Introduction to the Apache Server

ApacheCon 2000, London

Rich Bowen <Rich@Cre8tiveGroup.com>

Overview

Learn how to acquire, compile, install, and configure the Apache Web server. A great place to start if you are completely new to Apache, and trying to figure out where to start.

Introduction - What is Apache

The Apache web server project is more than just a piece of software. The Apache web server is the best, and most preferred, HTTP server software in use on the Internet today, and it was written entirely as a volunteer project, by volunteer programmers, in their spare time. That, in itself, is astonishing. That is, it is to people that are not familiar with the Open Source methodology, and Open Source projects like Linux, Perl, Sendmail, and a variety of others. The interesting thing about these volunteer-written, free software packages is that most of us, and our businesses, rely heavily on them, whether we are aware of it or not.

Before diving directly into talking about what Apache is, it is useful to talk about where Apache came from, and how it came to be.

The WWW

The Internet has been around for a long time. More than 30 years now. But for most of that time, it was entirely the domain of geeks and hobbyists. The main reason for this was that it was hard to use.

In 1991, Tim Berners-Lee developed something that he called the World Wide Web, while working at CERN. His purpose was to give quick and easy access to documents for geographically distributed people collaborating on projects. Along with a lot of help from the standards community (and, notably, Roy Fielding), they defined HTTP, HTML, URLs, and the other necessary components of making the Web a reality. He then went off, and with the help of colleagues around the world, communicating via email, developed the CERN web server, and a simple Web client, which he dubbed a ``browser." The name came about because there was very little of real value on the Web at that time, and all you ever really did was browse. Ironic that the name stuck!

NCSA

As more and more people got involved in the project, it was several Universities that contributed to the project the most. From very early on, one of the front-runners was the National Center for Supercomupting Activities (NCSA) at the University of Illinois at Urbana Champaign (UIUC). NCSA started working on the NCSA HTTPd (HyperText Transfer Protocol Daemon). Although that project is not active any more, you can still see the web site of the project at http://hoohoo.ncsa.uiuc.edu/ It still contains a wealth of information, most of which is still relevant, because the standards have not changed much in 8 years.

Rob McCool wrote the original code for the NCSA HTTPd, and this code was distributed without charge to the community, for them to use, with the understanding that if they fixed bugs, or added features, that they would then contribute them back to Rob to put into future versions.

The Apache Server

When Rob left the project, it left a problem. There were still a lot of people using his code, and actively making patches to the code, but there was no longer anyone collecting those patches.

In 1995, Brian Behlendorf and a small group of other developers started collecting these patches in a central repository. Brian got some space donated on a server, and set up a CVS tree so that developers could check in patches. And in April of 1995, they released the first official release (Version 0.6.2), which was given the name Apache, because it was ``a patchy server".

The Apache Group, as they were known at that time, had no formal organizational structure, never met, communicated only over email, and worked entirely in their free time, on a volunteer basis. Early the next year, Apache passed NCSA

as the most widely used server on the Internet, and is now used on more than 60% of all web servers on the Internet.

Apache's architecture

Since the 1.0 release of Apache (December 1, 1995) Apache has has a modular design. The core of the server is very light-weight, and all other functions are implemented as modules that plug in to the core. This means that you can keep the size of the executable down by leaving out functionality that you don't need. It also means that if there is some functionality missing that you do need, you can write your own custom module to plug into the core.

More recent history

In the last few years, Open Source has been getting a lot of press, because of Linux, Perl, and Apache. In 1998, IBM decided to abandon development of a web server engine to go into WebSphere - an application server for the web - and use Apache instead. This decision, along with Netscape's decision to release the source code for the Netscape browser, earlier that same year, showed the business world that Open Source was more than just a lot of long-haired anti-establishment types out to bring down the software industry, but that it was actually a good business model. It produces code more quickly, and that code is more reliable, because, in the words of Eric Raymond, with enough eyes, all bugs are shallow.

In June of 1999, The Apache Software Foundation was officially incorporated in the state of Delaware. The ASF has a much broader mission than just the Apache HTTP server, and has several other projects that exist under the larger umbrella of the ASF

The stated goals of the ASF are:

- provide a foundation for open, collaborative software development projects by supplying hardware, communication, and business infrastructure;
- create an independent legal entity to which companies and individuals can donate resources and be assured that those resources will be used for the public benefit;
- provide a means for individual volunteers to be sheltered from legal suits directed at the Foundation's projects; and,
- protect the 'Apache' brand, as applied to its software products, from being abused by other organizations.

Some of the better-known projects under the ASF are the Apache web server, mod_perl, mod_php, and Jakarta.

The future of Apache

At ApacheCon in Orlando, back in March, Apache 2.0 was released. This is largely a rewrite from earlier versions, and uses a threading model that will increase performance substantially on most platforms. As of this writing, version Alpha 6 of the 2.0 server has been released.

The Apache Group, as mentioned above, has become the Apache Software Foundation, and continues to take on new projects that seem to fit the larger vision that the ASF has for the future. Open Source, and open standards, produce better software. In the end, this makes life better for all of us, and we should support the ASF in all its endeavors, if only for purely selfish reasons.

Support for Apache

As an Open Source software product, Apache falls prey to the myth of no support.

There are two main ways to obtain support for Apache. First, there's the traditional email and Usenet methods. There are a variety of mailing lists on which you can obtain support for Apache. And there are two main Usenet groups - comp.infosystems.www.servers.unix and, for those running Apache on Windows, comp.infosystems.www.servers.mswindows

You can find information about the ``official'' Apache mailing lists on the Apache.org web site.

Another recommended mailing list for asking questions about Apache is the hwg-servers mailing list, run by the HTML Writers Guild. You can subscribe to that list, and find out more information about the list at http://www.hwg.org/lists/hwg-servers

Secondly, there's also commercial support for Apache, available through Covalent Technologies. Covalent offers support contracts for Apache, and they also have add-on products for Apache, such as Raven SSL. And the author of Comanche (which we'll discuss later) works at Covalent.

Obtaining and installing Apache

Apache is available as source code, and is probably available as a binary installation for your operating system, unless you are running something truly arcane and rare. And, of course, if you are, you can still get the source code, and compile it yourself.

Hardware/Software requirements

Apache runs on anything. Almost. It will almost certainly run on whatever you have. I've run Apache on a 386 with 4 MB of RAM. And I've run it on a 4-processor machine with 1GB of RAM. It was happy both places. The Apache.org web site does not list any hardware requirements. It will run on any hardware that runs the supported operating systems.

Apache will run on any flavor of *nix, and also on Microsoft Windows (95, 98, NT, 2000), Mac, and OS/2.

Obtaining

You can download the source code, absolutely free, of course, from the ASF web site at http://www.apache.org/

If you do want a binary distribution, just look in the binaries tree for your operating system - http://www.apache.org/dist/binaries/

Most distributions of Linux install Apache when you install the distribution. However, due to the nature of Open Source, what you install with your Linux is almost certainly not the latest version.

Compiling and installing

Most of the settings for your server, governing how it will operate, are done at the configuration stage, when you modify the configuration files that the server loads when it starts up. However, due to the modular architecture of Apache, a lot also depends on what modules you enable when you compile the server. The available configuration directives depend on the modules that are loaded.

You can either compile your server the quick, easy way, and get a default installation with the most common functionality, or you can get in there and pick and choose what you actually want.

The simple way

The installation process for Apache is really simple for most folks. If you are just wanting to set up a simple web site to do the normal things like serve web pages, and maybe do some CGI, the installation process looks like this:

```
tar -zxf apache_1.3.12.tar.gz
cd apache_1.3.12
./configure --prefix=/usr/local/apache
make
make install
/usr/local/apache/bin/apachectl start
```

Assuming you have a reasonably fast machine, this entire process does not take much more than 10 or 15 minutes, and you have a functioning web site. The `configure` process figures out reasonable settings for your system, and so the configuration files will have reasonable things in them so that you can immediately start serving web pages from your

server. The `--prefix` setting tells the configure process where you want to install the server. `/usr/local/apache` is the normal place to do this, but if you want to put it somewhere else, just specify that on the command line:

```
./configure --prefix=/home/rbowen/devserver
```

`apachectl` is a handy tool that Apache installs to make it simple to start, stop, and restart the server, as well as some other handy functionality. More about this a little later.

Advanced installation

If you're like me, you are not satisfied with the default installation. It does not have all the modules that I want, it has some stuff that I'd just as soon leave out, and there are some things I just do differently because I do them differently. I'm strange that way.

There are two ways to handle this that I'm going to talk about. First, you can actually edit the configuration file, and specifically choose what you want to compile into the server. Or, you can just throw everything in, but do it in such a way that you can go back and add and remove stuff at your leisure. I tend to go with the latter approach, but the former approach gets more coverage in the docs, and so is used more frequently.

You are advised to use the simple method above the first few times you install Apache. Also, in version 2.0, there will only be one installation method, and it will look more like the quick easy method above, than like these methods here.

Editing the configuration scripts

In our previous example, we ran a script called `configure` (``small-c configure'') in the main Apache directory. In this method, we're going to go down into the `src/` directory and actually look at the configuration files. After all, the motto of Linux is ``do it yourself.''

The process starts out the same:

```
tar -zxf apache_1.3.12.tar.gz
```

But rather than just going into the `apache_1.3.12` directory, you need to go down into the `src` directory:

```
cd apache_1.3.12/src
```

Then, using your favorite editor, edit the file `Configuration` (``big-C Configuration'') and comment, or uncomment, the lines that refer to options that you are interested in.

If you don't see a file called `Configuration`, copy the file `Configuration.tmpl` to `Configuration`, and use that as your template.

Once you have gone through and made sure that you have everything that you want in there, save the file, and run the following:

```
./Configure
make
make install
```

The simple method, which we talked about first, is doing all of this for you behind the scenes. If you run ``small-c'' `configuration`, you will have a file called `Configuration.apaci` created for you, which will then get used in this configuration step.

Dynamic Shared Objects (DSO's)

I get tired of rebuilding and reinstalling my web server every time I want to add in a new module, or when I decide to take one out, because I never really use it. This is where shared objects are handy. A shared object is something that gets loaded dynamically by a process when it needs it. This saves you from having to compile that code into the program executable, which, in turn, makes the executable smaller, and load up faster. By making your Apache modules

into shared objects, you can build everything into your server, but only actually use the parts that you want at any one time, and leave out everything else that you're not using.

On Windows, these are things called ``dynamic link libraries'', or DLLs. On Linux, they are called shared objects, or .so files.

In order to enable shared objects, you have to compile a module called `mod_so`, which, in turn, loads all the modules that you have compiled as shared objects. `mod_so` itself cannot be shared object, of course, because there would be no way to load it. Chicken, egg.

So, to build your Apache server to use shared objects, run the following commands:

```
./configure --prefix=/path/to/apache \
            --enable-module=most \
            --enable-shared=max
```

(You can either literally type those \ characters, or just put this command all on one line and omit the \'s)

What does this command do? Well, it compiles all of the modules that ship with Apache, except those that are considered experimental or unstable, and enables them all as DSO's.

This means that Apache will be loading up a bunch of modules that you don't really want, so you need to edit the configuration file and comment out those modules that you're not really going to use.

But it also means that if you want to add in a particular module, you can do so by putting it in the configuration file, rather than having to recompile your server from source. This is particularly handy if you change your server configuration a lot, or when you are testing out different configurations to see what it is that you want.

A server that is loading all the modules dynamically, rather than having those modules compiled in, takes a little longer to start up, but this penalty is paid only at server start, and after that the servers run at the same speed. That is, a server running modules as DSO's does not run any slower.

Configuring

Once you've compiled and installed your server, you need to configure it for your particular environment. Many of the configuration directives got set when you ran `configure` (or `Configure`), and so the server should work correctly immediately. However, you will probably want to change some things, since the default installation is very generic, and not precisely suited to your needs.

Apache, unlike most of its competitors in the web server market, lets you configure everything, down to the smallest detail. And if there's really something that you want to configure that you can't, you have the source code, so you can change it if you are so inclined.

Configuration files

The configuration for your Apache server is located in a file called `httpd.conf`, which is usually located at `/usr/local/apache/conf/httpd.conf`.

Note that if you installed Apache with a RPM (don't do that!) then the files will be in bizarre places that have no relation to logic. Uninstall the RPM, and install from source. It's a simple process, and reduces your pain in the long run.

Note: I made a comment like the above in one of my articles on ApacheToday.com, and got thoroughly chastized for it by some Red Hat fanatics that read the article. While I found their comments, and their reasons for their comments, to be rather amusing, I should emphasize that this is just my opinion, and should not be taken as some sort of transcendent truth. If you really want to spread your files all over your file system, go right ahead. You might notice, however, that a *default* installation of Apache puts everthing in `/usr/local/apache`, so it's a safe bet that the Apache developers agree with me on this one.

The format of `httpd.conf` is very simple.

There are comments, which consist of a hash sign (#) at the beginning of a line:

```
        # Based upon the NCSA server configuration files originally by Rob
McCool.
```

There are directives, which look like a name, followed by a value:

```
        ServerAdmin webmaster@rcbowen.com
```

There are sections, or containers, which look rather like HTML tags:

```
        <Directory /usr/local/apache/cgi-bin>
                AllowOverride None
        </Directory>
```

Sections can contain directives, and those directives apply to the the resources defined by the container definition. In the above example, the `AllowOverride` directive will apply to files located in the specified directory.

You can edit these configuration files with your favorite text editor. You need to restart the server when you are done editing the configuration files in order for the new configuration to take effect.

You can use the `apachectl` script to test your configuration file to make sure that you did not make any errors.

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

More about this below.

Comanche

One of the battles that *nix continually has to fight is the notion that it's hard to use. Much of this notion comes from the fact that everything you want to use on *nix has a configuration file, and every configuration file has a different format. Learning all these different formats is a pain, and it's so easy to get it wrong. Sendmail is one of the worst offenders in this arena, but even something as simple as Apache gets difficult to configure. Its modular architecture means that it can be extended forever, and every extension has its own configuration directives. This can be a little overwhelming.

Daniel Lopez took on this problem as his Master's thesis, and developed Comanche - the Configuration Manager for Apache. Comanche is a graphical configuration tool, written in Tcl, which lets you configure Apache in a more intuitive interface. It tells you what each directive means, and asks you questions that make sense. Your answers are put back into the configuration files in a format that Apache can understand.

Comanche can also be used to configure other applications, such as Samba, which have text configuration flies. There is not yet a plug-in for configuring Sendmail, but this is something that Daniel is frequently asked for, so perhaps there will be some day. And you can write your own extensions to Comanche to configure anything that has a text configuration file.

You can get Comanche at http://www.comanche.org/

Daniel now works at Covalent Technologies. Daniel is a member of the ASF, and you should attend his talks here at ApacheCon, if you have not already done so.

Starting, stopping, restarting

There are a variety of ways to control your Apache server. We'll focus on a script that ships with Apache, called `apachectl`, which does a few other things in addition to just starting, stopping, and restarting.

apachectl

`apachectl`, which presumably stands for ``Apache control'', is located in the bin directory of your Apache installation. It is a shell script which does many of the things that you'll want to do in controlling your Apache server. It can be run with any of the following arguments:

start

Starts the server.

stop

Stops the server.

restart

Restarts the server, if running, by sending a SIGHUP. If the server is not running, starts it.

fullstatus

Displays the full status of the server. Requires that `mod_status` is enabled, and that lynx is installed.

status

Displays a brief status report for the server. Requires that `mod_status` is enables, and that lynx is installed.

graceful

Does a graceful restart by sending a SIGUSR1, if the server is running. If the server is not running, it will start it. A graceful restart has the advantage over a simple restart in that child processes that are currently serving content will be permitted to complete their current connection before they are killed.

configtest

Reads the configuration file and parses it for syntax errors.

help

Displays usage information about the `apachectl` script.

Starting your Apache server on system restart

Linux has a process for starting processes on system startup. This consists of a directory `/etc/rc.d` containing scripts for each of the processes that you want to start.

If you place a file in `/etc/rc.d`, called `rd.httpd`, it will be run on server startup. `rc.httpd` should contain the following command:

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

If you're running Red Hat, or Mandrake, or one of the other Linuxes that looke like them, you'll find that there are a number of subdirectories of `/etc/rc.d` that look like `rc2.d`, `rc3.d`, and so on, which contain the startup scripts for all your various services. Actually, symlinks to them. On these systems you should create a file at `/etc/rc.d/init.d/httpd`, containing the command above. You should then create links to it from the directories `rc3.d` and `rc5.d`. Each of those directories corresponds to a runlevel. You'll usually be in either runlevel 3 or 5, so that's when you want to start Apache.

Integrating Apache with the rest of your business

The common wisdom is that every company needs a web site, because every company needs a web site. And so lots of companies have web sites which are nothing more than a electronic sales brochure.

With more and more people online every day, many users will expect to get just as good service from your web site as they would in person, or over the phone. In fact, the expectation is often higher. After all, this is a computer. They should be able to get direct access to the answers that they need, and it should be instantaneous.

Fortunately, I'm a technical guy, not a marketing guy, so I can only advise you on the tecnical aspects, not on business practices. However, I can say from experience, that if a company's web site does not provide me with the answers I need, I'm very likely to just go somewhere else for solutions. I don't have the time or patience to try to figure out bad web site.

CGI

The most common way to tie your web site to you database or other processing is with CGI programs. The Common Gateway Interface is a protocol to let your web server serve dynamically generated content from some process running on your server.

Of course, I've oversimplified it in that statement, but that's probably sufficient for our purposes.

A CGI program is a program written in any language you like, which formats its output in a certain way so that your browser can understand it. This allows you to write programs to put any of your existing databases onto your web site, and interact with your online customers in realtime, directly on your web site. You can let the customer customize their experience of your web site.

There are a plethora of good CGI tutorials on the web. (and even more bad ones!) But the basic concepts are pretty simple

MIME headers

Any output that your CGI program produces must be preceeded by a MIME header that tells the client (the browser) what sort of output they are receiving. This will look something like:

```
content-type: text/html
```

HTTP headers are followed by a blank line, which is how the client knows that the headers are done, and the next things that it sees are the body of the document. If you were to write a CGI program in Perl, for example, this would look like:

```
print "content-type: text/html\r\n\r\n";
```

`\r\n` is called a `crlf`, which is short for ``carriage return line feed.'' Frequently, you will see CGI programs that have just `\n`, rather than both, but it is more correct to use both. This used to be more of a big deal that it is now. Most web browsers are quite happy to accept one or the other.

Reading client input

Input from the client comes in to your program on STDIN. This means that you can read client input as though it was coming from the command line or from the keyboard.

Of course, most languages that you are likely to use for CGI programming have libraries readily available that will handle most of the mundane details of CGI programmig for you, and leave you to do your work.

For example, in Perl, there is the `CGI.pm` module, and a few others, such as the `CGI_Lite.pm` module, that handle such things as reading form input and managing cookies, and in the case of `CGI.pm` generating your headers and output. These modules are available from CPAN (The Comprehensive Perl Archive Network) at http://www.cpan.org/ and dozens of other mirror sites around the world.

In C, there are libraries available from Tom Boutell at Boutell.com which provide similar functionality from C.

Perl is the language of choice for CGI programming, because it is very conducive to the sort of rapid prototyping and development that is often demanded by the web.

Output in HTML (usually)

Since your output is going to a browser, you will almost always want to have your output in HTML. Occasionally, you'll want your output to be a gif image, or plain text.

Example CGI program

The following is an example CGI program written in Perl. It does not actually do anything useful, but it gives you an idea of what is the minimum necessary requirement for a CGI program.

```
#!/usr/bin/perl
print "content-type: text/html\r\n\r\n";
print "<b>Hello, World!</b>";
```

Not very exciting, is it?

Rather than provide a lot of example CGI programs, I'd encourage you to look at all the resources at the end of this paper for examples.

---

mod_perl

Something that you will eventually discover when using CGI is that it is slow. This has nothing to do with the quality of the code that you write, but is intrinsic to CGI. The problem is that every time a client requests a resource that involves running a CGI program, Apache has to launch that program. That takes a lot of time. This is the case whether the program is a Perl script or a compiled binary executable. Almost all of your time will be spent in the startup of that program, not in the actual run time of the program. There are programs with intensive database access, where this will not be the case, but they are in the minority.

There are a number of different technologies that address this problem. Most of them involve having your CGI programs somehow cached, so that when they are invoked, you don't have to pay that startup time, because they are already there in memory, ready to go.

Perhaps the most popular of these solutions is `mod_perl`. `mod_perl` is an Apache module that significantly enhances the performance of Perl CGI programs. It has other benefits, such as the ability to write Apache modules in Perl, but it is primarily used as a CGI enhancer.

Your Perl CGI programs are compiled, and kept in memory, so that every time the resource is requested, there is no time spent launching the Perl interpreter, or loading your program from disk.

`mod_perl` is extremely memory-intensive, since all this code is stored in memory. But the performance enhancements are several orders of magnitude, producing a very noticeable speed increase.

You can find out more about `mod_perl` at http://perl.apache.org/, or by attending one of the talks about `mod_perl` here at ApacheCon.

SSL and e-commerce

Even better than telling your customers about your business, is actually doing business online. You can put your catalog online, and let customers order, and pay for, your merchandise directly on your web site.

The one problem is that people are rather picky about who they give their credit card information to. And they are extrememly reluctant to type it in and submit it across the Internet without some assurance that what they are doing is secure.

By default, data that is passed to web servers is ``in the clear'', meaning that it is not encrypted, and anyone that is watching the wire would be able to see anything that went past. Like, for example, your credit card number. Even when you are in a password protected area on a web site, the username and password, and any data exchanged, is all passed in the clear.

One way around this is `SSL`. `SSL`, which stands for `Secure Socket Layers`, is a technology that encrypts traffic between the server and the client using a private key/public key technique. That means that only the people on the two ends can understand it. And even if someone were to intercept the entire message, they would not be able to decrypt it.

There are a number of SSL implementations that run on top of Apache. Two of the better known ones are Raven, from Covalent (http://www.covalent.com/raven/ssl/), and Stronghold (http://www.c2.net/products/sh2/). These are both commercial products.

The Open Source alternative is `mod_ssl` (http://www.modssl.org/), which runs in conjunction with `OpenSSL` (http://www.openssl.org).

With the recent changes in the crypto laws, there's a good chance that `mod_ssl` will ship as one of the standard Apache modules in future releases.

For more information on SSL, you should attend one of the SSL talks here at ApacheCon.

Authentication

Authentication is the process of verifying that you are who you say you are. This is usually accomplished by requesting some variety of username and password. There are a number of different implementations of this for use with Apache.

mod_auth

The ``standard'' Authorization technique is to use HTTP authentication provided by the Apache module called `mod_auth`. `mod_auth` is part of a standard installation of Apache, and is turned on by default.

To enable authentication for a particular directory, you need to do several things.

Create a password file

Using the `htpasswd` utility that comes with Apache, you need to create a password file, which tells apache what password is required for what username, in order to get to the resources in question.

The help for `htpasswd` says the following:

```
Usage:
    htpasswd [-cmdps] passwordfile username
    htpasswd -b[cmdps] passwordfile username password

    -c  Create a new file.
    -m  Force MD5 encryption of the password.
    -d  Force CRYPT encryption of the password (default).
    -p  Do not encrypt the password (plaintext).
    -s  Force SHA encryption of the password.
    -b  Use the password from the command line rather than prompting for
it.
    On Windows and TPF systems the '-m' flag is used by default.
    On all other systems, the '-p' flag will probably not work.
```

The `htpasswd` utility is (usually) located in `/usr/local/apache/bin`.

So, for example, to create a new password file, you would type:

```
htpasswd -c htpasswd rbowen
```

You will then be asked for the password that you want that user to have, and then you'll be asked to type it again to confirm it.

To add a password to an existing file, type the same command, but without the `-c`.

Create a group file

If you want to allow more than one user to have access to a particular resource, you can create a group of users. This is done by creating a group file which lists group names and the members in those groups. A line in the group file might look like this:

```
TCG: rbowen sungo chad tom
```

Put your files somewhere safe

You should store these files (the password file and the group file) somewhere outside of the document directory, so that they cannot be downloaded for leisurely off-line hacking.

Create a .htaccess file pointing at these files

In the directory that you want to protect, create a file called `.htaccess`, containing something like the following:

```
AuthName "Members Only"
AuthType Basic
AuthUserFile /path/to/htpasswd
AuthGroupFile /path/to/htgroup

Require group TCG
```

The `AuthName` is the string that appears in the authentication dialog that pops up when you visit a protected area.

`AuthType` is the method of authentication. It is one of `Basic` or `Digest`, but `Basic` is the only one of these methods that is widely implemented in browsers, so you should probably stick to that.

`AuthUserFile` and `AuthGroupFile` refer to the locations of the user file and group files that we created in the steps above.

`Require` is the directive that tells Apache what `user(s)` or `group(s)` can get the content specified. You can `Require` a particular user, or several users, rather than a particular group:

```
Require user Tom,Dick,Larry
```

The configuration detailed above will protect all files in a particular directory and all subdirectories thereof. You can also protect individual files with a <Files> section. See the documentation for more details.

mod_auth_db, mod_auth_mysql, etc

There are a variety of other modules that allow you to authenticate against usernames and passwords stored in a variety of other places, from DBM files, to MySQL databases, to Oracle databases, to Netware directory services. And a variety of other things. There are modules for authentication against a NT domain, or against Lotus Notes. Check out modules.apache.org for an impressive listing of Apache modules for these and other uses.

Log Files

Apache writes two log files as it runs - the `access_log`, which keeps a record of every request that your server receives, and the `error_log`, which keeps track of everything that goes wrong, or other less urgent information, such as server startup, stop, and restarts.

access_log

`access_log`, by default, is stored in the *common log format*, which contains the following information:

Address of the client

> The address of the remote machine requesting content from your server. This is usually just the IP address, but if you turn `HostNameLookups` on, this will be, whenever possible, the fully qualified domain name of the client.

ident

> The information returned by `ident`, or other similar lookup. This used to frequently actually contain the email address of the remote user, but this practice stopped as soon as it was realized that people were collecting this information for spam lists.

Username

> If the resource requested was password protected, this field will contain the username that was used to gain access.

date/time

> The date and time of the request.

Request

> The first line of the request that was made to the server

Status

> The return code the server returned to the client. 2xx messages mean everything went well. 3xx messages mean that the server redirected the request. 4xx messages mean the user did something wrong. 5xx messages mean that the server did something wrong.

Bytes sent

> How many bytes were actually sent to the client.

error_log

The `error_log` contains errors and various other messages that the server generates during operation. This is particularly useful for troublshooting CGI programs that are not behaving as expected.

Custom log files

With the `LogFormat` and `CustomLog` directives, you can create your own log files that contain whatever information you'd like to collect. See the Apache documentation for more details on generating these log formats.

Contributing to the project

Apache is a volunteer-driven project. That means that it relies largely on the users to contribute patches, suggestions, bug reports, and comments.

And big piles of money, of course.

If you think that you have any of the above, and you want to contribute them, here's how to go about it.

Within each project in the Apache Software Foundation, development is completely autonomous from the Foundation as a whole. Each project is left to manage affairs as best suits that project. Each project has its own web site off of the main www.apache.org web site, and you can usually find information on those sites about contributing code or documentation patches.

For example, for the XML projects, there is information listed at http://xml.apache.org/overview.html on how you can get involved in the project. There are links to the mailing lists (http://xml.apache.org/mail.html), the CVS repository (http://xml.apache.org/cvs.html), and the binary builds of the code (http://xml.apache.org/dist/).

If you would like to contribute financially to the foundation, you can find out more information about doing that at http://www.apache.org/foundation/contributing.html The ASF is not, at this time, a non-profit organization, so contributions are not tax-deductable. However, they might be elligible to be considered as part of the cost of doing business, if you use Apache software in your organization.

More information

Since Apache is entirely an online phenomenon, there is a lot of information online about it. Of course, the definitive source of information is the Apache Software Foundation, at http://www.apache.org/ but there other sites with additional information.

Covalent Technologies: http://www.covalent.com/ Covalent is the first company to offer commercial support for Apache. They also are the producers of Raven, one of the leading SSL solutions. And they host the development of Comanche, the GUI configuration tool for Apache.

ApacheWeek: http://www.apacheweek.com/ If you really want to keep up with what's happening with Apache, you need to subscribe to ApacheWeek. You'll get a weekly mailing with all the latest news. You'll know about the new releases before they happen. You'll know what the is going to be put into new versions, and what bugs have been found and fixed in the old versions. An absolute must for anyone dealing with Apache on a regular basis.

ApacheCon: http://www.apachecon.com/ Well, ApacheCon 2000 is already past, but there are more planned for the future. Submit papers and sign up to come to the next one!

ApacheToday: http://www.apachetoday.com/ Weekly articles about Apache from a few of the leading experts in the field. And me, too, into the bargain.

Apache Server Unleashed: http://www.apacheunleashed.com/ Buy the book. Don't wait for the movie.

Conclusion

Apache is the web server that you need to be using. There's really no question about it. 60% of all web site administrators can't be wrong.

When your web site is becoming such an important, integral part of your business, you really can't afford to be running anything but the best.