Pro Apache, Third Edition

PETER WAINWRIGHT

Apress™

Pro Apache, Third Edition

ISBN (pbk): 1-59059-300-6

Printed and bound in the United States of America 12345678910

Technical Reviewer and Contributing Author: Bradley Bartram

Editorial Board: Dan Appleman, Craig Berry, Gary Cornell, Tony Davis, Steven Rycroft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Tracy Brown Collins

Development Editor: Robert J. Denn

Copy Editor: Kim Wimpsett

Production Manager: Kari Brooks

Production Editor: Laura Cheu

Proofreader: Nancy Sixsmith

Compositor: Diana Van Winkle, Van Winkle Design Group

Indexer: Kevin Broccoli

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 7

Hosting More Than One Web Site

In many cases you may need to host more than one Web site. Providing a separate server for each site is certainly one solution, but it can be expensive both in terms of cost and in terms of maintenance. Assuming you want to serve multiple sites from a single server, you have four different approaches available to allow Apache to host more than one Web site:

User home pages: The simplest approach is to group all sites under one controlling hostname. This is most suitable for situations where the administrator wants to give individual users the ability to maintain their own home pages without reconfiguring the server each time a new user is added. Apache supports this model with the `UserDir` directive. Users may use limited configuration by creating `.htaccess` files inside their directory.

Separate servers: This approach solves the problem by running more than one instance of Apache at the same time. Each Web site is served by a different invocation of Apache configured with a different IP address and port number, all running on the same server. Although as an approach this is a little heavy, it consumes far more memory than a single Apache instance and there's no sharing of resources, but it helps to increase the security and reliability.

IP-based virtual hosting: Instead of serving different IP addresses with different servers, Apache can serve all addresses simultaneously from one configuration using the powerful *virtual hosting* feature. Not only does this allow multiple Web sites to share the same pool of server processes, it allows them to share configurations too, making the server both easier to administer and quicker to respond. However, each Web site still needs to have its own IP address. The server therefore needs either multiple network interfaces installed or, alternatively, the ability to multiplex several IP addresses on one interface, known as *aliased IPs*; all modern operating systems can do this.

Name-based virtual hosting: Apache also supports virtual hosting based on its name. This allows multiple Web sites to share the same IP address. This also uses requirements in HTTP/1.1 (specifically, the `Host:` header) that allow Apache to determine which host is the target of the request sent by the client. The advantage of name-based virtual hosting over IP-based virtual hosting is that the server's network configuration is much simpler. However, a drawback is incompatibility with pre-HTTP/1.1 clients. Fortunately, these are now rare and account for 1 percent or less of Internet traffic; but if that 1 percent might be important to you, then you need to consider name-based hosting carefully.

It's perfectly possible to mix all of these options in various ways. In addition, you can also combine several different approaches to defining IP or name-based virtual hosts.

In this chapter, I'll present all of the previous solutions for hosting more than one Web site, as well as show some of the issues surrounding virtual hosting. I'll also discuss dynamic virtual hosting and solutions for hosting very large numbers of Web sites.

Implementing User Directories with UserDir

Rather than configuring Apache to support virtual hosts, you can give users their own home pages with the `mod_userdir` module. This is a standard module distributed with Apache and compiled by default.

`mod_userdir` provides one directive, `UserDir`, that provides a limited form of redirection when Apache sees a URL of the correct form. You can also view it as a specialized form of the `Alias` and `Redirect` directives.

The main syntax of `UserDir` specifies a directory path that's used to map URLs beginning with a tilde (`~`).

> **NOTE** *The tilde character is shorthand for a user's home directory on Unix servers and borrowed by Apache.*

The directory can take one of three forms:

- A relative URL that's expanded using the user's account information
- An absolute URL to which the username is appended
- An absolute URL with a placeholder that's substituted with the username

Let's consider that the syntax is a relative URL, such as this:

```
UserDir public_html
```

In case of a relative URL, when Apache sees a URL that looks like this:

```
http://www.alpha-complex.com/~roygbiv/colors.html
```

it takes the username after the tilde and tries to expand it into the user's home directory, assuming that an account for `roygbiv` exists on the server. It then appends the relative URL followed by the rest of the request. If `roygbiv's` account is based in `/home/roygbiv`, the resultant pathname becomes this:

```
/home/roygbiv/public_html/colors.html
```

The effect is very much like an `Alias` directive, and in fact, you can replicate many forms of `UserDir` with `Alias`. One drawback to this method is that the home directory of some users may expand to a directory you don't want users anywhere near. `root`, for example, has a `root` directory for a `home` directory on many Unix systems, which would make the entire server's contents visible. Fortunately, you can disable accounts for consideration by `UserDir`, but you need to think carefully before relying on user account information held outside Apache.

Alternatively, you can specify an absolute URL. In this case, Apache makes no attempt to deduce the home directory of the user—indeed, the user doesn't have to have one or even exist as a user on the server—and simply substitutes the URL for the tilde:

```
UserDir /home/www/alpha-complex/users/
```

This would take the URL for previous `roygbiv` and convert it into this:

```
/home/www/alpha-complex/users/roygbiv/colors.html
```

A significant disadvantage of this approach is that although you don't need to create home directories for your users, providing only the Web directory leaves users no safe place to keep private files away from Web clients because everything is public. A far better approach is to use a placeholder as I'll discuss next.

More flexibly, you can specify an asterisk (*) in an absolute URL given to `UserDir` to tell Apache to substitute the username in the middle of the URL rather than append it. The following is a safer version of the relative URL example with the same effect for valid users:

```
UserDir /home/*/public_html
```

This substitutes `roygbiv` for *, so the resultant pathname is this:

```
/home/roygbiv/public_html/colors.html
```

This is the same as the first example. However, because this doesn't look up the user account details to determine the home directory, you ensure that only directories under `/home` will be accessed. You also avoid the need to actually have a user account on the system for a given user directory.

Enabling and Disabling Specific Users

As well as the directory syntax for specifying the true URL of a user directory, `UserDir` allows the usernames that can match this URL to be specified explicitly with the `enable` and `disable` keywords. There are two approaches to take with the access policy of user directories.

You can explicitly disable users you don't want mapped to directories; this approach looks something like this:

```
UserDir disable root admin webmaster fred jim Sheila
```

This allows any URL beginning with `/~` to be mapped, as long as the name following it isn't one of those listed, `~fred`, for instance.

I've taken care to disable root here too because it's bad for security reasons to allow the root directory (which is sometimes the home for root) to be accessed through the Web server; the prospect of arbitrary clients being able to see most aspects of the server's configuration isn't a pleasant one. In fact, it's not wise for the root to have a Web page at all. For this reason, Unix servers that have user directories expanded from the user's account should, at the bare minimum, have this:

```
UserDir disable root
```

Alternatively, you can disable all users and then enable only the users you want. This approach is more explicit:

```
UserDir disable
UserDir enable fred jim Sheila
```

This is more secure but requires editing the directive each time a new user is added. For a more flexible approach, you can instead use `mod_rewrite`, as detailed later in the chapter.

Redirecting Users to Other Servers

Another aspect of using `UserDir` with an absolute URL is that you can legally include a different server in the URL:

```
UserDir http://users.beta-complex.com/
```

This causes the server to issue a redirection (status `302`) to the client and works very much like a `Redirect` directive. The redirected URL includes the originally requested URL path minus the tilde. For example, for the following request:

```
http://www.alpha-complex.com/~roygbiv/colors.html
```

the client gets back the URL as this:

```
http://users.beta-complex.com/roygbiv/colors.html
```

You can instead use the * to define the destination URL more flexibly:

```
UserDir http://users.beta-complex.com/users/*/public
```

This results in a redirection to this:

```
http://www.beta-complex.com/users/roygbiv/public/colors.html
```

To access the user directory configuration of Apache running on the second server, use this (note that a URL is used, not a pathname):

```
UserDir http://users.beta-complex.com/~*/
```

This results in the client requesting the same URL that it started with but to a different server:

```
http://www.beta-complex.com/~roygbiv/colors.html
```

Alternative Ways to Implement User Directories

Much of what `mod_userdir` does can also be done by other modules. For example, the `AliasMatch` directive with the help of `mod_alias` can be used to much the same effect as `UserDir`. For example, the following:

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

is equivalent to this:

```
AliasMatch ^/~(.*)$ /home/www/users/$1
```

Likewise, you can do external redirection with either `UserDir` or `RedirectMatch` (of course, with `mod_alias` support). Thus, the following:

```
UserDir http://www.beta-complex/users/
```

is equivalent to this:

```
RedirectMatch ^/~(.*)$ http://www.beta-complex/users/$1
```

Both regular expressions match URLs starting with a slash followed by a tilde and append the remainder of the URL following the tilde to the end of the user directory location. The caret (^) ensures that only URLs starting with slash and tilde are matched, and the dollar ($) ensures that the rest of the URL is matched by the parentheses. In fact, because a regular expression such as `.*` will try to match as much as possible, again, it's good practice to use the dollar though not essential.

Finally, mod_rewrite allows much greater flexibility than any of the previous and allows an equivalent of the disable and enable keyword:

```
RewriteMap users txt:/usr/local/apache/auth/userdirs.txt
RewriteRule ^/~([^/]+)/(.*)$ /home/www/users/%{users:$1}/$2
```

This example uses a rewrite map. In this case, the txt: prefix indicates a text file, though you could also use a DBM database if you had a lot of user directories to deal with. The format of the file is simple and just contains a line for each user you want to map:

```
Stan stan     # a one-to-one map
Zev  zev      # old name, replaced by...
Xev  zev      # an alias for the above
Kai  dead/guy # you can put a subdirectory in here too
```

The regular expression in the rewrite rule extracts the username from any request that matches the initial slash-tilde condition and feeds it, in $1, to the rewrite map to find the corresponding directory. The result doesn't need to obviously correspond to the name if you don't want it to do so. You can even, as in this example, have two different usernames mapped to the same user directory. The remainder of the URL after the username, in $2, is appended to the newly constructed pathname. If the user doesn't have an entry in the file, the rewrite rule fails; this is how to enable or disable different users.

Separate Servers

Although Apache's support for virtual hosts is extensive and flexible, there are reasons to run separate invocations of Apache for different Web sites.

A Web site used for secure transactions should be protected from configuration loopholes introduced unwittingly into a main Web site. Separating the secure parts of the site and running them with a streamlined Apache, with a minimal set of modules and a simplified configuration, improves both the reliability and security of the server; it simply has fewer things that can go wrong with it. On platforms that support the concept, it can also run under a different user and group ID, isolating sensitive information and minimizing the possibility of the main server inadvertently revealing any part of it.

Because secure transactions are often involved in online ordering systems, it's desirable to have a preference for visitors who might spend money or are otherwise more important to satisfy. By running separate servers, you can maintain a more responsive secure server that should have spare capacity, even when the main server reaches its maximum number of simultaneous requests. You can also specify different values to Apache's RLimit directives (see Chapter 6) for each server to further tune their relative performance, so the main server never fully ties up the resource of the machine.

Both the Apache server and third-party modules such as mod_perl are released with improved features on a regular basis, both as official stable versions and as

intermediate development versions. A prudent Web server administrator doesn't simply replace a running server with a new release without testing it first; this is a third reason to run separate servers.

By configuring a new installation of Apache to work on a different port number, but utilizing the same configuration, you can check that your running system will happily transfer to the new release without disturbing the active server. Both instances may serve the same content from the same document root, or you can choose to clone the site and differentiate the servers slightly, using `Include` directives or `<IfDefine>` sections to share the majority of the servers' configurations.

Restricting Apache's Field of View

To run two or more separate invocations of Apache, you need to ensure that the servers don't attempt to listen to the same address and port because this isn't allowed, and whichever server was started second would be unable to initialize and would simply shut down again. For example, you want to run a normal public access server on the IP address 204.148.170.3 and a secure server with restricted functionality and authentication on 204.148.170.4. To stop the servers picking up requests intended for each other, you could give the main this directive:

```
# main server IP
Listen 204.148.170.3:80
```

And give the secure server this directive:

```
# secure server IP
Listen 204.148.170.4:80
```

This works for both Apache 1.3 and Apache 2. Administrators interested in migrating should note that `Port` and `BindAddress` are now deprecated in Apache 1.3 and are no longer available in Apache 2.

Alternatively, you can put both servers on the same IP address, as long as you have given them different port numbers. The secure server probably uses SSL for secure encrypted connections, so you constrain it to port 443, the SSL (or `HTTPS`) port:

```
Listen 204.148.170.3:443
```

You can support both these scenarios by having the secure server respond to requests on port 80 and 443 on its own IP, 204.148.170.4, and also port 443 on the main server's IP, 204.148.170.3. This works because the main server is only listening to port 80. The directives to achieve this are as follows:

```
Listen 204.148.170.4:80
Listen 204.148.170.4:443
Listen 204.148.170.3:443
```

Specifying Different Configurations and Server Roots

By default, Apache reads its configuration from a standard location, derived from the server root and configuration filename built into Apache when it was compiled. The default location is `/usr/local/apache/conf/httpd.conf` on Unix systems. Using the `--prefix` while compiling will change this location, as discussed in Chapter 2.

To run more than one server, you clearly need to put the configuration for each one of them in a separate location to differentiate the servers. Apache provides for this with the `-f` command-line option, which overrides the name of Apache's configuration file at startup:

```
# httpd -f conf/main-server.conf
# httpd -f conf/secure-server.conf
```

If it so happens that you only need one or two additional directives, you can also use the `-C` or `-c` options to specify them directly on the command line. Because this doesn't involve an alternate configuration file, Apache will read the default `httpd.conf` file as usual. `-C` defines directives in advance, whereas `-c` does so after reading the default configuration; in this case, it doesn't matter which you use:

```
# httpd -C "Listen 204.148.170.3:80"
```

This becomes impractical for configurations of any complexity, however.

You can use the `-d` option to specify a different server root for each invocation, thus providing each with its own complete configuration directory and related files. This is good to test a completely new Apache installation before replacing the main one. For example:

```
$ httpd -d /usr/local/testapache
```

This is most appropriate if your servers differ only in which modules they include, or you choose to build a new Apache with the same default paths as the existing one (because it's then easy to replace). Otherwise, if you're looking at two different Apache executables, you can build your two servers with different default server roots to start with, using the `--prefix` option to configure, and run each one from its installed location. If you want to share some parts of the installation you can customize the layouts more precisely, as covered in detail in Chapter 3.

Starting Separate Servers from the Same Configuration

Maintaining separate configuration files can be extra work, especially if the contents of each file are substantially the same. Apache allows you to use the same configuration file by adding conditional sections with the `<IfDefine>` (`Interface Definition`) container directive, which you saw in Chapter 4.

For example, you could combine the IP configuration for the two previous servers into one file with this:

```
<IfDefine main>
  Listen 204.148.170.3:80
</IfDefine>
<IfDefine secure>
  Listen 204.148.170.4:80
  Listen 204.148.170.4:443
  Listen 204.148.170.3:443
  SSLEngine on
</IfDefine>
```

The ability to create conditional sections gives you a lot of other possibilities. In this example, I've taken advantage of it to enable SSL for the secure server only.

You can now switch on the conditional sections with the `-D` command line option:

```
$ httpd -D main
$ httpd -D secure
```

There's no limit to the number of symbols you can define with the `-D` option, and so there's no limit to the number of different conditional sections you can specify, any of which can be enabled on a per-server basis. For example, you could extract the `SSLEngine on` directive into its own conditional section:

```
<IfDefine SSL>
  SSLEngine on
</IfDefine>
```

Then you can start any server with SSL by just adding the SSL symbol to the list on startup:

```
$ httpd -D secure -D SSL
```

Sharing External Configuration Files

You can also use Apache's `Include` directive to your advantage. By placing directives in common configuration files, you can create several `httpd.conf` files and have them share details of their configuration. The contents of the `conf` directory in such a scheme might look like this:

```
httpd.main1.conf
httpd.main2.conf
httpd.secure1.conf
httpd.secure2.conf
common.conf
common.secure.conf
```

Each of the `httpd` configuration files would then be started using a command similar to this:

```
$ httpd -f conf/httpd.main2.conf
```

You could then use `Include` in each master `httpd.conf` file as appropriate. For example, for your first main server you might have this:

```
# main server configuration httpd.main1.conf

# main server IP
Listen 204.148.170.3:80

# now include common configuration
Include conf/common.conf
```

Here, `common.conf` is essentially just `httpd.conf` with any `Listen`, `Port`, or `BindAddress` directives removed. In Apache 2, where `Port` and `BindAddress` are in any case no longer permitted, attempting to start from the common configuration alone will fail because Apache 2 requires that at least one `Listen` be specified.

In this example, the secure servers would include both the `common.conf` file and `common.secure.conf` files, the latter containing common configuration for secure servers only:

```
Include conf/common.conf
Include conf/common.secure.conf
```

You can also use the `-f` option multiple times to read several different configuration files, but the `Include` solution is probably more elegant.

IP-Based Virtual Hosting

Virtual hosting involves a single Apache instance serving several different Web sites from a single configuration. You have a choice of two similar approaches:

- IP-based hosting
- Name-based hosting

> **NOTE** *The advantage of IP-based virtual hosting is that each host has its own unique IP address and therefore doesn't need the client to identify it by name. This will work with older browsers and clients that don't comply with HTTP/1.1 (specifically, don't send a* `Host:` *header).*

Multiple IPs, Separate Networks, and Virtual Interfaces

For IP-based virtual hosting to work, each hostname that Apache serves must be associated with a unique IP address or port number. Because domain names don't carry port number information, this means you need a separate IP address to have different domain names for each host.

There are two ways in which you can have separate IP addresses:

- Install multiple network cards, and assign a different IP address to each one.
- Assign multiple IP addresses to the same interface. This method is also known as *multihoming* or *IP aliasing*.

Separate Network Cards

Using separate network cards is a practical solution for a small number of Web sites in some circumstances. One instance would be a host serving pages to both external clients and an intranet, and you might already have dual network interfaces on the server for security reasons. In this case, you can assign addresses to network interfaces, like this:

```
204.148.170.3  eth0   external site - www.alpha-complex.com
192.168.1.1    eth1   internal site - internal.alpha-complex.com
127.0.0.1      lo0    localhost address
```

However, for hosting many sites, this is clearly not a practical solution; you need a platform that can support virtual network interfaces.

Virtual Interfaces

Most modern platforms support the ability to have multiple IP addresses assigned to the same network interface out of the box. Others, including older versions of Solaris, have patches available that can add this facility to the standard installation. This allows you to assign addresses to interfaces like so:

```
204.148.170.3    eth0:1     www.alpha-complex.com
204.148.170.4    eth0:2     www.beta-complex.com
204.148.170.5    eth0:3     www.troubleshooter.com
204.148.170.6    eth0:4     users.alpha-complex.com
204.148.170.7    eth0:5     secure.alplacomplex.com
```

Note that to actually assign the name to the IP address, you need to create entries in the DNS servers for your network, or else you won't be able to access the host without an IP address. For `localhost`, you can do the same with entries in the host's file.

> **TIP** *The actual process for configuring virtual network interfaces varies from platform to platform, so consulting the operating system documentation is a necessary step before attempting this. Linux administrators should also check out the Virtual Services HOWTO, which explains how to set up virtual IP addresses for many system services, including Apache.*

On most Unix servers, you can use the `ifconfig` command to assign multiple addresses like this:

```
/sbin/ifconfig lo:1 192.168.1.162 netmask 255.255.255.128
/sbin/ifconfig lo:2 192.168.1.163 netmask 255.255.255.128
/sbin/ifconfig lo:3 192.168.1.164 netmask 255.255.255.128
/sbin/ifconfig lo:4 192.168.1.165 netmask 255.255.255.128
/sbin/ifconfig lo:5 192.168.1.166 netmask 255.255.255.128
```

This creates five virtual interfaces on the local loopback interface `lo` (the name of the interface varies). This is ideal for testing IP-based virtual hosts locally because the loopback interface can't by definition be accessed through an external network. When you finish testing, you can expose these IP addresses simply by aliasing them to a different interface, such as `eth0`.

The `netmask` parameter probably isn't necessary for this example, but it would allow you to use IP addresses 192.168.1.1 to 192.168.1.126 for a different interface. Note that the primary address of the interface, traditionally 127.0.0.1 for the loopback interface, doesn't need to be in the same network or share the same `netmask`.

> **NOTE** *If the server platform doesn't support the multiplexing of IP addresses on one interface, then IP-based virtual hosting isn't possible. In such a situation, name-based virtual hosting comes to the rescue.*

Configuring What Apache Listens To

To service multiple IP addresses (or ports), you have to tell Apache which addresses to listen to in the main server configuration. Apache won't pass on connections to virtual hosts, even if they're configured.

Apache 1.3, by default, listens to any address available on the server and looks for connections on port 80. This is equivalent to the following `BindAddress` and `Port` directives:

```
# Apache 1.3 only - listen to all interfaces on port 80
BindAddress *
Port 80
```

However, Apache 2 bans both `BindAddress` and `Port` from the configuration. Apache 2 requires that at least one `Listen` directive is present in the configuration. In Apache 2, the equivalent of the 1.3 default defined with `Listen` would be this:

```
# Apache 1.3 and Apache 2 - listen to all interfaces on port 80
Listen 80
```

`Listen` takes a port number or, an IP address and port number together, to define both an address and port at the same time. For example:

```
Listen 204.148.170.3:80
```

Multiple `Listen` directives merge rather than override each other; to specify more than one address and port, just add more `Listen` directives:

```
Listen 204.148.170.3:80
Listen 204.148.170.4:80
Listen 204.148.170.5:80
Listen 204.148.170.6:80
Listen 204.148.170.7:443
```

Because `Listen` directives merge, you can cause Apache to listen to all addresses on several different port numbers. Of course, this is impossible with the now deprecated `Port` directive of Apache 1.3:

```
# http
Listen 80
# https
Listen 443
# proxies
Listen 8080
```

`Listen` only takes an IP address as part of its parameter; it doesn't accept a `hostname`, unlike the older `BindAddress`. This is a deliberate design decision to counter the potential security issues and cost involved with the DNS lookups that was incurred by specifying `hostnames`.

From Apache 2, you can also specify IPv6 addresses to `Listen`. Because IPv6 uses colons to separate the different parts of the address, you need to add some square brackets to differentiate the port. For example, to assign the IPv6 loopback address, you can use either of the following:

```
Listen [0:0:0:0:0:0:0:1]:80

Listen [::1]:80
```

These directives are identical, but the second uses an abbreviated IPv6 address for simplicity. Note that if you specify only a port number to `Listen`, it'll listen to both IPv4 and IPv6 addresses if both are present.

Defining IP-Based Virtual Hosts

Virtual hosts can be defined with the `<VirtualHost>` container directive. The full syntax of the `<VirtualHost>` container is as follows:

```
<VirtualHost IP:port IP:port...>
  # virtual host directives
  ...
</VirtualHost>
```

In most cases, you only want to serve one IP address and use the port number specified by the main server (with a `Port` or `Listen` directive). In this case, use this:

```
<VirtualHost IP>
  # virtual host directives
  ...
</VirtualHost>
```

Technically, the `VirtualHost` directive accepts hostnames and IP addresses. However, as mentioned in the previous section, using `hostnames` is discouraged. This is because it requires Apache to perform DNS lookups, putting an extra burden on the server and making Apache vulnerable to DNS spoofing attacks (see Chapter 10 for more details).

The primary object of the `<VirtualHost>` container is to include a set of alternative directives to distinguish it from the main host; Table 7-1 describes these directives.

Table 7-1. `VirtualHost` *Directives*

Directive	Description
`ServerName`	The canonical name of this host
`ServerAlias`	One or more aliases for this host, if desired
`ServerAdmin`	The name of the Web administrator for this host
`DocumentRoot`	Where this host's contents (such as HTML files) are
`ErrorLog`	The error log for this host
`CustomLog`	The access log for this host

For example, a typical `VirtualHost` directive looks something like the following:

```
<VirtualHost 204.148.170.3>
  ServerName www.alpha-complex.com
  ServerAlias alpha-complex.com *.alpha-complex.com
  ServerAdmin webmaster@alpha-complex.com
  DocumentRoot /home/www/alpha-complex
  ErrorLog logs/alpha-complex_errors
  TransferLog logs/alpha-complex_log
</VirtualHost>
```

If any of these directives, apart from `ServerName`, aren't specified for a given virtual host, they're inherited from the main server-level configuration. It's common for `ServerAdmin`, `ErrorLog`, and `TransferLog` to be inherited from the main server, causing the main server's logs to be used for all logging and the main `ServerAdmin` address to be used in error messages.

In IP-based virtual hosts, the presence of `ServerName` has nothing to do with the name the virtual host responds to, which is defined by the IP address or `hostname` in the `VirtualHost` directive itself. Rather, `ServerName` defines the name of the host used in self-referential external URLs; without it, Apache is forced to do a DNS lookup of the virtual host's IP address to discover the name.

As it stands, this host will receive requests for this IP address on all ports the server has been told to listen to. The previous `Listen` directive example specifies port `80` for this address. In this case, only requests on port `80` would be directed to this virtual host. If you were listening to more than one port, you could ensure that this virtual host only responded to port `80` connections by changing the `VirtualHost` directive to read as follows:

```
<VirtualHost 204.148.170.3:80>
  # virtual host directives
  ...
</VirtualHost>
```

Alternatively, you can have the same virtual host directives apply to more than one IP address and port by specifying them in `VirtualHost`:

```
<VirtualHost 204.148.170.3:80 204.148.170.7:443>
  # virtual host directives
  ...
</VirtualHost>
```

You can also tell a VirtualHost to respond to all ports that Apache is listening to by setting a wildcard for the port number:

```
<VirtualHost 204.148.170.3:*>
  # virtual host directives
  ...
</VirtualHost>
```

This is different from not specifying a port number at all. In Apache 1.3, without a port number, the virtual host responds to the default port defined by Port (or 80, if no Port directive has been issued). In Apache 2, the virtual host responds to the port of the previous matching Listen.

When a virtual host directive matches more than one IP address or more than one port, Apache needs to know which server name and port to use to construct self-referential URLs so that redirections cause clients to come back to the right place. In Apache 1.3, you do this with ServerName and Port inside the <VirtualHost> container:

```
<VirtualHost 204.148.170.3:80 204.148.170.7:443>
  ServerName secure.alpha-complex.com
  Port 443
  # virtual host directives
  ...
</VirtualHost>
```

In Apache 2, the Port directive has been removed and the port number is, instead, given as part of the server name:

```
<VirtualHost 204.148.170.3:80 204.148.170.7:443>
  ServerName secure.alpha-complex.com:443
  # virtual host directives
  ...
</VirtualHost>
```

If a server name or port isn't defined for the virtual host, the settings of the main server are inherited. Apache 1.3 administrators should note that the Port directive has no other effect in virtual hosts. The Port directive sets the default port in the server-level configuration.

However, whether Apache actually uses the server name and port defined or takes its cue from the request made by the client depends on the setting of the UseCanonicalName directive:

```
# Use server name and port defined by configuration
UseCanonicalName on

# Use server name and port requested by client
UseCanonicalName off
```

You can specify UseCanonicalName in both the main configuration and inside virtual hosts. In Apache 2, you can additionally specify it in <Directory> containers, which may be useful for mass virtual hosting with mod_vhost_alias or mod_rewrite; both modules are covered with respect to virtual hosting a little later in the chapter.

You can also specify an argument of DNS to force Apache to carry out a reverse DNS lookup. However, this is a costly operation and is only recommended in situations where it's unavoidable or security is of great concern. I'll discuss network security issues more in Chapter 10.

Virtual Hosts and the Server-Level Configuration

A <VirtualHost> container can enclose almost any directive acceptable to the main server configuration, including other containers such as <Directory> and <Location>. In addition, virtual hosts inherit all the directives defined in the server location, including Directory and Location directives, Alias and ScriptAlias directives, and so on. This allows a set of defaults to be specified for all virtual hosts that individual hosts can then override or supplement according to their needs. Location and Directory defined at the server level differ significantly in how they operate in conjunction with virtual hosts.

<Directory> affects any URL that resolves to a file within its scope, irrespective of the virtual host that initially handled the request. Depending on how they're configured, some virtual hosts might access the directory through different aliased URLs, and others might not access it at all. You can also use a wildcard directory to specify behavior on a per-virtual host basis, where it applies only to hosts that actually have the directory, as in this example:

```
#define a Directory container that defines CGI directories for all virtual hosts
<Directory /home/www/*/cgi-bin/>
  Options +ExecCGI
  AddHandler cgi-script .cgi .pl
</Directory>
```

<Location> specifies a relative URL, so it applies equally to all virtual hosts:

```
<Location />
  AddHandler server-parsed .shtml
</Location>
```

In both cases, you can override or refine the default behavior of a server-level container within a given virtual host by defining a similar container inside the virtual-host definition. The following example removes the previous server-level handler for a particular virtual host:

```
<VirtualHost 204.148.170.3:80>
  # virtual host directives
  ...
```

```
  <Location />
    RemoveHandler .shtml
  <Location>
</VirtualHost>
```

Many of Apache's modules provide a series of directives to configure their behavior and one directive to enable or disable that behavior; `mod_ssl` and `mod_rewrite` are two of them. It's common, for example, to define an SSL configuration for Apache at the server level and then selectively switch it `on` or `off` for individual hosts:

```
# SSL configuration directives
...
SSLEngine off

<VirtualHost 204.148.170.7:443>
  # virtual host directives
  ...
  SSLEngine on
</VirtualHost>
```

Other modules that can benefit from the same approach include `mod_rewrite`, `mod_usertrack`, `mod_session`, and `mod_vhost_alias`.

Specifying Virtual Host User Privileges

One major advantage of virtual hosts over user home pages is the ability, on Unix servers, to use Apache's `suExec` wrapper for determining the user privileges of CGI scripts. This can be useful for servers that have multiple virtual hosts all edited and maintained by different users. In Apache 2, you can instead use the `perchild` MPM, a more advanced solution to the same problem.

The suExec Wrapper

As you saw in Chapter 6, if the `suExec` wrapper is enabled (which you can check for by running `httpd -l` and examining the `suExec` status on the last line), you can set the user and group under which each virtual host runs by adding either a `User` and `Group` directive in Apache 1.3 or a `SuExecUserGroup` directive in Apache 2:

```
<VirtualHost 204.148.170.5>
  # virtual host directives
  ...
  # Apache 1.3
  #User roygbiv
  #Group troubleshooters
  # Apache 2
  SuExecUserGroup roygbiv troubleshooters
</VirtualHost>
```

If you don't specify suExec's directives (in either previous case), virtual hosts inherit their user and group from the main server. You may not want this, so administrators using suExec should ensure that every virtual host is correctly set up.

The perchild MPM

The suExec security wrapper only affects CGI scripts and external filters, not Apache itself. Significantly, scripting modules such as mod_perl run their code within Apache itself and can't use suExec because this sits between Apache and another external process, and there's no external process there.

In Apache 1.3, this is simply a problem you either have to live with or solve by running entirely separate invocations of Apache each under a different user and group. In Apache 2, however, Unix servers can take advantage of a unique feature, the perchild MPM. It allows you to assign user and group IDs directly to Apache processes and by extension any external processes such as CGI scripts that they run. This makes the suExec wrapper entirely redundant.

I briefly covered perchild in Chapter 3, along with the other MPMs available for Apache. Apache chooses its MPM at build time, and you can't change it subsequently, so if you want perchild and aren't currently using it, you'll have to rebuild the server. Because perchild isn't the default MPM for Unix, you must ask for it explicitly by supplying the --with-mpm=perchild option to configure and then rebuilding Apache. If you have Apache correctly built with perchild, then you should see the following when you ask Apache how it was built by otherwise executing httpd -V:

```
-D APACHE_MPM_DIR="server/mpm/perchild"
```

Most of the configuration issues surrounding any MPM are performance related; I'll discuss how to configure perchild in detail when I cover MPMs in detail in the next chapter. For now, I'll just present the unique features of perchild that relate to virtual hosting.

Like the worker MPM, perchild implements a hybrid process model, which is to say it uses both multiple processes and multiple threads within each process. Unlike worker, however, the number of processes is fixed, and perchild adjusts the number of threads within each process according to the demand for individual virtual hosts. This is the opposite of worker, which adjusts the number of processes according to demand but maintains a fixed number of threads per process. perchild requires the number of processes to be fixed because it differs from all other MPMs in one significant way: It allows each child process spawned by the server to run with its own unique user and group.

Furthermore, and crucial to the subject of this chapter, you can assign each of these children to a particular virtual host to have all requests directed to that host processed under a specific user and group. Each virtual host is permanently tied to its assigned child, so any and all activities it carries out run under that identity.

Running processes under assigned user and group IDs has significant security benefits, as I discussed when I covered the suExec CGI security wrapper in Chapter 6. The difference between perchild and suExec is that the latter only affects processes

started by Apache; the server itself, and any handlers or filters you install into it, runs under the original user and group defined for the main server. With `perchild`, every request received for a particular virtual host is processed under the user and group of the child process assigned to it, including embedded scripts such as `mod_perl` handlers. This was previously impossible to achieve in Apache 1.3 and is a major step forward in isolating virtual hosts from each other in a large virtual-hosting configuration.

To configure Apache to use the `perchild` MPM, you first need to define a number of server processes at least equal to the number of virtual hosts that you want to run under their own user and group. For example, for five server processes, you use this:

```
NumServers 5
```

This server-level directive is unique to the `perchild` MPM because it's the only MPM that uses a fixed number of processes with a variable number of threads per process. You can define more servers, if you like. Any extra server processes retain the server-level user and group identity, as defined by the `User` and `Group` directives, and will serve requests for the main server, should any arise.

Next, allocate one or more server processes to each user and group pair you want to use, which you do with the `ChildPerUserID` directive. Assuming you have five processes configured as in the previous example, you could write:

```
ChildPerUserID alphabadm alphagroup 1
ChildPerUserID betavadm betagroup 1
ChildPerUserID primeuadm alphagroup 2
```

In this example, three pairs of user and group ID are defined. The first two have one process allocated each, with the third allocated two. Because originally five processes were specified, there's one left over, which will be allocated to the main server and run under the server's `User` and `Group` settings. There's no reason why different child processes have to run under different IDs, and in this case two processes share the same group, `alphagroup`. If you simply omit one of the directives, then that child process runs with the `User` and `Group` of the server-level configuration.

If you want, you can also specify the user and group IDs as numeric values:

```
ChildPerUserID 1002 999 2
```

If `1002` is the user ID for `primeuadm` and `999` is the group ID for `alphagroup`, then this is equivalent to the last directive previously. You can similarly use numeric IDs with `AssignUserID`, in addition to using user and groups names as shown next.

All that now remains is to tell Apache which virtual host corresponds to each child process. To do that, use the `AssignUserID` directive, which takes a user and group name as arguments. Other than the addition of this directive, the virtual host definition is the same as before:

```
Listen 443
Listen 80
# server level directives
```

```
...

<VirtualHost 204.148.170.3:*>
  ServerName www.alpha-complex.com
  AssignUserID alphabadm alphagroup

  # virtual host directives
  ...
</VirtualHost>

<VirtualHost 204.148.170.4:80>
  ServerName www.beta-complex.com:80
  AssignUserID betavadm betagroup

  # virtual host directives
  ...
</VirtualHost>

<VirtualHost 204.148.170.5:443>
  ServerName secure.troubleshooter.com:443
  AssignUserID primeuadm alphagroup
  SSLEngine on

  # virtual host directives
  ...
</VirtualHost>
```

If you don't assign a user and group, or you attempt to assign a user and group for which no child process exists, then the assignment simply doesn't work and that virtual host continues to run with the main server's user and group. The assignment of child processes doesn't affect the rest of the virtual host configuration; you can create default hosts, mix IP-based and name-based hosts, and so forth. The only difference is that each virtual host runs with its own user and group and, therefore, so do any external processes that it starts.

`perchild` is a uniquely Unix-based solution to the problem of user permissions; it requires a platform that supports multiple forked processes and threads, and it also understands ownership permissions. Only Unix platforms provide all three of these ingredients.

As yet, `perchild` doesn't work with other virtual hosting options such as `mod_vhost_alias`, covered later in the chapter. It has scaling limitations because it requires many processes, and although you can assign the same child process to more than one virtual host, there eventually comes a limit beyond which `perchild` can't help you. It's ideal for hosting 20 virtual hosts, but rather less ideal for 2,000. Ironically, `suExec` may be more appropriate in this case because it's unlikely that every one of hundreds of virtual hosts will try to run an external CGI script at the same time. For midrange scenarios, however, `perchild` is an appealing option.

Excluded Directives

Several directives don't make sense in virtual host configurations and are either ignored or explicitly forbidden by Apache in virtual host containers. In fact, many of them have been removed in Apache 2 (see Table 7-2).

Table 7-2. Excluded Directives

Directive	Description
`ServerType`	The server type, `standalone` or `inetd`, is a global directive and affects the operation of all Apache subprocesses and virtual hosts. Note that this directive no longer exists in Apache 2.
`StartServers, MaxSpareServers, MinSpareServers, MaxRequests, PerChild`	All these directives control the management of Apache's subprocesses for handling individual HTTP requests. Virtual hosts share the pool of servers between them and have no direct relation to Apache processes. Any process can handle a request for any virtual host.
`BindAddress, Listen`	For a virtual host to receive a connection request, the main server configuration needs to be told to listen to the relevant IP address and port. Specifying these directives in a `<VirtualHost>` container would have no useful effect and is in fact illegal. `BindAddress` no longer exists in Apache 2. Note that Apache 1.3 allows the `Port` directive in virtual hosts. In this context, it specifies the canonical port number as set in the `SERVER_PORT` environment variable and is used in self-referential external URLs. In Apache 2, the `ServerName` directive has been extended to perform the same duty, and `Port` is no longer valid.
`ServerRoot`	The server root defines the location of configuration information and other resources (such as icons or the default `cgi-bin`) common to all virtual hosts. A virtual host can move some of the files normally found under the server root with appropriate directives such as `ErrorLog` for the error log.
`PidFile`	The process ID of the main server can only be set by the main server; virtual hosts in any case have no direct relationship with individual server processes.
`TypesConfig`	The name of the file for media type (MIME type) definitions can only be set on a global basis. However, virtual hosts inherit this information and can override or supplement it with `AddType`.
`NameVirtualHost`	`NameVirtualHost` defines an IP address on which name based virtual hosting is to be performed and makes no sense in a virtual host context. The directives for setting the name of a virtual host are `ServerName` and `ServerAlias`.

Default Virtual Hosts

Normally, the main Apache server deals with any valid IP address and port that aren't governed by a virtual host configuration. However, the special symbol `_default_` allows you to catch requests with a virtual-host container instead:

```
<VirtualHost _default_>
  # virtual host directives
  ...
</VirtualHost>
```

Without further qualification, this will match all IP addresses to which Apache is listening, using either the `Port` directive of the main server configuration (in Apache 1.3 only) or the last `Listen` directive given before the `<VirtualHost>` container definition. Assuming that the server configuration contained:

```
Listen 80
Listen 443
```

the previous would be equivalent to this:

```
<VirtualHost _default_:443>
  # virtual host directives
  ...
</VirtualHost>
```

Because it's more likely that you'd want to default to port 80, you should probably list the two `Listen` directives the other way around. Alternatively, if you wanted to catch all otherwise undefined IP addresses on all possible ports that Apache listens to (as defined by multiple `Listen` port directives), you could use the wildcard port identifier seen earlier:

```
<VirtualHost _default_:*>
  # virtual host directives
  ...
</VirtualHost>
```

A default host with a wildcard port number must appear after default hosts with specific port numbers, or else it'll override them. This doesn't affect explicit virtual hosts, which are always matched before a default host of any kind.

> **NOTE** *If you define a default virtual host such as in the previous example, then the main server can never get a request and becomes merely a convenient place for defining defaults. This is a valid approach to server configuration because (in a typical Apache configuration file) the main server directives are spread out. Overriding them with a virtual host whose definition can fit onto the screen can greatly aid in legibility.*

The power of default virtual hosts is limited. For example, you can't configure a virtual host to respond to all IP addresses on an external network. A few approaches get around this problem, which I'll discuss in more detail later in the chapter.

Name-Based Virtual Hosting

Name-based virtual hosting is a new form of virtual hosting based on requirements of HTTP/1.1. Instead of requiring that each virtual host have its own IP address, many domain names are multiplexed over a single IP address. This greatly simplifies network configuration and eliminates the need for multiple interfaces in hardware or software.

Apache can determine which Web site is being asked for by examining the `Host:` header sent by the client as part of an HTTP request. This header is obligatory for HTTP/1.1 but was optional for HTTP/1.0, so name-based virtual hosting doesn't work reliably with HTTP/1.0 clients. Fortunately, the number of non-HTTP/1.1 clients is small and decreasing, but it can still be a concern. As you'll see later, the `ServerPath` directive can provide a partial solution to this problem.

Although name-based hosting absolves the server administrator of the need to reconfigure the network settings of the server, the domain names that it must respond to still have to be entered into the DNS configuration of the network name servers; otherwise, external clients won't be able to reach the virtual hosts.

Defining Named Virtual Hosts

The principal difference between the configuration of IP-based virtual hosts and name-based ones is the `NameVirtualHost` directive. This directive marks an IP address as a target for multiplexing multiple name-based hosts, rather than a single IP-based host. For example:

```
NameVirtualHost 204.148.170.5
```

If you want to restrict the port number as well, you can do this with the following:

```
NameVirtualHost 204.148.170.5:80
```

Once an IP address has been marked for use in name-based hosting, you can define as many `<VirtualHost>` containers as you like using this IP address, differentiating them with different values in their `ServerName` directives:

```
<VirtualHost 204.148.170.5>
  ServerName users.alpha-complex.com
  # virtual host directives
  ...
</VirtualHost>

<VirtualHost 204.148.170.5>
  ServerName secure.alpha-complex.com
  # virtual host directives
  ...
</VirtualHost>

<VirtualHost 204.148.170.5>
  ServerName www.alpha-complex.com
  # virtual host directives
  ...
</VirtualHost>
```

When Apache receives a request for the IP address 204.148.170.5, it recognizes it as being for one of the name-based hosts defined for that address. It then checks the `ServerName` and `ServerAlias` directives looking for a match with the hostname supplied by the client in the `Host`: header. If it finds one, it then goes forward and processes the URL according to the configuration of that host; otherwise, it returns an error.

You can specify the `NameVirtualHost` directive multiple times; each one marks a different IP address and allows name-based virtual hosting on it. This allows virtual hosts to be partitioned into different groups. One address hosts one group of virtual hosts, and another IP address hosts a different group. There are few reasons why this could be useful:

- Bandwidth limiting by IP address
- Separating secure sites using SSL from insecure ones
- Using `allow` or `deny` to provide some hosts more privileges than others

If you know that you want to use all your available IP addresses for name-based virtual hosting, you can mark all addresses that Apache is listening to by using *:

```
# Mark all IP addresses for use with named virtual hosts
NameVirtualHost *
```

However, there are a couple of implications of using the `NameVirtualHost` directive to nominate an IP address for named hosting:

> Once an IP address has been defined for use in name-based hosting, it can no longer be used to access the main server. If Apache hasn't been configured to listen to other IP addresses, the main server is effectively out of reach. In the case of a wildcard, this is certainly and irrevocably true.

A named IP address can't ever match the main server or a default server defined with the _default_ token, even if the host specified in the request doesn't match any of the named virtual hosts in the configuration. The NameVirtualHost directive effectively captures all requests to the specified IP address whether the requested host exists. If none of the virtual hosts match, then the first-named virtual host for the IP address is chosen as a default host. Thus, to define a default host for a marked IP address, you must specify the host that should handle the request first from among all the hosts that include the IP address in their <VirtuaHost> definitions.

Server Names and Aliases

As you saw previously, the key to name-based virtual hosts is the ServerName directive that provides the actual name of the host, allowing Apache to compare it to the client request and determine which name-based virtual host is being asked for.

The ServerName directive is important. It defines the true name of the host and is the name Apache will use to construct self-referential URLs with, just as with IP-based virtual hosts. A host can have as many aliases as it wants, but it must always have a server name, even if it never matches (though this would be strange).

You can also have the virtual host respond to other hostnames by specifying an alias. You can define aliases with the ServerAlias directive, which accepts a list of domain names and in addition accepts the wildcard characters, * and ?.

For example, the following virtual host responds to www.alpha-complex.com and www.alpha-prime.com, plus any host that has the word *complex* along with a three-letter top-level domain (for example, .com and .org, but not .co.uk):

```
<VirtualHost 204.148.170.3>
  ServerName www.alpha-complex.com
  ServerAlias www.alpha-prime.com *complex*.???
  ServerAdmin ...
  DocumentRoot ...
  ErrorLog ...
  TransferLog ...
</VirtualHost>
```

Defining a Default Host for Name-Based Virtual Hosting

Because name-based virtual hosting captures all requests for a given IP address, you can't use a default virtual host to catch invalid domain names, as you could with IP-based virtual hosting. The first-named host defined for the IP address is treated as the default. However, you can do the equivalent of an explicit default host with a very relaxed ServerAlias:

```
<VirtualHost 204.148.170.3:*>
  ServerName www.alpha-complex.com:80
  ServerAlias *
```

```
  RewriteEngine On
  RewriteRule .* - [R]
</VirtualHost>
```

Specify a port number of * to catch all ports that Apache listens to, as well as a `ServerAlias` of * to match anything the client sends to you in the `Host:` header. A port number and a rewrite rule redirect clients to the main host, `www.alpha-complex.com`. The redirection uses the values of the canonical server name and port to construct the redirected URL. Because you're just redirecting any URL you get, don't bother with the `ServerAdmin`, `DocumentRoot`, or `ErrorLog` and `TransferLog` directives.

> **NOTE** *This example is for Apache 2. For Apache 1.3, a separate Port* `80` *directive should replace the* `:80` *at the end of the* `ServerName` *directive.*

Mixing IP-Based and Name-Based Hosting

There's no reason why you can't use IP-based and name-based virtual hosts in the same server configuration. The only limitation is that any IP address that has been specified with `NameVirtualHost` for name-based virtual hosts can't be used for an IP-based host.

The following example demonstrates both name-based and IP-based hosts in the same configuration. The example also contains a default host for unmatched IP addresses, a secure host simultaneously working as both, name-based and IP-based virtual host, and a default host for unrecognized `hostnames` on the name-based IP address:

```
### Set up the main server's name and port as a fallback ###
### (it should never be matched) ###
ServerName localhost:80
# for Apache 1.3 use ServerName localhost, Port 80

### Set up the ports we want to listen to - list 80 last so it becomes ###
### the default for virtual hosts that do not express a port preference ###
Listen 443
Listen 80

### Set up the main server ###

# because we have a default server these directives are inherited
ServerAdmin webmaster@alpha-complex.com
DocumentRoot /home/www/alpha-complex/
ErrorLog logs/error_log
TransferLog logs/access_log

# User and Group - always set these
User httpd
Group httpd
```

```
### IP-based virtual hosts ###

# A standard IP-based virtual host on port 80
<VirtualHost 204.148.170.3>
  ServerName www.alpha-complex.com
  ServerAdmin webmaster@alpha-complex.com
  DocumentRoot /home/www/alpha-complex/
  ErrorLog logs/alpha-complex_error
  TransferLog logs/alpha-complex_log
</VirtualHost>

# Another standard IP-based virtual host on port 80 and 443
<VirtualHost 204.148.170.4:*>
  ServerName www.beta-complex.com
  ServerAdmin webmaster@beta-complex.com
  DocumentRoot /home/www/alpha-complex/
  ErrorLog logs/alpha-complex_error
  TransferLog logs/alpha-complex_log
</VirtualHost>

### Name-based virtual hosts ###

# Nominate an IP address for name-based virtual hosting
NameVirtualHost 204.148.170.5

# A name-based virtual host on port 80
<VirtualHost 204.148.170.5>
  ServerName www.troubleshooter.com
  ServerAlias *.troubleshooter.*
  ServerAdmin webmaster@troubleshooter.com
  DocumentRoot /home/www/troubleshooter/
  ErrorLog logs/troubleshooter_error
  TransferLog logs/troubleshooter_log
</VirtualHost>

# add more virtual hosts here ...

# a name-based virtual host on port 443
<VirtualHost 204.148.170.5:443>
  ServerName secure.troubleshooter.com
  ServerAdmin webmaster@troubleshooter.com
  DocumentRoot /home/www/troubleshooter-secure/
  ErrorLog logs/secure.troubleshooter_error
  TransferLog logs/secure.troubleshooter_log
</VirtualHost>

# add more virtual hosts here ...

# this host responds to both the name-based IP and its own dedicated IP
<VirtualHost 204.148.170.5 204.148.170.7:443>
  # this name resolves to 204.148.170.7
```

```
  ServerName secure.alpha-complex.com:443
  # this alias matches hosts on the name-based IP
  ServerAlias secure.*
  ServerAdmin secure@alpha-complex.com
  DocumentRoot /home/www/alpha-complex/
  ErrorLog logs/alpha-complex_sec_error
  TransferLog logs/alpha-complex_sec_log
  # this assumes we've specified the other SSL directives elsewhere
  <Location /secure/>
    SSLEngine on
  </Location>
</VirtualHost>

# this host catches requests for users.alpha-complex.com on any port number
# on the name-based virtual host IP
<VirtualHost 204.148.170.5:*>
  ServerName users.alpha-complex.com
  ServerAdmin webmaster@alpha-complex.com


  DocumentRoot /home/www/alpha-complex/users/
  ErrorLog logs/alpha-complex_usr_error
  TransferLog logs/alpha-complex_usr_log
</VirtualHost>

# this host catches all requests not handled by another name-based virtual host
# this must come after other name-based hosts to allow them to match first
<VirtualHost 204.148.170.5:*>
  ServerName wildcard.alpha-complex.com
  # catch all hosts that don't match anywhere else
  ServerAlias *
  DocumentRoot /home/www/alpha-complex/
  ErrorLog logs/alpha-complex_error
  TransferLog logs/alpha-complex_log
</VirtualHost>

### Default IP-based virtual host ###

# this host catches all IP addresses and port numbers not
# already handled elsewhere and redirects them to www.alpha-complex.com. Given the
# configuration above, the only thing it can catch at present is a request for
# 204.148.170.3 on port 443, or an IP address other than 204.148.170.3-5
<VirtualHost _default_:*>
  ServerName www.alpha-prime.com:80

  RewriteEngine On
  RewriteRule .* - [R]
</VirtualHost>
```

It's unlikely you'd ever want to configure Apache with these many different styles of virtual hosts, but it does demonstrate how flexible virtual hosts can be.

Issues Affecting Virtual Hosting

Whichever virtual hosting scheme you pick, there are some important caveats and considerations for any Web server administrators who want to implement virtual hosts on their server. I'll now discuss some of the most important ones:

- Log files and file handles
- Virtual hosts and server security
- Secure HTTP and virtual hosts
- Handling HTTP/1.0 clients with name-based virtual hosts

Log Files and File Handles

Most operating systems impose a limit on how many files any one application can open at any given time. This can present a problem for servers hosting many virtual hosts where each host has its own access and error log because each host will consume two file handles. Also, Apache needs one file handle for each network connection, plus one or two others for its own internal use. In fact, there are usually three kinds of limits:

- A *soft limit* that's set low but can be upgraded to the hard limit by the application or on startup with the `ulimit` system command. Apache will automatically try to do this if it runs out of file handles.
- A *hard limit* that's normally unset (and so, defaults to the kernel limit) but once set won't allow the soft limit to increase past it. Only the root user is allowed to change the hard limit.
- A *kernel limit* that's the absolute maximum the operating system supports. This varies from platform to platform. For example, Linux allows 256 or 1024, depending on the kernel, but can be patched to allow more.

Depending on the platform in question, these limits can be retrieved and set with a variation of the `ulimit` command, which is usually built into Unix shells. For more details, consult the manual page for the shell.

To find out the soft limit, the following usually works:

```
$ ulimit -S -n
```

To set the limit, specify a number:

```
$ ulimit -S -n 512
```

This can be either added to the startup script for Apache so it's set at runtime or put in a wrapper script for the startup script to run. For example, you could rename `httpd` to `httpd.bin` and then create a script called `httpd` with this:

```
#!/bin/sh
ulimit -S -n 1024
/usr/local/apache/bin/httpd.bin $*
```

The `$*` at the end of the last line ensures that any command-line options you give to the script are passed along to Apache.

However, sooner or later, you'll come up against the kernel file handle limit, at which point you can no longer get away with more file handles. There are a few solutions to this problem, loosely divided into increasing the number of handles available, reducing the number of handles you use at any one time, or side stepping the problem altogether.

On Unix systems, the file handle limit can often be raised, but only to a point, and some platforms have specific peculiarities or work-arounds:

- Solaris allows 1024 but only the first 255 for functions in the standard library; because error logs use the standard library, only around 110 virtual hosts can be supported. Apache allows this to be increased to around 240 by changing a limit within Apache so that file handles are only allocated below 256 when there's no choice. This is the *high slack line* and can be fed to Apache at build time in `EXTRA_CFLAGS` with `-DHIGH_SLACK_LINE=256`.

- Linux allows 256 file handles as a kernel limit, but this can be patched to 1024 or higher with one of several freely available patches.

- Other platforms vary according to their specific implementation. Consult the Apache documentation and the manual pages for the platform in question for more information.

- Reducing the number of handles is most easily achieved by combing the error and access logs for each virtual host into two master logs, saving one file handle per virtual host for each log. To do this, you just have to omit the logging directives into your virtual host definitions. On sites where each virtual host is individually maintained this can be inconvenient, but you can modify the log format so that each virtual host is at least distinguishable from the others.

- For name-based hosts, use the `%V` format, which expands to the `hostname`:

  ```
  LogFormat "%V: %h %l %u %t \"%r\" %>s %b"
  ```

- For IP-based hosts, you use the `%A` format, which expands to the IP address:

  ```
  LogFormat "%A: %h %l %u %t \"%r\" %>s %b"
  ```

You can also use the %V format with IP-based hosts if you specify `UseCanonicalName dns`. However, this is an expensive option if logging is the only reason for it. You're better off postprocessing the logs with the `logresolve` program, a support program supplied with Apache.

- Having combined the log files, you can use the `split-logfile` program, another support program supplied with Apache, to generate separate log files subsequently. From the logs directory, invoke this:

```
$ ../support/split-logfile < access.log
```

  Log files for each virtual host will be generated based on their name (or IP address). Log information will be appended to any log file that already exists.

- You can also redirect logging to an external program by using the piped application format of the `ErrorLog`, `TransferLog`, and `CustomLog` directives. This program will be limited by the same file handle limit as Apache; but because it's independent, it won't have file handles taken up by network connections. For example:

```
TransferLog |/usr/local/apache/bin/my_logging_application
```

  Because all logging information passes through this program, you can use it to selectively log only those things you care about or split the log into multiple files based on any criteria you like. You can also use the `split-logfile` program mentioned previously as a piped application.

- Alternatively, you can use Apache's `syslog` option for logs and dump the whole problem on the system-logging daemon (a.k.a. `syslogd`), which is designed to solve such problems. You can give each virtual host a different identifier to generate unique logs for each host. See more details on logging errors in Chapter 9.

- You can also use Apache's `syslog` option to relay logging information to a system log daemon running on a different server entirely. Not only does this absolve the Web server of the need to raise its file handle limits, but also it removes the need to write to disk, which saves both time and disk space. This is most effective if the logging server is attached to the Web server via a separate network interface, so it doesn't cut into the network traffic between the server and its clients. In any case, the logging server should be on a secured intranet; allowing your logs to leak onto the Internet is poor security indeed.

Virtual Hosts and Server Security

The presence of virtual hosts can change a number of things in relation to the security of the server. First, virtual host logs are just as significant as a potential security hole and should be treated with the same care to ensure that they're not world writeable

and preferably not world readable either. For instance, `500—Internal Server Errors` can often include sensitive information such as username, passwords, and so on, which are mistakenly dumped to `STDERR` on failure; this gets put there often because the scripts fail before they can output a header.

Servers hosting virtual sites often move the logs for a given virtual host to its special directory by using directives such as this:

```
DocumentRoot /home/www/site/html
ErrorLog /home/www/site/logs/error_log
TransferLog /home/www/site/logs/access_log
```

This is a fine strategy, as long as the `logs` directory and the files inside it don't have insecure permissions. Frequently, users who log in to the server to maintain a Web site can break file protections if they own the directories in question. Keeping all the log files in a place where only the Web server administrator has control over them is therefore much safer. On Unix systems, there's no problem with keeping log files in `/usr/local/apache/logs/` and giving the virtual host log directories symbolic links to them. Site administrators will be able to access their logs from their own accounts, but they can't alter the file permissions.

First, it's important to note that you should never place log files under the document root or alias them so that a URL can reach them.

Second, it's certainly a good idea to enable a security wrapper such as `suExec` or `CgiWrap` when hosting multiple virtual hosts where the administrators of each Web site are independent of each other. Although they increase the security risk associated with an individual site, they substantially reduce the possibility of damage to other virtual hosts and the server as a whole. Chapter 6 covers both `suExec` and `CgiWrap` in more detail.

Secure HTTP and Virtual Hosts

It's frequently desirable to have the same host respond to both normal and secure HTTP connections. However, Apache won't match more than one virtual host definition to the same request, so you can't define the standard server directives such as `ServerName` and `DocumentRoot` in one virtual host and then have a second SSL host inherit those directives.

One way around this is to use the main server for one of the hosts:

```
Listen 80
Listen 443

ServerName www.alpha-complex.com
DocumentRoot /home/www/alpha-complex
ServerAdmin webmaster@alpha-complex.com
ErrorLog logs/alpha-complex_error

TransferLog logs/alpha-complex_log
```

```
<VirtualHost 204.148.170.3:443>
  # the main server's ServerName, DocumentRoot etc. are inherited
  ... SSL directives ...
  SSLEngine on
  # override the Transfer log only
  logs/alpha-complex_log
</VirtualHost>
```

Because you only have one main server, you can use this method only once. You can also use two virtual hosts but only if you specify the standard server directives twice:

```
<VirtualHost 204.148.170.3:80>
  ... virtual host directives ...
</VirtualHost>

<VirtualHost 204.148.170.3:443>
  ... virtual host directives (again) ...
  ... SSL directives ...
</VirtualHost>
```

You might be tempted to use a wildcard port number for the first `<VirtualHost>` container previously, so it matches both ports 80 and 443, and then remove the virtual host directives from the second container. However, this doesn't work because Apache will match a request for port 443 to the first virtual host and never see the second virtual host definition with the SSL directives.

> **TIP** *An alternative way of achieving a more efficient configuration is to use one of the dynamic virtual host configuration techniques covered in the next section; for a large number of virtual hosts, this is an ideal solution.*

Another approach is to avoid using SSL within Apache and instead use an SSL wrapper application such as `SSLwrap`. These work by interposing themselves between Apache and the network so the client can carry out an encrypted dialogue with the wrapper while Apache receives and transmits regular unencrypted messages.

> **NOTE** *Administrators who plan to keep SSL separate from Apache with the use of packages such as* `SSLwrap` *should be aware that this doesn't work correctly with name-based virtual hosts. The reason is that client connections encounter the SSL wrapper before they get as far as Apache. Apache therefore has no control over whether SSL should be enabled or which certificate should be used.*

Handling HTTP/1.0 Clients with Name-Based Virtual Hosts

Name-based virtual hosting relies on the client sending a `Host:` header with each HTTP request to identify which virtual host it wants. Accordingly, HTTP/1.1 requires a `Host:` header as part of every client request. However, clients that predate HTTP/1.1 don't always send a `Host:` header, so Apache is unable to resolve the virtual host.

In these circumstances, Apache defaults to using the first name-based virtual host in the configuration file for the IP address used. This might not be quite what you want, so Apache provides a partial solution with the `ServerPath` directive. This is only allowed in `<VirtualHost>` containers and defines a URL prefix that, when matched by a client request, causes Apache to serve the URL not from the first virtual host listed but from the virtual host whose `ServerPath` matched.

This doesn't fix the problem, but it gives older clients another way to get to the page they really want by adjusting their URL to include the `ServerPath` for the virtual host they really want. The adjusted URL then has this form:

```
http://<name of any named virtual host>/<server path>/<original URL>
```

For example, you might define two named virtual hosts, the first of which is the default and the second of which defines a special `ServerPath`:

```
NameVirtualHost 204.148.170.5

<VirtualHost 204.148.170.5>
  ServerName www.beta-complex.com
  ... virtual host directives ...
</VirtualHost>

<VirtualHost 204.148.170.5>
  ServerName secure.beta-complex.com
  ServerPath /secure
  ... virtual host directives ...
</VirtualHost>
```

HTTP/1.1 clients can reach the server `secure.beta-complex.com` and retrieve its home page with the URL:

```
http://secure.beta-complex.com/index.html
```

HTTP/1.0 clients that tried to access this URL would instead get this:

```
http://www.beta-complex.com/index.html
```

HTTP/1.0 clients can instead retrieve the index page for `secure.beta-complex.com` with the following:

```
http://www.beta-complex.com/secure/index.html
```

or with this:

```
http://secure.beta-complex.com/secure/index.html
```

Of course, there's no way for an HTTP/1.0 client to know this unless you tell them. So to create an HTTP/1.0-compatible server, you can create a special named virtual host as your first server. This server isn't accessible to HTTP/1.1 clients because it has an unmatchable server name. It consists only of an index page of the server's real named hosts:

```
NameVirtualHost 204.148.170.5

# Virtual Host for HTTP/1.0 clients
<VirtualHost 204.148.170.5>
  ServerName this.is.never.matched.by.an.HTTP.1.1.client
  DocumentRoot /usr/local/apache/http10clients
</VirtualHost>
... the real named virtual hosts ...
```

You would then create an index page with links (relative or absolute but not fully qualified with a protocol and server name) such as the following:

```
<html>
  <head>
    <title>Welcome HTTP/1.0 clients!</title>
  </head>
  <body>
    <h1>Index of Sites hosted on this Server:</h1>
    <hr>
    <br>
    <ul>
      <a href=/www/index.html>www.beta-complex.com</a>
      <a href=/secure/index.html>secure.beta-complex.com</a>
    </ul>
  </body>
</html>
```

A smarter scheme than this could involve a CGI script that took note of the URL the client asked for, searched the virtual hosts to discover which it was valid on, and then presented a list of matches or, in the case of a single match, went straight to it.

Dynamic Virtual Hosting

Apache's support for virtual hosts is comprehensive, but it's also long-winded if you want to configure more than a few virtual hosts. Configuring a server with 300 virtual hosts by hand isn't much fun.

Ideally, you'd like to automate the configuration of virtual hosts so that Apache can either determine the virtual-host configuration when it starts up or be absolved of the need to know the actual names of virtual hosts entirely.

Fortunately, Apache provides several options:

- From Apache 1.3.9, you can use the `mod_vhost_alias` module.
- You can fake virtual hosts with `mod_rewrite`.
- You can use `mod_perl` to dynamically generate the configuration.

Of these, `mod_vhost_alias` is the simplest to use, so I'll discuss it first.

Mass Hosting with Virtual-Host Aliases

`mod_vhost_alias` is a module introduced with Apache 1.3.9, specifically designed to address the needs of hosting many virtual hosts. It isn't built by default, so Apache needs to be rebuilt or the module should be added with `LoadModule` and `AddModule`.

Instead of creating `<VirtualHost>` container directives, define one virtual document root with tokens into which the virtual host's `hostname` or IP address is inserted. You can also use a variant of `ScriptAlias` to locate `cgi-bin` directories on a per-virtual host basis.

Basic Virtual-Host Aliasing

The basic operation of `mod_vhost_alias` allows you to implement a dramatic reduction in the number of lines in the configuration file. A named virtual host configuration can be replaced by just two directives:

```
UseCanonicalName off
VirtualDocumentRoot /home/www/%0
```

Here `%0` contains the complete `hostname` supplied by the client in the `Host:` header. The `UseCanonicalName` directive is needed to ensure Apache deduces the name of the host from the client rather than generating it from the `ServerName` or `Port` directives. In this case, it'll always return the name of the main server, so self-referential URLs will fail.

Now when a client asks for a URL such as `http://www.alpha-complex.com/index.html`, Apache translates this with the `VirtualDocumentRoot` directive into `/home/www/www.alpha-complex.com/index.html`.

An IP-based virtual hosting scheme is similar, but you have to get the name of the host from DNS because the client hasn't supplied it, and you only have the IP address:

```
UseCanonicalName DNS
VirtualDocumentRoot /home/www/%0
```

This has precisely the same effect on incoming URLs as the previous named virtual-host example.

In Apache 2, if you have a combination of aliased virtual hosts and regular ones, you constrain the `UseCanonicalName` directive to a directory so that it applies only to the aliased hosts:

```
<Directory /home/www/aliased_hosts>
  UseCanonicalName DNS
</Directory>
VirtualDocumentRoot /home/www/aliased_hosts/%0
```

Because using DNS is undesirable from a performance and security standpoint, using `mod_vhost_alias` is a better option. This also allows you to define the virtual document root in terms of the IP address rather than the `hostname` with `VirtualDocumentRootIP`:

```
VirtualDocumentRootIP /home/www/%0
```

Now when a client asks for a URL, Apache uses the IP address that the request was received on and maps the URL to an IP address, like this:

```
/home/www/204.148.170.3/index.html
```

This isn't a universal solution to the problems, though. `mod_vhost_alias` doesn't allow you to specify individual error or access logs or an administrator's e-mail address.

Keeping Hosts in Subdirectories with Named Virtual Aliasing

The interpolation features available for the pathname of `VirtualDocumentRoot` and `VirtualDocumetRootIP` are a lot more powerful than just inserting the whole `hostname` or IP address with `%0`. In fact, a `hostname` can be split into parts, even down to single characters, extracted with multiple placeholders and inserted into the pathname in several different places. Table 7-3 describes the tokens to achieve this.

Table 7-3. Placeholder Tokens

Token	Description	Example	Result
%%	A % sign	%%	%
%p	The port number	%p	80
%0	The whole name or IP address	%0	www.server3.alpha-complex.com
%N	The Nth part of the name or IP address counting forward	%1 %2 %3	www server3 alpha-complex
%-N	The Nth part of the name or IP address counting backward	%-1 %-2 %-3	com alpha-complex server3
%N+	The Nth part of the name and all succeeding parts, counting forward	%1+ %2+ %3+	Same as %0 server3.alpha-complex.com alpha-complex.com
%-N+	The Nth part of the name and all preceding parts, counting backward	%-1+ %-2+ %-3+	Same as %0 www.server3.alpha-complex www.server3

You can use the tokens listed in Table 7-3 to place the document roots of virtual hosts into individual subdirectories rather than keep them all in the same base directory. For example, you have three hosts in each second-level domain, `www`, `users`, and `secure`. You can subdivide your hosts into top, second, and host-name directories with this:

```
VirtualDocumentRoot /home/www/%-1/%-2/%-3
```

This maps URLs of the following:

```
http://www.alpha-complex.com/index.html
http://secure.beta-complex.com/index.html
http://users.troubleshooters.org/index.html
into:
/home/www/com/alpha-complex/www/index.html
/home/www/com/beta-complex/secure/index.html
/home/www/org/troubleshooters/users/index.html
```

You could also provide different document roots for different port numbers:

```
VirtualDocumentRoot /home/www/%2+/%p
```

This maps URLs of the following:

```
http://www.alpha-complex.com:80/index.html
http://www.beta-complex.com:443/secure/index.html
into:
/home/www/alpha-complex.com/80/index.html
/home/www/beta-complex.com/443/secure/index.html
```

However, if your objective is to reduce the number of subdirectories, this may not always work. For example, if most of your hosts are in the .com top-level domain, you'll still end up with a lot of directories in `/home/www/com/`. To solve this problem, you can also split the `hostname` parts themselves.

The syntax for extracting name subparts is similar to the syntax for the parts themselves (see Table 7-4).

Table 7-4. Extraction Tokens

Syntax	Description	Example	Result

`%N.M`	The Mth character of the Nth part, counting forward	`%3.1`
		`%3.2`
		`%3.3`
			a
			l
			p

`%N.-M`	The Mth character of the Nth part, counting backward	`%3.-1`
		`%3.-2`
		`%3.-3`
			x
			e
			l

`%N.M+`	The Mth and succeeding characters of the Nth part, counting forward	`%3.1+`
		`%3.2+`
		`%3.3+`
			Same as `%3`
			`pha-complex`
			`ha-complex`

`%N.-M+`	The Mth and preceding characters of the Nth part, counting backward	`%3.-1+`
		`%3.-2+`
		`%3.-3+`
			Same as `%3`
			`alpha-comple`
			`alpha-compl`

In this table, `M` may be either a positive or a negative number, as illustrated previously. You can use this syntax to subdivide directories on the first letter of the second-level domain name:

```
VirtualDocumentRoot /home/www/%-1/%-2.1/%-2.2+/%-3
```

This maps the following URL:

```
http://www.alpha-complex.com/index.html
```

into this filename:

```
/home/www/com/a/lpha-complex/www/index.html
```

A side effect of the use of dots for specifying subparts is that you can't use it to mean a literal dot after a placeholder, which is ironically a likely place to want to put it. As a consequence, you can't extract and use the last and penultimate parts of the domain name with `%-2.%-1` because the dot is taken to be the start of a subpart specification. Although this might seem like an insoluble problem, the solution is actually straightforward—add a subpart that returns the whole thing:

```
VirtualDocumentRoot /home/www/%-2.1+.%-1
```

The `.1+` has no effect on the interpolated value of `%2`, but it prevents the second dot from being mistaken as part of the format.

Keeping Hosts in Subdirectories with IP Virtual Aliasing

The syntax for subdividing hostnames also works for IP addresses. The only difference is that the parts are now the four octets of the IP address rather than the different levels of the domain name. The IP address 204.148.170.3 splits into this:

```
%0                    204.148.170.3
%1 OR %-4             204
%2 OR %-3             148
%3 OR %-2             170
%4 OR %-1             3
```

You can also include succeeding or preceding numbers:

```
%2+               148.170.3
%-2+              204.148.170
```

You can split the individual numbers:

```
%0.1 OR %0.-3          2
%0.2 OR %0.-2          0
%0.3 OR %0.-1          4
```

And finally, you can include succeeding or preceding digits:

```
%0.2+                   04
%0.-2+                  20
```

To put this into practice, you can split hosts according to the last two numbers of their IP address, assuming you have several class C network addresses to manage:

```
VirtualDocumentRootIP /home/www/%4/%3
```

Because there are always exactly four octets in an IP address, this is identical to the following:

```
VirtualDocumentRootIP /home/www/%-1/%-2
```

Either way, the maximum number of subdirectories you can now have is 254 (because the numbers 0 and 255 have special meaning to TCP/IP). If you wanted to cut this down further, you could use one of the following:

```
# subdivide hosts into sub 100, 100-199 and 200+ directories
VirtualDocumentRootIP /home/www/%4/%3.-3/%3.2+

# subdivide hosts by last digit of last octet
VirtualDocumentRootIP /home/www/%4/%3.-2+/%3.-1

# subdivide hosts by first octet and all three digits of last octet
VirtualDocumentRootIP /home/www/%4/%3.-3/%3.-2/%3.-1
```

All these examples count the elements of the last octet backward for a good reason; mod_vhost_alias returns an underscore (_) for values that are out of range. The number 3 resolves to the directory 3/_/_ when aliased with %N.1/%N.2/%N.3. To get the more correct behavior you want, you count from the end with %N.-3/%N.-2/%N.-1, which produces _/_/3.

Virtual Script Aliasing

The mod_vhost_alias also supplies two directives for specifying CGI script directories that use an interpolated directory path. For example, you can keep a virtual host's cgi-bin directory next to its document root by putting both into a subdirectory, like so:

```
VirtualDocumentRoot /home/www/%0/html/
VirtualScriptAlias /home/www/%0/cgi-bin/
```

or like so:

```
VirtualDocumentRootIP /home/www/%0/html/
VirtualScriptAliasIP /home/www/%0/cgi-bin/
```

The VirtualScriptAlias and VirtualScriptAliasIP directives work similarly to ScriptAlias. Unlike ScriptAlias, though, you can't specify the origin directory to map; the directory is instead hardwired to /cgi-bin. The only way to change this is to preemptively rewrite the URL with mod_rewrite, but it may be simpler, in this case, to just use mod_rewrite for the whole virtual host configuration (of which I'll give an example later).

If you want to use a single cgi-bin directory for all your virtual hosts, you can just use a normal ScriptAlias directive:

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

You can also use ScriptAliasMatch to do something apparently similar to VirtualScriptAlias with this:

```
ScriptAliasMatch /cgi-bin/ /home/www/.*/cgi-bin/
```

However, this isn't the same. Although it does enable the individual cgi-bin directories for each virtual host, it enables all of them for all virtual hosts, so each virtual host is able to execute CGI scripts from any other host's cgi-bin. The VirtualScriptAlias directives enable only the cgi-bin directory for the host to which it belongs.

Logging Aliased Virtual Hosts

Given that you can't split logs for different virtual hosts using mod_vhost_alias, you need some way to distinguish them in the common error and access logs. To do this, you can use the %V and %A log format tokens to record the virtual hostname and IP

address, respectively. In this respect, handling log files for aliased virtual hosts is no different from handling a combined log for ordinary virtual hosts, which I've already discussed. To recap, you can redefine the standard transfer log to include named virtual host identities with this:

```
LogFormat "%V: %h %l %u %t \"%r\" %>s %b"
```

For IP-based virtual hosting, you can either use `UseCanonicalName` DNS or log the IP address instead with this:

```
LogFormat "%A: %h %l %u %t \"%r\" %>s %b"
```

In both cases, you can subsequently divide the combined log file into separate logs for each host using the `split-logfile` program, supplied as part of the standard Apache distribution.

As an intermediate solution, you can also separate groups of virtual hosts into different logs by constraining the scope of `mod_vhost_alias` with a `<VirtualHost>` container.

Constraining Aliased Virtual Hosts with a Virtual Host Container

Strange as it might seem, you can put directives such as `VirtualDocumentRoot` inside a `<VirtualHost>` container. The usefulness of this might not be immediately apparent, but in fact the two virtual host strategies can complement each other well. You can use the container to collect directives that should apply to all the aliased hosts at a given IP address and port number and then use aliasing to separate the individual document roots for the hosts that are matched by the container. For example:

```
<VirtualHost 204.148.170.3>
  ServerName server1.alpha-complex.com
  ServerAdmin webmaster@alpha-complex.com
  ServerAlias server[0-9].alpha-complex.com
  VirtualDocumentRoot /home/www/%1.1+.alpha-complex/web/
  VirtualScriptAlias /home/www/%1.1+.alpha-complex/cgi-bin/
  ErrorLog logs/alpha-complex_error
  TransferLog logs/alpha-complex_log
</VirtualHost>
```

This virtual host container matches requests for hosts `server0` to `server9` in the domain `alpha-complex.com`. The ten virtual hosts are given separate existences using `VirtualDocumentRoot` and `VirtualScriptAliasIP` directives to place their document roots and CGI directories into different locations. For `server0`, it's equivalent to this:

```
DocumentRoot /home/www/server0.alpha-complex/web/
ScriptAlias /cgi-bin/ /home/www/server0.alpha-complex/cgi-bin/
```

As described earlier, use the apparently redundant subpart specifier, `.1+`, to allow the following dot to be accepted by the directive.

Placing the aliasing directives within the virtual host allows you to constrain them to the IP address of the container (you could've added a port number, too). It also allows you to give them shared access and error logs separate from that of the main server or any other aliased hosts in the same configuration.

Mapping Hostnames Dynamically with mod_rewrite

Rather than specifying dozens of virtual host directives, you can use `mod_rewrite` to effectively fake name-based virtual hosts without actually configuring them:

```
# switch on rewriting rules
RewriteEngine on
# test for a host header and extract the domain name
RewriteCond %{HTTP_HOST} ^www\.(.+)$
# rewrite the URL to include the domain name in the path
RewriteRule ^/(.+)$ /home/www/%1/$1 [L].
```

For this to work, the DNS servers for the network must resolve any names you want to serve to the IP address for name-based hosts, but you'd have to do this anyway, of course. When a client asks for a URL such as `http://www.alpha-complex.com/index.html`, the `Rewrite` rule converts this into the file path `/home/www/alpha-complex.com/index.html`.

The beautiful thing about this is that it requires no knowledge of virtual hostnames by Apache, and you don't even need to restart Apache to add a new virtual host. All you have to do is add the new `hostname` to the DNS configuration of the name servers and create the appropriate directory.

This trick relies on the `Host:` header, so it only works reliably for HTTP/1.1 clients (HTTP/1.0 clients may choose to send a `Host:` header but aren't required to do so). You can catch HTTP/1.0 clients with another `RewriteRule`. Because the previous rule ended with an `[L]` flag to force immediate processing, this will only get used for requests without a `Host:` header:

```
RewriteRule ^.* http://www.alpha-complex.com/http10index.html [R,L]
```

If you really wanted to get efficient, you could also add a condition to test for the existence of the file being requested or, alternatively, the validity of the URL (which catches aliases missed by a file test):

```
# switch on rewriting rules
RewriteEngine on
# test for a Host: header via HTTP_HOST environment variable,
# and extract domain
RewriteCond %{HTTP_HOST} ^www\.(.+)$

# rewrite the URL to include the domain name in the resultant pathname
RewriteRule ^/(.+)$ /home/www/%1/$1 [C]
```

```
# test the new pathname to see if it actually matches a file
RewriteCond ^(.+)$ -f
# if it does, discontinue further rule processing
RewriteRule ^.* - [L]
```

With this solution, you don't get the chance to specify different access or error logs, and you also don't get to specify a different administrator mail address. But you can implement as many virtual hosts as you want in only four or five lines.

A more advanced solution could use a file map to relate domain names to pathnames (in effect, document roots), which can be anywhere in the file system:

```
RewriteMap vhost_docroot /usr/local/apache/conf/vhost.map
RewriteCond %{HTTP_HOST} ^www\.(.+)$
RewriteRule ^/(.+)$ %{vhost_docroot:%1}/$1 [C]
```

The `vhost.map` file would contain entries such as this:

```
alpha-complex.com    /home/www/alpha-complex
beta-complex.com     /usr/mirrors/betasite/www/beta-complex
```

Although `mod_vhost_alias` achieves a similar task, `mod_rewrite` can allow you greater flexibility in how you process URLs and, in particular, how you deal with invalid hostnames. In some cases, this might be a preferable alternative.

Generating On the Fly and Included Configuration Files with mod_perl

`mod_perl` is possibly the most powerful module available for Apache, embedding a complete Perl interpreter into the server. As well as allowing for Apache modules to be written in Perl and CGI scripts to be sped up with the Apache `Registry` module, `mod_perl` also allows Perl scripts to be embedded into the configuration file. This provides an extremely powerful tool for generating on-the-fly configurations.

> **NOTE** `mod_perl 2` *is currently still beta and, although very capable, should be used with care.*

Embedded Perl appears in Apache's configuration inside a `<Perl>...</Perl>` container, also known as a *Perl section*. Anything inside this container is executed by `mod_perl` when Apache starts. You specify configuration directives simply by assigning a package variable of the same name. For example:

```
<Perl>
  $ServerAdmin="webmaster@alpha-complex.com";
</Perl>
```

In Perl, this is a basic assignment of a string to a scalar variable. In a Perl section, it becomes a configuration directive because the variable corresponds to the name of a configuration directive understood by Apache. Because a Perl section programs Apache through variables, you don't want to use a lexical variable (declared with `my`, as in `my $ServerAdmin`) because these don't survive outside the section. Conversely, temporary variables should be lexical to make sure they don't end up in Apache's configuration.

Other kinds of configuration are achieved with different kinds of data structure. For example, directives that take multiple arguments can be specified either as a list or a string to equal effect. For example, this scalar-string assignment configures a directive with three values:

```
$DirectoryIndex="index.html index.shtml index.cgi";
```

You can achieve the same effect by assigning a list to an array, like this:

```
@DirectoryIndex=("index.html","index.shtml","index.cgi");
```

This is where Perl becomes more fun because you can create and manipulate arrays in all sorts of ways. For example, you can also achieve the previous with this:

```
foreach my $suffix qw(html shtml cgi) {
  push @DirectoryIndex "index.$suffix";
}
```

Virtual hosts (and indeed any kind of container) are implemented as hashes, with the keys being the names of the directives inside the container. The following is a more complex example that defines three virtual hosts. You define the source data as lexical `my` variables, so they don't interfere with the actual configuration:

```
<Perl>
  # the network address of all our hosts
  my $network="204.148.170.";

  # IP address to hostname definitions
  my %hosts={
    3 => "www.alpha-complex.com",
    4 => "users.alpha-complex.com",
    5 => "www.beta-complex.com"
  };

  # generate a virtual host definition for each host listed in %hosts
  foreach my $host_ip (sort keys %hosts) {
    my $servername=$hosts{$host_ip};
    $VirtualHost{$network.$host_ip}={
      ServerName => $servername,
      DocumentRoot => "/home/www/$servername/web",
      ServerAdmin => "webmaster@$servername",
```

```
      TransferLog => "/home/www/$servername/logs/access_log common",
      ErrorLog => "/home/www/$servername/logs/error_log";
    }
  }
</Perl>
```

Although deliberately limited in its abilities, this script already allows you to add a new virtual host with just one new line. Of course, you have access to the whole Perl language, including the ability to read files of configuration information.

You can define other containers inside the virtual host by assigning a hash as the value of a key in the virtual host hash and, indeed, for any container that's nested inside another. If you want to define two containers of the same type in the same place, you just create a hash of hashes. The following code shows part of the Perl section in the previous example adapted to place two `<Directory>` containers inside each virtual host definition:

```
$VirtualHost{$network.$host_ip}={
  ... other definitions ...
  Directory => {
    "/home/www/$servername/web/cgi-bin" => {
            Options => "+ExecCGI"
    },

    "/home/www/$servername/web/private" => {
            Order => "allow, deny",
            Deny => "from all",
            Allow => "from 127.0.0.1",
    },
  };
}
```

Although an understanding of Perl is useful for making sense of this example, the general form of it is easily adaptable even without it.

As a more complete and actually more useful example, the following script will generate a series of IP-based or name-based virtual hosts, including a mixture of the two, based on the contents of an external file. It'll also generate the corresponding `Listen` and, where appropriate, `NameVirtualHost` directives. Finally, it'll work both as an embedded Perl section and as a stand-alone script. In the latter case, the configuration is written out to standard output and can be redirected to a file that in turn can be included into the server configuration with `Include`. You can also use this output to check the results you generate are actually correct. You could also use the `Apache::PerlSections` module that comes with `mod_perl` to dump out the configuration as Perl variable definitions.

First, create a configuration file containing the host details:

```
# conf/vhosts.conf - configuration file for mod_perl generated hosts
#
# File format:
```

```
#
# IP(:port), hostname, admin e-mail, document root, aliases
#   additional directive
#   additional directive
#   ...
# IP(:port), hostname, admin e-mail, document root, aliases
# ...
#


# For IP vhosts ignore the aliases column. For Named vhosts remember to add
# appropriate NameVirtualHost directives to httpd.conf

204.148.170.3:443, secure.alpha-complex.com, secure@alpha-complex.com,
        alpha-complex/secure, shop.alpha-complex.com
        SSLEngine on
204.148.170.4, users.alpha-complex.com, users@alpha-complex.com, alpha-complex/users
204.148.170.5:8080, proxy.alpha-complex.com, proxy@alpha-complex.com, proxy
        ProxyRequests On
        AllowCONNECT 443 23
204.148.170.6, www.beta-complex.com, webmaster@beta-complex.com, beta-complex

# define this last so wildcard alias catches
204.148.170.3, www.alpha-complex.com, webmaster@alpha-complex.com,
         alpha-complex, *.alpha-complex.com
```

Now write the Perl script and embed it into `http.conf` in a `<Perl>` container:

```
... rest of httpd.conf ...
# generate virtual hosts on the fly with Perl
<Perl>
#!/usr/bin/perl -w
#line <n>
# The above along with the __END__ at the bottom allows us to check the
# syntax of the section with 'perl -cx httpd.conf'. Change the '<n>' in
# '#line <n>' to whatever line in httpd.conf the Perl section really starts

# Define some local variables. These are made local so Apache doesn't
# see them and try to interpret them as configuration directives
local ($ip,$host,$admin,$vroot,$aliases);
local ($directive,$args);

# Open the virtual hosts file
open (FILE,"/usr/local/apache139/conf/vhosts.conf");

# Pull lines from the file one by one
while (<FILE>) {
  # Skip comments and blank lines
  next if /^\s*(#|$)/;
```

```
  # If the line starts with a number it's the IP of a new host
  if (/^\d+/) {

    # Extract core vhost values and assign them
    ($ip,$host,$admin,$vroot,$aliases)=split /\s*,\s*/,$_;
    $VirtualHost{$ip}={
      ServerName => $host,
      ServerAdmin => $admin,
      DocumentRoot => "/home/www/".$vroot,
      ErrorLog => "logs/".$host."_error",
      TransferLog => "logs/".$host."_log"
    };

    # If we have any aliases, assign them to a ServerAlias directive
    $VirtualHost{$ip}{ServerAlias}=$aliases if $aliases;

    # If the IP has a port number attached, infer and add a Port directive
    $VirtualHost{$ip}{Port}=$1 if ($ip=~/:(\d+)$/);

  # Otherwise it's an arbitrary additional directive for the current host
  } elsif ($ip) {
    # Note this only handles simple directives, not containers
    ($directive,$args)=split / /,$_,2;
    $VirtualHost{$ip}{$directive}=$args;
  }
}

# All done
close (FILE);

# Tell 'perl -cx' to stop checking
__END__
# back to httpd.conf
</Perl>
```

This generates `VirtualHost` directives from the configuration file:

```
<VirtualHost 204.148.170.3:443>
  ServerName secure.alpha-complex.com
  ServerAdmin secure@alpha-complex.com
  DocumentRoot /home/www/alpha-complex
  ErrorLog logs/www.alpha-complex.com_error
  TransferLog logs/www.alhpacomplex.com_log
  ServerAlias shop.alpha-complex.com buy.alpha-complex.com
  Port 443
  SSLEngine on
</VirtualHost>
```

Of course, you don't actually see these definitions because they're internal to the server. This is just what they would look like if you had defined them by hand. Better, you can keep the script as a separate file, allowing you to run it independently of Apache and include it in the Perl section:

```
<Perl>
  require "/usr/local/apache/conf/genvhosts.pl";
</Perl>
```

or even just like so:

```
PerlRequire /usr/local/apache/conf/genvhosts.pl
```

The Perl section automatically places its contents into the `Apache::ReadConfig` package, so it's equally valid (but usually redundant) to say this:

```
$Apache::ReadConfig::ServerAdmin="webmaster@alpha-complex.com";
```

The upshot of this is that if you want an external script to generate a configuration that Apache will process, then you need to ensure that it places the configuration details into this package rather than the default `main` package. The `genvhosts.pl` script previously makes sure of this by putting the package `Apache::ReadConfig` at the start of the script.

Because the script will also run outside of the server, you can also `Include` the results:

```
cd /usr/local/apache/conf/
genhosts.pl > Vhosts.conf

... in httpd.conf ...
Include conf/Vhosts.conf
```

Of course, you could use any scripting language to create a configuration file this way because Apache no longer needs to know how to interpret it itself. When you add a new host, you just have to remember to rerun the script to regenerate the file before restarting Apache with `apachectl graceful`.

Although you get everything you could want with this kind of strategy, it does have a downside—Apache's configuration becomes as large as if you'd specified each host in the file and requires the same memory overhead to store it. The advantage of `mod_vhost_alias` and `mod_rewrite` is that Apache's runtime configuration after parsing remains small, giving improved performance in cases where large numbers of hosts are involved. You'll take a closer look at `mod_perl` in Chapter 12.

Summary

In this chapter, you saw how to configure Apache to serve several different Web sites using user directories, separate invocations of Apache, and finally virtual hosts. You've considered the issues involved in user directories and discussed the capabilities of the `mod_userdir` module and how other directives can simulate it. You also dealt with the problem of running several different Apache configurations at the same time, as well as how to simplify this task through the sharing of common configuration files.

You've looked at the difference between IP and name-based virtual hosts and how to configure each, as well as combining the two kinds of host together in the same configuration. You also considered some of the security issues surrounding virtual hosts where other administrators are responsible for the content of individual hosts and used the `perchild` MPM to fully partition hosts under different owners.

You finished off by considering the demands of mass hosting and examined several ways to deal with large numbers of hosts, first using the `mod_vhost_alias` and `mod_rewrite` modules and then using embedded Perl sections and external scripts to automate virtual-host configurations in different ways.