

Advanced Apache Configuration

Prepared and presented by: Ken Coar
VP of Conference Planning,
Apache Software Foundation and
Senior Software Engineer,
IBM Corporation

Topics:

- User Authentication
- Dynamic Content (CGI and Embedded Scripts)
- Virtual Hosts
- Content Negotiation
- The Proxy Server

Some Definitions

Config File:

A text file containing *directives* that control how the Web server should operate. Config files live in the `conf` subdirectory under the *ServerRoot*.

Container:

A paired set of *directives* that define a *scope* within which the enclosed directives have effect. Sometimes a container and the directives enclosed within it are called a *stanza*.

Directive:

A one-line text command in a *config file* which instructs the server in some aspect of its operation. Directive names are not case-sensitive, but the arguments that follow them on the line usually are. Examples: `User`, `DirectoryIndex`.

DocumentRoot:

The top of the directory tree where your Web content actually lives.

Internet Media Type:

An attribute of a document that describes the format of the content. The IMT is split into a major and minor portion, separated by a slash. Some examples are `text/plain` for normal unformatted text or prose; `text/html` for text that contains HTML tags; `image/gif` denotes a binary image in GIF format; and `application/octet-stream` is frequently used to identify unknown or arbitrary binary content. IMTs are not case-sensitive; `text/plain` and `Text/Plain` are equivalent. This same major/minor syntax is used in other cases, such as when a browser indicates what types of content it can accept (e.g., `text/*`).

MIME Type:

MIME is an acronym for Multipurpose Internet Mail Extensions, a set of standards defining how mail of various types and contents can be exchanged. Since the *MIME* system defined a very flexible way of assigning attributes to message bodies, it was adopted by the Web, so you'll frequently hear about the 'MIME type' of a Web document. The more general term is *Internet Media Type* (*q.v.*), though.

Overrides:

The types of settings that can be changed by directives in *.htaccess* files. If the `FileInfo` keyword is in the list of allowed overrides, for example, that means that directives that affect file information in *.htaccess* files can *override* any settings declared at a broader *scope*.

Scope:

A bounded portion of your server's URI space or filesystem. Scopes are nestable, and *directive* effects typically are inherited from higher-level scopes. The narrowest applicable scope ultimately defines the attributes of Web resources within it.

ServerRoot:

The top of the directory tree where the server application itself, and all the configuration files, lives.

Stanza:

See *container*.

Virtual Hosting:

A means of making a single system appear to be multiple ones.

Implementing user authentication

This topic is covered in detail by other sessions at this conference, so this section is only the high-altitude view.

‘Authentication’ and ‘authorisation’ frequently get confused. Authentication is the process of proving an association or identity between an agent (the end-user in our case) and a defined degree of trust. It’s essentially making someone show an ID card and prove that it’s hers. Authorisation, which comes *after* authentication, is the process of matching up this proven identity with a set of access rights. It’s quite possible for the authentication step to be passed, but to find the user isn’t authorised to access something.

Let’s deal with authentication first.

There are two ways in which Apache can do authentication. In common security parlance, they’re called *discretionary* and *nondiscretionary* access control, depending upon whether the information supplied (called the *credentials*) is at the discretion of the user trying to gain access. A good way to think about it is that authentication is controlled by what you know, what you have, what you are, or some combination of these.

Nondiscretionary mechanisms rely on aspects of the request that are fundamental and unchangeable, like the IP address. A client system can’t change its IP address, or the server wouldn’t be able to reach it through the net. A door lock that decided whether or not to let you in based on your DNA would be nondiscretionary in nature; you can’t change your genes. This is a ‘what you are’ system.

Discretionary mechanisms rely on something the user provides, such as a username and password. To use the door lock idea again, a combination lock is a ‘what you know’ type, and a regular key lock is a ‘what you have’ type. The problems with these are that multiple people can be privy to the secret for ‘what you know’ systems, and someone could steal (or make a copy of) the key for a ‘what you have’ lock.

The nondiscretionary controls in Apache are based on the client’s IP address or domain/host name. Access can be explicitly allowed or denied on an address-by-address basis, or by CIDR or subnet address, or by name. In the case of controls that use host or domain names, Apache does what’s called a *double-reverse lookup*, which means that it translates the user-agent’s IP address into a name, and then translates that name back into a set of addresses again. If the original address is in the final list, it’s considered valid and the allow or deny rule is put into effect; otherwise, the client doesn’t match the condition and the server treats it as not being in the claimed domain at all.

Discretionary access controls depend upon the credentials supplied by the user in response to the pop-up username/password dialogue box the browser presents when the server tells it the requested resource has restrictions. The credentials the user supplies are compared against whatever databases have been defined to Apache as being applicable for the resource; a match on both means authentication has succeeded and the processing can proceed to access checking.

Note: These user credentials are transmitted over the net in what amounts to plaintext; *i.e.*, they’re not encrypted at all. This makes them easy to intercept and steal. In addition, most (if not all) Web servers out there now – including Apache – don’t put any restrictions on the number of consecutive authentication failures from a particular source, so a cracker can easily bang away at a Web server with a dictionary attack.

The access checking mechanisms in Apache are quite simple: they merely specify what combination of successfully authenticated credentials are allowed access – all others are denied. These combinations can be specific usernames, any valid username, and valid username that is Apache’s databases as being part of a particular group, and so on.

Dynamic Content

This topic is also covered in by other sessions, so this is another high-level overview.

Dynamic content consists of resources that are tailored somehow before being presented to the end-user, or that allows the end-user to interact with the display somehow. The tailoring may be based upon the user's preferences, or the results of earlier requests, or the latest stock-ticker results, or the phase of the moon, or anything at all.

There are three basic ways of implementing dynamic content. Two are performed by the server providing the resource, and the third is performed by the end-user's browser or application. Not only is the latter highly dependent upon the user's application to function correctly, but since it isn't a server aspect at all and therefore isn't related to Apache, I'll say no more about it other than to mention that JavaScript is an example.

CGI Scripts

CGI, the Common Gateway Interface, describes a way for servers (primarily Web servers) to pass some or all of the responsibility for handling client requests to external applications or scripts. CGI scripts can be incredibly flexible and powerful, but they have the disadvantage of being slow and expensive to operate on high-volume sites. There are some ways of improving this, though, such as FastCGI and mod_perl.

Since the server is 'subcontracting' part of the request processing work, this is obviously a server-side solution.

Embedded Scripts

Another way of making content dynamic is to actually embed instructions of some sort in the resource content itself, so that a scripting engine can process them at delivery time. This is how JavaScript works; the instructions are aimed at the end-user's browser. This method has the disadvantage of being at the mercy of the browser application; it might not support the scripting language, or support only a different dialect, or the end-user may have disabled it.

Two other examples are PHP and SSI (server-side includes), both of which are processed by the server itself, and thus are immune to the vagaries of the spectrum of client browsers. The disadvantages to this method include the extra time and system resources that are needed to do the processing on the server.

Setting up virtual hosts

Virtual hosting is a way of setting up your server¹ so that it can appear to be multiple Web sites at once, depending upon how it's accessed by clients. The Apache Web server software allows you do create as many such virtual site identities as your system's limits will allow; this can be on the order of a few dozen to several thousand, depending upon your operating system.

A virtual host is declared in the Apache configuration files with the `<VirtualHost>` container directive. This can be a fairly complex directive, syntax-wise, but I'm only going to cover the basics here.

¹ Virtual hosting is potentially available with other types of server operations, too, such as FTP and mail. The availability depends upon the specific software in use.

In the simplest case, as shown in Figure 1, a `<VirtualHost>` stanza starts by defining the IP address to which the virtual host identity applies.

Note: You *can* specify a hostname rather than an IP address in the `<VirtualHost>` declaration, but this is discouraged; changes in IP addresses often silently break existing Apache configurations this way. The IP address is completely unequivocal.

```
<VirtualHost 10.0.0.1>
  ServerName WWW.Mydomain.Com
  ServerAdmin Webmaster@Mydomain.Com
  DocumentRoot /home/httpd/mydomain
  CustomLog logs/mydomain_log CLF
</VirtualHost>
```

Figure 1 A Sample Virtual Host

Within the stanza come the directives that apply only to that virtual host. Almost all of the standard Apache directives can appear in here, but there are two that are essential to proper operation:

- **ServerName** – It is critical that Apache know the name associated with a virtual host declaration, so that it can be sure it assigns requests to the right one and can construct references to it. The `ServerName` directive does this.
- **DocumentRoot** – This tells the Apache software where the tree of content for this particular virtual host can be found. Obviously, if you have multiple vhosts on a machine you're probably going to want them to have unique content; this is how you accomplish that.

Other directives that are frequently found inside `<VirtualHost>` containers include the various logging controls (`CustomLog`, `TransferLog`, `ErrorLog`, and so forth). One of the unfortunate side-effects of these directives is that each open log file typically means one fewer client connections that can be handled. There's usually an operating system-imposed limit on the number of open I/O channels, so it's generally a better idea to combine the access log files into one that includes the target virtual host ID – you can then split that single file out into multiple *per*-host logs later, or even at run-time by sending the combined access log to a splitter process with something like the following

```
CustomLog "|logfile-splitter" "%v %h %l %u %t \"%r\" %s %b"
```

which will prepend the logging virtual host's name to the front of a Common Log Format record. The Apache distribution includes a Perl script that can demonstrate this on a combined log file; see the file `src/support/split-logfile`. This script has *not* been tested as a run-time splitter, only with files that have already been created and aren't open.

Name *versus* Address

In the early days of the Web, virtual hosting was done strictly on the basis of IP addresses. That is, if a single system were to pose as five different Web sites, it would need to have five different IP addresses assigned to it.

This simple mechanism is called *address-based* virtual hosting. It has the advantage of instant and unambiguous identification of which host the client wants, based on the address to which it connects. It has the disadvantage of requiring a fair amount of infrastructural maintenance, getting and keeping all the IP addresses registered properly – which is complicated by the fact that the name-to-address and address-to-name translation tables are usually maintained by different people on different systems.

The industry received a real ease-of-implementation boost with the advent of *name-based* hosting. Name-based hosts allow a single IP address to have multiple identities. This means that only one side of the translation table – the name-to-address one – needs to be maintained. The main disadvantage of this mechanism is that it requires additional information from the client, after the connection has been

established, in order to determine which virtual host is involved. If the client doesn't supply that information, the server has no way of knowing which Web site it wants.

If you're setting up address-based virtual hosts, you need only prepare a `<VirtualHost>` stanza for each and specify the appropriate address in the opening directive. The address identifies the host, not any names that may be assigned to it, so it's actually possible that an address-based host might end up dealing for requests for a Web site name that isn't listed in its declaration, but *is* assigned to its address.

Setting up name-based virtual hosts is a little more complicated, but only a little. Regardless of how many IP addresses may be assigned to a system, only one of them may be used for name-based hosting. You need to tell Apache which address will be used this way with the `NameVirtualHost` directive:

```
NameVirtualHost 127.32.64.128
```

Every time Apache spots this address in a `<VirtualHost>` declaration it knows that the stanza is defining a name-based host.

While address-based hosts are identified solely by their addresses, name-based hosts need a name. IP addresses can be assigned to multiple names, so a single address-based host can respond to multiple names if they're associated with its address. In order to pull this trick with a name-based host, the assignment of multiple names must be done inside the Apache config files, since it can be done by the DNS name service. The two main directives that name a virtual host are `ServerName` and `ServerAlias`. The `ServerName` takes a single name and defines the primary identity of the host, while `ServerAlias` can take a list of space-separated alternate names, as shown in Figure 2. You can also include multiple `ServerAlias` directives in a `<VirtualHost>` container; their effect is cumulative.

```
NameVirtualHost 10.0.0.1
<VirtualHost 10.0.0.1>
    ServerName WWW.Mydomain.Com
    ServerAlias WWW.X.Com WWW.Y.Org
    ServerAdmin Webmaster@Mydomain.Com
    DocumentRoot /home/httpd/mydomain
    CustomLog logs/mydomain_log CLF
</VirtualHost>
<VirtualHost 10.0.0.1>
    ServerName WWW.Other.Org
    ServerAlias WWW.Z.Net WWW.A.Org
    ServerAdmin Webmaster@Other.Org
    DocumentRoot /home/httpd/other
    CustomLog logs/other_log CLF
</VirtualHost>
```

Figure 2 Name-Based Virtual Host

Each of the two name-based virtual hosts defined in Figure 2 will respond to three different Web site names, as long as the requests come in to IP address 10.0.0.1.

```
<VirtualHost _default_>
    ServerName WWW.Default.Com
    DocumentRoot /home/httpd/default
    CustomLog logs/default_log CLF
    ErrorLog logs/default_error_log
</VirtualHost>
```

Figure 3 The _default_ Virtual Host

What happens to requests that come in to the `NameVirtualHost` address, but don't specify a name that matches one of those in a `<VirtualHost>` container? By default they'll be assigned to the first name-based virtual host defined in the config files. You can change this behaviour and define a special host to deal with these by using a special name in the

`<VirtualHost>` declaration: `_default_` (see Figure 3).

The Default/Global Server

Many if not most of the configuration aspects of your virtual hosts can be inherited from the *global* server environment – that is, the directives that appear outside *any* `<VirtualHost>` containers.

Before the Web really took off, have a single Web site *per* server was pretty much the rule. Since there was only one host, it wasn't really considered virtual, and so the directives defining the server's behaviour were just listed normally in the server config files. Now that servers can have multiple identities this 'global' environment usually isn't used to handle requests any more, but rather defines a set of default conditions that can be inherited – and overridden -- by the specific virtual host definitions. Unfortunately, this capability has evolved rather than having been designed from the outset, and so the way in which any particular directive gets inherited is largely specific to that directive. Defining all the inheritance rules is outside the scope of this presentation, but a good topic for a chapter in a book about Apache.

Example directives whose effects can be safely inherited include the `ErrorLog` and `CustomLog` directives, and the environment-variable settings created by `SetEnv` and `BrowserMatch`.

Content and language negotiation

Although the term implies that the client and the server dicker back and forth until they arrive at a mutually satisfactory compromise solution, that's not really what happens when a document is negotiated.

Instead, the client provides a set of preferences concerning the preferred attributes of a document, and the server tries to come up with the best match from the versions available. If there's only one version of a document, then of course that's what the server is going to send, unless there's something about it which is completely unacceptable.

The set of all versions of a document that are available to the server is called the resource's list of *variants*. The ways in which they differ, such as by language or encoding, are called its *dimensions*. The Apache server can negotiate along the content-type, encoding, and language dimensions; if you have two versions of a document that are the same in these attributes but differ in some other way, Apache cannot use the other attributes in its calculations.

If the Apache server is unable to calculate its way to a single unique document that best fits the clients preferences, it will respond with a page saying so but also listing all the variants it *did* find. This allows the end-user to possibly select one manually, since the automatic selection didn't work.

The primary use of negotiation on the Web to-day is to select translations of text. Typically a work is prepared in a default language such as English, and the result put in a file named `text.html` or some such. Then translated versions are prepared, and each is put into an appropriate-named file: `text.html.da` for Danish, `text.html.fr` for French, `text.html.en-gb` for English (Great Britain dialect), and so on.

Note: Apache *does not* provide translation facilities! It will make decisions amongst documents that *you* provide and label as being in different languages, but it won't provide any content itself.

You can either prepare a list of variants yourself, which gives you greater control but requires more work, or you can allow Apache's `mod_negotiation` module to calculate the list itself by scanning the directory. The latter method is more common, since it is easier and automatically picks up new variants as soon as their available, but it *does* impose a slight performance penalty due to the directory scan. Hopefully the directory information will be cached by the operating system and so won't require constantly pounding on the disk.

The automatic scanning capability is controlled by the `MultiViews` option. Scopes which include this option are candidates for automatic negotiation. You enable this with one of the directives shown in Figure 4. The second format is preferred, since it selectively enables the option without disturbing any other options settings in effect. You can selectively *disable* the option within a scope with a corresponding `Options -MultiViews` statement.

```
Options MultiViews
or
Options +Multiviews
or
Options -Multiviews
```

Figure 4 Controlling Variant Scanning

If you choose to prepare your own variant list, you do so by creating a *type-map* file which lists all of the document's variants and their negotiation attributes. One negative side-effect of this method is that it is typically *not* transparent – in order to activate the negotiation process, the client needs to access the type-map file itself rather than one of the variants. In other words, the client would need to ask for `foo.var` rather than `foo.html`.

Apache uses the filename suffixes in its negotiation calculations. The filename

```
<http://www.apache.org/docs/content_negotiation.html>
<http://www.apache.org/docs/mod/mod_negotiation.html>
```

Figure 5 Negotiation Documentation

`text.html.fr.gz` actually appears on all three of Apache's available dimensions: the `.html` identifies the content-type, the `.fr` identifies the language, and the `.gz` identifies the encoding. That's typically the order in which dimension suffixes appear, but as far as Apache is concerned the file could easily have been named `text.gz.html.fr`.

Note: It's best to list your negotiation suffixes in the type/language/encoding order, however, in order to keep things simple and avoid confusion.

Although the language dimension is usually applied to documents containing text, it works just as well for other types of content. You can use it to distinguish between GIF images containing French and Spanish logos, for instance.

In order to let Apache know that a particular suffix is associated with a language, you use the `AddLanguage` directive. This specifies both the language code used by the Web software, and the suffix *you're* going to use to represent it. Typically these are the same, but they need not be; for instance, the Web language code for Polish is "pl", but that also happens to be the common suffix for Perl scripts, so you might resolve this by telling Apache you'll use ".po" to represent Polish documents:

```
AddLanguage pl .po
```

As with most directives of this type, you can specify one language code and multiple suffixes.

Order is unimportant when it comes to Apache's calculations. All variants along a particular dimension are considered to be of equal value, which can lead to the 'I couldn't choose the best one, so here's a lit – *you* pick the one you want' response page. There are two ways of imposing priority on the selection process; one is defined by the Web standards, and one is specific to Apache.

Priorities along negotiation dimensions are assigned by *quality*, or *q*, values. Q-values range from 0.000 (absolute worst quality, totally unacceptable) to 1.000 (absolutely perfect, accept no substitutes). Unfortunately, there is no easy way, short of using `.var` type-map files, to assign quality values to specific variants.

The Apache `LanguagePriority` directive lists language codes in the order they should be considered; codes specified in `AddLanguage` directives but not listed in `LanguagePriority` are all considered

equal and come after the last one on the `LanguagePriority` list when the calculations are performed. This priority list provides a tie-breaking capability in the case of no clear winner due to quality settings along the language dimension.

The Apache server also applies some simple heuristics to fill in the blanks:

- Any variants with a language code that isn't listed in a `LanguagePriority` directive (*i.e.*, they appear only in an `AddLanguage` line) are given an implicit Q-value of 0.001 – as low as possible without being unusable.
- Any variants that Apache recognises as scripts (*e.g.*, with a `.cgi` suffix) are assigned a very low quality value, based on the presumption that prepared material is of higher quality than content constructed on the fly.

Apache as a proxy server

In Web terms, a proxy server is one that stands between the end user and the server that has the master copy of documents being requested. In practice, proxy servers are frequently used in firewalled environments to provide a single point of contact between the users inside and the Web outside the firewall. The users ask for a page, the browser sends the request to the proxy server, the proxy server sends the request to the origin server, and the data flows back the same way.

A second function commonly associated with proxy servers is caching. It's so common, in fact, that many consider the cache function to be an integral part of the proxy function. The caching portion of a proxy server's operation involves keeping a copy of many of the documents that it fetches for its users, and checking future requests to see if they can be satisfied from the local copy rather than having to bother the network with another fetch.

Apache's proxy module provides both of these capabilities. Before you leap to activate it, however, consider the following potential drawbacks:

- If you only have one proxy server, all of the users that reference it will be unable to access the Web if it goes down; it's a single point of failure.
- Using a caching proxy server is not a universal win; having all requests go through one, for instance, can result in up to double the amount of time it takes for the end-user to get its document. The cache consumes disk space, and keeping the cache fresh requires server CPU cycles. And finally, if you have a very diverse user base, with few overlapping interests or a very broad spectrum of them, the cache may end up thrashing because it's filled with documents that get thrown out to make room without having been accessed more than once or twice.

To use the proxy functions in your Apache Web server, the `mod_proxy` module must be activated, either by being statically linked in (less common than formerly), or turned on at run-time with directives such as those shown in Figure 6.

```
LoadModule proxy_module libexec/libproxy.so
:
AddModule mod_proxy.c
```

Figure 6 Activating the Proxy Module

Note: The location of these directives in the configuration file, particularly the `AddModule` line relative to other `AddModule` lines, is extremely important. If you use the `APACI` (`./configure; make; make install`) method, these are set up properly in your configuration files.

Once the proxy module is active in your server's configuration, you need to actually instruct it to do work, with the various proxy directives.

- `ProxyRequests` – This is the 'master directive,' which enables and disables proxy operation overall. It takes a single keyword argument, which must be either `On` or `Off`.
- `ProxyRemote` – This is used to tell the Apache server that certain requests are to be forwarded to yet another system rather than being handled locally, allowing for cascading of proxies. See the documentation for more detailed information.
- `ProxyBlock` – This directive allows you to selectively block the fetching of certain resources through the proxy. It takes a list of space-separated case-insensitive phrases or URI substrings which should be blocked. "`ProxyBlock foo`" would block all requests in which "foo" appeared, in any combination of upper- and lower-case, including "`http://foo.com/bar/`" and "`http://bar.org/zardofoo.html`". You can block all proxy requests with "`ProxyBlock *`".
- `ProxyPass` – Maps a remote server's URI space into that of the local server, making the documents appear as though they were on the local system. This is a very powerful capability, allowing you to pull together multiple server namespaces into a single virtual one. As `Alias` maps URI space on to the filesystem, `ProxyPass` maps local URI space onto a remote server's. For example:

```
ProxyPass /frobozz/ http://www.frobozz.cc/
```

This would make the content of the Frobozz.CC Web site appear as though it were in the `/frobozz/` directory of yours.

Note: This can be very rude, and lead to charges of fraud if done egregiously and without permission. However, it can be very useful for stitching your own sites together.

Note: This directive doesn't involve the client at all, and isn't dependent upon the other aspects of proxy operation, so it still works even if you've got a "`ProxyRequests Off`" directive in your config file.

- `CacheRoot` – Defines the directory that's the top of the cache hierarchy. The Apache proxy module stores its cache in a directory tree under this location, so it should point to a filesystem large enough for your desired cache size. The directory hierarchy must be writable by the Apache server user (*e.g.*, `httpd` or `nobody`). Don't expect to be able to snoop through the cache tree and have all the names make sense; the Apache server uses its own naming conventions for the stuff it stores.
- `CacheSize` – This directive specifies the largest number of kilobytes of space you want to allocate to the proxy cache.

Note: The total amount of space in the cache tree *may* grow beyond this size, but it should be trimmed back the next time the Apache garbage collector runs.

- `CacheDirLevels` – This directive controls how deep the cache directory tree is allowed to grow, and probably shouldn't be changed from the default.
- `CacheDirLength` – Used to define the number of characters used in naming each level of cache subdirectory. Leave at the default.
- `NoCache` – Use this directive to identify proxy-accessed documents that shouldn't be cached. The format is the same as for the `ProxyBlock` directive.

- `CacheGCInterval` – This defines, in hours, how often the cache garbage collector should run. When the GC starts up, it scans the cache and throws out old or too-large items until the space is below the `CacheSize` threshold again.
- `CacheMaxExpire` – This sets the maximum age of cached documents in hours. Regardless of other aspects of the cache, no cached resource is permitted to hang around longer than this; once it reaches this age, it will automatically be thrown out the next time the garbage collector runs. The default value is 24 hours.
- `CacheLastModifiedFactor` – Two of the resource attributes that the proxy uses to test the freshness of items in the cache are any `Expires` and `Last-Modified` header fields it saw when it put them into the cache. Items with an `Expires` value will be considered stale once that date and time comes and goes. In a complex way, this directive controls how the proxy calculates the sell-by date of cached items that don't have an `Expires` value but *do* have a `Last-Modified` one. Basically, the floating-point value of the directive (the default is 0.1) is multiplied by the difference between the `Last-Modified` date and the current time, and the result is added to the time the item was put into the cache. The result is the cache expiration date of the item. If the `Last-Modified` date was seven days ago, it was two hours ago, and the value of the directive is the default of 0.1, then the expiration time will be set to $7 \times 24 \times 0.1 - 2$, or 14.8, hours from now. The net effect is that documents that haven't changed recently are considered more stable and worthy of caching for longer. This value is only calculated when a resource is first put into the cache, and is minimised with the value of the `CacheMaxExpire` directive.
- `CacheDefaultExpire` – If a resource has neither a `Last-Modified` nor an `Expires` field associated with it, it is given this value, which is a number of hours, as its default cache lifetime.

Usually a proxy server is set up as a completely separate virtual host from the rest of the server's operation (which is likely to be handling normal requests). You can do this with a `<VirtualHost>` stanza similar to that shown in Figure 7. The `<IfModule>` container will only be evaluated if the proxy module is part of the running server.

```
<IfModule mod_proxy.c>
  Listen 10.0.0.1:8080
  <VirtualHost 10.0.0.1:8080>
    ProxyRequests On
    DocumentRoot /home/httpd/htdocs/proxy
  </VirtualHost>
</IfModule>
```

Figure 7 A Proxy Host