirect that would require the client to request the document again. Here are examples of each of the possible redirect forms:

```
#HTTP Error 401 (Unauthorized); display subscription page
ErrorDocument 401 /subscription_info.html

#HTTP Error 404 (Not found); redirect to error script
ErrorDocument 404 /cgi-bin/bad_urls.pl

#HTTP Error 500 (Internal error) Redirect to backup server
ErrorDocument 500 http://jackal2.hiwaay.net
```


NOTE If you attempt to redirect a request with an `ErrorDocument 401` directive (which means that the client is unauthorized to access the requested document), the redirect must refer to a local document. Apache will not permit an external redirect for this HTTP error.

The *DefaultType* Directive

A very rarely used directive in the general server scope, `DefaultType` can redefine the default MIME content type for documents requested from the server. If this directive is not used, all documents not specifically typed elsewhere are assumed to be of MIME type `text/html`. Chapter 15 discusses MIME content types.

Apache reads its MIME-type-to-filename-extension mappings from a file named `mime.types`, which is found in the Apache configuration directory. This file contains a list of MIME content types, each optionally followed by one or more filename extensions. This file is used to determine the MIME content header sent to the client with each resource sent.

The `DefaultType` directive is generally used in a directory scope, to redefine the default type of documents retrieved from a particular directory. In the following example, the default MIME type for all documents in the `/images` directory under `ServerRoot` is defined to be `image/gif`. That way, the server doesn't rely on an extension (like `.gif`) to determine the resource type. A file with no extension at all, when served from this directory, will be sent to the requesting user with an HTTP header identifying it as MIME type `image/gif`.

```
<Directory /images>
   DefaultType image/gif
</Directory>
```

Controlling Server Processes

The following directives are used to control the Linux processes when Apache is run on that platform. All remaining directives control system settings for the Apache server processes.

The *PidFile* Directive

In Chapter 2, I noted that the Linux Apache server usually runs as a pool of listening processes, all of which are under the control of a single main server process that *never* responds to client requests. The `PidFile` directive defines the location and filename of a text file that contains the process ID (or PID) of the main Apache server. Processes that need to know the Apache server process ID—for example, the `apachectl` utility, discussed in Chapter 10—read the PID from this file. It is rarely necessary to change the Apache default PID file, which is stored as `httpd.pid` in the `logs` directory under the Apache `ServerRoot`.

This directive changes the default to place the PID file in the location that Red Hat Linux uses:

```
PidFile "/var/run/apache.pid"
```

NOTE Note that the location of Apache's PID file is determined when Apache is installed. If you change the value of the `PidFile` directive in the original `httpd.conf` file, you will also need to change the location in the `apachectl` utility (discussed in Chapter 10) so that it can find the file in its new location. Open `apachectl` (in the `bin` directory of the Apache installation) with your favorite text editor, and change this file to point to the file in its new location: `PIDFILE=/usr/share/apache2/httpd.pid`.

The *User* Directive

This directive defines the Linux user that owns the child processes created to handle user requests. This directive is meaningful only when the Apache server is started as `root`. If the server is started as any other user, it cannot change ownership of child processes.

The default behavior of Apache on Linux systems is to change the ownership of all child processes to `UID -1` (which corresponds to user `nobody` in the standard Linux `/etc/password` file). This is the preferred way to run Apache on Linux systems.

In the following example, I've chosen to run Apache as `www`, a special Web-specific user account that I create on all my Web servers. For ease of administration, Apache resources on my server are usually owned by user `www` and group `wwwteam`.

```
User www
```

The *Group* Directive

Like the `User` directive, this directive is used to change the ownership of the child processes created to handle user requests. Instead of changing the user ownership of these processes, however, this directive changes the group ownership.

The default behavior of Apache on Linux systems is to change the group ownership of all child process to *group ID* (GID) –1, which corresponds to the `nobody` group in `/etc/groups`. It often makes more sense to change the group ownership of the Apache server processes than it does to change the user ownership. On Linux servers where I want to give several users read/write access to my Apache configuration files and Web resources, I normally set up a special group with a name like `webteam`:

```
Group webteam
```

I place all the Web developers' accounts in this group and also change the Apache configuration to run server processes owned by this group.

As it must with the `User` directive, a stand-alone Apache server must be started as `root` to use the group directive. Otherwise, the server can't change the group ownership of any child processes it spawns.

The *Listen* Directive

Apache now provides just one core directive, the `Listen` directive, to define the IP addresses and TCP port numbers on which it listens for and accepts client connections.

The `Listen` directive incorporates all of the functionality of the `BindAddress` and `Port` directives that existed in Apache versions earlier than 2.0, but has several important advantages over them (in addition to eliminating the confusing overlap of functionality between the older directives). The `Listen` directive has a global server scope and has no meaning inside a container. Apache is smart enough to detect and warn if the `Listen` directive is used in the wrong context. Placing a `Listen` directive inside a virtual host container, for example, generates this error:

```
Syntax error on line 1264 of /usr/local/apache/conf/httpd.conf:
Listen cannot occur within <VirtualHost> section
```

If `Listen` specifies only a port number, the server listens to the specified port on all system network interfaces. If a single IP address and a single port number are given, the server listens only on that port and interface.

Multiple `Listen` directives may be used to specify more than one address and port to listen to. The server will respond to requests from any of the listed addresses and ports. For example, to make the server accept connections on both port 80 and port 8080, use these directives:

```
Listen 80
Listen 8080
```

To make the server accept connections on two specific interfaces and port numbers, identify the IP address of the interface and the port number and separate them by a colon, as in this example:

```
Listen 192.168.1.3:80
Listen 192.168.1.5:8080
```

Although `Listen` is very important in specifying multiple IP addresses for IP-based virtual hosting (discussed in detail in Chapter 6), the `Listen` directive does *not* tie an IP address to a specific virtual host. Here's an example of the Listen directive used to instruct Apache to accept connections on two interfaces, each of which uses a different TCP port.

```
Listen 192.168.1.1:80
Listen 216.180.25.168:443
```

I use this configuration to accept ordinary HTML requests on port 80 on my internal network interface; connections on my external interface (from the Internet) are accepted only on TCP port 443, the default port for SSL connections (as we'll see in Chapter 14).

The *Options* Directive

The `Options` directive controls which server features are available in a particular directory. The value can be set to `None`, in which case none of the extra features is enabled, or to one or more of the following:

`ExecCGI` Permits execution of CGI scripts.

`FollowSymLinks` The server will follow symbolic links (symlinks) in this directory. Following symlinks does not change the pathname used to match against `<Directory>` sections. This option is ignored if set inside a `<Location>` section.

`Includes` Permits Server-Side Includes (SSI).

`IncludesNOEXEC` Server-Side Includes are permitted, but the `#exec` and `#include` commands of SSI scripts are disabled.

`Indexes` If a URL that maps to a directory is requested, and there is no `DirectoryIndex` (for example, `index.html`) in that directory, then the server will return a formatted listing of the directory.

`MultiViews` Allows content-negotiated MultiViews. As discussed in Chapter 15, MultiViews are one means of implementing content negotiation.

`All` Includes all options except for MultiViews. This is the default setting.

`SymLinksIfOwnerMatch` The server will only follow symbolic links for which the target file or directory is owned by the same user ID as the link. Like `FollowSymLinks`, this option is ignored if set inside a `<Location>` section.

Normally, if multiple options apply to a directory, the most specific one is used and the other options are ignored. However, if all the options on the `Options` directive are preceded by a plus (+) or minus (–) character, then the options are merged. Any options preceded by a plus are added to the options currently in effect, and any options preceded by a minus are removed from the options currently in effect.

Since the default setting for the `Options` directive is `All`, the configuration file that is provided with Apache contains the following section, which enables only `FollowSymLinks` for every directory onunder Apache's `DocumentRoot`.

```
<Directory />
    Options FollowSymLinks
</Directory>
```

The following examples should clarify the rules governing the merging of options. In the first example, only the option `Includes` will be set for the `/web/docs/spec` directory:

```
<Directory /web/docs>
```

```
    Options Indexes FollowSymLinks
</Directory>
<Directory /web/docs/spec>
      Options Includes
</Directory>
```

In the example below, only the options `FollowSymLinks` and `Includes` are set for the `/web/docs/spec` directory:

```
<Directory /web/docs>
   Options Indexes FollowSymLinks
</Directory>
<Directory /web/docs/spec>
   Options +Includes -Indexes
</Directory>
```

Using either `-IncludesNOEXEC` or `-Includes` disables Server-Side Includes. Also, the use of a plus or minus sign to specify a directive has no effect if no options list is already in effect. No options list will be created if only these modifiers are used to specify new options.

> ***WARNING*** Be aware that the default setting for Options is All. For that reason, you should always ensure that this default is overridden for every Web-accessible directory. The default configuration for Apache includes a `<Directory>` container to do this; do not modify or remove it.

The Container Directives

The scope of an Apache directive is often restricted using special directives called *container directives*. In general, these container directives are easily identified by the enclosing <> brackets. The *conditional directives* `<IfDefine>` and `<IfModule>`, which are not container directives, are an exception. Container directives require a closing directive that has the same name and begins with a slash character (much like HTML tags).

A container directive encloses other directives and specifies a limited scope of applicability for the directives it encloses. A directive that is not enclosed in a container directive is said to have global scope and applies to the entire Apache server. A global directive is overridden locally by the same directive when it is used inside a container. The following sections examine each type of container directive.

The *<VirtualHost>* Container

The <VirtualHost> container directive encloses directives that apply only to a specific *virtual host*. As discussed further in Chapter 6, a virtual host is a Web site hosted on your server that is identified by a hostname alias. For example, assume your server is www.aulds.com and that it hosts a Web site for a local bait and tackle shop. That shop, however, does not want its customers connecting to www.aulds.com for information; it wants customers to use the Web site www.worms.com. You can solve this problem by creating a virtual host for www.worms.com on the real host www.aulds.com. The format of the <VirtualHost> container directive is

```
<VirtualHost address>
   directives
</VirtualHost>
```

The directives you enclose in the <VirtualHost> container will specify the correct hostname and document root for the virtual host. Naturally, the server name should be a value that customers of the Web site expect to see when they connect to the virtual host. Additionally, the file served to the customers needs to provide the expected information. In addition to these obvious directives, almost anything else you need to customize for the virtual host can be set in this container. For example:

```
<VirtualHost 192.168.1.4>
   ServerAdmin webmaster@host1.com
   DocumentRoot /home/httpd/wormsdocs
   ServerName www.worms.com
   ErrorLog logs/worms.log
   TransferLog logs/worms.log
</VirtualHost>
```

The example above defines a single virtual host. In Chapter 6, we'll see that this is one form of virtual host, referred to as *IP-based*. The first line defines the Internet address (IP) for the virtual host. Only connections made to the Apache server on this IP address are handled by the virtual server for this site, which might be only one of many virtual sites being hosted on the same server. Each directive defines site-specific values for configuration parameters that, outside a <VirtualHost> container directive, normally refer to the entire server. The use of each of these in the general server context was shown earlier in this chapter.

The *<Directory>* and *<DirectoryMatch>* Containers

The <Directory> container encloses directives that apply to a filesystem directory and its *subdirectories*. The directory must be expressed by its full pathname or with wildcards. The example below illustrates a <Directory> container that sets the Indexes and FollowSymLinks options for all directories under /home/httpd/ that begin with user:

```
<Directory /home/httpd/user*>
```

```
    Options Indexes FollowSymLinks
</Directory>
```

<Directory> containers are always evaluated so that the shortest match (widest scope) is applied first, and longer matches (narrower scope) override those that may already be in effect from a wider container. For example, the following container disables all overrides for every directory on the system (/ and all its subdirectories):

```
<Directory />
    AllowOverrides None
</Directory>
```

If the httpd.conf file includes a second <Directory> container that specifies a directory lower in the filesystem hierarchy, the directives in the container take precedence over those defined for the filesystem as a whole. The following container enables FileInfo overrides for all directories under /home (which hosts all user home directories on most Linux systems):

```
<Directory /home/*>
    AllowOverrides FileInfo
</Directory>
```

The <Directory> container can also be matched against regular expressions by using the ~ character to force a regular expression match:

```
<Directory ~ "^/home/user[0-9]{3}">
```

The <DirectoryMatch> directive is specifically designed for regular expressions, however, and should normally be used in place of this form. This container directive is exactly like <Directory>, except that the directories to which it applies are matched against regular expressions. The following example applies to all request URLs that specify a resource beginning with /user and followed by exactly three digits. (The ^ character denotes "beginning of string," and the {3} means to match exactly three occurrences of the previous character—in this case, any member of the character set [0–9].)

```
<DirectoryMatch "^/user[0-9]{3}">
    order deny,allow
    deny from all
    allow from .foo.com
</Directory>
```

This container directive would apply to a request URL like the following:

```
http://jackal.hiwaay.net/user321
```

because the <DirectoryMatch> container directive looks for directories (relative to DocumentRoot) that consist of the word *user* followed by three digits.

Introduction to Regular Expressions

Many Apache configuration directives accept regular expressions for matching patterns. Regular expressions are an alternative to wildcard pattern matching and are usually an extension of a directive's wildcard pattern matching capability. Indeed, I have heard regular expressions (or *regexps*) described as "wildcards on steroids."

A brief sidebar can hardly do justice to the subject, but to pique your interest, here are a few regexp tags and what they mean:

^ and $ — Two special and very useful tags that mark the beginning and end of a line. For example, `^#` matches the # character whenever it occurs as the first character of a line (very useful for matching comment lines), and `#$` would match # occurring as the very last character on a line. These pattern-matching operators are called *anchoring operators* and are said to "anchor the pattern" to either the beginning or the end of a line.

* and ? — The * character matches the preceding character zero or more times, and ? matches the preceding pattern zero or one time. These operators can be confusing, because they work slightly differently from the same characters when used as "wildcards." For example, the expression `fo*` will match the pattern *foo* or *fooo* (any number of *o* characters), but it also matches *f*, which has zero *o*'s. The expression `ca?` will match the *c* in *score*, which seems a bit counterintuitive because there's no a in the word, but the `a?` says zero or one *a* character. Matching zero or more occurrences of a pattern is usually important whenever that pattern is optional. You might use one of these operators to find files that begin with a name that is optionally followed by several digits and then an extension. Matching for `^filename\d*.gif` will match *filename001.gif* and *filename2.gif*, but also simply *filename.gif*. The `\d` matches any digit (0–9); in other words, we are matching zero or more digits.

+ — Matches the preceding character one or more times, so `ca+` will not match *score*, but will match *scare*.

. — The period character matches any single character except the newline character. In effect, when you use it, you are saying you don't care what character is matched, as long as some character is matched. For example, `x.y` matches *xLy* but not *xy*; the period says the two must be separated by a single character. The expression `x.*y` says to match an *x* and a *y* separated by zero or more characters.

{*n*} — This operator (a number between braces) matches *n* occurrences of the preceding character. For example, `so{2}` matches *soot*, but not *sot*.

Introduction to Regular Expressions (continued)

If you're an experienced Linux system administrator, you're already familiar with regular expressions from using `grep`, `sed`, and `awk`. And if you're an experienced Perl user, you probably also have some knowledge of regular expressions. The GNU C++ Regular Expressions Library and Windows Scripting Host (WSH) even allow expressions in Microsoft's JavaScript or VBScript programs.

The only way to develop proficiency in using regexps is to study examples and experiment with them. Entire books have been written on the power of regular expressions (well, at least one) for pattern matching and replacement.

Some useful resources on regexps:

Mastering Regular Expressions Second Edition, by Jeffrey E.F. Friedl (O'Reilly, 2002)

`http://www.delorie.com/gnu/docs/regex/regex_toc.html`

`http://lib.stat.cmu.edu/scgn/v52/section1_7_0_1.html`

The *<Files>* and *<FilesMatch>* Containers

The `<Files>` container encloses directives that apply only to specific files, which should be specified by filename (using wildcards when necessary). The following example allows access to files with the `.OurFile` extension only by hosts in a specific domain:

```
<Files *.OurFile>
   order deny,allow
   deny from all
   allow from .thisdomain.com
</Files>
```

Like the `<Directory>` container, `<Files>` can also be matched against regular expressions by using the `~` character to force a regular expression match. The following line, for example, matches filenames that end in a period character (escaped with a backslash) immediately followed by the characters *xml*. The `$` in regular expressions denotes the end of the string. Thus we are looking for filenames with the extension `.xml`.

```
<Files ~ "\.xml$">
   Directives go here
</Files>
```

The `<FilesMatch>` directive is specifically designed for regular expressions, however, and should normally be used in place of this form.

<FilesMatch> is exactly like the <Files> directive, except that the specified files are defined by regular expressions. All graphic images might be defined, for example, using:

```
<FilesMatch  "\.(gif|jpe?g|png)$">
   some directives
</FilesMatch>
```

This regular expression matches filenames with the extension *gif* or *jpg* or *jpeg* or *png*. The *or* is denoted by the | (vertical bar) character. Notice the use of the ? character after the e, which indicates zero or one occurrences of the preceding character (e). In other words, a match is made to *jp*, followed by zero or one *e*, followed by *g*.

The *<Location>* and *<LocationMatch>* Containers

The <Location> container encloses directives that apply to specific URLs. This is similar to <Directory>, because most URLs contain a reference that maps to a specific directory relative to Apache's DocumentRoot. The difference is that <Location> does not access the filesystem, but considers only the URL of the request. Most directives that are valid in a <Directory> context also work in a <Location> container; directives that do not apply to a URL are simply ignored because they are meaningless in a <Location> context.

The <Location> functionality is especially useful when combined with the SetHandler directive. For example, to enable status requests, but only from browsers at foo.com, you might use the following (note that *status* is not a directory; it is a part of the URL, and actually invokes a server-generated status page:

```
<Location /status>
   SetHandler server-status
   order deny,allow
   deny from all
   allow from .foo.com
</Location>
```

You can also use extended regular expressions by adding the ~ character, as described for the <Directory> and <Files> container directories; but a special container directive, <LocationMatch>, is specifically designed for this purpose and should be used instead.

<LocationMatch> is exactly like the <Location> container directive, except that the URLs are specified by regular expressions. The following container applies to any URL that contains the substring */www/user* followed immediately by exactly three digits; for example, */www/user101*:

```
<LocationMatch "/www/user[0-9]{3}">
    order deny,allow
    deny from all
    allow from .foo.com
</Location>
```

The *<Limit>* and *<LimitExcept>* Containers

<Limit> encloses directives that apply only to the HTTP methods specified. The <Limit> directive can be applied to any HTTP request method discussed in Chapter 1. In the following example, user authentication is required only for requests that use the HTTP methods POST, PUT, and DELETE. Any other request method (GET, for example) will not result in a prompt for username and password:

```
<Limit POST PUT DELETE>
   require valid-user
</Limit>
```

<LimitExcept> encloses directives that apply to all HTTP methods *except* those specified. The following example shows how authentication can be required for all HTTP methods other than GET:

```
<LimitExcept GET>
   require valid-user
</Limit>
```

Perl Sections

If you are using the mod_perl module, it is possible to include Perl code to automatically configure your server. Sections of the httpd.conf containing valid Perl code and enclosed in special <Perl> container directives are passed to mod_perl's built-in Perl interpreter. The output of these scripts is inserted into the httpd.conf file before it is parsed by the Apache engine. This allows parts of the httpd.conf file to be generated dynamically, possibly from external data sources like a relational database on another machine.

Since this option absolutely requires the use of mod_perl, it is discussed in far more detail with this sophisticated module in Chapter 8.

Apache's Order of Evaluation for Containers

When multiple containers apply to a single incoming request, Apache resolves them in the following order:

1. Apache will first evaluate any <Directory> container (except for those that match regular expressions) and merge any .htaccess files it finds that apply to the request. <Directory> containers are always evaluated from widest to narrowest scope, and directives found in .htaccess files override those in <Directory> containers that apply to the same directory.
2. Directives found in <DirectoryMatch> containers and <Directory> containers that match regular expressions are evaluated next. Directives that apply to

the request override those in effect from <Directory> or .htaccess files (item 1 of this list).

3. After directives that apply to the directory in which the resource resides, Apache applies directives that apply to the file itself. These come from <Files> and <FilesMatch> containers, and they override directives in effect from <Directory> containers. For example, if an .htaccess file contains a directive that denies the requester access to a directory, but a directive in a <Files> container specifically allows access to the file, the request will be granted, because the contents of the <Files> container override those of the <Directory> container.
4. Finally, any directives in <Location> or <LocationMatch> containers are applied. These directives are applied to the request URL and override directives in all other containers. If a directive in a <Location> container directly conflicts with the same directive in either a <Directory> or a <Files> container, the directive in the <Location> container will override the others.

Containers with narrower scopes always override those with a wider scope. For example, directives contained in <Directory /home/httpd/html> override those in <Directory /home/httpd> for the resources in its scope. If two containers specify exactly the same scope (for example, both apply to the same directory or file), the one specified last takes precedence.

The following rather contrived example illustrates how the order of evaluation works:

```
<Files index.html>
   allow from 192.168.1.2
</Files>

<Directory /home/httpd/html>
   deny from all
</Directory>
```

In this example, the <Directory> container specifically denies access to the /home/httpd/html directory to all clients. The <Files> directive (which precedes it in the httpd.conf file) permits access to a single file index.html inside that directory, but only to a client connecting from IP address 192.168.1.2. This permits the display of the HTML page by that client, but not any embedded images; these can't be accessed because the <Files> directive does not include them in its scope. Note also that the order of the containers within the configuration file is *not* important; it is the order in which the containers are resolved that determines which takes precedence. Any <Files> container directives will always take precedence over <Directory> containers that apply to the same resource(s).

The *.htaccess* File

Although an Apache server is usually configured completely within the `httpd.conf` file, editing this file is not always the most efficient configuration method. Most Apache administrators prefer to group directory-specific directives, particularly access-control directives, in special files located within the directories they control. This is the purpose of Apache's `.htaccess` files.

In addition to the convenience of having all the directives that apply to a specific group of files located within the directory that contains those files, `.htaccess` files offer a couple of other advantages. First, you can grant access to modify `.htaccess` files on a per-directory basis, allowing trusted users to modify access permissions to files in specific directories without granting those users unrestricted access to the entire Apache configuration. Second, you can modify directives in `.htaccess` files without having to restart the Apache server (which is the only way to read a modified `httpd.conf` file).

By default, the Apache server searches for the existence of an `.htaccess` file in every directory from which it serves resources. If the file is found, it is read and the configuration directives it contains are merged with other directives already in effect for the directory. Unless the administrator has specifically altered the default behavior (using the `AllowOverrides` directive as described below), all directives in the `.htaccess` file override directives already in effect. For example, suppose `httpd.conf` contained the following `<Directory>` section:

```
<Directory /home/httpd/html/Special>
   order deny,allow
   deny from all
</Directory>
```

All access to the directory `/home/httpd/html/Special` would be denied. This may be exactly what the administrator wants, but it is more likely that the directory exists under the Web server root so that someone can get to it with a browser. This can be accomplished by creating an `.htaccess` file in the `Special` directory with directives like the following, which overrides the directives already active for the directory:

```
allow from 192.168.1.*
```

Here, we've used a wildcard expression to specify a range of IP addresses (possibly the Web server's local subnet) that can access resources in the `Special` directory.

The *AllowOverrides* Directive

By default, whenever Apache receives a request for a resource, it searches for an `.htaccess` file in the directory where that resource resides and in every parent directory of

that directory on the filesystem. Remember, this search is not limited to `DocumentRoot` and its subdirectories, but extends all the way up the filesystem hierarchy to the root directory ("/"). It treats each of these exactly as if it were a `<Directory>` container for the directory in which it is located. The directives in all `.htaccess` files found in the requested resource's tree are merged with any other directives already in effect for that directory. Those lower in the filesystem hierarchy override those higher in the tree; this means you can grant permission to access a directory even if that permission was denied to a higher-level directory (and, consequently, all of its subdirectories). After merging all the relevant `.htaccess` files with all directives from all applicable `<Directory>` containers, Apache applies them according to the order of evaluation described earlier.

What I've just described is the default behavior of Apache with regard to `.htaccess` files. You can modify this behavior through the special directive `AllowOverrides`, which controls how `.htaccess` files are handled. The `AllowOverrides` directive specifies which directives, when found in an `.htaccess` file, are allowed to override conflicting directives that are already in effect. `AllowOverrides` is not used to enable or disable directives, but to specify types of directives that can be overridden in `.htaccess` files.

The following is a list of all permissible arguments to the `AllowOverrides` directive. Each enables or disables a set of directives when these directives are found in `.htaccess` files. Consult the table in Appendix A for the applicable `AllowOverrides` for each directive for which an override can be specified; the `AllowOverrides` directive does not apply to directives shown in that table with "N/A" in the Override column.

`All` This enables all `.htaccess` overrides. Therefore, all directives that are permissible in an `.htaccess` file can be used to override settings in the `httpd.conf` file.

WARNING The default behavior of Apache is to search for `.htaccess` files in each directory in the path of a resource as if `AllowOverrides All` had been specified for all directories. This makes the server hard to secure, because anyone who can write a file into any of the directories from which Apache serves files can create a bogus `.htaccess` file that can be used to subvert system security. It is always best to use `AllowOverrides` to disable `.htaccess` files in all directories, enabling the use of `.htaccess` files only for specific purposes and locations, on a case-by-case basis. Disabling the search for `.htaccess` files also has the added benefit of improving Apache performance (as discussed in Chapter 12).

`None` This disables `.htaccess` overrides. If `AllowOverrides None` is specified for a directory, Apache will not read an `.htaccess` even if it exists in that direc-

tory. If `AllowOverrides None` is specified for the system `root` ("/") directory, no directory will ever be searched for an `.htaccess` file.

`Authconfig` Allows the use of all user/group authorization directives (`Authname`, `Authuserfiles`, `Authgroupfile`, `Require`), which are discussed in detail in Chapter 13.

`FileInfo` Allows the use of directives controlling document types.

`Indexes` Allows the use of directives controlling directory indexing.

`Limit` Allows the use of directives that control access based on the browser hostname or network address.

`Options` Allows the use of special directives, currently limited to the directives `Options` and `XBitHack`.

Setting Up User Home Directories

In nearly every server used to support multiple users, it is useful to provide individual users with their own Web home directories. This is a very common practice among Internet Service Providers that support Web hosting for their users. Providing user home directories is similar to virtual hosting in some respects, but it is much simpler to implement. The functionality is provided by a standard Apache module (`mod_userdir`) that is available to the Apache server by default.

Specifying Username-to-Directory Mappings

If you intend to allow users to publish their own Web pages, the `UserDir` directive indicates the name of a directory that, if found in the users' home directories, contains Web pages that are accessed with a URL of the form `http://`*`hostname`*`/~`*`username`*`/`, where *`username`* is a Linux user account on the server (or virtual server) *`hostname`*. The Apache default is to name this directory `public_html`. There is absolutely nothing wrong with this default value, but for years, since I first administered a CERN 3.0 server, I have chosen to name this directory `WWW`. A simple change to the `UserDir` directive in `httpd.conf` let me reconfigure this value for all users on the server:

```
UserDir WWW
```

Now once I add this line to Apache's `httpd.conf` file and restart the server, each user on my system can place files in a `~/WWW` subdirectory of their home directory that Apache can serve. Requests to a typical user's Web files look like:

```
http://jackal.hiwaay.net/~caulds/index.html
```

The `UserDir` directive specifies a filename or pattern that is used to map a request for a user home directory to a special repository for that user's Web files. The `UserDir` directive can take one of three forms:

A Relative Path This is normally the name of a directory that, when found in the user's home directory, becomes the `DocumentRoot` for that user's Web resources:

```
UserDir public_html
```

This is the simplest way to implement user home directories, and the one I recommend because it gives each user a Web home underneath their system home directories. This form takes advantage of the fact that `~`*`username`* is always Linux shorthand for "user account's home directory." By specifying users' home directories as a relative path, the server actually looks up the user's system home (in the Linux `/etc/passwd` file) and then looks for the defined Web home directory beneath it).

WARNING Be careful when using the relative path form of the `UserDir` directive. It can expose directories that shouldn't be accessible from the Web. For example, when using the form `http://`*`hostname`*`/~root/`, the Linux shortcut for `~root` maps to a directory in the filesystem reserved for system files on most Linux systems. If you had attempted to designate each user's system home directory as their Web home directory (using `UserDir/`), this request would map to the `/root` directory. When using the relative directory form to designate user Web home directories, you should lock out any accounts that have home directories on protected filesystems (see the section titled "Enabling/Disabling Mappings" later in this chapter). The home directory of the root account (or *superuser*) on Linux systems should be protected. If someone was able to place an executable program in one of root's startup scripts (like `.profile` or `.bashrc`), that program would be executed the next time a legitimate user or administrator logged in using the `root` account.

An Absolute Path An absolute pathname is combined with the username to identify the `DocumentRoot` for that user's Web resources:

```
UserDir /home/httpd/userstuff
```

This example would give each user their own directory with the same name as their user account underneath `/home/httpd/userstuff`. This form gives each user a Web home directory that is *outside* their system home directory. Maintaining a special directory for each user, outside their system home directory, is

not a good idea if there are a lot of users. They won't be able to maintain their own Web spaces as they could in their respective home directories, and the entire responsibility will fall on the administrator. Use the absolute form for defining user Web home directories only if you have a small number of users, preferably where each is knowledgeable enough to ensure that their Web home directory is protected from other users on the system.

An Absolute Path with Placeholder An absolute pathname can contain the * character (called a *placeholder*), which is replaced by the username when determining the `DocumentRoot` path for that user's Web resources. Like the absolute path described above, this form can map the request to a directory outside the user's system home directory:

```
UserDir  /home/httpd/*/www
```

Apache substitutes the username taken from the request URL of the form `http://`*`hostname`*`/~`*`username`*`/` to yield the path to each user's Web home directory, as in this example:

```
/home/httpd/caulds/www
```

If all users have home directories under the same directory, the placeholder in the absolute path can mimic the relative path form, by specifying

```
UserDir  /home/*/www
```

The behavior of the lookup is slightly different, though, using this form. In the relative path form, the user's home directory is looked up in `/etc/passwd`. In the absolute path form, this lookup is not performed, and the user's Web home directory must exist in the specified path. The advantage of using the absolute path in this manner is that it prevents URLs like `http://`*`hostname`*`/~root` from mapping to a location that Web clients should never access.

The disadvantage of using the "absolute path with placeholder" form is that it forces all Web home directories to reside under one directory that you can point to with the absolute path. If you needed to place user Web home directories in other locations (perhaps even on other filesystems) you will need to create symbolic links that point the users' defined Web home directories to the actual location of the files. For a small to medium-sized system, this is a task that can be done once for each user and isn't too onerous, but for many users, it's a job you might prefer to avoid.

The use of the `UserDir` directive is best illustrated by example. Each of the three forms of the directive described above would map a request for

```
http://jackal.hiwaay.net/~caulds/index.html
```

into the following fully qualified path/filenames, respectively:

1. `~caulds/public_html/index.html`
2. `/home/httpd/userstuff/caulds/index.html`
3. `/home/httpd/caulds/www/index.html`

Redirecting Requests for User Home Directories

Chapter 9 provides a detailed discussion of Apache's tools for redirection, but the topic is worth a quick preview here, in the context of user home directories.

A server cannot *force* a browser to retrieve a resource from an alternate location. It sends a status code showing that the server couldn't respond to the browser's requests and a `Location` directive indicating an alternate location. The browser is politely asked to redirect its request to this alternate location. In the case of `UserDir`, the server issues a redirect request to the client, which will in all likelihood request the resource again from the specified alternate location, and the user is none the wiser. The argument to `UserDir` can also take the form of a URL rather than a directory specification, in which case the mapping is sent back to the client as a redirect request. This is most useful when redirecting requests for users' home directories to other servers. The following `UserDir` directive:

```
UserDir http://server2.hiwaay.com/~*/
```

would cause a request for

```
http://jackal.hiwaay.net/~caulds/docfiles/index.html
```

to generate a URL redirect request that would send the requester to the following resource, which is on a separate server:

```
http://server2.hiwaay.net/~caulds/docfiles/index.html
```

Enabling/Disabling Mappings

Another form of the `UserDir` directive uses the keywords `enabled` or `disabled` in one of three ways.

First,

```
UserDir disabled <username1 username2 …>
```

disables username-to-directory mappings for the space-delimited list of usernames. Example:

```
UserDir disabled root webmaster
```

> ***WARNING*** If you are running a 1.3 version of Apache, it is strongly recommended that your configuration include a `UserDir disabled root` declaration.

Next, using the `disabled` keyword without username:

```
UserDir disabled
```

turns off all username-to-directory mappings. This form is usually used prior to a `UserDir enabled` directive that explicitly lists users for which mappings are performed.

And finally,

```
UserDir enabled <username1 username2 …>
```

enables username-to-directory mappings for the space-delimited list of usernames. It usually follows a `UserDir disabled` directive that turns off username-to-directory mappings for all users (all are normally enabled). Example:

```
UserDir disabled
UserDir enabled caulds csewell webteam
```

Using suEXEC with User Directories

Most sites that support user directories also allow users to create and run their own CGI processes. It is easy to see how allowing users to write and run CGI programs that run with the permissions of the Web server could be disastrous. Such a script would have the same access privileges that the Web server itself uses, and this is normally not a good thing. To protect the Web server from errant or malicious user-written CGI scripts, and to protect Web users from one another, user CGI scripts are usually run from a program called a CGI wrapper. A *CGI wrapper* is used to run a CGI process under different user and group accounts than those that are invoking the process. In other words, while ordinary CGI processes are run under the user and group account of the Apache server (by default, that is user `nobody` and group `nobody`), using a CGI wrapper, it is possible to invoke CGI processes that run under different user and group ownership. In most cases, this is used to restrict access by a script to only files and directories that are accessible to the effective owner of the script, that is, the user account under which suEXEC is running the script. This is most often used to give ordinary (nonprivileged) system users the opportunity to run CGI scripts that can have access only to other files owned by that user.

There are several such CGI wrappers, but one program of this type called suEXEC is a standard part of Apache in all versions after version 1.2 (though not enabled by the default installation). SuEXEC is very easy to install, and even easier to use. There are two ways in which suEXEC is useful to Apache administrators. The most important use

for suEXEC is to allow users to run CGI programs from their own directories that run under their user and group accounts, rather than that of the server.

The second way in which suEXEC is used with Apache is with virtual hosts. When used with virtual hosts, suEXEC changes the user and group accounts under which all CGI scripts defined for each virtual host are run. This is used to give virtual host administrators the ability to write and run their own CGI scripts without compromising the security of the primary Web server (or any other virtual host).

Configuring Apache to Use suEXEC

The suEXEC tool is very easy to set up using the APACI installation script. APACI's `configure` script is provided with a number of options that are used to configure suEXEC. The most important of these is `--enable-suexec`, which is required to enable suEXEC. All of the other options have default values that you can find by peeking into the makefile in the top Apache source directory. On my system, I chose to use all the available options when running `configure`. Even when the default values are acceptable, I include them in my `build.sh` script, borrowing the default values from the makefile and modifying them where I desire.

Listing 4.1 shows the complete `build.sh` script I used to build Apache version 2.0.36 with suEXEC support. The lines that begin with `--with-suexec-` are a little different than most configure options. Instead of defining the Apache configuration, they provide values that are compiled into the resulting `suexec` wrapper as a security measure. As "hard-coded" values, defined at compile-time, they can't be overridden with directives or command-line arguments. They are fixed until `suexec` is recompiled (or replaced by a bogus version of `suexec` by a malicious party, which is your biggest security risk using `suexec`). The default permissions on `suexec` (and the `bin` directory in which it is placed during the install) should not be changed. They ensure that only a user with administrative rights can alter or replace the wrapper.

Listing 4.1 A `build.sh` Script for Building Apache 2.0.36 with suEXEC Support

```
#!/bin/sh
##
##  config.status -- APACI auto-generated configuration restore script
##
##  Use this shell script to re-run the APACI configure script for
##  restoring your configuration. Additional parameters can be supplied.
##

##LIBS=`perl -MExtUtils::Embed -e ldopts` \
##CFLAGS=`perl -MExtUtils::Embed -e ccopts` \
./configure \
```

```
"--with-layout=Apache" \
"--enable-mods-shared=all" \
"--enable-ssl" \
"--enable-proxy" \
"--enable-proxy-http" \
"--enable-proxy-ftp" \
"--enable-cache" \
"--enable-disk-cache" \
"--enable-mem-cache" \
"--enable-suexec" \
"--with-suexec-caller=nobody" \
"--with-suexec-docroot=/home/httpd/" \
"--with-suexec-logfile=/usr/local/apache2/logs/suexec_log" \
"--with-suexec-userdir=public_html" \
"--with-suexec-uidmin=100" \
"--with-suexec-gidmin=100" \
"--with-suexec-safepath=/usr/local/bin:/usr/bin:/bin" \
"$@"
```

To build and install Apache with suEXEC, I enter three lines in the Apache source directory (`build.sh` is a shell script that contains the lines shown above):

```
# sh build.sh
# make
# make install
```

Apache will still start, even if suEXEC is unavailable, but suEXEC will be disabled. You have to keep an eye on this; it is unfortunate that, when suEXEC is disabled, no warning is given when Apache is started, and nothing is written into Apache's error log. The error log will only show when suEXEC is enabled. You can check inside Apache's error log (which is in `logs/error.log` under the Apache installation directory, unless you've overridden this default value). If all is OK, the error log will contain the following line, with a timestamp, usually just before the line indicating that Apache has been started:

```
suEXEC mechanism enabled (wrapper: /usr/local/apache2/bin/suexec)
```

If suEXEC is not enabled when Apache is started, verify that you have the `suexec` wrapper program, owned by `root`, in Apache's `bin` directory:

```
# ls -al /usr/local/apache2/bin/suexec
-rwsr-xr-x    1 root     root   57543 Jun  5 20:19 suexec
```

Note the `s` in the user permissions. This indicates that the `setuid` bit is set. In other words, the file, when executed, will run under the user account of the file's owner. For example, the Apache `httpd` process that invokes `suexec` will probably be running under the `nobody` account. The `suexec` process it starts, however, will run under the `root`

account, because root is the owner of the file suexec. Only root can invoke the Linux setuid and setgid system functions to change the ownership of processes it spawns as children (the CGI scripts that run under its control). If suexec is not owned by root, and does not have its user setuid bit set, correct this by entering the following lines while logged in as root:

```
# chown root /usr/local/apache2/bin/suexec
# chmod u+s /usr/local/apache2/bin/suexec
```

If you wish to disable suEXEC, the best way is to simply remove the user setuid bit:

```
# chmod u-s /usr/local/apache2/bin/suexec
```

This not only disables suEXEC, but it also renders the suEXEC program a bit safer because it will no longer run as root (unless directly invoked by root).

Using suEXEC

While suEXEC is easy to set up, it's even easier to use. Once it is enabled in your running Apache process, any CGI script that is invoked from a user's Web directory will execute under the user and group permissions of the owner of the Web directory. In other words, if I invoke a script with a URL like http://jackal.hiwaay.net/~caulds/cgi-bin/somescript.cgi, that script will run under caulds's user and group account. Note that all CGI scripts that will run under the suEXEC wrapper must be in the user's Web directory (which defaults to public_html but can be redefined by the --with suexec-userdir configuration) or a subdirectory of that directory.

For virtual hosts, the user and group accounts under which CGI scripts are run are defined by the SuexecUserGroup directive found in the virtual host container (this single directive replaces the use of the User and Group directives in pre-2.0 versions of Apache):

```
<VirtualHost 192.168.1.1>
  ServerName vhost1.hiwaay.net
  ServerAdmin caulds@hiwaay.net
  DocumentRoot /home/httpd/NamedVH1
  SuexecUserGroup vh1admin vh1webteam
</VirtualHost>
```

Simple Request Redirection

Chapter 9 discusses the redirection of HTTP requests in detail, particularly using mod_rewrite, which permits the use of a series of rules to perform very sophisticated URL rewriting. URL rewriting is a highly flexible way to redirect requests, but it is also quite complicated. For simple redirection, the Alias core directive is very useful. As an example of how Alias permits easy access to HTML documents outside the Document-

Root directory on my server, I'll demonstrate how I redirected access to the documentation for the MySQL database on my system, which resides outside Apache's DocumentRoot directory.

Many applications for Linux include documentation in the form of linked HTML pages. These are ordinarily outside the hierarchy of resources that has the Apache DocumentRoot at its top. Apache documentation is no exception. Some provision should be made to allow access to these pages. On my system, the Apache documentation pages are installed in /usr/local/apache2/htdocs. I used the Alias directive to alias two directories outside my DocumentRoot to URLs that appear inside the DocumentRoot resource tree. The first of these is the documentation for the MySQL database, which placed its documentation in /usr/share/doc/mysql-3.23.49 when installed on my system.

```
# pwd
/usr/share/doc/mysql-3.23.49
# ls
COPYING  COPYING.LIB  INSTALL-SOURCE  manual.texi  manual.txt
mysqld_error.txt  README
```

I symbolically linked the top-level HTML file to one that Apache will read when the requested URL names only the directory and not a particular file (that is, where it matches one of the names specified in DirectoryIndex):

```
# ln -s manual_toc.html index.html
```

Using a symbolic link, rather than copying the file or renaming it, ensures that only one copy of the file exists, but can be accessed by either name. The last step was the insertion of two Alias directives into httpd.conf. Place these in a manner that seems logical to you, probably somewhere in the main server configuration section of the file, so that you can easily locate the directives at a later date.

```
Alias /MySQL/ /usr/share/doc/mysql-3.23.49
Alias /ApacheDocs/ "/usr/local/apache2/manual/"
```

Any user can now access these sets of documentation on my server using these URLs:

```
http://jackal.hiwaay.net/MySQL/
http://jackal.hiwaay.net/ApacheDocs/
```

Providing Directory Indexes

I'm ending this chapter with a discussion of a very important set of Apache directories that are not actually part of the core module, but are such an essential part of the standard

distribution that they are used on every Apache server. You might notice that most Web pages are not retrieved by the specific filename. Rather than entering a URL like this:

```
http://jackal.hiwaay.net/dirname/index.html
```

you generally enter a URL like the following:

```
http://jackal.hiwaay.net/dirname
```

This URL actually maps to a directory on the server (a directory named `dirname` beneath the directory defined in the Apache configuration as `DocumentRoot`). It is only through a standard Apache module named `mod_dir` that a specific page is served to clients that send a request URL that maps to a directory. Without `mod_dir`, the second form, which does not specify a unique resource, would be invalid and would produce an HTTP 404 (Not Found) error.

The `mod_dir` module serves two important functions. First, whenever a request is received that maps to a directory but does not have a trailing slash (/) as in:

```
http://jackal.hiwaay.net/dirname
```

`mod_dir` sends a redirection request to the client indicating that the request should be made, instead, to the URL:

```
http://jackal.hiwaay.net/dirname/
```

This requires a second request on the part of the client to correct what is, technically, an error in the original request. Though the time required to make this second request is usually minimal and unnoticed by the user, whenever you express URLs that map to directories rather than files, you should include the trailing slash for correctness and efficiency.

The second function of `mod_dir` is to look for and serve a file defined as the index file for the directory specified in the request. That page, by default, is named `index.html`. This can be changed using `mod_dir`'s only directive, `DirectoryIndex`, as described below. The name of the file comes from the fact that it was originally intended to provide the requestor with an index of the files in the directory. While providing directory indexes is still useful, the file is used far more often to serve as a default HTML document, or Web page, for the root URL; this is often called the home page. Remember that this behavior is not a given; `mod_dir` must be included in the server configuration and enabled for this to work.

The *DirectoryIndex* Directive

As mentioned, the default value of the file served by `mod_dir` is `index.html`. In other words, if the Apache configuration contains no `DirectoryIndex` directive, it will look

for and attempt to serve a file named `index.html` whenever a request URL resolves to a directory. Although this is a default behavior, the standard Apache configuration will create the following line in the `httpd.conf` file that it installs:

```
DirectoryIndex index.html index.html.var
```

The `.var` entry is a *type-map file*, which allows Apache to serve its default page in a language based on the language preference in the requesting user's browser settings. The use of type-map files is described in Chapter 15.

The last change I made was to add a second filename to the `DirectoryIndex` directive. I added an entry for `index.htm` to cause the Apache server to look for files of this name, which may have been created on a system that follows the Microsoft convention of a three-character filename extension. The files are specified in order of preference from left to right, so if it finds both `index.html` and `index.htm` in a directory, it will serve `index.html`.

```
# DirectoryIndex index.html index.html.var
DirectoryIndex index.html index.htm
```

Fancier Directory Indexes

I've described the behavior of Apache when a request is received that maps to a directory on the server. Through `mod_dir`, Apache serves a file from the directory defined in the `DirectoryIndex` directive (or `index.html` if `DirectoryIndex` is not specified). In cases where no such file exists, Apache uses a second module, `mod_autoindex`, to prepare an index or listing of the files in the directory. Figure 4.1 shows the default directory index that `mod_autoindex` will serve to the requesting client.

The default `httpd.conf` for an Apache 2.0 server will contain the following line in the main server configuration section:

```
IndexOptions FancyIndexing VersionSort
```

This directive enables the use of *fancy indexing* from `mod_autoindex`, which allows the administrator full control over every aspect of the listing Apache prepares for a directory that does not contain an index file. The default `httpd.conf` provided with the Apache distribution uses many of the directives that I'll describe in the following sections to set up the default fancy directory for use on your server.

Older versions of Apache included `mod_autoindex`, but disabled the fancy indexing by default. Unless manually reconfigured, an Apache server older than 2.0 displays a directory listing as shown in Figure 4.2. Removing the `IndexOptions FancyIndexing` directive from your server configuration will produce a listing with this generic format.

Figure 4.1 A fancy Apache directory listing

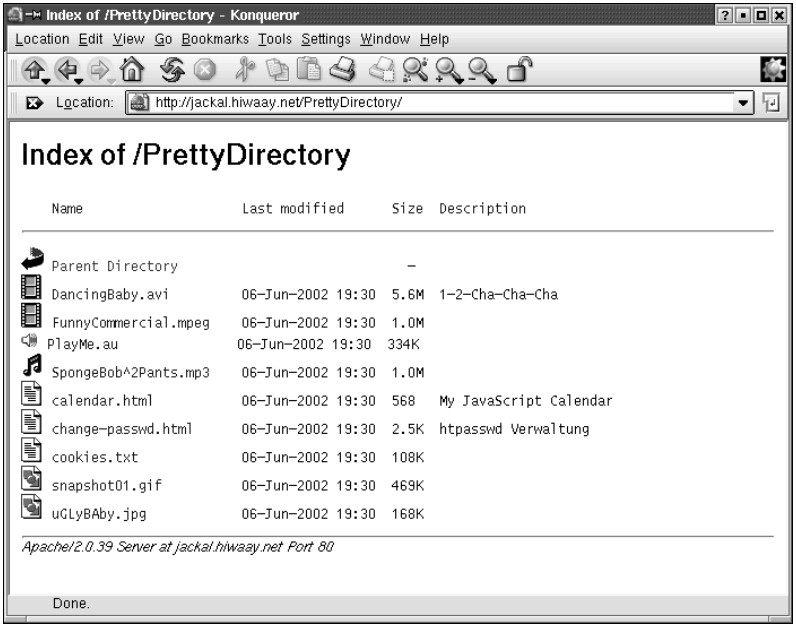
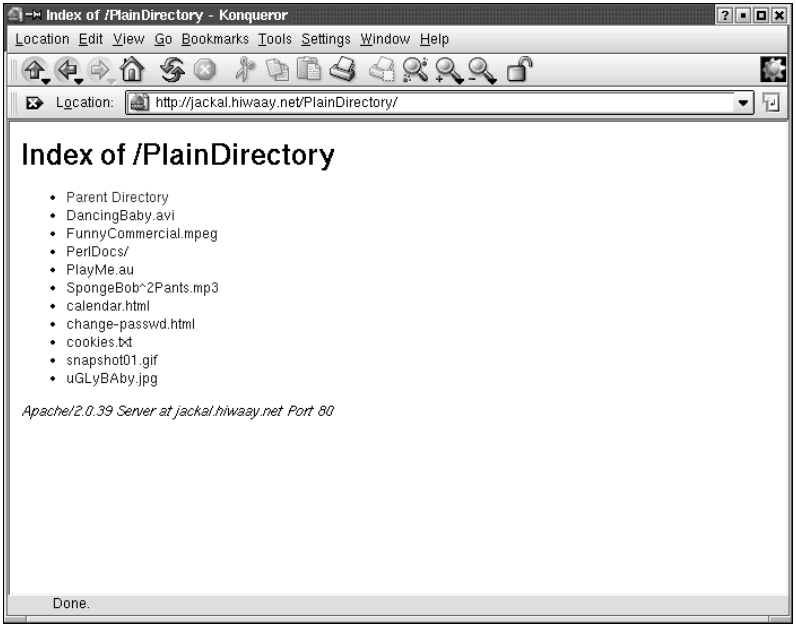


Figure 4.2 A plain Apache directory listing



The `VersionSort` indexing option used in the default Apache 2.0 configuration file (also new in Apache versions later than 2.0) causes Apache to attempt to sort files that contain version numbers in a logical manner. This is so that, for example, a file named `httpd-2.0.9.gz` will be listed before one named `httpd-2.0.12.gz`, unlike a simple alphanumeric sort.

Index Options

`IndexOptions` can also be used to set a number of other options for configuring directory indexing. Among these are options to specify the size of the icons displayed, suppress the display of any of the columns besides the filename, and whether or not clicking the column heading sorts the listing by the values in that column. Many of these options can be utilized to disable formatting features that aren't properly displayed in older browsers. Although these indexing options allow a great amount of flexibility in determining the resulting display of directory listings, most users are perfectly content with the Apache-provided default values. Table 4.1 depicts all possible options that can be used with the `IndexOptions` directive.

Table 4.1 Index Options

Index Option	Description
`DescriptionWidth=`*`n`*	Defines the character width of the file description column.
`FancyIndexing`	Enables fancy indexing.
`FoldersFirst`	Lists subdirectories (folders) first, before all files in the displayed directory.
`HTMLTable`	Uses HTML tables to display the directory (may not work with some browsers).
`IconsAreLinks`	Makes icons part of the clickable anchor for the filename.
`IconHeight=`*`pixels`*	Sets the height (in pixels) of the icons displayed in the listing. Like the HTML tag <`IMG HEIGHT=`*`n`*`…`>.
`IconWidth=`*`pixels`*	Sets the width (in pixels) of the icons displayed in the listing. Like the HTML tag <`IMG WIDTH=`*`n`*`…`>.
`IgnoreClient`	Ignores all query variables from the client, including sort column and order.

Table 4.1 Index Options (*continued*)

Index Option	Description
`NameWidth=n`	Sets the width (in characters) of the filename column in the listing, truncating characters if the name exceeds this width. Specifying `NameWidth=*` causes the filename column to be as wide as the longest filename in the listing.
`ScanHTMLTitles`	Causes the file description to be extracted from the HTML <TITLE> tag, if it exists. The `AddDescription` directive overrides this setting for individual files.
`SuppressColumnSorting`	Disables the normal behavior of inserting clickable headers at the top of each column that can be used to sort the listing.
`SuppressDescription`	Disables the display of the Description column of the listing.
`SuppressHTMLPreamble`	Disables automatic HTML formatting of the header file if one is specified for the listing. No <HTML>, <HEAD>, or <BODY> tags can precede the header file, and they must be manually placed into the file contents if desired.
`SuppressIcon`	Disables the printing of the icon in directory listings.
`SuppressLastModified`	Disables the display of the Last Modified column of the listing.
`SuppressRules`	Disables the printing of the horizontal rules in directory listings.
`SuppressSize`	Disables the display of the Size column in the listing.
`TrackModified`	Returns the date of last modification and entity tag (or ETag) information of the current directory inside the HTTP response header. This information is useful to caching engines.
`VersionSort`	Attempts to sort filenames containing version numbers in a logical order.
`None`	Disables fancy indexing.

Options are always inherited from parent directories. This behavior is overridden by specifying options with a + or – prefix to add or subtract the options from the list of options that are already in effect for a directory. Whenever an option is read that does

not contain either of these prefixes, the list of options in effect is immediately cleared. Consider this example:

```
IndexOptions +ScanHTMLTitles -IconsAreLinks SuppressSize
```

If this directive appears in an `.htaccess` file for a directory, regardless of the options inherited by that directory from its higher-level directories, the net effect will be the same as this directive:

```
IndexOptions SuppressSize
```

This is because as soon as the `SuppressSize` option was encountered without a `+` or `–` prefix, the current list of options was immediately cleared.

Specifying Icons

In addition to `IndexOptions`, `mod_autoindex` provides other directives that act to configure the directory listing. You can, for example, provide a default icon for unrecognized resources. You can change the icon or description displayed for a particular resource, either by its MIME type, filename, or encoding type (GZIP-encoded, for example). You can also specify a default field and display order for sorting, or identify a file whose content will be displayed at the top of the directory.

The *AddIcon* Directive

`AddIcon` specifies the icon to display for a file when fancy indexing is used to display the contents of a directory. The icon is identified by a relative URL to the icon image file. Note that the URL you specify is embedded directly in the formatted document that is sent to the client browser, which then retrieves the image file in a separate HTTP request.

The `name` argument can be a filename extension, a wildcard expression, a complete filename, or one of two special forms. Examples of the use of these forms follow:

```
AddIcon /icons/image.jpg *jpg*
AddIcon (IMG, /icons/image.jpg) .gif .jpg .bmp
```

The second example above illustrates an alternate form for specifying the icon. When parentheses are used to enclose the parameters of the directive, the first parameter is the alternate text to associate with the resource; the icon to be displayed is specified as a relative URL to an image file. The alternate text, `IMG`, will be displayed by browsers that are not capable of rendering images. A disadvantage of using this form is that the alternate text cannot contain spaces or other special characters. The following form is *not* acceptable:

```
AddIcon ("JPG Image", /icons/image.jpg) .jpg
```

There are two special expressions you can use in place of a filename in the AddIcon directive that specify which images to use as icons in the directory listing. Use ^^BLANKICON^^ to specify an icon for blank lines in the listing, and use ^^DIRECTORY^^ to specify an icon for directories in the listing:

```
AddIcon /icons/folder.gif ^^DIRECTORY^^
AddIcon /icons/blank.gif ^^BLANKICON^^
```

There is one other special case that you should be aware of. The parent of the directory whose index is being displayed is indicated by the ".." filename. You can change the icon associated with the parent directory with a directive like the following:

```
AddIcon /icons/up.gif ..
```


NOTE The Apache Software Foundation recommends using AddIconByType rather than AddIcon whenever possible. Although there appears to be no real difference between these (on a Linux system, the MIME type of a file is identified by its filename extension), it is considered more proper to use the MIME type that Apache uses for the file, rather than directly examining its filename. There are often cases, however, when no MIME type has been associated with a file and you must use AddIcon to set the image for the file.

The *AddIconByType* Directive

AddIconByType specifies the icon to display in the directory listing for files of certain MIME content types. This directory works like the AddIcon directive just described, but it relies on the determination that Apache has made of the MIME type of the file (as discussed in Chapter 15, Apache usually determines the MIME type of a file based on its filename).

```
AddIconByType /icons/webpage.gif text/html
AddIconByType (TXT, /icons/text.gif) text/*
```

This directive is used almost exactly like AddIcon. When parentheses are used to enclose the parameters of the directive, the first parameter is the alternate text to associate with the resource; the icon to be displayed is specified as a relative URL to an image file. The last parameter, rather than being specified as a filename extension, is a MIME content type. Look in conf/mime.types under the Apache home directory (typically /usr/local/apache2) for a list of types that Apache knows about.

The *AddIconByEncoding* Directive

AddIconByEncoding is used to specify the icon displayed next to files that use a certain MIME encoding. As discussed in Chapter 15, MIME encoding generally refers to file

compression schemes and therefore determines what action is required to decode the file for use. Some typical encoding schemes and examples of the use of this directive are:

```
AddIconByEncoding /icons/gzip.gif x-gzip
AddIconByEncoding /icons/tarimage.gif x-gtar
```

Specifying a Default Icon

A special directive, `DefaultIcon`, is used to set the icon that is displayed for files with which no icon has been associated, i.e., none of the other directives mentioned above. The directive simply identifies an image file by relative URL:

```
DefaultIcon  /icons/unknown.pcx
```

Adding Alternate Text for Images

When an image is displayed in a Web page, the HTML tags used to embed the image in the page provide for an alternate text string that is displayed in browsers that cannot display graphics. This text string is also displayed as pop-up text if the user of a graphical browser right-clicks or, depending on your Web browser, when the cursor pauses or "hovers" over the image.

There are three directives that are provided by `mod_autoindex` for setting the alternate text associated with a file in the fancy directory listing. Each of these directives is analogous to one of the `AddIcon` directives shown above and uses the same syntax.

The *AddAlt* Directive

The `AddAlt` directive specifies an alternate text string to be displayed for a file, instead of an icon, in text-only browsers. Like its `AddIcon` counterpart, the directive specifies a filename, partial filename, or wildcard expression to identify files:

```
AddAlt "JPG Image" *jpg*
AddAlt "Image File" .gif .jpg .bmp
```

Note that it is possible to use a quoted string with the `AddAlt` directive, which can contain spaces and other special characters. This is not possible when specifying alternate text using the special form of `AddIcon` as shown above.

The *AddAltByType* Directive

`AddAltByType` sets the alternate text string to be displayed for a file based on the MIME content type that Apache has identified for the file. This directive works very much like its counterpart, `AddIconByType`.

```
AddAltByType "HTML Document"  text/html
```

The *AddAltByEncoding* Directive

`AddAltByEncoding` sets the alternate text string to be displayed for a file, based on the MIME content encoding of the file, as determined by Apache.

```
AddAltByEncoding "GZipped File" x-gzip
```

Specifying File Descriptions

The `AddDescription` directive is used to specify a text string to be displayed in the Description column of the listing for specific files. Files can be identified by a partial or full pathname:

```
AddDescription "My Home Page" index.html
```

Note that this example sets a description to apply to all files named `index.html`. To apply the description to a specific file, use its full and unique pathname:

```
AddDescription "My Home Page" /home/httpd/html/index.html
```

`AddDescription` can also be used with wildcard filenames to set descriptions for entire classes of files (identified by filename extension in this case):

```
AddDescription "PCX Image" *.pcx
AddDescription "TAR File" *.tgz *.tar.gz
```

When multiple descriptions apply to the same file, the first match found is the one used in the listing; so always specify the most specific match first:

```
AddDescription "Powered By Apache Logo" poweredby.gif
AddDescription "GIF Image" *.gif
```

In addition to `AddDescription`, there is one other way that `mod_autoindex` can determine values to display in the Description column of a directory listing. If `IndexOptions ScanHTMLTitles` is in effect for a directory, `mod_autoindex` will parse all HTML files in the directory, and extract descriptions for display from the `<TITLE>` elements of the documents. This is handy if the directory contains a relatively small number of HTML documents or is infrequently accessed. Enabling this option requires that every HTML document in the directory be opened and examined. For a large number of files, this can impose a significant workload, so the option is disabled by default.

Adding a Header and a Footer

The `mod_autoindex` module supplies two directives that allow you to insert the contents of a file at the top of the index listing as a page header or at the bottom of the listing as a page footer.

The HeaderName directive specifies a filename using a URI relative to the one used to access the directory. The contents of this file are placed into the listing immediately after the opening <BODY> tag of the listing. It is usually a good idea to maintain the header file in the same directory it describes, which makes it easy to reference by its filename. The default httpd.conf file that comes with Apache contains the following directive, which doesn't really need to be changed:

```
HeaderName HEADER.html
```

Files identified by the HeaderName directive must be of the major MIME content type text. If the file is identified as type text/html (generally by its extension), it is inserted verbatim; otherwise it is enclosed in <PRE> and </PRE> tags. A CGI script can be used to generate the information for the header (either as HTML or plain text), but you must first associate the CGI script with a MIME main content type (usually text), as follows:

```
AddType text/html .cgi
HeaderName HEADER.cgi
```

The ReadmeName directive works almost identically to HeaderName to specify a file (again relative to the URI used to access the directory being indexed) that is placed in the listing just before the closing </BODY> tag. I changed the default httpd.conf file that comes with Apache to better describe its function of placing a "footer" on the directory page:

```
#   Apache default
# ReadmeName README.html
#   My revised directive
ReadmeName FOOTER.html
```

Ignoring Files

The IndexIgnore directive specifies a set of filenames that are ignored by mod_autoindex when preparing the index listing of a directory. The filenames can be specified by wildcards:

```
IndexIgnore FOOTER*
```

> ***NOTE*** The default httpd.conf file provided with Apache contains an IndexIgnore directive that prevents filenames beginning with README or HEADER from being displayed in the index listing by mod_autoindex. This makes these filenames obvious (but not necessary) choices for use as headers and footers for directory listings.

Ordering the Index Listing

The IndexOrderDefault directive is used to change the default order of the index listing generated by mod_autoindex, which is to sort the list in ascending order by filename.

This directive takes two arguments. The first must be either `Ascending` or `Descending` to indicate the sort direction; the second names a single field as the primary sort key and can be `Name`, `Date`, `Size`, or `Description`:

```
IndexOrderDefault Descending Size
```

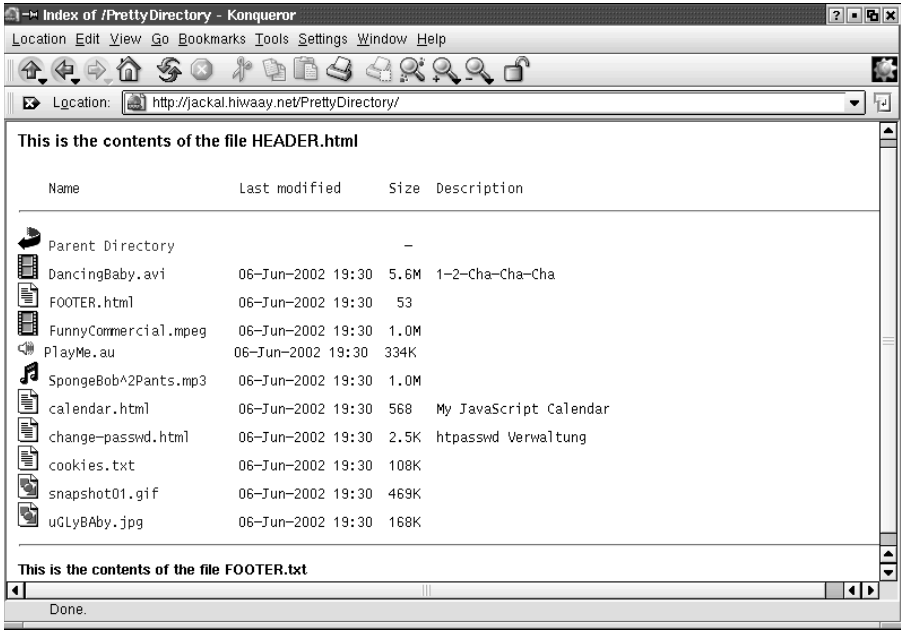
The secondary sort key is always the filename in ascending order.

Example

In order to illustrate typical uses of some of the `mod_autoindex` directives discussed, I created an `.htaccess` file in the same directory that was illustrated in Figure 4.2. This file contains the following directives, all of which are used by `mod_autoindex` to customize the index listing for the directory. The result of applying these directives is shown in Figure 4.3.

```
IndexOptions FancyIndexing VersionSort +ScanHTMLTitles
AddIcon /icons/small/sound.gif .au
AddDescription "1-2-Cha-Cha-Cha" DancingBaby.avi
AddAltByType "This is a JPG Image" image/jpeg
ReadmeName FOOTER.html
```

Figure 4.3 A customized `mod_autoindex` listing



The `IndexOptions` directive is used to enable the extraction of file descriptions from the `<TITLE>` tags of HTML formatted files (technically, files of MIME content type text/HTML). You can see this illustrated in Figure 4.3 by the files with `.html` extensions. If any of these files had the name `index.html`, the index listing would not have been generated at all; instead, `index.html` would have been sent (by `mod_dir`) to the client.

I've also provided an example of adding an icon using the `AddIcon` directive and a file description using `AddDescription`. The results of these directives can be easily seen in Figure 4.3. The alternate text for JPEG images (added with the `AddAltByType` directive) is not displayed in the figure, but would be seen in place of the image icon in text-only browsers. It would also appear in a graphical browser in a pop-up dialog box when the cursor is paused over the associated icon. This gives the page developer a handy way to add help text to a graphics-rich Web page, which can be particularly useful when the icon or image is part of an anchor tag (clickable link) and can invoke an action.

To demonstrate the `HeaderName` and `ReadmeName` directives, I included files with these names, the contents of which were included as, respectively, the header and footer for the directory page. Note that I overrode Apache's default value for the `ReadmeName` directive, which looks for a file named `README.html`, a name that I felt wasn't descriptive of its purpose as a page footer. Both files consisted of a single line of (boring) text. The header file contained HTML-formatting tags (`<H3> </H3>`) that caused it to be rendered in larger, bolder characters, and the footer used `<H4>` tags so it was slightly less prominent. There is no reason why either the header or footer couldn't be longer and contain far more elaborate formatting. Use your imagination.

In Sum

This chapter has covered a lot of ground, because so much of Apache's functionality is now incorporated into the configuration directives provided by its core modules, rather than through optional extension modules. We began with the essential concept of directive *context*, or the scope within which particular directives are valid. We then looked at the directives used to configure the basic server environment and how the server listens for connections. These directives are fundamental to Apache's operation, and every administrator needs to be familiar with them.

Later sections of the chapter explored the directives used to create and manage user home directories. These are not only an essential function for any ISP installation of an Apache server; they are also widely used in intranets.

The next chapter moves beyond the core modules to the use of freely downloadable third-party modules and the techniques you can use to incorporate them into your Apache server.