C H A P T E R 2

The web server

In this chapter we'll discuss the different aspects of the web server, by which we mean both the physical machine and the program that responds to clients. It may seem somewhat confusing to mean two or more different things at different times, but the usage is normal in the industry. Context should make it clear when we mean hardware or software.

2.1 WHAT MAKES A GOOD WEB SERVER

First we'll start with the physical machine and basic setup. In deciding what is good for the task, we first have to consider what the task is. Is this a server for internal documentation that gets a handful of lookups an hour, or is it a major commercial site that is expecting millions of visitors per day? Obviously those are extremes, so let's consider three scenarios:

- *Community site*—The server is exposed to the Internet and provides news and information to a group of hobbyists, a profession or some similar association. The traffic is mostly reads, with some messages written back to the site. Downtime is annoying but won't result in lawsuits.
- *Intranet site*—This is a server that supports a workgroup within a company's protected network. It has a fast network connection to all the clients, and will host a handful of applications as well as the bulk of the group's online documentation. The server needs to be available during work hours.
- *Store front*—An e-commerce server that hosts a database of product information and creates catalog pages on the fly. Security and constant availability are the highest priority, followed closely by speed.

Those three cases will be used as examples throughout the book, and each gets its own chapter.

2.1.1 Hardware

Obviously the three scenarios have varying hardware needs, depending on the traffic, of course; a popular news site could be the busiest of the three. But in general, traffic level is the first consideration for choosing a machine. A fast CPU makes for a more responsive server; the operators of the e-commerce site might consider a multiprocessor system so that users won't have to fight for computing time.

To decide how much memory our systems need, we have to consider what applications the servers run. By applications we mean both what the web server does to serve up pages, and the other programs running on the machine. The requirements to run a large database might swamp the web server by comparison. If it does, there should probably be a dedicated database server (that will be discussed in chapter 12) If the site offers mostly static pages (HTML files that are sent as-is to the client) then the web server itself won't need as much memory, but if that site also has intense traffic demanding those pages, then, to improve performance, we might need a lot of memory for caches. A site that generates most of its pages dynamically (such as the e-commerce site) will need more memory because each web server process will put greater demand on the system.

Network bandwidth is the next consideration. Consider the size of a typical response to a client and multiply that by the expected number of requests in a time period to figure out your bandwidth requirements. A site that serves large graphics files to a few users per hour might need more bandwidth than a site with intense traffic for textual stock quotes.

Disk space and I/O bandwidth come last, in part because disk space is so cheap these days that only sites with intensive requirements need to even consider that part of the issue. But if your site runs database applications and performance is a chief concern, disk I/O may be the limiting factor. Consider the best SCSI bus and drives for the e-commerce site, and also look into trading memory for application time by caching the results of database queries.

We'll discuss all these issues in more detail later on. If I recommend using a tool that causes memory consumption to soar I'll point out that you need to reconsider your configuration. All of chapter 12 is devoted to performance analysis and problem resolution.

2.1.2 Operating system

Since this is a book about Open Source tools, I'm going to recommend freely available OSs, but let me first say this: beyond certain considerations, the OS doesn't matter that much. If you've already paid for a commercial version of Unix (or your company says you have to use one) or even (horrors!), one of those shrink-wrapped box OSs, you can run almost anything discussed in this book. With that in mind, let's consider what is important in the choice of OS.

Server and application software—Does the necessary and desired software run on this OS? That's the most important consideration, beyond any brand name or feature. Software compatibility is the reason that a mediocre and buggy operating system dominated desktop computing through the '90s, and it will continue to outweigh other factors.

In the case of our example systems, we want an operating system that runs the Apache web server, our scripting language of choice (preferably Perl, although there are others), and the applications that the web clients will run via the server. There also may be a preferred database and other packages that will support the applications developed for the site.

We're in luck here: nearly any OS with a recent release can run Apache and the major utilities. There are caveats for some operating systems, so check the documentation thoroughly before committing to a decision, but we're largely free to choose one based on other factors.

Performance—The OS can be a major factor in system performance. This is a very complex issue, with reams of academic work on the relative merits of such choices as micro versus monolithic kernels and other design factors. To evaluate it in terms that are important to a web server, go back to the same issues used to evaluate hardware: does the operating system support the amount of memory needed by the server? Does it provide for the kind of network connection to be used? What file system options are there and what features do they provide?

Again, nearly any OS that supports the desired software will have the performance features needed for a web server, but some offer goodies worth considering: journaled file systems that recover quickly from crashes, for instance.

Hardware requirements—Didn't we already cover hardware? In this case, we mean any specific requirements for this OS. If some higher power has mandated the use of a particular OS, then we have to use the hardware it supports. If you are making this choice yourself, you may have hardware at hand that you want to use, such as the ever-popular PC that can be recycled into a Linux system.

In either case, make sure that the hardware choice doesn't limit your performance. For instance, an older PC with IDE disk controllers might not support the update speed needed for an e-commerce site, and a prior-generation motherboard could surprise you with the amount of memory it can handle.

Support costs—If downtime will be costly for your site, then you must have adequate support for your operating system. Adequate may be mailing list or newsgroup support for an Open Source OS, if you are comfortable with the rapidity of responses you see when others have emergencies (check the archives). If your OS doesn't have such venues, then you will need commercial support that promises to help you in time of need.

However you choose to arrange your support, figure the ongoing costs in your TCO.

2.1.3 Re-evaluation and installation

Having examined hardware and OSs and made a choice, go back to the beginning and re-evaluate it. The costs for an adequate machine and a commercial operating system may surprise you. If you are purchasing all this for a high-traffic site, you should never buy a machine that is only adequate, because if your site is successful, you will find yourself buying another machine soon.

If your first round of evaluations included a commercial OS, consider Linux or one of the BSDs. The hardware coverage is very broad, as is choice of software for the web server and application languages, and support is free. While the cost of the OS and initial software are not a large fraction of TCO, having more money for memory or disk bandwidth at the start can help you avoid a costly migration early in your server's life.

Install the OS with an eye toward the goal: a fast, secure web site. That means avoiding unneeded services that will have to be disabled later, even if they would be nice to have at the start. Assume that the server will be maintained via file copies and minimal shell access, and don't bother installing your favorite GUI, desktop, and editor. Less is more (more disk space, more security, more peace of mind later).

Where there is a need to assume things, for the rest of the book I'll assume that the web server is running a reasonably modern Linux distribution or Unix operating system, with adequate hardware for the task at hand. In chapter 12 we'll discuss options for improving performance with a marginal system.

2.2 SECURING THE SITE

Whether you have just built a brand new machine or you are recycling an existing server, it's time for a security review. It may seem early in the process to be worrying about this, but in my experience, starting with a secure system is better than trying to lock down a finished installation. The former builds good habits early, while the latter is prone to loose threads.

The particulars are OS-specific, but for most Unix-like systems the procedure is roughly the same:

1 Go to the password file (`/etc/passwd` or `/etc/shadow`) and disable shell access for all accounts that aren't necessary for a functioning server. You can take that to mean everything but root, although some people prefer to have one non-privileged account with normal access.

2 If you are running inetd, open `/etc/inetd.conf` (or whatever the configuration file is on your system) and start commenting out services. Which services do you need? Possibly none, in which case you can just shut down inetd all together: chances are however, that you'll use ftp for maintaining your site, and you'll need telnet to log in and do the rest of the configuration. Consider replacing both of these with ssh; it provides scp to perform secure, password-encrypted file transfers as well as a secure shell that doesn't expose passwords to plain text network traffic. In chapter 11 we'll discuss rsync and other configuration management tools that will ease your site-management tasks.

3 Moving on to tougher tasks, find out what scripts and configuration files your system uses to start other services. Some have a master script (`rc.boot`), some have a program that executes a series of scripts (often located in `/etc/rc.d`). On my system, `/etc/rc.d` has the boot-time scripts: `rc.sysinit` runs first, then rc executes a series of scripts from one of the subdirectories, then `rc.local` executes. Examine the scripts to find out what services they start, and which of those services respond to commands from the outside world.

4 Disable services that you don't need for a web server. Some things you should not need are: nfs or other networked file systems; network printer services; SMB or other Windows connectivity services; Yellow Pages (yp) or NIS services; and any remote management utilities you can live without on your system. If you aren't expecting to receive email on this site, you can shut down sendmail, imap, pop, and other such tools. You will probably find inetd's startup somewhere along the line, and you can shut it down also if you aren't using its services.

5 Create a nonprivileged account that the web server will use to own its files, with a name such as www, web, or apache. If anyone other than the system administrator will be putting files on the server, let him use that account; otherwise disable shell access.

6 Change the root password, preferably using mkpasswd or a similar tool to generate a random string of 10–15 letters and numbers. Do the same for any remaining accounts with shell access. I keep such passwords in a small notebook that I can lock in a drawer or cabinet.

Now you are ready to reboot your system and verify that all is well and your system is secure. While booting you may notice other services that should be disabled, so go back to the research step and find out how to remove them. You might also investigate tools such as nessus (http://www.nessus.org) that will help check your security.

Some systems don't need to be locked down this tightly. In particular, an intranet server can be considered secure if it has its own root password and there are no shell accounts that have privileges to change configurations. Since the server is inside a protected network, you can take advantage of nfs and other LAN-level services to make your workgroup's life easier.

If your server has an open connection to the Internet, all these steps are required and should be taken before you proceed with any other installations. From this point on you'll be working with the machine either by logging in directly or via a remote session (preferably protected by ssh). There should be no other way to get to the server's files.

Systems exposed to the Internet get attacked. It is a sad fact of life that crackers are always finding new ways to find and exploit vulnerable systems. But protecting your system isn't as difficult as the media sometimes portrays: remove vulnerabilities by disabling services that your system doesn't need, and tightly secure the rest with passwords that can't be guessed (or generated from a dictionary) and use configurations that make sense.

From here on, I'll assume that the system is secure. As configuration issues or new potential vulnerabilities come up, I'll highlight the steps needed to keep things that way.

2.3 THE CASE FOR APACHE

Now it is time to apply the Open Source value considerations and buyer's guide principles to a real-world choice: what web server should you use for your site?

In the rest of the book I'll present alternatives where possible, but in this case there is only one strong choice: the Apache web server. There are other Open Source choices (including thttpd, which we'll discuss in section 2.9), but most are either experimental or specialized; Apache is hardened by years of use at *millions* of web sites. Of the commercial choices available, many are actually made from Apache itself; the code is offered under a BSD-like license that doesn't constrain commercial use. In fact, the developers encourage other companies to use it as a reference implementation for the HTTP protocol.

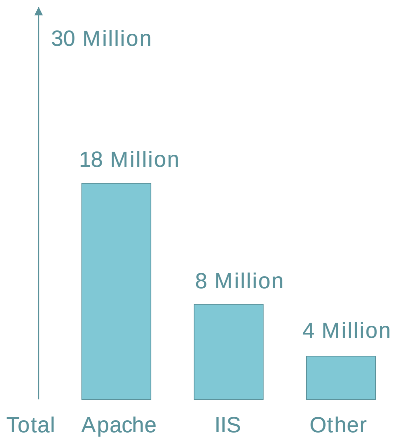

Figure 2.1 Apache market share

The Apache mindshare is one of the largest in Open Source. Estimates of the number of active developers run in the hundreds of thousands, second only to Linux. Development began in 1995, using the code from the original NCSA server (all of which has been replaced in the intervening years). By early 1996 the Apache server was the most popular web server in use by sites polled by Netcraft (http://www.netcraft.com/survey), and in 1998 it gained over 50 percent of the market, making it more popular than all other commercial and Open Source servers combined. Its market share in 2001 has passed 60 percent, running on 18 million servers, more than twice that of Microsoft IIS (the only strong competition). This is illustrated in figure 2.1.

Apache is developed by the Apache Group, a cadre of programmers that was started when a number of web administrators and programmers decided to make an official release of the NCSA server based on their patches. From that beginning they embarked on full-scale redevelopment of the server into their own product, with a stable release in December 1995. The group develops the server and also coordinates the volunteer efforts of countless other contributors, evaluating and accepting patches from outside their number. The membership changes with time as some drop out and other volunteer contributors are invited to join.

2.3.1 Installation

There are a number of options for installing the web server. Prebuilt binary distributions are available for a number of operating systems, and for most sites this is the best way to go. The source distribution is always there of course (this being Open Source), and building Apache from source is quick and easy on most operating systems. Since version 1.3.20, Apache's configuration and build scripts have included support for the free CygWin compiler and tools for Windows operating systems.

The binary distributions on Apache sites (http://www.apache.org) are built with the default set of modules and options. We'll cover what those are in other sections of the book which discuss the need for something that isn't a default. You may need to build from source if your site requires special modules, although it is possible to use a binary distribution with support for dynamically loaded libraries to add in the extras. If you want to strip down Apache by taking out things you won't use or that constitute security risks for your site, make your own binaries. I recommend doing so anyway—it isn't difficult. I'll show you how I built special modules on my system.

Given the number of people who build Apache by hand, it's no surprise that there are programs and tools made just for this purpose. Go to http://www.apache-tools.com/ and search for the Configurator category, where you will find Apache Toolbox, Comanche, and other helpers.

If you download a binary distribution from the Apache site or another distribution site, you will need to find out where it installs the server's main directory. When building from source on Linux or Unix the main directory is `/usr/local/apache` (unless you override the default when setting up the build). I'll refer to that as Apache's home directory from here on, and any relative file paths (those that don't begin with a leading /) are relative to that point.

2.3.2 First tests

Naturally you'll want to see results. Apache works on most systems without any further configuration effort, so let's get started. Your distribution directory contains directions on how to launch the server; for Linux or Unix systems, run this command:

```
bin/apachectl start
```

You should get a response back that says the server started. Fire up your favorite web browser and direct it to your own system. For Linux and Unix users, http://localhost/ or your real host name should work. If all is well, you'll get the default page Apache puts up as part of the installation, directing you to the included online documentation.

If the server doesn't start, or you don't get that page, round up the usual suspects:

1. If `apachectl` didn't start the server, examine the error message. If it reports a problem with the configuration file then the default configuration doesn't work on your system; go on to section 2.4, which is about changing the things you'll probably want to change anyway.
2. Apache may be configured to use another port; try http://localhost:8080/ (or your real host name) instead.
3. Look in the log files, `logs/error_log` and `logs/access_log`. Run-time errors are reported to `error_log`, so if the server started but then had a problem talking to your browser a message will appear there. If there aren't any messages in `error_log`, check `access_log`, which logs every transfer to each connected client. If there aren't any messages in `access_log`, then your browser didn't connect to your server in the first place.
4. Check for other conflicts on your system. Is there already a web server installed that is using the HTTP port? By default, Apache listens for requests on port 80, but it can be configured to use a different port if that's a problem for your system.

2.4 APACHE CONFIGURATION

Like most Open Source programs, Apache gets its knowledge of the system from text files.[1] Get out your favorite text editor and examine the contents of the `conf` directory in Apache's home; it contains a handful of files which are all readable by humans. Lines beginning with '#' (or any text after a '#' that's not in a quoted string) are comments; the rest is information for the server.

The main configuration file for Apache is `conf/httpd.conf`. The other files in the directory are auxiliaries (such as `magic` and `mime.types`) or vestigial configuration files that are left over from the way NCSA did things (`access.conf` and `srm.conf`). The .default files provide the original configuration info shipped with Apache, and can be used as a reference if you want to go back to the way things started.

Note that in discussing configuration here, we mean *run-time configuration*. Apache also has *compile-time configuration*, which uses a separate configuration file (`src/Configuration` in the source distribution) as well as other sources that say how Apache should be put together. One of Apache's strengths is the flexibility each site's management has in deciding what to include in its server. We'll discuss these compile-time options in each section that requires them.

2.4.1 The httpd.conf configuration file

A quick look at `httpd.conf` can be intimidating; it's a big file and it seems very complex. You don't need to digest it all at once, though, and as you've perhaps seen, Apache runs just fine without changing a single line.

The good news is that the file documents all of the defaults and explains which sections you might want to change and why. For further information you can look up each of those options (and the ones not included by default) in the Apache documentation that was installed with the distribution. Assuming you have a working server and browser, go back to the default page and click the documentation link. You'll get a page of documentation topics, including run-time configuration directives; go to that page for a list of everything that can be set.

The configuration file contains comments (any unquoted text after a '#') and *directives*; a directive is either a one-line command or a section containing a set of further directives.

One-line directives begin with the directive name followed by arguments:

```
MinSpareServers 5
MaxSpareServers 10
```

[1] Configuration files for Unix utilities are usually plain text delimited by white space. There is a movement toward using XML for configuration information; this format looks more complex at a glance, but it is also very readable once you get used to all the `<brackets>`. XML-based configuration files are easier for programs to parse, which in turn makes it simpler to write configuration helper programs.

These lines contain two directives, setting the values of the `MinSpareServers` and `MaxSpareServers` parameters to `5` and `10` respectively, meaning that Apache will keep at least five servers ready and at most 10 inactive servers). The lines can start and end with white space, and the amount of white space between the directive name and the arguments isn't significant as long as there is some separation. If it reads correctly to you, chances are good it will read fine for Apache.

Sections are bracketed by a pair of tags in the style of XML and HTML:

```
<Location /server-status>
    SetHandler server-status
    Order deny,allow
    Deny from all
    Allow from .your_domain.com
</Location>
```

The Location section starts with `<Location` *`argument`*`>` and ends with `</Location>`; between the < > brackets of the opening tag the directive works as a one-liner, with the directive name followed by arguments. Any number of directives can appear between the opening and closing tags, including other tag pairs that close off further nested directives. By convention we indent the enclosed directives to show that they apply only to surrounding tags, but again the white space isn't significant to Apache.

The most common of these section tags are `<Location>`, `<Directory>`, `<Files>`, and, if you are managing more than one host on a server, `<VirtualHost>`. `<Location>` specifies directives for a particular URL, while `<Directory>` applies to directories on a file system and `<Files>` applies to files that match a pattern. If multiple blocks all match a given file, `<Location>` has the highest importance, so you can set different rules for a file when it is accessed by different URLs. After that, `<Files>` overrules `<Directory>`.

2.4.2 Things you need to change

Actually, Apache doesn't need much configuration. Look for these sections and change them if the conditions apply to your site.

- *`ServerRoot "/usr/local/apache"`*—Put the home directory for the Apache server here if you aren't using the default. Then search through the rest of the file and change the same string everywhere that it appears.
- *`Port 80`*—If you have another web server running already, you'll need to tell your new Apache server to use a different port. 8080 is a common alternative.
- *`ServerAdmin root@localhost`*—Replace the default address with your email address.
- *`#ServerName localhost`*—This directive is commented out by default, and the server will use the regular host name for reporting its name (in error messages for example). If you'd rather use a different name, uncomment the line by removing the leading '#' and then replace the default with the name you want. Virtual hosts require this directive in each site's configuration section.

2.5 PRODUCTION SERVER

When actively building web sites, a development group often needs a set of servers in different roles: a “toy” server where the developers can experiment with things, a production server where the finished product runs, and perhaps something in between.

If you are working alone and your server isn’t exposed to the great wide world, the minimum configuration is fine for getting started and learning what to do next. If, however, you are setting up a group of servers, you should secure your production server immediately. While it’s unlikely that some cracker is lurking in the shadows waiting to pounce on your hapless web site, securing the server from the start will establish good habits all around by making developers learn how things need to work in the final installation.

Apache’s default configuration has reasonable security practices in place, but not as good as you might like. Read through the Security Tips page in the Apache online documentation, set the protections on your files and directories as shown there, then consider how permissive you want to be with your users.

After deciding on your site’s policies, you’ll want to look for the following directives in `httpd.conf` and change them accordingly:

`AllowOverride`—Permits or disables the use of local configuration files (usually called `.htaccess`) in directories, so examine carefully each occurrence in `httpd.conf`. In general, a production server should disable this at the top directory, and then enable it again if needed in specific directories or locations:

```
<Directory />
    AllowOverride None
</Directory>
```

This directive turns off the use of local configuration files in all directories. If you need to enable it again for a given directory—say, the document directory of a trusted user—you can use `AllowOverride` again to enable some features:

```
<Directory /home/sysmgr/public_html>
    AllowOverride FileInfo AuthConfig Limit
</Directory>
```

This allows sysmgr to set some directives in a .htaccess file in `/home/sysmgr/public_html`. Not all directives will be honored; look up `AllowOverride`’s documentation to see exactly what we’ve permitted.

Note that allowing local configuration files slows down the server, as it has to parse the local file for each request to a URL that permits them. This is not how you want to run your primary applications.

`Options`—Permits or disables a grab bag of server features, including the execution of CGI scripts and browsing of directories. For a secure server, the top directory should turn off everything:

```
<Directory />
    Options None
</Directory>
```

Options None is as tight as it gets, but you might consider allowing FollowSymLinks or SymLinksIfOwnerMatch. These two permit the use of symbolic links on Linux and Unix, which is convenient as long as users don't have general write permissions to important directories. SymLinksIfOwnerMatch allows links only if the link and the file it points to are owned by the same user. Using just FollowSymLinks is the best option performance-wise, since it doesn't require Apache to do extra look-ups along file paths. See chapter 12 for more information.

Again, the Options directive can be used in specific <Directory> or <Location> sections to open up permissions as needed.

What if we had this section and the previous one setting AllowOverride? They would both apply; Apache neatly merges sections that apply to the same target in the order they occur. The same applies to <Location>, <Files>, and so on. Sane administrators will want to do that merging manually though, so that it is more obvious what directives apply to which sections.

UserDir—This directive controls the mapping of username URLs; that is, http://www.example.site/~user. The default is to enable user directories for all users and map them to public_html, meaning that http://www.example.site/~bob/resume.html gets mapped to /home/bob/public_html/resume.html (assuming /home/bob is Bob's home directory). The argument can either be disabled or enabled (with a list of users following either) or a path can be mapped onto the user's home directory.

If your site doesn't have general user accounts, you can turn this feature off: setting UserDir disabled will turn off the mapping functions. Better yet, if you build Apache from source files you can leave out mod_userdir entirely. See the helpful files on compile-time configuration in the source distribution. If you want to let a few users do this but disable it for the rest, then use:

```
UserDir public_html
UserDir disabled
UserDir enabled bob carol ted alice
```

The more permissive variation is to disable user directories for sensitive accounts and leave it open otherwise:

```
UserDir disabled root www nobody ftp
```

If you allow user directories, then you should also have a <Directory> section that specifies AllowOverride and Options to control what users can do. The default httpd.conf contains such a section (possibly commented out), so modify it accordingly:

```
<Directory /home/*/public_html>
    AllowOverride None
    Options Indexes SymLinksIfOwnerMatch
</Directory>
```

This section disables the use of local configuration files and allows browsing of directories (that's what Indexes does) and symbolic links that are owned properly.

Mapping ~user URLs to the user's `public_html` directory is a typical scheme, but `UserDir` can also be set to map those requests onto an entirely separate directory tree with a subdirectory for each user. For example, this arrangement directs the requests to appropriate directories under `/www/users`:

```
UserDir /www/users
```

Whether using subdirectories of the user's home or a separate directory tree, each user chooses what files to expose on the web site by moving those files to the appropriate directory. Directory ownership and write permissions should be set accordingly. `UserDir` should never map requests directly to a user's home directory since that could make all subdirectories visible, thus removing the active choice. Another explanation given for this arrangement is that by not exposing the user's home directory, you also don't expose the various hidden files (those beginning with '.' on Linux and Unix) that contain sensitive information such as passwords for mail servers, which are stored by some mail clients. It's not a good idea to use such things on a server that has an open Internet connection.

A third possibility is to direct ~user URLs to another server entirely:

```
UserDir http://another.example.site/home_pages
```

This version causes `UserDir` to redirect requests for http://www.example.site/~user to http://another.example.site/home_pages/user.

`ScriptAlias`—Specifies a directory that contains executable scripts. This is one way to get the Apache server to execute a script for a URL, and is very secure as long as the administrator controls the scripts. A typical setting is:

```
ScriptAlias /cgi-bin/ /usr/local/cgi/
```

The first argument matches the beginning of a URL, and the second specifies the directory that contains the script for the match. For example, http://www.example.site/cgi-bin/hello_world.cgi would map to `/usr/local/cgi/hello_world.cgi`. Note that directories containing executable scripts should not be viewable by browsers. Set `DocumentRoot` appropriately and don't mingle the two.

That's one way to handle executable scripts. The others are via the `ExecCGI` option in the `Options` directive and by setting special handlers for given files. We'll cover special handlers in later chapters. Think twice about using `ExecCGI`, especially in user directories. In conjunction with `AddHandler`, this option can lead to Apache running arbitrary code. Even if your users are trustworthy, you have to trust that they are taking proper precautions with their directories, passwords, and so forth.

If you have opened up directory permissions, then also use a `<Directory>` section to lock down each script directory:

```
<Directory "/usr/local/cgi">
    AllowOverride None
    Options None
</Directory>
```

The `ScriptAlias` tells Apache that the directory contains executable scripts in spite of `Options None`, so everything is set and ready for testing.

After making all these changes, restart Apache and test the server again. You can even skip to the next chapter and grab a few CGI sample scripts to make sure that works. Your production server is now secure and running!

2.6 DEVELOPMENT SERVER

While it is possible to build a site using a single server, it's often handy to have a separate place for working on new programs and content (a "toy" server where security is loosened and developers can try out their code without a lot of bother). As compared to the production environment, a development server can allow open access to configuration files, documents, and scripts so that programmers can drop in new works quickly. Of course, this assumes that the server is not on an open Internet connection—it should be in a protected network or otherwise configured to refuse requests from outside its LAN.

Here are a few possible configuration scenarios:

2.6.1 Allow everything

The following directives allow Apache to serve documents and scripts from users' directories:

```
UserDir /home
<Directory /home/*>
    AllowOverride All
    Options All
    AddHandler cgi-script .pl .cgi
</Directory>
```

The `UserDir` directive maps ~user URLs to the user's home directory (assuming, of course, your users' directories are under `/home`), allowing any unprotected document to be read. We then open up permissions in those directories with `AllowOverride All` and `Options All`, and tell Apache to treat any file ending in .pl or .cgi as an executable script.

Each developer can tailor what Apache shows and does by appropriate use of .htaccess files, starting with one in his home directory that will be inherited by each subdirectory.

It's hard to imagine giving the users much more than this, but of course it is possible: we could give each user his own server! If you want to maintain the illusion of a single site but map each user to a different server, look at combining UserDir with a Redirect for each user to send requests to developer sites.

2.6.2 Allow documents and scripts

This scenario lets users publish documents and run CGI scripts, but only from certain directories:

```
UserDir public_html
ScriptAlias /~user/cgi-bin/ "/home/user/cgi-bin/"
<Directory /home/*/public_html>
    AllowOverride None
    Options Indexes FollowSymLinks
</Directory>
```

The UserDir directive maps requests for http://www.example.site/~user to /home/user/public_html, while ScriptAlias similarly sends http://www.example.site/~user/cgi-bin/ requests to /home/user/cgi-bin (and tells the server to execute the resulting file—no need for an AddHandler directive). Directories are browsable (Options Indexes) and users can manage their documents using symbolic links (FollowSymLinks), but can't override server configuration or options in general (AllowOverride None).

ScriptAlias tells Apache that the given directory contains executable scripts. The previous example works only for a user named "user." For this scenario to work, we need to add a ScriptAlias line for each user who is allowed to run CGI scripts. If all users are permitted, we can handle this in one directive using the Match variant of ScriptAlias:

```
ScriptAliasMatch ^/~([^/]+)/cgi-bin(.*) /home/$1/cgi-bin$2
```

The parts that look like cartoon characters swearing are *regular expression* matches; see your Apache documentation for more information on the Match variations of directives and how to use regular expressions in them. Briefly, this particular match looks for URLs of the form /~user/cgi-bin/* and translates them to /home/user/cgi-bin/*, where the real user name is substituted for "user" in both cases.

2.6.3 Allow approved documents

As shown in one of the production server configurations, UserDir can also be used to map ~user URLs to a separate directory tree. This variation stores users' public documents in subdirectories under /www/users:

```
UserDir /www/users
<Directory "/www/users/*">
    AllowOverride None
    Options None
</Directory>
```

This is as tight as it gets, assuming the directory protections are set properly. If write permission is restricted to the system administrator then only those files permitted by the boss will be displayed on the web site. In such a scenario it makes no sense to discuss executing scripts, since a CGI could display arbitrary files (and leads one to wonder if this is actually a development site).

2.7 USING APACHECTL

Having chosen a configuration and set things up properly, test your server again. This is a good idea after any change to a configuration file, even if the changes look trivial. If your configuration allows CGI then take an example from the next chapter and try that out too.

The apachectl program includes helpful code for verifying configuration files. This command checks that things are all proper:

```
/usr/local/apache/bin/apachectl configtest
```

There are three options for restarting the server, in order of severity: graceful, immediate, and hard.

A graceful restart allows Apache to complete work in progress; servers with open connections won't restart until they finish their transfers or time out waiting for clients. A server won't close an open log file until the server shuts down. Your users will appreciate having a chance to finish their business, but you might find it troubling if you are looking for an immediate change.

Trigger this restart using apachectl or kill:

```
/usr/local/apache/bin/apachectl graceful
```

or

```
kill -USR1 httpd
```

If you use apachectl, it will also run a configtest for you automatically, so you can be assured the server will start up again properly.

An immediate restart tells Apache to close log files and open connections and then read the configuration files again:

```
/usr/local/apache/bin/apachectl restart
```

or

```
kill -HUP httpd
```

The apachectl program will confirm the configuration files with configtest before stopping the server. If the server isn't running in the first place, it will just start it for you without an error.

A hard restart is necessary if you have changed the Apache program itself or made system changes that the server won't normally track:

```
/usr/local/apache/bin/apachectl stop
/usr/local/apache/bin/apachectl start
```

or

```
kill -9 httpd
/usr/local/apache/bin/apachectl start
```

Shutting the server down with apachectl is preferable, but not always possible if things are going wrong.

2.8 SERVING DOCUMENTS

Your site now has a functional web server that can present static documents and run CGI scripts. Most sites have some static content, and it's easy to manage once you learn how Apache translates a URL into a file path.

We've discussed URL mapping previously in the sections on `UserDir`. When Apache receives a request for a particular URL, it translates it into a file path and then figures out how to serve up that file. The rules for the latter can be quite complicated. Later chapters will explore them in examples that use handlers to take over this process for certain files or directories. The mapping rules themselves are generally simple:

1. If Apache is configured to handle multiple sites using `<VirtualHost>` directives, it looks at the site specification to see which set of rules to use. The site is the part of the URL between http:// and the next /. If there aren't any virtual hosts, this part of the URL is ignored and the general rules are applied.
2. The section of the URL after the site specification is the path, composed of words separated by /s; it can end in a file name such as `document.html`, a trailing path component or just a /. Apache evaluates `UserDir`, `ScriptAlias`, `<Location>`, and other directives to see if they match this path.
3. If the beginning of the path matches a `ScriptAlias` directive, the rest of the path is mapped onto the given directory and the resulting file is executed as a CGI script. Similarly `Alias` directives, `Redirects`, and other rewriting rules are applied.
4. If Apache is built with mod_userdir and `UserDir` isn't disabled, it checks to see if the path begins with a ~. If so, the first path component is considered to be a user name and the `UserDir` rules discussed previously are applied to map the rest of the URL as a file path onto the user's document directory.
5. If the rules in number four didn't satisfy the most, the path is considered as a file path relative to the directory given by the `DocumentRoot` directive, usually the `htdocs` subdirectory of Apache's home. Any `<Directory>` directives that match some or all of that file path are applied to the URL.
6. If the path ends in a file name and that file exists, `<File>` directives are checked and the file is served if permissions allow. The browser receives the file and displays it according to its rules and idiosyncrasies.

7 If the path ends in a trailing /, Apache checks for the existence of a default file (usually `index.html`) and serves that if it exists. Otherwise, if directory browsing is allowed (`Options Indexes`), Apache creates a document on the fly that represents a directory listing. Depending on the other options you allow and the number of icons supplied for file types, this directory listing can look like a desktop file browser or an FTP site listing.

8 If Apache hasn't figured out any other way of handling the path, it sends back an error document. These too are configurable; some sites have nice, apologetic error documents that offer an email address for sending complaints, while others reroute the user to a site navigation page. My favorite variations are those that use haiku:

You step into the stream,
but the water has moved on.
Document not found.

There is a bit more to it than that; when Apache decides what file to send to the browser, it also tries to figure out what type of file it is so that it can send along appropriate headers to the browser. These types are managed in the `mime.types` file in Apache's configuration directory. If you are serving up unusual content, you'll need to add types to this file so that browsers know what to do with the documents you send.

Suppose a browser sends a request for http://www.example.site/hello_web.html to our server. The path consists of just a file name, so there isn't much in the way of analysis to do; Apache looks up `hello_web.html` in the `DocumentRoot` directory and sends it back to the browser.

That file contains an IMG tag specifying a relative URL for images/hi.jpg. Assuming the browser is displaying images, it sends a request for that URL back to Apache. The URL has a path, images, and a file name, hi.jpg. Apache looks for directives that apply to the path, and finding none, maps it onto the document root as a simple directory path. It sends the file back with appropriate headers and the image is displayed.

That's static document handling in a nutshell. As I mentioned, there are plenty of ways to make even this process more complicated. Apache has a rich set of directives for rewriting URLs and managing changes of directories, file names, and so on that are inevitable over the lifetime of a site.

If that's all you need, Apache will serve you and your documents well. You might consider an alternative, however, one specially built for speed at just this task.

2.9 THTTPD

thttpd is a small and versatile web server. Its flexibility begins with its name: the 't' stands for tiny, turbo or throttling, take your pick. The author (Jef Poskanzer) offers the software from its web page (http://www.acme.com/software/thttpd/thttpd.html). While thttpd's design goals are much the same as those for Apache—a secure, stable web server that handles static documents and other tasks—thttpd is built for speed, while Apache is meant to be a general platform for many other pieces of software.

thttpd's feature list shows its focus: it serves static documents with a minimal implementation of the HTTP 1.1 protocol, offers a few utilities such as CGI processing, and a unique throttle utility which lets a webmaster allocate bandwidth to different sections of a site. It handles virtual hosts (see chapter 11), an absolute necessity for modern web sites. Other features such as redirection and server-side include (SSI) are given over to external programs, keeping thttpd small.

One reason for its lean size is it runs as a single process, unlike Apache with its separate children. The server listens on its socket for incoming requests and handles each in turn, but it doesn't bog down on any particular client. thttpd uses nonblocking I/O via `select` to fill each waiting socket and move on to the next, so it can feed documents to a large number of browsers concurrently. That single process is smaller than just one normal Apache child (of which a typical configuration has at least several hanging around waiting for work).

Thus by keeping a tight focus and using a minimum of system resources, thttpd provides a very high performance level for servers that run in small spaces. In a race with Apache, thttpd can serve documents faster per second than the larger server can, but the author points out that the bandwidth limitations of most sites limit the performance of either server more than memory or other system resources. Don't expect thttpd to improve on a site that has a network bottleneck.

In chapter 12 we'll discuss how thttpd can be used as a front-end server to handle static documents while a more heavyweight Apache serves dynamic content. Don't consider it just for a secondary role, however; it's a fine server for many workloads.

It is also easy to extend thttpd for more than static documents. PHP users can use the popular mod_php module directly to run their applications, and Freshmeat lists a thttpd variant, pthttpd, which has an embedded Perl interpreter for speeding up the kind of code I'll be talking about in the rest of the book. The web page for thttpd lists add-ons for SSL (see chapter 6) and other frequently needed options.

If you need only to publish static documents, Apache or thttpd will work fine for you. Chances are good you want more from your web server, however, and dynamic content is where the action is, so let's go on to the tools that make the exciting stuff happen.