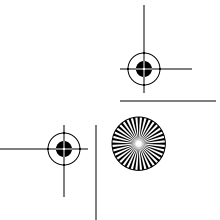
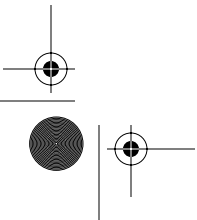
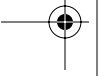
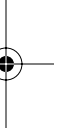
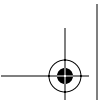
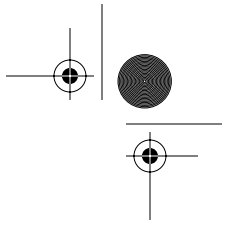


# PART I

# THE BASICS







# BASIC CONCEPTS

## 1.1 Introduction

This chapter introduces the concepts that are the foundation of the Apache Web server. The other chapters in the book sling the jargon discussed herein pretty freely, so if you're not familiar with terms like *directive*, *module*, and so forth, you probably need to read this chapter through.

Most of the Apache-specific techniques described below are recognizably evolved from standard programming doctrine. However, if you are not a programmer, or even if you are just new to Apache, reading this chapter will help speed your ascent of the learning curve.

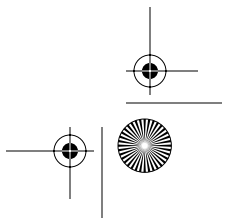
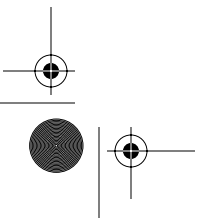
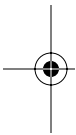
## 1.2 Configuration Files

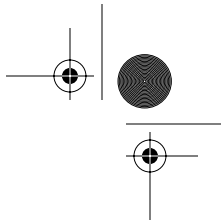
Apache is an extremely configurable piece of software. There are thousands of possible combinations of values for hundreds of configuration variables. Those of you who have used Unix or DOS are undoubtedly familiar with the concept of the command-line option. The basic principle behind directives is the same as command-line options, but rather than respecify each of the dozens or hundreds of configuration variables on the command line every time Apache is invoked, the values are collected into configuration files, which are read automatically by the server at startup time. The variables that Apache reads to determine what it is supposed to do were previously contained in three files:

<code>httpd.conf</code>	The main configuration file containing the variables that specify server-specific configuration information.
<code>httpd.conf</code>	Server Resource Management—historically, this file contained the variables that specified how server resources were to be used.
<code>access.conf</code>	Historically, this file contained the variables relating to access control.

Currently the practice of using all three configuration files is an anachronism. The practice is still supported for purposes of backward compatibility, but is discouraged. *These days you will usually find all configuration directives in the `httpd.conf` file.*

By default, configuration files are stored in a subdirectory named `conf` under the main Apache directory. As with most everything else in Apache, the location and names of these





files is configurable, but for purposes of discussion I will usually assume the default location. Chapter 3, “Configuring Apache,” contains more information about how to locate and specify the main Apache directory and the various files.

## 1.3 Directives

As mentioned above, Apache configuration values are specified by variables contained in configuration files. These variables are known as *directives*. Ninety percent of Apache administration is spent figuring out what directives you want to set and what you want to set them to.<sup>1</sup>

Note that *not all possible directives are automatically recognized by Apache*. A fairly sizable subset of the directives (known as the *core directives*) are enabled by default, but whether or not Apache recognizes the rest depends on what modules are compiled in to the specific version of Apache that you are running. If you include a directive in your configuration file but it doesn’t appear to have any effect, the executable you are running may not have the right module compiled in. To get a listing of your compiled-in modules, type the following command:

```
httpd -l
```

Note that shared object modules are loaded at runtime and will not be displayed by this command. See the section “Modules” in this chapter for more information.

## 1.4 Limiting the Scope of Directives

Not all directives are universally applied. Sometimes it is useful to have a particular directive set one way for one portion of your server, but the opposite way (or not at all) for another portion.

The most obvious example occurs when hosting multiple sites. It is possible to have more than one Web site handled by the same Apache server. This practice is discussed more fully in Chapter 5, “Hosting Multiple Sites,” but for now just take it on faith that two sites (for example, *www.christiansite.org* and *www.muslimsite.org*) can be serviced by the same program on the same machine. When hosting multiple Web sites, you will usually want to specify different directives for each site; at the very least, they will have different names.

Excluding Virtual Hosts (which are discussed later), the scope of a particular directive can be limited in three ways:

- by directory, using the **<Directory>** directive, the **<DirectoryMatch>** directive, or the **.htaccess** file
- by URL, using the **<Location>** and **<LocationMatch>** directives
- by file, using the **<File>** and **<FileMatch>** directives

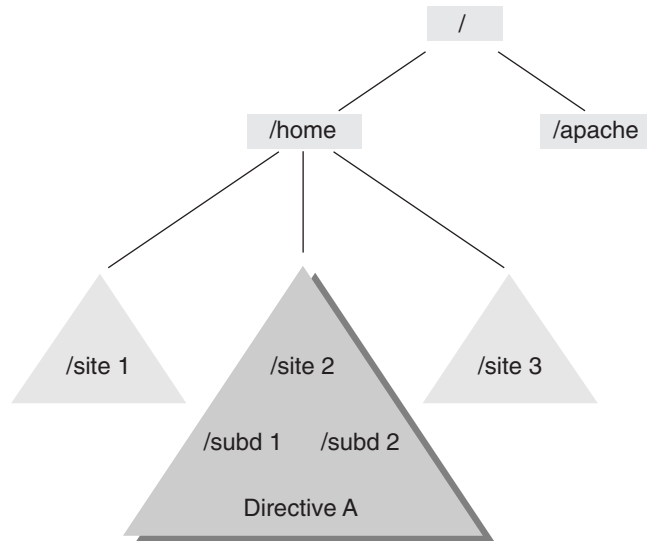
---

1. The other ninety percent is spent tweaking your compilation.

### 1.4.1 Limiting Scope to a Directory, v.1: <Directory> and <DirectoryMatch>

Say you have a directive that you wish to apply only to the **/home/site2** directory and any subdirectories it may have. For purposes of this example, it doesn't matter what the directive actually does, only that you want it to apply to the chunk of the directory tree rooted at **/home/site2** and not to the rest of your site. I'll use an imaginary directive, **DirectiveA**, in this example. If you want **DirectiveA** to apply to **/home/site2** and any subdirectories it may have, but not to the rest of the server, enclose it in **<Directory>** brackets as shown below.

```
<Directory /home/site2>
  DirectiveA
</Directory>
```

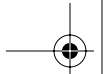


**Figure 1.1** Scope of DirectiveA.

The **<DirectoryMatch>** directive works much like the **<Directory>** directive, except it takes a regular expression rather than an actual directory name as an argument. For example,

```
<DirectoryMatch "/home/site[1-3]">
  DirectiveA
</DirectoryMatch>
```

would match with the directories **/home/site1**, **/home/site2**, and **/home/site3**.



## 1.4.2 Limiting Scope to a Directory, v.2: Using .htaccess Files

Another way to accomplish much the same thing is to create a special file containing configuration directives and store that file in the directory to which you want the configuration information to apply. By default, such files are named **.htaccess**. If that name doesn't suit you, you can change it to most anything you like by means of the **AccessFileName** directive.

In order for the directives in **.htaccess** to have an effect, Apache must first be informed that such files may exist and that it should look for them. This is accomplished by means of the **AllowOverride** directive. **AllowOverride** is not just a simple on/off switch, however. It may also be used to specify what *kinds* of directives to use in a **.htaccess** file. The possible options are specified in Table 1.1.

**Table 1.1** AllowOverride

<i>Option</i>	<i>Effect</i>
All	Allows use of all directives
None	No directives allowed; searching for the file is disabled
AuthConfig	Allows use of <b>AuthDBMGroupFile</b> , <b>AuthDBMUserFile</b> , <b>AuthGroupFile</b> , <b>AuthName</b> , <b>AuthType</b> , <b>AuthUserFile</b> , <b>require</b>
FileInfo	Allows use of <b>AddEncoding</b> , <b>AddLanguage</b> , <b>AddType</b> , <b>DefaultType</b> , <b>ErrorDocument</b> , <b>LanguagePriority</b>
Indexes	Allows use of <b>AddDescription</b> , <b>AddIcon</b> , <b>AddIconByEncoding</b> , <b>AddIconByType</b> , <b>DefaultIcon</b> , <b>DirectoryIndex</b> , <b>FancyIndexing</b> , <b>HeaderName</b> , <b>IndexIgnore</b> , <b>IndexOptions</b> , <b>ReadMeName</b>
Limit	Allows use of <b>allow</b> , <b>deny</b> , and <b>order</b> directives
Options	Allows use of <b>Options</b> and <b>XBitHack</b>

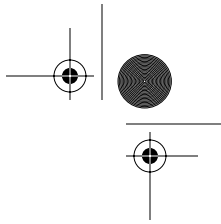
More than one option may be specified at a time. For example, to allow any **Options** directives and **FileInfo** directives to be overridden by the contents of an **.htaccess** file:

```
AllowOverride Options FileInfo
```

The fact that **.htaccess** files apply to subdirectories as well as to the directory in which they are found means that when someone attempts to access files in the **/home/site2/cgi-bin**, Apache must search for a file named **.htaccess** (or whatever you specified) in that directory *and in all the directories above it in the directory tree*. As you might imagine, this hurts Apache's performance. If there is no pressing need to allow the use of **.htaccess** files, you should disable them as follows:

```
AllowOverride none
```





### 1.4.3 Limiting Scope to a URL: <Location> and <LocationMatch>

Just as the <Directory> directive limits the scope of the directives, which it brackets to a location within the file system, the <Location> directive limits the scope of the directives, which it brackets to a URL. In practice, <Location> and <Directory> can be effectively the same. The distinction between the two is that the <Location> directive need not necessarily apply to an actual physical location within the file system.

Typically, the location directives specify a sub-URL with the implication that it is local to the specific server. For example, say you have configured a server named *www.example.com* and it uses the following location directive:

```
<Location /status>
    SetHandler server-status
</Location>
```

In that case, Apache would activate the *server-status* handler when the following URL is requested:

```
http://www.example.com/status
```

Don't worry too much if you've never heard of handlers. They're a way of telling Apache to do nondefault things when certain locations are accessed. They will be discussed in more detail later in this chapter. The key point here is that if you don't specify a server name, Apache will assume that any location you specify is relative to the local server.

### 1.4.4 Limiting Scope to a Virtual Host

For completeness, let me say here that you may also limit the scope of a directive by creating a virtual host within the local server. The term *virtual host* refers to an Apache server that has been configured so that it will respond to requests for more than one server. For example, a small Web hosting company might have dozens or hundreds of Web sites running on the same piece of hardware and serviced by the same instance of Apache. When properly configured, Apache is bright enough to distinguish among the virtual hosts and pretend to the outside world that each one is an entirely separate entity. The means by which this is accomplished are discussed in detail in Chapter 5.

### 1.4.5 Limiting Scope by File: <File> and <FileMatch>

The <Files> and <FilesMatch> directives are also conceptually very similar to the <Directory> and <DirectoryMatch> directives. The difference, of course, is that they apply only to individual files or, in the case of <FilesMatch>, to individual files as matched by regular expressions. For example, the following directive makes the *.htaccess* configuration file inaccessible to anyone:

```
<Files .htaccess>
    Order deny, allow
    Deny from all
</Files>
```



Note that unlike the other scope-related directives discussed above, **<File>** and **<FileMatch>** may be meaningfully included in a **.htaccess** file.

## 1.5 Modules

As mentioned earlier, Apache includes a chunk of basic functionality, called the core, with all implementations. The core implements the concept of directives, the ability to read configuration files, some access control, the ability to extend itself, and half a dozen other fundamental abilities. Specifically, the directives found in Appendix A, “Core Directives,” will always be present in any standard Apache distribution.

In addition to this, Apache is designed to allow the basic functionality to be augmented by subsections of bonus code, which may or may not be linked into a particular executable. These subsections are called *modules*, and a fairly sizable selection of the most useful ones are enabled by default in the standard distribution. To generate a listing of the modules included in your executable, use the `-l` option:

```
httpd -l
```

Whether or not a particular module is included in the current executable can be controlled by the server administrator (you, presumably) either at compile time or by using the **AddModule** and **ClearModuleList** directives. In the case of shared object files, modules can be dynamically controlled at runtime using the **LoadModule** directive (see “Dynamic Shared Objects” in this chapter).

The order in which modules are loaded is significant and should not be tampered with lightly. However, if you wish to specify your own load order (and potentially exclude some of the default modules), you may empty the internal load list with the **ClearModuleList** directive:

```
ClearModuleList
```

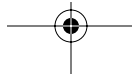
You should immediately repopulate the load list with a sequence of **AddModule** directives:

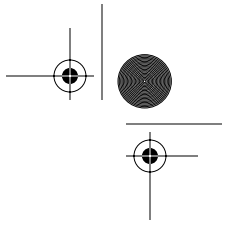
```
AddModule mod_access.c
```

When included, a module is an integral part of the executing `httpd` process, with the same process ID and access to the same system resources. Contrast this to the CGI programs, which may be included in your Web sites. CGI processes are distinct from the `httpd` executable that invoked them and must communicate with it via system interprocess communication resources, a *much* slower method. Whenever speed is critical, write a module.

Apache was designed with modules in mind. At various stages of its execution, the core `httpd` code will poll any executables that have been included to find out what they have to say about how to handle the configuration files, the `http` source on the local sites, and user requests. It's all very orderly.

The techniques involved in creating your own module are fairly advanced, but not secret. Anyone can write their own Apache module, and many people do. In fact, some of the most useful modules were written by third-party developers in response to their own needs and were only later included in Apache distributions. See Chapter 12, “Module Construction,” for a fuller exploration of the module creation process.





## 1.6 Dynamic Shared Objects

Many Unix implementations provide the capability of specifying which portions of an executable are to be included and which are to be excluded at runtime. The term for these pre-compiled chunks is *shared objects*. Apache makes extensive use of the shared object capability.

If a module is to be loaded as a shared object, it must be compiled using the **apxs** (APache eXtenSion) program. Third-party modules (**mod\_perl**, **mod\_php**) typically come with their own instructions for doing this. Some platforms require dynamic shared object (DSO) functionality on the Apache core features in order to force the linker to export the symbol table for later use by third-party modules. If you need to enable DSO functionality on the core Apache features, use the following option in your configure script:

```
--enable-rule=SHARED_CORE
```

Then compile and rebuild, and reinstall httpd as discussed in Chapter 2, “Installing Apache.”

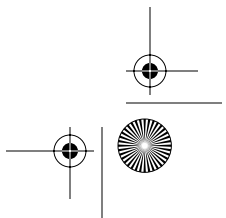
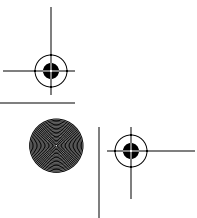
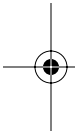
The apache module **mod\_so** enables you to specify at runtime which shared object modules are to be included via the **LoadModule** directive. You need to specify the name of the module and the path to the shared object file in the directive.

```
LoadModule perl_module libexec/libperl.so
```

Typically (and by default), such files are found in a library directory named **libexec** under the **ServerRoot** directory.

## 1.7 Handlers

Modules sometimes provide specific handlers, which are methods of processing files or requests in an unusual way. Sometimes handlers are named so that they can be referred to in configuration directives. Named handlers and their associated modules are listed below in Table 1.2.



**Table 1.2** Named Handlers and Their Modules

<i>Handler</i>	<i>Module</i>	<i>Effect</i>
send-as-is	mod_asis	Serve file and headers as-is.
cgi-script	mod_cgi	Attempt to execute and serve output
imap-file	mod_imap	Image map rule file.
server-info	mod_info	Display server configuration information.
server-parsed	mod_include	Locate and replace server-side includes.
server-status	mod_status	Display server status information.
type-map	mod_negotiation	Parse as type map file.

As described in the modules section of Table 1.2, you must include a module in the current httpd executable before you can access its handler.

Handlers may be associated with file extensions by means of the **AddHandler** directive. For example, to tell Apache to pass all files with a .pl extension over to the cgi-script handler, use the following directive:

```
AddHandler cgi-script .pl
```

The other method of invoking a particular handler is the **SetHandler** directive. **SetHandler** is intended for use within a **<Directory>** or **<Location>** container directive, as it associates a handler with all content in a particular location. For example, to treat all files within the **/images** directory as image map rule files, use the following directive:

```
<Location /images>
  SetHandler imap-file
</Location>
```

## 1.8 MIME Types

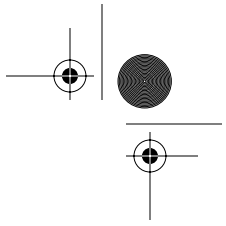
MIME is an acronym for Multimedia Internet Mail Extensions. The idea behind MIME types is to enable a program to determine what kind of data a file contains by looking at the file's extension. By default, Apache's MIME types and their associated extensions are found in a file named **mime.types** in the **conf** directory. This is, of course, configurable; use the **TypesConfig** directive to set it to some other value. For example,

```
TypesConfig /etc/mime.types
```

puts the **mime.types** file in the **/etc** directory.

To simply associate a MIME type with a file extension, use the **AddType** directive. For example,

```
AddType application/x-httpd-php .php
```



## 1.8 MIME Types

11

tells Apache that files which end in *.php* contain PHP4-enriched HTML data. Apache uses MIME types to decide what sort of preprocessing is required by files before delivery to a client. When fiddling with MIME types, it is recommended that you limit yourself to using the **AddType** directive and refrain from modifying the **mime.types** file.

Another useful directive is **DefaultType**. Servers are expected to inform their clients what type of data they are transmitting. Use the **DefaultType** directive to specify a type to be transmitted when nothing else applies.

```
DefaultType text/html
```

