APACHE

An HTTP Server

Reference Manual

Contents

Preface

This manual documents version 1.0 of the Apache HTTP server This server is a product of the Apache project, which was organised in an attempt to answer some of the concerns regarding active development of a freely available HTTP server. The goal of this project is to provide a secure, efficient and extensible server which provides HTTP services in accordance with current HTTP standards.

The Apache httpd server is designed to be both compatible with the NCSA httpd version 1.3, and also provide frequently requested features, such as

- DBM databases for authentication.
- Customised responses to errors and problems.
- Multiple `directoryindex` directives.
- Unlimited numbers of `Alias` and `Redirect` directives.
- Content−based document negotiation.
- Virtual servers.

The Apache Licence

This software consists of voluntary contributions made by many individuals on behalf of the Apache Group and was originally based on public domain software written at the National Center for Supercomputing Applications, University of Illinois, Urbana–Champaign. For more information on the Apache Group and the Apache HTTP server project, please see http://www.apache.org/.

Chapter 1

Compiling and Installing Apache

1.1 Downloading Apache

Information on the latest version of Apache can be found on the Apache web server at http://www.apache.org/. This will list the current release, any more recent beta–test release, together with details of mirror web and anonymous ftp sites.

1.2 Compiling Apache

This release of Apache supports the notion of 'optional modules'. However, the server has to know which modules are compiled into it, in order for those modules to be effective; this requires generation of a short bit of code ('`modules.c`') which simply has a list of them.

If you are satisfied with our standard module set, and expect to continue to be satisfied with it, then you can just edit the stock `Makefile` and compile as you have been doing previously. If you would like to select optional modules, however, you need to run the configuration script.

To do this:

1. Edit the file '`Configuration`'. This contains the per–machine config settings of the Makefile, and also an additional section at the bottom which lists the modules which have been compiled in, and also names the files containing them. You will need to:
   (a) Select a compiler and compilation options as appropriate to your machine.
   (b) Uncomment lines corresponding to those optional modules you wish to include (among the Module lines at the bottom of the file) or add new lines corresponding to custom modules you have written.
   Note that DBM auth has to be explicitly configured in, if you want it; just uncomment the corresponding line.
2. Run the 'Configure' script:

```
% Configure
Using 'Configuration' as config file
```

```
%
```

This generates new versions of the Makefile and of modules.c. If you want to maintain multiple configurations, you can say, e.g.,

```
% Configure -file Configuration.ai
Using alternate config file Configuration.ai
%
```

3. Type ‘make’.

   The modules we place in the Apache distribution are the ones we have tested and are used regularly by various members of the Apache development group. Additional modules contributed by members or third parties with specific needs or functions are available at <URL:http://www.apache.org/dist/contrib/modules/>. There are instructions on that page for linking these modules into the core Apache code.

1.3 Installing Apache

After compilation, you will have a binary called ‘httpd’ in the `src/` directory. A binary distribution of Apache will supply this file.

The next step is to edit the configuration files for the server. In the subdirectory called ‘conf’ you should find distribution versions of the three configuration files: `srm.conf-dist`, `access.conf-dist` and `httpd.conf-dist`. Copy them to `srm.conf`, `access.conf` and `httpd.conf` respectively.

First edit `httpd.conf`. This sets up general attributes about the server; the port number, the user it runs as, etc. Next edit the `srm.conf` file; this sets up the root of the document tree, special functions like server-parsed HTML or internal imagemap parsing, etc. Finally, edit the `access.conf` file to at least set the base cases of access.

Finally, make a call to httpd, with a -f to the full path to the httpd.conf file. I.e., the common case:

```
/usr/local/etc/apache/src/httpd -f
/usr/local/etc/apache/conf/httpd.conf
```

The server should be now running.

By default the `srm.conf` and `access.conf` files are located by name; to specifically call them by other names, use the AccessConfig and ResourceConfig directives in `httpd.conf`.

Chapter 2

Starting Apache

2.1 Invoking Apache

The `httpd` program is either invoked by the Internet daemon `inetd` each time a connection to the HTTP service is made, or alternatively it may run as a daemon which executes continuously, handling requests. Whatever method is chosen, the ServerType directive must be set to tell the server how it is to run.

2.2 Command line options

The following options are recognised on the httpd command line:

`-d` *serverroot*
: Set the initial value for the ServerRoot variable to *serverroot*. This can be overridden by the ServerRoot command in the configuration file. The default is `/usr/local/etc/httpd`.

`-f` *config*
: Execute the commands in the file *config* on startup. If *config* does not begin with a `/`, then it is taken to be a path relative to the ServerRoot. The default is `conf/httpd.conf`.

`-X`
: Run in single–process mode, for internal debugging purposes only; the daemon does not detach from the terminal or fork any children. Do *NOT* use this mode to provide ordinary web service.

`-v`
: Print the version of httpd, and then exit.

`-?`
: Print a list of the httpd options, and then exit.

2.3 Configuration files

The server will read three files for configuration directives. Any directive may appear in any of these files. The the names of these files are taken to be relative to the server root; this is set by the ServerRoot directive, or the `-d` command line flag. Conventionally, the files are:

`conf/httpd.conf`
: Contains directives that control the operation of the server daemon. The filename may be overridden with the `-f` command line flag.

`conf/srm.conf`
: Contains directives that control the specification of documents that the server can provide to clients. The filename may be overridden with the ResourceConfig directive.

`conf/acces.conf`
: Contains directives that control access to documents. The filename may be overridden with the AccessConfig directive.

However, these conventions need not be adhered to.

The server also reads a file containing mime document types; the filename is set by the TypesConfig directive, and is `conf/mime.types` by default.

2.4 Log files

2.4.1 pid file

On daemon startup, it saves the process id of the parent httpd process to the file `logs/httpd.pid`. This filename can be changed with the PidFile directive. The process–id is for use by the administrator in restarting and terminating the daemon; A HUP signal causes the daemon to re–read its configuration files and a TERM signal causes it to die gracefully.

If the process dies (or is killed) abnormally, then it will be necessary to kill the children httpd processes.

2.4.2 Error log

The server will log error messages to a log file, `logs/error_log` by default. The filename can be set using the ErrorLog directive; different error logs can be set for different virtual hosts.

2.4.3 Transfer log

The server will typically log each request to a transfer file, `logs/access_log` by default. The filename can be set using a TransferLog directive; different transfer logs can be set for different virtual hosts.

Chapter 3

Apache Core Features

These configuration parameters control the core Apache features, and are always available.

3.1 AccessConfig directive

Syntax: AccessConfig *filename*
Default: `AccessConfig conf/access.conf`
Context: server config, virtual host
Status: core

The server will read this file for more directives after reading the ResourceConfig file. *Filename* is relative to the ServerRoot. This feature can be disabled using:

```
AccessConfig /dev/null
```

Historically, this file only contained <Directory> sections; in fact it can now contain any server directive allowed in the *server config* context.

3.2 AccessFileName directive

Syntax: AccessFileName *filename*
Default: `AccessFileName .htaccess`
Context: server config, virtual host
Status: core

When returning a document to the client the server looks for an access control file with this name in every directory of the path to the document, if access control files are enabled for that directory. For example:

```
AccessFileName .acl
```

before returning the document /usr/local/web/index.html, the server will read /.acl, /usr/.acl, /usr/local/.acl and /usr/local/web/.acl for directives, unless they have been disabled with

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

3.3 AllowOverride directive

Syntax: AllowOverride *override override ...*
Default: `AllowOverride All`
Context: directory
Status: core

When the server finds an .htaccess file (as specified by AccessFileName) it needs to know which directives declared in that file can override earlier access information.

Override can be set to `None`, in which case the server will not read the file, `All` in which case the server will allow all the directives, or one or more of the following:

AuthConfig
Allow use of the authorization directives (AuthDBMGroupFile, AuthDBMUserFile, AuthGroupFile, AuthName, AuthType, AuthUserFile and require).

FileInfo
Allow use of the directives controlling document types (AddEncoding, AddLanguage, AddType, DefaultType and LanguagePriority).

Indexes
Allow use of the directives controlling directory indexing (AddDescription, AddIcon, AddIconByEn–coding, AddIconByType, DefaultIcon, DirectoryIndex, FancyIndexing, HeaderName, IndexIgnore, IndexOptions and ReadmeName).

Limit
Allow use of the directives controlling host access (allow, deny and order).

Options
Allow use of the directives controlling specific directory features (Options and XBitHack).

3.4 AuthName directive

Syntax: AuthName *auth-domain*
Context: directory, .htaccess
Override: AuthConfig
Status: core

This directive sets the name of the authorization realm for a directory. This realm is given to the client so that the user knows which username and password to send. It must be accompanied by AuthType and require directives, and directives such as AuthUserFile and AuthGroupFile to work.

3.5 AuthType directive

Syntax: AuthType *type*
Context: directory, .htaccess
Override: AuthConfig
Status: core

This directive selects the type of user authentication for a directory. Only `Basic` is currently implemented. It must be accompanied by AuthName and require directives, and directives such as AuthUserFile and AuthGroupFile to work.

3.6 BindAddress directive

Syntax: BindAddress *saddr*
Default: `BindAddress *`
Context: server config
Status: core

A Unix® http server can either listen on for connections to every IP address of the server machine, or just one IP address of the server machine. *Saddr* can be

- *
- An IP address
- A fully−qualified internet domain name

If the value is *, then the server will listen for connections on every IP address, otherwise it will only listen on the IP address specified.

This option can be used as an alternative method for supporting virtual hosts instead of using <VirtualHost> sections.

3.7 DefaultType directive

Syntax: DefaultType *mime-type*
Default: `DefaultType text/html`
Context: server config, virtual host, directory, .htaccess
Override: FileInfo
Status: core

There will be times when the server is asked to provide a document whose type cannot be determined by its MIME types mappings.

The server must inform the client of the content–type of the document, so in the event of an unknown type it uses the `DefaultType`. For example:

```
DefaultType image/gif
```

would be appropriate for a directory which contained many gif images with filenames missing the .gif extension.

3.8 <Directory> directive

Syntax: <Directory *directory*> ... </Directory>
Context: server config, virtual host
Status: Core.

<Directory> and </Directory> are used to enclose a group of directives which will apply only to the named directory and sub–directories of that directory. Any directive which is allowed in a directory context may be used. *Directory* is either the full path to a directory, or a wildcard string. In a wildcard string, '?' matches any single character, and '*' matches any sequences of characters. Example:

```
<Directory /usr/local/httpd/htdocs>
Options Indexes FollowSymLinks
</Directory>
```

If multiple directory sections match the directory (or its parents) containing a document, then the directives are applied in the order of shortest match first, interspersed with the directives from the .htaccess files. For example, with

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

<Directory /home/*>
AllowOverride FileInfo
</Directory>
```

The for access to the document `/home/web/dir/doc.html` the steps are:

- Apply directive `AllowOverride None` (disabling `.htaccess` files).
- Apply directive `AllowOverride FileInfo` (for directory `/home/web`).
- Apply any FileInfo directives in `/home/web/.htaccess`

The directory sections typically occur in the access.conf file, but they may appear in any configuration file. <Directory> directives cannot nest, and cannot appear in a <Limit> section.

3.9 DocumentRoot directive

Syntax: DocumentRoot *directory-filename*
Default: `DocumentRoot /usr/local/etc/httpd/htdocs`
Context: server config, virtual host
Status: core

This directive sets the directory from which httpd will serve files. Unless matched by a directive like Alias, the server appends the path from the requested URL to the document root to make the path to the document. Example:

```
DocumentRoot /usr/web
```

then an access to `http://www.my.host.com/index.html` refers to `/usr/web/index.html`.

3.10 ErrorDocument directive

Syntax: ErrorDocument *error-code document*
Context server config, virtual host
Status: core

In the event of a problem or error, Apache can be configured to do one of four things,

1. behave like NCSA httpd 1.3
2. output a customized message
3. redirect to a local URL to handle the problem/error
4. redirect to an external URL to handle the problem/error

2–4 are configured using `ErrorDocument`, which is followed by the HTTP response code and a message or URL.

Messages in this context, begin with a single quote (`"`), which does not form part of the message itself. Apache will sometime offer additional information regarding the problem/error. This can be embedded into the message using `%s`

URLs will begin with a slash (/) for local URLs, or will be a full URL which the client can resolve. Examples:

```
ErrorDocument 500 /cgi-bin/tester
ErrorDocument 404 /cgi-bin/bad_urls.pl
ErrorDocument 401 http://www2.foo.bar/subscription_info.html
ErrorDocument 403 "Sorry can't allow you access today
```

See Also: documentation of customizable responses.

3.11 ErrorLog directive

Syntax: ErrorLog *filename*
Default: `ErrorLog logs/error_log`
Context: server config, virtual host
Status: core

The error log directive sets the name of the file to which the server will log any errors it encounters. If the filename does not begin with a slash (/) then it is assumed to be relative to the ServerRoot. Example:

```
ErrorLog /dev/null
```

This effectively turns off error logging.

3.12 Group directive

Syntax: Group *unix-group*
Default: `Group #-1`
Context: server config
Status: core

The Group directive sets the group under which the server will answer requests. In order to use this directive, the standalone server must be run initially as root. *Unix-group* is one of:

A group name
Refers to the given group by name.

 # followed by a group number.
Refers to a group by its number.

It is recommended that you set up a new group specifically for running the server. Some admins use user `nobody`, but this is not always possible or desirable.

Note: if you start the server as a non–root user, it will fail to change to the specified, and will instead continue to run as the group of the original user.

SECURITY: See User for a discussion of the security considerations.

3.13 IdentityCheck directive

Syntax: IdentityCheck *boolean*
Default: `IdentityCheck off`
Context: server config
Status: core

This directive enables RFC931–compliant logging of the remote user name for each connection, where the client machine runs identd or something similar. This information is logged in the access log. *Boolean* is either `on` or `off`.

The information should not trusted in any way except for rudimentary usage tracking.

3.14 <Limit> directive

Syntax: <Limit *method method* ... > ... </Limit>
Context: any
Status: core

<Limit> and </Limit> are used to enclose a group of access control directives which will then apply only to the specified access methods, where *method* is any valid HTTP method. Any directive except another <Limit> or <Directory> may be used; the majority will be unaffected by the <Limit>. Example:

```
<Limit GET POST>
require valid-user
</Limit>
```

If an access control directive appears outside a <Limit> directive, then it applies to all access methods.

3.15 MaxClients directive

Syntax: MaxClients *number*
Default: `MaxClients 150`
Context: server config
Status: core

The MaxClients directive sets the limit on the number of simultaneous requests that can be supported; not more than this number of child server processes will be created.

3.16 MaxRequestsPerChild directive

Syntax: MaxRequestsPerChild *number*
Default: `MaxRequestsPerChild 0`
Context: server config
Status: core

The MaxRequestsPerChild directive sets the limit on the number of requests that an individual child server process will handle. After MaxRequestsPerChild requests, the child process will die. If MaxRequestsPerChild is 0, then the process will never expire.

Setting MaxRequestsPerChild to a non–zero limit has two beneficial effects:

- it limits the amount of memory that process can consume by (accidental) memory leakage;
- by giving processes a finite lifetime, it helps reduce the number of processes when the server load reduces.

3.17 MaxSpareServers directive

Syntax: MaxSpareServers *number*
Default: `MaxSpareServers 10`
Context: server config
Status: core

The MaxSpareServers directive sets the desired maximum number of *idle* child server processes. An idle process is one which is not handling a request. If there are more than MaxSpareServers idle, then the parent process will kill off the excess processes.

Tuning of this parameter should only be necessary on very busy sites. Setting this parameter to a large number is almost always a bad idea.

See also MinSpareServers and StartServers.

3.18 MinSpareServers directive

Syntax: MinSpareServers *number*
Default: `MinSpareServers 5`
Context: server config
Status: core

The MinSpareServers directive sets the desired minimum number of *idle* child server processes. An idle process is one which is not handling a request. If there are fewer than MinSpareServers idle, then the parent process creates new children at a maximum rate of 1 per second.

Tuning of this parameter should only be necessary on very busy sites. Setting this parameter to a large number is almost always a bad idea.

See also MaxSpareServers and StartServers.

3.19 Options directive

Syntax: Options *option option ...*
Context: server config, virtual host, directory, .htaccess
Override: Options
Status: core

The Options directive controls which server features are available in a particular directory.

option can be set to `None`, in which case none of the extra features are enabled, or one or more of the following:

All
All options except for MultiViews.

ExecCGI
Execution of CGI scripts is permitted.

FollowSymLinks
The server will follow symbolic links in this directory.

Includes
Server–side includes are permitted.

IncludesNOEXEC
Server–side includes are permitted, but the #exec command and #include of CGI scripts are disabled.

Indexes
If a URL which maps to a directory is requested, and the there is no DirectoryIndex (e.g. index.html) in that directory, then the server will return a formatted listing of the directory.

MultiViews
Ccontent negotiatad MultiViews are allowed.

SymLinksIfOwnerMatch
The server will only follow symbolic links for which the target file or directory is owned by the same user id as the link.

If multiple Options could apply to a directory, then the most specific one is taken complete; the options are not merged. For example:

```
<Directory /web/docs>
Options Indexes FollowSymLinks
</Directory>
<Directory /web/docs/spec>
Options Includes
</Directory>
```

then only `Includes` will be set for the /web/docs/spec directory.

3.20 PidFile directive

Syntax: PidFile *filename*
Default: `PidFile logs/httpd.pid`
Context: server config
Status: core

The PidFile directive sets the file to which the server records the process id of the daemon. If the filename does not begin with a slash (/) then it is assumed to be relative to the ServerRoot. The PidFile is only used in standalone mode.

It is often useful to be able to send the server a signal, so that it closes and then reopens its ErrorLog and TransferLog, and re–reads its configuration files. This is done by sending a SIGHUP (kill −1) signal to the process id listed in the PidFile.

3.21 Port directive

Syntax: Port *number*
Default: `Port 80`
Context: server config
Status: core

The Port directive sets the network port on which the server listens. *Num* is a number from 0 to 65535; some port numbers (especially below 1024) are reserved for particular protocols. See `/etc/services` for a list of some defined ports; the standard port for the http protocol is 80.

Port 80 is one of Unix's special ports. All ports numbered below 1024 are reserved for system use, i.e. regular (non–root) users cannot make use of them; instead they can only use higher port numbers.

To use port 80, you must start the server from the root account. After binding to the port and before accepting requests, Apache will change to a low privileged user as set by the User directive.

If you cannot use port 80, choose any other unused port. Non–root users will have to choose a port number higher than 1023, such as 8000.

SECURITY: if you do start the server as root, be sure not to set User to root. If you run the server as root whilst handling connections, your site may be open to a major security attack.

3.22 require directive

Syntax: require *entity-name entity entity...*
Context: directory, .htaccess
Override: AuthConfig
Status: core

This directive selects which authenticated users can access a directory. The allowed syntaxes are:

- require user *userid userid ...*

  Only the named users can access the directory.

- require group *group-name group-name ...*

  Only users in the named groups can access the directory.

- require valid–user

  All valid users can access the directory.

If `require` appears in a <Limit>section, then it restricts access to the named methods, otherwise it restricts access for all methods. Example:

```
AuthType Basic
AuthName somedomain
AuthUserFile /web/users
AuthGroupFile /web/groups
Limit <GET POST>
require group admin
</Limit>
```

Require must be accompanied by AuthName and AuthType directives, and directives such as AuthUserFile and AuthGroupFile (to define users and groups) in order to work correctly.

3.23 ResourceConfig directive

Syntax: ResourceConfig *filename*
Default: `ResourceConfig conf/srm.conf`
Context: server config, virtual host
Status: core

The server will read this file for more directives after reading the httpd.conf file. *Filename* is relative to the ServerRoot. This feature can be disabled using:

```
ResourceConfig /dev/null
```

Historically, this file contained most directives except for server configuration directives and <Directory>. sections; in fact it can now contain any server directive allowed in the *server config* context.

See also AccessConfig.

3.24 ServerAdmin directive

Syntax: ServerAdmin *email-address*
Context: server config, virtual host
Status: core

The ServerAdmin sets the e–mail address that the server includes in any error messages it returns to the client.

It may be worth setting up a dedicated address for this, e.g.

```
ServerAdmin www-admin@foo.bar.com
```

as users do not always mention that they are talking about the server!

3.25 ServerName directive

Syntax: ServerName *fully-qualified domain name*
Context: server config, virtual host
Status: core

The ServerName directive sets the hostname of the server; this is only used when creating redirection URLs. If it is not specified, then the server attempts to deduce it from its own IP address; however this may not work reliably, or may not return the preferred hostname. For example:

```
ServerName www.wibble.com
```

would be used if the canonical (main) name of the actual machine were `monster.wibble.com`.

3.26 ServerRoot directive

Syntax: ServerRoot *directory-filename*
Default: `ServerRoot /usr/local/etc/httpd`
Context: server config
Status: core

The ServerRoot directive sets the directory in which the server lives. Typically it will contain the subdirectories `conf/` and `logs/`. Relative paths for other configuration files are taken as relative to this directory.

3.27 ServerType directive

Syntax: ServerType *type*
Default: `ServerType standalone`
Context: server config
Status: core

The ServerType directive sets how the server is executed by the system. *Type* is one of

inetd
The server will be run from the system process inetd; the command to start the server is added to `/etc/inetd.conf`

standalone
The server will run as a daemon process; the command to start the server is added to the system startup scripts. (`/etc/rc.local` or `/etc/rc3.d/....`)

Inetd is the lesser used of the two options. For each http connection received, a new copy of the server is started from scratch; after the connection is complete, this program exits. There is a high price to pay per connection, but for security reasons, some admins prefer this option.

Standalone is the most common setting for ServerType since it is far more efficient. The server is started once, and services all subsequent connections. If you intend running Apache to serve a busy site, standalone will probably be your only option.

SECURITY: if you are paranoid about security, run in inetd mode. Security cannot be guaranteed in either, but whilst most people are happy to use standalone, inetd is probably least prone to attack.

3.28 StartServers directive

Syntax: StartServers *number*
Default: `StartServers 5`
Context: server config
Status: core

The StartServers directive sets the number of child server processes created on startup. As the number of processes is dynamically controlled depending on the load, there is usually little reason to adjust this parameter.

See also MinSpareServers and MaxSpareServers.

3.29 TimeOut directive

Syntax: TimeOut *number*
Default: `TimeOut 1200`
Context: server config
Status: core

The TimeOut directive sets the maximum time that the server will wait for the receipt of a request and the completion of a request, in seconds. So if it takes more than TimeOut seconds for a client to send a request or receive a response, the server will break off the connection. Thus TimeOut limits the maximum a transfer can take; for large files, and slow networks transfer times can be large.

3.30 User directive

Syntax: User *unix-userid*
Default: `User #-1`
Context: server config
Status: core

The User directive sets the userid as which the server will answer requests. In order to use this directive, the standalone server must be run initially as root. *Unix-userid* is one of:

A username
Refers to the given user by name.

 # followed by a user number.
Refers to a user by their number.

The user should have no privileges which result in it being able to access files which are not intended to be visible to the outside world, and similarly, the user should not be able to execute code which is not meant for httpd requests. It is recommended that you set up a new user and group specifically for running the server. Some admins use user `nobody`, but this is not always possible or desirable.

Notes: If you start the server as a non–root user, it will fail to change to the lesser privileged user, and will instead continue to run as that original user. If you do start the server as root, then it is normal for the parent process to remain running as root.

SECURITY: Don't set User (or Group) to `root` unless you know exactly what you are doing, and what the dangers are.

3.31 <VirtualHost> directive

Syntax: <VirtualHost *addr*> ... </Directory>
Context: server config
Status: Core.

<VirtualHost> and </VirtualHost> are used to enclose a group of directives which will apply only to a particular virtual host. Any directive which is allowed in a virtual host context may be used. When the server receives a request for a document on a particular virtual host, it uses the configuration directives enclosed in the <VirtualHost>section. *Addr* can be

- The IP address of the virtual host
- A fully qualified domain name for the IP address of the virtual host.

Example:

```
<VirtualHost host.foo.com>
ServerAdmin webmaster@host.foo.com
DocumentRoot /www/docs/host.foo.com
ServerName host.foo.com
ErrorLog logs/host.foo.com-error_log
TransferLog logs/host.foo.com-access_log
</VirtualHost>
```

Currently, each VirtualHost must correspond to a different IP address for the server, so the server machine must be configured to accept IP packets for multiple addresses. If the machine does not have multiple network interfaces, then this can be accomplished with the `ifconfig alias` command (if your OS supports it), or with kernel patches like VIF (for SunOS(TM) 4.1.x).

Chapter 4

Apache Standard Modules

4.1 Module mod_access

This module is contained in the `mod_access.c` file, and is compiled in by default. It provides access control based on client hostname or IP address.

4.1.1 allow

Syntax: allow from *host host ...*
Context: directory, .htaccess
Override: Limit
Status: Base
Module: mod_access

The allow directive affects which hosts can access a given directory; it is typically used within a <Limit> section. *Host* is one of the following:

`all`
: all hosts are allowed access

A (partial) domain-name
: host whose name is, or ends in, this string are allowed access.

A full IP address
: An IP address of a host allowed access

A partial IP address
: The first 1 to 3 bytes of an IP address, for subnet restriction.

Example:

```
allow from .ncsa.uiuc.edu
```

All hosts in the specified domain are allowed access.

Note that this compares whole components; `bar.edu` would not match `foobar.edu`.

See also deny and order.

4.1.2 deny

Syntax: deny from *host host ...*
Context: directory, .htaccess
Override: Limit
Status: Base
Module: mod_access

The deny directive affects which hosts can access a given directory; it is typically used within a <Limit> section. *Host* is one of the following:

`all`
: all hosts are denied access

A (partial) domain-name
: host whose name is, or ends in, this string are denied access.

A full IP address
: An IP address of a host denied access

A partial IP address
: The first 1 to 3 bytes of an IP address, for subnet restriction.

Example:

```
deny from 16
```

All hosts in the specified network are denied access.

Note that this compares whole components; `bar.edu` would not match `foobar.edu`.

See also allow and order.

4.1.3 order

Syntax: order *ordering*
Default: `order deny,allow`
Context: directory, .htaccess
Override: Limit
Status: Base
Module: mod_access

The order directive controls the order in which allow and deny directives are evaluated. *Ordering* is one of

deny,allow
the deny directives are evaluated before the allow directives.

allow,deny
the allow directives are evaluated before the deny directives.

mutual-failure
Only those hosts which appear on the allow list and do not appear on the deny list are granted access.

Example:

```
order deny,allow deny from all allow from .ncsa.uiuc.edu
```

Hosts in the ncsa.uiuc.edu domain are allowed access; all other hosts are denied access.

4.2 Module mod_alias

This module is contained in the `mod_alias.c` file, and is compiled in by default. It provides for mapping different parts of the host filesystem in the the document tree, and for URL redirection.

4.2.1 Alias

Syntax: Alias *url-path directory-filename*
Context: server config, virtual host
Status: Base
Module: mod_alias

The Alias directive allows documents to be stored in the local filesystem other than under the DocumentRoot. URLs with a (%–decoded) path beginning with *url-path* will be mapped to local files beginning with *directory-filename*. Example:

```
Alias /image /ftp/pub/image
```

A request for http://myserver/images/foo.gif would cause the server to return the file /ftp/pub/images/foo.gif.

See also ScriptAlias.

4.2.2 Redirect

Syntax: Redirect *url-path url*
Context: server config, virtual host
Status: Base
Module: mod_alias

The Redirect directive maps an old URL into a new one. The new URL is returned to the client which attempts to fetch it again with the new address. *Url-path* a (%–decoded) path; any requests for documents beginning with this path will be returned a redirect error to a new (%–encoded) url beginning with *url*. Example:

```
Redirect /service http://foo2.bar.com/service
```

If the client requests http://myserver/service/foo.txt, it will be told to access http://foo2.bar.com/service/foo.txt instead.

Note: Redirect directives take precedence over Alias and ScriptAlias directives, irrespective of their ordering in the configuration file.

4.2.3 ScriptAlias

Syntax: ScriptAlias *url-path directory-filename*
Context: server config, virtual host
Status: Base
Module: mod_alias

The ScriptAlias directive has the same behaviour as the Alias directive, except that in addition it marks the target directory as containing CGI scripts. URLs with a (%–decoded) path beginning with *url-path* will be mapped to scripts beginning with *directory-filename*. Example:

```
ScriptAlias /cgi-bin/ /web/cgi-bin/
```

A request for http://myserver/images/foo would cause the server to run the script /web/cgi–bin/foo.

4.3 Module mod_asis

This module is contained in the `mod_asis.c` file, and is compiled in by default. It provides for `.asis` files. Any document with mime type `httpd/send-as-is` will be processed by this module.

4.3.1 Purpose

To allow file types to be defined such that Apache sends them without adding HTTP headers.

This can be used to send any kind of data from the server, including redirects and other special HTTP responses, without requiring a cgi–script or an nph script.

4.3.2 Usage

In the server configuration file, define a new mime type called `httpd/send-as-is` e.g.

```
AddType httpd/send-as-is asis
```

this defines the `.asis` file extension as being of the new `httpd/send-as-is` mime type. The contents of any file with a `.asis` extension will then be sent by Apache to the client with almost no changes. Clients will need HTTP headers to be attached, so do not forget them. A Status: header is also required; the data should be the 3–digit HTTP response code, followed by a textual message.

Here's an example of a file whose contents are sent *as is* so as to tell the client that a file has redirected.

```
Status:  302 Now where did I leave that URL
Location:  http://xyz.abc.com/foo/bar.html
Content-type:  text/html

<HTML>
<HEAD>
<TITLE>Lame excuses'R'us</TITLE>
</HEAD>
<BODY>
<H1>Fred's exceptionally wonderful page has moved to
<A HREF="http://xyz.abc.com/foo/bar.html">Joe's</A> site.
</H1>
</BODY>
</HTML>
```

Notes: the server always adds a Date: and Server: header to the data returned to the client, so these should not be included in the file. The server does *not* add a Last–Modified header; it probably should.

4.4 Module mod_auth

This module is contained in the `mod_auth.c` file, and is compiled in by default. It provides for user authentication using textual files.

4.4.1 AuthGroupFile

Syntax: AuthGroupFile *filename*
Context: directory, .htaccess
Override: AuthConfig
Status: Base
Module: mod_auth

The AuthGroupFile directive sets the name of a textual file containing the list of user groups for user authentication. *Filename* is the absolute path to the group file.

Each line of the group file contains a groupname followed by a colon, followed by the member usernames separated by spaces. Example:

```
mygroup:  bob joe anne
```

Note that searching large groups files is *very* inefficient; AuthDBMGroupFile should be used instead.

Security: make sure that the AuthGroupFile is stored outside the document tree of the webserver; do *not* put it in the directory that it protects. Otherwise, clients will be able to download the AuthGroupFile.

See also AuthName, AuthType and AuthUserFile.

4.4.2 AuthUserFile

Syntax: AuthUserFile *filename*
Context: directory, .htaccess
Override: AuthConfig
Status: Base
Module: mod_auth

The AuthUserFile directive sets the name of a textual file containing the list of users and passwords for user authentication. *Filename* is the absolute path to the user file.

Each line of the user file file contains a username followed by a colon, followed by the crypt() encrypted password. The behaviour of multiple occurrences of the same user is undefined.

Note that searching user groups files is inefficient; AuthDBMUserFile should be used instead.

Security: make sure that the AuthUserFile is stored outside the document tree of the webserver; do *not* put it in the directory that it protects. Otherwise, clients will be able to download the AuthUserFile.

See also AuthName, AuthType and AuthGroupFile.

4.5 Module mod_cgi

This module is contained in the `mod_cgi.c` file, and is compiled in by default. It provides for execution of CGI scripts. Any file with mime type `application/x-httpd-cgi` will be processed by this module.

4.5.1 Summary

Any file that has the mime type `application/x-httpd-cgi` will be treated as a CGI script, and run by the server, with its output being returned to the client. Files acquire this type either by having a name ending in an extension defined by the AddType directive, or by being in a ScriptAlias directory.

When the server invokes a CGI script, it will add a variable called `DOCUMENT_ROOT` to the environment. This variable will contain the value of the DocumentRoot configuration variable.

4.5.2 CGI Environment variables

The server will set the CGI environment variables as described in the CGI specification, with the following provisos:

REMOTE_HOST
This will only be set if the server has not been compiled with `MINIMAL_DNS`.

REMOTE_IDENT
This will only be set if IdentityCheck is set to `on`.

REMOTE_USER
This will only be set if the CGI script is subject to authentication.

4.6 Module mod_dir

This module is contained in the `mod_dir.c` file, and is compiled in by default. It provides for directory indexing.

4.6.1 Summary

This module controls the directory indexing. The index of a directory can come from one of two sources:

- A file written by the user, typically called `index.html`. The DirectoryIndex directive sets the name of this file.
- Otherwise, a listing generated by the server. The other directives control the format of this listing. The AddIcon, AddIconByEncoding and AddIconByType are used to set a list of icons to display for various file types; for each file listed, the first icon listed that matches the file is displayed.

4.6.2 AddDescription

Syntax: AddDescription *string file file...*
Context: server config, virtual host, directory, .htaccess
Override: Indexes
Status: Base
Module: mod_dir

This sets the description to display for a file, for FancyIndexing. *File* is a file extension, partial filename, wildcard expression or full filename for files to describe. *String* is enclosed in double quotes (`"`). Example:

```
AddDescription "The planet Mars" /web/pics/mars.gif
```

4.6.3 AddIcon

Syntax: AddIcon *icon name name ...*
Context: server config, virtual host, directory, .htaccess
Override: Indexes
Status: Base
Module: mod_dir

This sets the icon to display next to a file ending in *name* for FancyIndexing. *Icon* is either a (%−escaped) relative URL to the icon, or of the format (*alttext*,*url*) where *alttext* is the text tag given for an icon for non−graphical browsers.

Name is either ^^DIRECTORY^^ for directories, ^^BLANKICON^^ for blank lines (to format the list correctly), a file extension, a wildcard expression, a partial filename or a complete filename. Examples:

```
AddIcon (IMG,/icons/image.xbm) .gif .jpg .xbm
AddIcon /icons/dir.xbm ^^DIRECTORY^^
AddIcon /icons/backup.xbm *~
```

AddIconByType should be used in preference to AddIcon, when possible.

4.6.4 AddIconByEncoding

Syntax: AddIconByEncoding *icon mime-encoding mime-encoding ...*
Context: server config, virtual host, directory, .htaccess
Override: Indexes
Status: Base
Module: mod_dir

This sets the icon to display next to files with *mime-encoding* for FancyIndexing. *Icon* is either a (%−escaped) relative URL to the icon, or of the format (*alttext*,*url*) where *alttext* is the text tag given for an icon for non−graphical browsers.

Mime-encoding is a wildcard expression matching required the content−encoding. Examples:

```
AddIconByEncoding /icons/compress.xbm x-compress
```

4.6.5 AddIconByType

Syntax: AddIconByType *icon mime-type mime-type ...*
Context: server config, virtual host, directory, .htaccess
Override: Indexes
Status: Base
Module: mod_dir

This sets the icon to display next to files of type *mime-type* for FancyIndexing. *Icon* is either a (%−escaped) relative URL to the icon, or of the format (*alttext*,*url*) where *alttext* is the text tag given for an icon for non−graphical browsers.

Mime-type is a wildcard expression matching required the mime types. Examples:

```
AddIconByType (IMG,/icons/image.xbm) image/*
```

4.6.6 DefaultIcon

Syntax: DefaultIcon *url*
Context: server config, virtual host, directory, .htaccess
Override: Indexes
Status: Base
Module: mod_dir

The DefaultIcon directive sets the icon to display for files when no specific icon is known, for FancyIndexing. *Url* is a (%–escaped) relative URL to the icon. Examples:

```
DefaultIcon /icon/unknown.xbm
```

4.6.7 DirectoryIndex

Syntax: DirectoryIndex *local-url local-url ...*
Default: `DirectoryIndex index.html`
Context: server config, virtual host, directory, .htaccess
Override: Indexes
Status: Base
Module: mod_dir

The DirectoryIndex directive sets the list of resources to look for, when the client requests an index of the directory by specifying a NULL file at the end of the a directory name. *Local-url* is the (%–encoded) URL of a document on the server relative to the requested directory; it is usually the name of a file in the directory. Several URLs may be given; the server will return the first one that it finds. If none of the resources exist, then the server will generate its own listing of the directory. Example:

```
DirectoryIndex index.html
```

then a request for `http://myserver/docs/` would return `http://myserver/docs/index.html` if it exists, or would list the directory if it did not.

Note that the documents do not need to be relative to the directory;

```
DirectoryIndex index.html index.txt /cgi-bin/index.pl
```

would cause the CGI script `/cgi-bin/index.pl` to be executed if neither `index.html` or `index.txt` existed in a directory.

4.6.8 FancyIndexing

Syntax: FancyIndexing *boolean*
Context: server config, virtual host, directory, .htaccess
Override: Indexes

Status: Base
Module: mod_dir

The FancyIndexing directive sets the FancyIndexing option for a directory. *Boolean* can be `on` or `off`. The IndexOptions directive should be used in preference.

4.6.9 HeaderName

Syntax: HeaderName *filename*
Context: server config, virtual host, directory, .htaccess
Override: Indexes
Status: Base
Module: mod_dir

The HeaderName directive sets the name of the file that will be inserted at the top of the index listing. *Filename* is the name of the file to include, and is taken to be relative to the directory being indexed. The server first attempts to include *filename*`.html` as an HTML document, otherwise it will include *filename* as plain text. Example:

```
HeaderName HEADER
```

when indexing the directory `/web`, the server will first look for the HTML file `/web/HEADER.html` and include it if found, otherwise it will include the plain text file `/web/HEADER`, if it exists.

See also ReadmeName.

4.6.10 IndexIgnore

Syntax: IndexIgnore *file file ...*
Context: server config, virtual host, directory, .htaccess
Override: Indexes
Status: Base
Module: mod_dir

The IndexIgnore directive adds to the list of files to hide when listing a directory. *File* is a file extension, partial filename, wildcard expression or full filename for files to ignore. Multiple IndexIgnore directives add to the list, rather than the replacing the list of ignored files. By default, the list contains '`.`'. Example:

```
IndexIgnore README .htaccess *~
```

4.6.11 IndexOptions

Syntax: IndexOptions *option option ...*
Context: server config, virtual host, directory, .htaccess
Override: Indexes

Status: Base
Module: mod_dir

The IndexOptions directive specifies the behaviour of the directory indexing. *Option* can be one of

FancyIndexing
This turns on fancy indexing of directories.

IconsAreLinks
This makes the icons part of the anchor for the filename, for fancy indexing.

ScanHTMLTitles
This enables the extraction of the title from HTML documents for fancy indexing. If the file does not have a description given by AddDescription then httpd will read the document for the value of the TITLE tag. This is CPU and disk intensive.

SuppressLastModified
This will suppress the display of the last modification date, in fancy indexing listings.

SuppressSize
This will suppress the file size in fancy indexing listings.

SuppressDescription
This will suppress the file description in fancy indexing listings.

This default is that no options are enabled. If multiple IndexOptions could apply to a directory, then the most specific one is taken complete; the options are not merged. For example:

```
<Directory /web/docs>
IndexOptions FancyIndexing
</Directory>
<Directory /web/docs/spec>
IndexOptions ScanHTMLTitles
</Directory>
```

then only `ScanHTMLTitles` will be set for the /web/docs/spec directory.

4.6.12 ReadmeName

Syntax: ReadmeName *filename*
Context: server config, virtual host, directory, .htaccess
Override: Indexes
Status: Base
Module: mod_dir

The ReadmeName directive sets the name of the file that will be appended to the end of the index listing. *Filename* is the name of the file to include, and is taken to be relative to the directory being indexed. The server first attempts to include *filename*`.html` as an HTML document, otherwise it will include *filename* as plain text. Example:

```
ReadmeName README
```

when indexing the directory /web, the server will first look for the HTML file /web/README.html and include it if found, otherwise it will include the plain text file /web/README, if it exists.

See also HeaderName.

4.7 Module mod_imap

This module is contained in the mod_imap.c file, and is compiled in by default. It provides for .map files, replacing the functionality of the imagemap CGI program. Any document with mime type application/x-httpd-imap will be processed by this module.

4.7.1 Summary

In order to use server–parsed imagemap files, you must first compile the module into your server, and add the following line to the server configuration file. This entry indicates that imagemap files will be named with a .map extension.

```
AddType application/x-httpd-imap map
```

4.7.2 New Features

The imagemap module adds some new features that were not possible with previously distributed imagemap programs.

- URL references relative to the Referer: information.
- Default <BASE> assignment through a new map field base_uri.
- No need for imagemap.conf file.
- Point references.

base_uri options:

map
: Provides the default and old behaviour of *map relative* reference.

referer
: Uses the Referer: header information to reference a URL relative to the current document.

http://whateverurl
: Has the effect of setting <BASE> to that URL making all references relative to that <BASE>.

Mapfile Example

```
default http:/lincoln/
base_uri referer
rect ..  0,0 77,27
poly http://www.inetnebr.com/ 78,0 194,27
circle http://www.inetnebr.com/lincoln/feedback/ 195,0 305,27
rect search_index 306,0 419,27
point http://www.zyzzyva.com/ 420,0 549,27
```

Referencing your mapfile

```
<A HREF="http:/maps/imagmap1.map">
<IMG ISMAP SRC="http:/images/imagemap1.gif">
</A>
```

4.8 Module mod_include

This module is contained in the `mod_include.c` file, and is compiled in by default. It provides for server–parsed html documents, known as SPML documents. Any document with mime type `text/x-server-parsed-html` or `text/x-server-parsed-html3` will be parsed by this module, with the resulting output given the mime type `text/html`.

4.8.1 SPML – Include file Format

The document is parsed as an HTML document, with special commands embedded as SGML comments. A command has the syntax:

`<!--#`*element attribute*=*value attribute*=*value* ... `-->`

The value will often be enclosed in double quotes; many commands only allow a single attribute–value pair.

The allowed elements are:

config
This command controls various aspects of the parsing. The valid attributes are:

errmsg
The value is a message that is sent back to the client if an error occurs whilst parsing the document.

sizefmt
The value sets the format to be used which displaying the size of a file. Valid values are `bytes` for a count in bytes, or `abbrev` for a count in Kb or Mb as appropriate.

timefmt
The value is a string to be used by the `strftime(3)` library routine when printing dates.

echo

This command prints one of the include variables, defined below. If the variable is unset, it is printed as `(none)`. Any dates printed are subject to the currently configured `timefmt`. Attributes:

var

The value is the name of the variable to print.

exec

The exec command executes a given shell command or CGI script. The IncludesNOEXEC Option disables this command completely. The valid attributes are:

cgi

The value specifies a (%–encoded) URL relative path to the CGI script. If the path does not begin with a (/), then it is taken to be relative to the current document. The document referenced by this path is invoked as a CGI script, even if the server would not normally recognise it as such. However, the directory containing the script must be enabled for CGI scripts (with ScriptAlias or the ExecCGI Option).

The CGI script is given the PATH_INFO and query string (QUERY_STRING) of the original request from the client; these cannot be specified in the URL path. The include variables will be available to the script in addition to the standard CGI environment.

If the script returns a Location: header instead of output, then this will be translated into an HTML anchor.

The `include virtual` element should be used in preference to `exec cgi`.

cmd

The server will execute the given string using `/bin/sh`. The include variables are available to the command.

fsize

This command prints the size of the specified file, subject to the `sizefmt` format specification. Attributes:

file

The value is a path relative to the directory containing the current document being parsed.

virtual

The value is a (%–encoded) URL–path relative to the current document being parsed. If it does not begin with a slash (/) then it is taken to be relative to the current document.

flastmod

This command prints the last modification date of the specified file, subject to the `timefmt` format specification. The attributes are the same as for the `fsize` command.

include

This command inserts the text of another document or file into the parsed file. Any included file is subject to the usual access control. If the directory containing the parsed file has the Option IncludesNOEXEC set, and the including the document would cause a program to be executed, then it will not be included; this prevents the execution of CGI scripts. Otherwise CGI scripts are invoked as normal using the complete URL given in the command, including any query string.

An attribute defines the location of the document; the inclusion is done for each attribute given to the include command. The valid attributes are:

file

The value is a path relative to the directory containing the current document being parsed. It cannot contain `../`, nor can it be an absolute path. The `virtual` attribute should always be used in preference to this one.

virtual
The value is a (%–encoded) URL relative to the current document being parsed. The URL cannot contain a scheme or hostname, only a path and an optional query string. If it does not begin with a slash (/) then it is taken to be relative to the current document.

A URL is constructed from the attribute, and the output the server would return if the URL were accessed by the client is included in the parsed output. Thus included files can be nested.

4.8.2 Include variables

These are available for the `echo` command, and to any program invoked by the document.

DATE_GMT
The current date in Greenwich Mean Time.

DATE_LOCAL
The current date in the local time zone.

DOCUMENT_NAME
The filename (excluding directories) of the document requested by the user.

DOCUMENT_URI
The (%–decoded) URL path of the document requested by the user. Note that in the case of nested include files, this is *not* then URL for the current document.

LAST_MODIFIED
The last modification date of the document requested by the user.

4.8.3 XBitHack

Syntax: XBitHack *status*
Default: `XBitHack off`
Context: server config, virtual host, directory, .htaccess
Override: Options
Status: Base
Module: mod_include

The XBitHack directives controls the parsing of ordinary html documents. *Status* can have the following values:

off
No special treatment of executable files.

on
Any file that has the user–execute bit set will be treated as a server–parsed html document.

full
As for `on` but also test the group–execute bit. If it is set, then set the Last–modified date of the returned file to be the last modified time of the file. If it is not set, then no last–modified date is sent. Setting this bit allows clients and proxies to cache the result of the request.

4.9 Module mod_log_common

This module is contained in the `mod_log_common.c` file, and is compiled in by default. It provides for logging of the requests made to the server using the Common Logfile Format.

4.9.1 Log file format

The log file contains a separate line for each request. A line is composed of several tokens separated by spaces:

> host ident authuser date request status bytes

If a token does not have a value then it is represented by a hyphen (-). The meanings and values of these tokens are as follows:

host
: The fully-qualified domain name of the client, or its IP number if the name is not available.

ident
: If IdentityCheck is enabled and the client machine runs identd, then this is the identity information reported by the client.

authuser
: If the request was for an password protected document, then this is the userid used in the request.

date
: The date and time of the request, in the following format:

```
date = [day/month/year:hour:minute:second zone]
day = 2*digit
month = 3*letter
year = 4*digit
hour = 2*digit
minute = 2*digit
second = 2*digit
zone = (`+' | `-') 4*digit
```

request
: The request line from the client, enclosed in double quotes (").

status
: The three digit status code returned to the client.

bytes
: The number of bytes in the object returned to the client, not including any headers.

4.9.2 TransferLog

Syntax: TransferLog *file-pipe*
Default: `TransferLog logs/transfer_log`

Context: server config, virtual host
Status: Base
Module: mod_log_common

The TransferLog directive sets the name of the file to which the server will log the incoming requests. *File-pipe* is one of

A filename
A filename relative to the ServerRoot.

'|' followed by a command
A program to receive the agent log information on its standard input. Note the a new program will not be started for a VirtualHost if it inherits the TransferLog from the main server.

Security: if a program is used, then it will be run under the user who started httpd. This will be root if the server was started by root; be sure that the program is secure.

4.10 Module mod_mime

This module is contained in the `mod_mime.c` file, and is compiled in by default. It provides for determining the types of files from the filename.

4.10.1 Summary

This module is used to determine the mime types of documents. Some mime types indicate special processing to be performed by the server, otherwise the type is returned to the client so that the browser can deal with the document appropriately.

The filename of a document is treated as being composed of a basename followed by some extensions, in the following order:

base.type.language.enc

The *type* extension sets the type of the document; types are defined in the TypesConfig file and by the AddType directive. The *language* extension sets the language of the document, as defined by the AddLanguage directive. Finally, the *enc* directive sets the encoding of the document, as defined by the AddEncoding directive.

4.10.2 AddEncoding

Syntax: AddEncoding *mime-enc extension extension...*
Context: server config, virtual host, directory, .htaccess
Override: FileInfo
Status: Base
Module: mod_mime

The AddEncoding directive adds to the list of filename extensions which filenames may end in for the specified encoding type. *Mime-enc* is the mime encoding to use for documents ending in *extension*. Example:

```
AddEncoding x-gzip gz
AddEncoding x-compress Z
```

This will cause files ending in .gz to be marked as encoded using the x–gzip encoding, and .Z files to be marked as encoded with x–compress.

4.10.3 AddLanguage

Syntax: AddLanguage *mime-lang extension extension...*
Context: server config, virtual host, directory, .htaccess
Override: FileInfo
Status: Base
Module: mod_mime

The AddLanguage directive adds to the list of filename extensions which filenames may end in for the specified content language. *Mime-lang* is the mime language of files with names ending *extension*, after any content encoding extensions have been removed. Example:

```
AddEncoding x-compress Z
AddLanguage en .en
AddLanguage fr .fr
```

Then the document `xxxx.en.Z` will be treated as being a compressed English document. Although the content language is reported to the client, the browser is unlikely to use this information. The AddLanguage directive is more useful for content negotiation, where the server returns one from several documents based on the client's language preference.

4.10.4 AddType

Syntax: AddType *mime-type extension extension...*
Context: server config, virtual host, directory, .htaccess
Override: FileInfo
Status: Base
Module: mod_mime

The AddType directive adds to the list of filename extensions which filenames may end in for the specified content type. *Mime-enc* is the mime type to use for documents ending in *extension*. after content–encoding and language extensions have been removed. Example:

```
AddType image/gif GIF
```

It is recommended that new mime types be added using the AddType directive rather than changing the TypesConfig file.

Note that, unlike the NCSA httpd, this directive cannot be used to set the type of particular files.

4.10.5 TypesConfig

Syntax: TypesConfig *filename*
Default: `TypesConfig conf/mime.types`
Context: server config
Status: Base
Module: mod_mime

The TypesConfig directive sets the location of the mime types configuration file. *Filename* is relative to the ServerRoot. This file sets the default list of mappings from filename extensions to content types; changing this file is not recommended. Use the AddType directive instead. The file contains lines in the format of the arguments to an AddType command:

> *mime-type extension extension ...*

The extensions are lower–cased. Blank lines, and lines beginning with a hash character ('#') are ignored.

4.11 Module mod_negotiation

This module is contained in the `mod_negotiation.c` file, and is compiled in by default. It provides for content negotiation. Any document with mime type `application/x-type-map` will be processed by this module.

4.11.1 Summary

Content negotiation, or more accurately content selection, is the selection of the document that best matches the clients capabilities, from one of several available documents. There are two implementations of this.

- A type map (a file with the mime type `application/x-type-map`) which explicitly lists the files containing the variants.
- A MultiViews search (enabled by the MultiViews Option, where the server does an implicit filename pattern match, and choose from amongst the results.

Type maps

A type map has the same format as RFC822 mail headers. It contains document descriptions separated by blank lines, with lines beginning with a hash character ('#') treated as comments. A document description consists of several header records; records may be continued on multiple lines if the continuation lines start with spaces. The leading space will be deleted and the lines concatenated. A header record consists of a keyword name, which always ends in a colon, followed by a value. Whitespace is allowed between the header name and value, and between the tokens of value. The headers allowed are:

Content-Encoding:
: The encoding of the file. Currently only two encodings are recognised by http; `x-compress` for compressed files, and `x-gzip` for gzipped files.

Content-Language:
The language of the variant, as an Internet standard language code, such as `en`.

Content-Length:
The length of the file, in bytes. If this header is not present, then the actual length of the file is used.

Content-Type:
The MIME media type of the document, with optional parameters. parameters are separated from the media type and from one another by semi–colons. Parameter syntax is name=value; allowed parameters are:

level
the value is an integer, which specifies the version of the media type. For `text/html` this defaults to 2, otherwise 0.

qs
the value is a floating–point number with value between 0. and 1. It indications the 'quality' of this variant.

Example:

```
Content-Type:  image/jpeg; qs=0.8
```

URI:
The path to the file containing this variant, relative to the map file.

MultiViews

A MultiViews search is enabled by the MultiViews Option. If the server receives a request for `/some/dir/foo` and `/some/dir/foo` does *not* exist, then the server reads the directory looking for all files named `foo.*`, and effectively fakes up a type map which names all those files, assigning them the same media types and content–encodings it would have if the client had asked for one of them by name. It then chooses the best match to the client's requirements, and returns that document.

4.11.2 LanguagePriority

Syntax: LanguagePriority *mime-lang mime-lang...*
Context: server config, virtual host, directory, .htaccess
Override: FileInfo
Status: Base
Module: mod_mime

The LanguagePriority sets the precedence of language variants for the case where the client does not express a preference, when handling a MultiViews request. The list of *mime-lang* are in order of decreasing preference. Example:

```
LanguagePriority en fr de
```

For a request for `foo.html`, where `foo.html.fr` and `foo.html.de` both existed, but the browser did not express a language preference, then `foo.html.fr` would be returned.

4.12 Module mod_userdir

This module is contained in the `mod_userdir.c` file, and is compiled in by default. It provides for user–specific directories.

4.12.1 UserDir

Syntax: UserDir *directory*
Default: `UserDir public_html`
Context: server config, virtual host
Status: Base
Module: mod_userdir

The UserDir directive sets the real directory in a user's home directory to use when a request for a document for a user is received. *Directory* is either `disabled`, to disable this feature, or the name of a directory. If not disabled, then a request for a URL beginning with `http://myserver/~`*unix-username* will be translated to a filename beginning with *home-dir/directory*, where *home-dir* is the home directory for user *unix-username*. Example:

```
UserDir public_html
```

Then a request for `http://myserver/~foo56/adir/file.html` will return the file `http://myserver/home/foo56/public_html/adir/file.html`.

Chapter 5

Apache Extension Modules

5.1 Module mod_auth_dbm

This module is contained in the `mod_auth_dbm.c` file, and is not compiled in by default. It provides for user authentication using DBM files. See the DBM user documentation.

5.1.1 AuthDbmGroupFile

Syntax: AuthGroupFile *filename*
Context: directory, .htaccess
Override: AuthConfig
Status: Extension
Module: mod_auth_dbm

The AuthDBMGroupFile directive sets the name of a DBM file containing the list of user groups for user authentication. *Filename* is the absolute path to the group file.

The group file is keyed on the username. The value for a user is a comma–separated list of the groups to which the users belongs. There must be no whitespace within the value, and it must never contain any colons.

Security: make sure that the AuthDBMGroupFile is stored outside the document tree of the webserver; do *not* put it in the directory that it protects. Otherwise, clients will be able to download the AuthDBMGroupFile.

See also AuthName, AuthType and AuthDBMUserFile.

5.1.2 AuthDBMUserFile

Syntax: AuthDBMUserFile *filename*
Context: directory, .htaccess
Override: AuthConfig
Status: Extension

Module: mod_auth_dbm

The AuthDBMUserFile directive sets the name of a DBM file containing the list of users and passwords for user authentication. *Filename* is the absolute path to the user file.

The user file is keyed on the username. The value for a user is the crypt() encrypted password, optionally followed by a colon and arbitrary data. The colon and the data following it will be ignored by the server.

Security: make sure that the AuthDBMUserFile is stored outside the document tree of the webserver; do *not* put it in the directory that it protects. Otherwise, clients will be able to download the AuthDBMUserFile.

See also AuthName, AuthType and AuthDBMGroupFile.

5.2 Module mod_cookies

This module is contained in the `mod_cookies.c` file, and is not compiled in by default. It provides for Netscape(TM) cookies. There is no documentation available for this module.

5.2.1 CookieLog

Syntax: CookieLog *filename*
Context: server config, virtual host
Status: Experimental
Module: mod_cookies

The CookieLog directive sets the filename for logging of cookies. The filename is relative to the ServerRoot.

5.3 Module mod_dld

This module is contained in the `mod_dld.c` file, and is not compiled in by default. It provides for loading of executable code and modules into the server at start–up time, using the GNU dld library.

5.3.1 Summary

The optional dld module is a proof–of–concept piece of code which loads other modules into the server as it is configuring itself (the first time only; for now, rereading the config files cannot affect the state of loaded modules), using the GNU dynamic linking library, DLD. It isn't compiled into the server by default, since not everyone has DLD, but it works when I try it. (Famous last words.)

Note that for some reason, `LoadFile /lib/libc.a` seems to be required for just about everything.

Note: that DLD needs to read the symbol table out of the server binary when starting up; these commands will fail if the server can't find its own binary when it starts up, or if that binary is stripped.

5.3.2 LoadFile

Syntax: LoadFile *filename filename* ...
Context: server config
Status: Experimental
Module: mod_dld

The LoadFile directive links in the named object files or libraries when the server is started; this is used to load additional code which may be required for some module to work. *Filename* is relative to ServerRoot.

5.3.3 LoadModule

Syntax: LoadModule *module filename*
Context: server config
Status: Experimental
Module: mod_dld

The LoadModule directive links in the object file or library *filename* and adds the module structure named *module* to the list of active modules. *Module* is the name of the external variable of type `module` in the file. Example:

```
LoadModule ai_backcompat_module modules/mod_ai_backcompat.o
LoadFile /lib/libc.a
```

loads the module in the modules subdirectory of the ServerRoot.

5.4 Module mod_log_agent

This module is contained in the `mod_log_agent.c` file, and is not compiled in by default. It provides for logging of the client user agents.

5.4.1 AgentLog

Syntax: AgentLog *file-pipe*
Default: `AgentLog logs/agent_log`
Context: server config, virtual host
Status: Extension
Module: mod_log_agent

The AgentLog directive sets the name of the file to which the server will log the UserAgent header of incoming requests. *File-pipe* is one of

A filename
A filename relative to the ServerRoot.

'|' followed by a command
A program to receive the agent log information on its standard input. Note the a new program will not be started for a VirtualHost if it inherits the AgentLog from the main server.

Security: if a program is used, then it will be run under the user who started httpd. This will be root if the server was started by root; be sure that the program is secure.

This directive is provided for compatibility with NCSA 1.4.

5.5 Module mod_log_config

This module is contained in the `mod_log_config.c` file, and is not compiled in by default. It provides for logging of the requests made to the server, using a user–specified format.

5.5.1 Summary

This is an EXPERIMENTAL module, which implements the TransferLog directive (same as the common log module), and an additional directive, LogFormat. Bugs would not surprise me.

The argument to the LogFormat is a string, which can include literal characters copied into the log files, and '%' directives as follows:

```
%...h:           Remote host
%...l:           Remote logname (from identd, if supplied)
%...u:           Remote user (from auth; may be bogus if return
                 status (%s) is 401)
%...t:           Time, in common log format time format
%...r:           First line of request
%...s:           Status.  For requests that got internally redirected,
                 this is status of the {\bf original} request --- %...>s
                 for the last.
%...b:           Bytes sent.
%...{Foobar}i:  The contents of Foobar: header line(s) in the request
                 sent to the client.
%...{Foobar}o:  The contents of Foobar: header line(s) in the reply.
```

The '...' can be nothing at all (e.g. `"%h %u %r %s %b"`), or it can indicate conditions for inclusion of the item (which will cause it to be replaced with '–' if the condition is not met). Note that there is no escaping performed on the strings from %r, %...i and %...o; some with long memories may remember that I thought this was a bad idea, once upon a time, and I'm still not comfortable with it, but it is difficult to see how to 'do the right thing' with all of '%..i', unless we URL–escape everything and break with CLF.

The forms of condition are a list of HTTP status codes, which may or may not be preceded by '!'. Thus, '%400,501{User–agent}i' logs User–agent: on 400 errors and 501 errors (Bad Request, Not Implemented) only; '%!200,304,302{Referer}i' logs Referer: on all requests which did **not** return some sort of normal status.

The default LogFormat reproduces CLF; see below.

The way this is supposed to work with virtual hosts is as follows: a virtual host can have its own LogFormat, or its own TransferLog. If it doesn't have its own LogFormat, it inherits from the main server. If it doesn't have its own TransferLog, it writes to the same descriptor (meaning the same process for '| ...').

That means that you can do things like:

```
<VirtualHost hosta.com>
LogFormat "hosta ..."
...
</VirtualHost>

<VirtualHost hosta.com>
LogFormat "hostb ..."
...
</VirtualHost>
```

... to have different virtual servers write into the same log file, but have some indication which host they came from, though a %v directive may well be a better way to handle this. Look for more changes to come to this format.

5.5.2 LogFormat

Syntax: LogFormat *string*
Default: `LogFormat "%h %l %u %t \"%r\" %s %b"`
Context: server config, virtual host
Status: Experimental
Module: mod_log_config

This sets the format of the logfile.

5.5.3 TransferLog

Syntax: TransferLog *file-pipe*
Default: `TransferLog logs/transfer_log`
Context: server config, virtual host
Status: Experimental
Module: mod_log_config

The TransferLog directive sets the name of the file to which the server will log the incoming requests. *File-pipe* is one of

A filename
A filename relative to the ServerRoot.

'|' followed by a command
A program to receive the agent log information on its standard input. Note the a new program will not be started for a VirtualHost if it inherits the TransferLog from the main server.

Security: if a program is used, then it will be run under the user who started httpd. This will be root if the server was started by root; be sure that the program is secure.

5.6 Module mod_log_referer

This module is contained in the `mod_log_referer.c` file, and is not compiled in by default. It provides for logging of the documents which reference documents on the server.

5.6.1 Log file format

The log file contains a separate line for each refer. Each line has the format

uri -> *document*

where *uri* is the (%–escaped) URI for the document that references the one requested by the client, and *document* is the (%–decoded) local URL to the document being referred to.

5.6.2 RefererIgnore

Syntax: RefererIgnore *string string ...*
Context: server config, virtual host
Status: Extension
Module: mod_log_referer

The RefererIgnore directive adds to the list of strings to ignore in Referer headers. If any of the strings in the list is contained in the Referer header, then no referrer information will be logged for the request. Example:

```
RefererIgnore www.ncsa.uiuc.edu
```

This avoids logging references from www.ncsa.uiuc.edu.

5.6.3 RefererLog

Syntax: RefererLog *file-pipe*
Default: `RefererLog logs/referer_log`
Context: server config, virtual host
Status: Extension
Module: mod_log_referer

The RefererLog directive sets the name of the file to which the server will log the Referer header of incoming requests. *File-pipe* is one of

A filename
A filename relative to the ServerRoot.

'|' followed by a command
A program to receive the referrer log information on its standard input. Note the a new program will not be started for a VirtualHost if it inherits the RefererLog from the main server.

Security: if a program is used, then it will be run under the user who started httpd. This will be root if the server was started by root; be sure that the program is secure.

This directive is provided for compatibility with NCSA 1.4.

Chapter 6

Apache API notes

These are some notes on the Apache API and the data structures you have to deal with, etc. They are not yet nearly complete, but hopefully, they will help you get your bearings. Keep in mind that the API is still subject to change as we gain experience with it. (See the TODO file for what *might* be coming). However, it will be easy to adapt modules to any changes that are made. (We have more modules to adapt than you do).

A few notes on general pedagogical style here. In the interest of conciseness, all structure declarations here are incomplete ––– the real ones have more slots that I'm not telling you about. For the most part, these are reserved to one component of the server core or another, and should be altered by modules with caution. However, in some cases, they really are things I just haven't gotten around to yet. Welcome to the bleeding edge.

Finally, here's an outline, to give you some bare idea of what's coming up, and in what order:

- Basic concepts.
  - Handlers, Modules, and Requests
  - A brief tour of a module
- How handlers work
  - A brief tour of the `request_rec`
  - Where request_rec structures come from
  - Handling requests, declining, and returning error codes
  - Special considerations for response handlers
  - Special considerations for authentication handlers
  - Special considerations for logging handlers
- Resource allocation and resource pools
- Configuration, commands and the like
  - Per–directory configuration structures
  - Command handling
  - Side notes ––– per–server configuration, virtual servers, etc.

6.1 Basic concepts.

We begin with an overview of the basic concepts behind the API, and how they are manifested in the code.

6.1.1 Handlers, Modules, and Requests

Apache breaks down request handling into a series of steps, more or less the same way the Netscape server API does (although this API has a few more stages than NetSite does, as hooks for stuff I thought might be useful in the future). These are:

- URI –> Filename translation
- Auth ID checking [is the user who they say they are?]
- Auth access checking [is the user authorized *here*?]
- Access checking other than auth
- Determining MIME type of the object requested
- 'Fixups' ––– there aren't any of these yet, but the phase is intended as a hook for possible extensions like `SetEnv`, which don't really fit well elsewhere.
- Actually sending a response back to the client.
- Logging the request

These phases are handled by looking at each of a succession of *modules*, looking to see if each of them has a handler for the phase, and attempting invoking it if so. The handler can typically do one of three things:

- *Handle* the request, and indicate that it has done so by returning the magic constant `OK`.
- *Decline* to handle the request, by returning the magic integer constant `DECLINED`. In this case, the server behaves in all respects as if the handler simply hadn't been there.
- Signal an error, by returning one of the HTTP error codes. This terminates normal handling of the request, although an ErrorDocument may be invoked to try to mop up, and it will be logged in any case.

Most phases are terminated by the first module that handles them; however, for logging, 'fixups', and non–access authentication checking, all handlers always run (barring an error). Also, the response phase is unique in that modules may declare multiple handlers for it, via a dispatch table keyed on the MIME type of the requested object. Modules may declare a response–phase handler which can handle *any* request, by giving it the key `*/*` (i.e., a wildcard MIME type specification). However, wildcard handlers are only invoked if the server has already tried and failed to find a more specific response handler for the MIME type of the requested object (either none existed, or they all declined).

The handlers themselves are functions of one argument (a `request_rec` structure. vide infra), which returns an integer, as above.

6.1.2 A brief tour of a module

At this point, we need to explain the structure of a module. Our candidate will be one of the messier ones, the CGI module —– this handles both CGI scripts and the `ScriptAlias` config file command. It's actually a great deal more complicated than most modules, but if we're going to have only one example, it might as well be the one with its fingers in every place.

Let's begin with handlers. In order to handle the CGI scripts, the module declares a response handler for them. Because of `ScriptAlias`, it also has handlers for the name translation phase (to recognise `ScriptAlias`ed URIs), the type–checking phase (any `ScriptAlias`ed request is typed as a CGI script).

The module needs to maintain some per (virtual) server information, namely, the `ScriptAlias`es in effect; the module structure therefore contains pointers to a functions which builds these structures, and to another which combines two of them (in case the main server and a virtual server both have `ScriptAliases` declared).

Finally, this module contains code to handle the `ScriptAlias` command itself. This particular module only declares one command, but there could be more, so modules have *command tables* which declare their commands, and describe where they are permitted, and how they are to be invoked.

A final note on the declared types of the arguments of some of these commands: a `pool` is a pointer to a *resource pool* structure; these are used by the server to keep track of the memory which has been allocated, files opened, etc., either to service a particular request, or to handle the process of configuring itself. That way, when the request is over (or, for the configuration pool, when the server is restarting), the memory can be freed, and the files closed, *en masse*, without anyone having to write explicit code to track them all down and dispose of them. Also, a `cmd_parms` structure contains various information about the config file being read, and other status information, which is sometimes of use to the function which processes a config–file command (such as `ScriptAlias`). With no further ado, the module itself:

```
/* Declarations of handlers. */

int translate_scriptalias (request_rec *);
int type_scriptalias (request_rec *);
int cgi_handler (request_rec *);

/* Subsidiary dispatch table for response-phase handlers, by MIME type */

handler_rec cgi_handlers[] = {
{ "application/x-httpd-cgi", cgi_handler },
{ NULL }
};

/* Declarations of routines to manipulate the module's configuration
 * info.  Note that these are returned, and passed in, as void *'s;
 * the server core keeps track of them, but it doesn't, and can't,
 * know their internal structure.
 */

void *make_cgi_server_config (pool *);
void *merge_cgi_server_config (pool *, void *, void *);
```

```
/* Declarations of routines to handle config-file commands */

extern char *script_alias(cmd_parms *, void *per_dir_config, char *fake,
                          char *real);

command_rec cgi_cmds[] = {
{ "ScriptAlias", script_alias, NULL, RSRC_CONF, TAKE2,
    "a fakename and a realname"},
{ NULL }
};

module cgi_module = {
   STANDARD_MODULE_STUFF,
   NULL,                     /* initializer */
   NULL,                     /* dir config creator */
   NULL,                     /* dir merger --- default is to override */
   make_cgi_server_config,   /* server config */
   merge_cgi_server_config,  /* merge server config */
   cgi_cmds,                 /* command table */
   cgi_handlers,             /* handlers */
   translate_scriptalias,    /* filename translation */
   NULL,                     /* check_user_id */
   NULL,                     /* check auth */
   NULL,                     /* check access */
   type_scriptalias,         /* type_checker */
   NULL,                     /* fixups */
   NULL                      /* logger */
};
```

6.2 How handlers work

The sole argument to handlers is a `request_rec` structure. This structure describes a particular request which has been made to the server, on behalf of a client. In most cases, each connection to the client generates only one `request_rec` structure.

6.2.1 A brief tour of the `request_rec`

The `request_rec` contains pointers to a resource pool which will be cleared when the server is finished handling the request; to structures containing per–server and per–connection information, and most importantly, information on the request itself.

The most important such information is a small set of character strings describing attributes of the object being requested, including its URI, filename, content–type and content–encoding (these being filled in by the translation and type–check handlers which handle the request, respectively).

Other commonly used data items are tables giving the MIME headers on the client's original request, MIME headers to be sent back with the response (which modules can add to at will), and environment variables for any subprocesses which are spawned off in the course of servicing the request. These tables are manipulated using the `table_get` and `table_set` routines.

Finally, there are pointers to two data structures which, in turn, point to per–module configuration structures. Specifically, these hold pointers to the data structures which the module has built to describe the way it has been configured to operate in a given directory (via `.htaccess` files or `<Directory>` sections), for private data it has built in the course of servicing the request (so modules' handlers for one phase can pass 'notes' to their handlers for other phases). There is another such configuration vector in the `server_rec` data structure pointed to by the `request_rec`, which contains per (virtual) server configuration data.

Here is an abridged declaration, giving the fields most commonly used:

```
struct request_rec {

  pool *pool;
  conn_rec *connection;
  server_rec *server;

  /* What object is being requested */

  char *uri;
  char *filename;
  char *path_info;
  char *args;           /* QUERY_ARGS, if any */
  struct stat finfo;    /* Set by server core;
                         * st_mode set to zero if no such file */

  char *content_type;
  char *content_encoding;

  /* MIME header environments, in and out.  Also, an array containing
   * environment variables to be passed to subprocesses, so people can
   * write modules to add to that environment.
   *
   * The difference between headers_out and err_headers_out is that
   * the latter are printed even on error, and persist across internal
   * redirects (so the headers printed for ErrorDocument handlers will
   * have them).
   */

  table *headers_in;
  table *headers_out;
  table *err_headers_out;
  table *subprocess_env;

  /* Info about the request itself... */

  int header_only;     /* HEAD request, as opposed to GET */
  char *protocol;      /* Protocol, as given to us, or HTTP/0.9 */
  char *method;        /* GET, HEAD, POST, etc. */
  int method_number;   /* M_GET, M_POST, etc. */
```

```
  /* Info for logging */

  char *the_request;
  int bytes_sent;

  /* A flag which modules can set, to indicate that the data being
   * returned is volatile, and clients should be told not to cache it.
   */

  int no_cache;

  /* Various other config info which may change with .htaccess files
   * These are config vectors, with one void* pointer for each module
   * (the thing pointed to being the module's business).
   */

  void *per_dir_config;   /* Options set in config files, etc. */
  void *request_config;   /* Notes on *this* request */

};
```

6.2.2 Where request_rec structures come from

Most `request_rec` structures are built by reading an HTTP request from a client, and filling in the fields. However, there are a few exceptions:

- If the request is to an imagemap, a type map (i.e., a `*.var` file), or a CGI script which returned a local 'Location:', then the resource which the user requested is going to be ultimately located by some URI other than what the client originally supplied. In this case, the server does an *internal redirect*, constructing a new `request_rec` for the new URI, and processing it almost exactly as if the client had requested the new URI directly.
- If some handler signaled an error, and an `ErrorDocument` is in scope, the same internal redirect machinery comes into play.
- Finally, a handler occasionally needs to investigate 'what would happen if' some other request were run. For instance, the directory indexing module needs to know what MIME type would be assigned to a request for each directory entry, in order to figure out what icon to use.

  Such handlers can construct a *sub-request*, using the functions `sub_req_lookup_file` and `sub_req_lookup_uri`; this constructs a new `request_rec` structure and processes it as you would expect, up to but not including the point of actually sending a response. (These functions skip over the access checks if the sub–request is for a file in the same directory as the original request).

  (Server–side includes work by building sub–requests and then actually invoking the response handler for them, via the function `run_sub_request`).

6.2.3 Handling requests, declining, and returning error codes

As discussed above, each handler, when invoked to handle a particular `request_rec`, has to return an `int` to indicate what happened. That can either be

- OK — the request was handled successfully. This may or may not terminate the phase.
- DECLINED — no erroneous condition exists, but the module declines to handle the phase; the server tries to find another.
- an HTTP error code, which aborts handling of the request.

Note that if the error code returned is `REDIRECT`, then the module should put a `Location` in the request's `headers_out`, to indicate where the client should be redirected *to*.

6.2.4 Special considerations for response handlers

Handlers for most phases do their work by simply setting a few fields in the `request_rec` structure (or, in the case of access checkers, simply by returning the correct error code). However, response handlers have to actually send a request back to the client.

They should begin by sending an HTTP response header, using the function `send_http_header`. (You don't have to do anything special to skip sending the header for HTTP/0.9 requests; the function figures out on its own that it shouldn't do anything). If the request is marked `header_only`, that's all they should do; they should return after that, without attempting any further output.

Otherwise, they should produce a request body which responds to the client as appropriate. The primitives for this are `rputc` and `rprintf`, for internally generated output, and `send_fd`, to copy the contents of some `FILE *` straight to the client.

At this point, you should more or less understand the following piece of code, which is the handler which handles `GET` requests which have no more specific handler; it also shows how conditional `GET`s can be handled, if it's desirable to do so in a particular response handler — `set_last_modified` checks against the `If-modified-since` value supplied by the client, if any, and returns an appropriate code (which will, if nonzero, be USE_LOCAL_COPY). No similar considerations apply for `set_content_length`, but it returns an error code for symmetry.

```
int default_handler (request_rec *r)
{
    int errstatus;
    FILE *f;

    if (r->method_number != M_GET) return DECLINED;
    if (r->finfo.st_mode == 0) return NOT_FOUND;

    if ((errstatus = set_content_length (r, r->finfo.st_size))
        || (errstatus = set_last_modified (r, r->finfo.st_mtime)))
        return errstatus;

    f = fopen (r->filename, "r");

    if (f == NULL) {
        log_reason("file permissions deny server access",
                   r->filename, r);
        return FORBIDDEN;
```

```
    }

    register_timeout ("send", r);
    send_http_header (r);

    if (!r->header_only) send_fd (f, r);
    pfclose (r->pool, f);
    return OK;
}
```

Finally, if all of this is too much of a challenge, there are a few ways out of it. First off, as shown above, a response handler which has not yet produced any output can simply return an error code, in which case the server will automatically produce an error response. Secondly, it can punt to some other handler by invoking `internal_redirect`, which is how the internal redirection machinery discussed above is invoked. A response handler which has internally redirected should always return `OK`.

(Invoking `internal_redirect` from handlers which are *not* response handlers will lead to serious confusion).

6.2.5 Special considerations for authentication handlers

Stuff that should be discussed here in detail:

- Authentication–phase handlers not invoked unless auth is configured for the directory.
- Common auth configuration stored in the core per–dir configuration; it has accessors `auth_type`, `auth_name`, and `requires`.
- Common routines, to handle the protocol end of things, at least for HTTP basic authentication (`get_basic_auth_pw`, which sets the `connection->user` structure field automatically, and `note_basic_auth_failure`, which arranges for the proper `WWW-Authenticate:` header to be sent back).

6.2.6 Special considerations for logging handlers

When a request has internally redirected, there is the question of what to log. Apache handles this by bundling the entire chain of redirects into a list of `request_rec` structures which are threaded through the `r->prev` and `r->next` pointers. The `request_rec` which is passed to the logging handlers in such cases is the one which was originally built for the initial request from the client; note that the bytes_sent field will only be correct in the last request in the chain (the one for which a response was actually sent).

6.3 Resource allocation and resource pools

One of the problems of writing and designing a server–pool server is that of preventing leakage, that is, allocating resources (memory, open files, etc.), without subsequently releasing them. The resource pool machinery is designed to make it easy to prevent this from happening, by allowing resource to be allocated in such a way that they are *automatically* released when the server is done with them.

The way this works is as follows: the memory which is allocated, file opened, etc., to deal with a particular request are tied to a *resource pool* which is allocated for the request. The pool is a data structure which itself tracks the resources in question.

When the request has been processed, the pool is *cleared*. At that point, all the memory associated with it is released for reuse, all files associated with it are closed, and any other clean–up functions which are associated with the pool are run. When this is over, we can be confident that all the resource tied to the pool have been released, and that none of them have leaked.

Server restarts, and allocation of memory and resources for per–server configuration, are handled in a similar way. There is a *configuration pool*, which keeps track of resources which were allocated while reading the server configuration files, and handling the commands therein (for instance, the memory that was allocated for per–server module configuration, log files and other files that were opened, and so forth). When the server restarts, and has to reread the configuration files, the configuration pool is cleared, and so the memory and file descriptors which were taken up by reading them the last time are made available for reuse.

It should be noted that use of the pool machinery isn't generally obligatory, except for situations like logging handlers, where you really need to register cleanups to make sure that the log file gets closed when the server restarts (this is most easily done by using the function `pfopen`, which also arranges for the underlying file descriptor to be closed before any child processes, such as for CGI scripts, are `exec`ed), or in case you are using the timeout machinery (which isn't yet even documented here). However, there are two benefits to using it: resources allocated to a pool never leak (even if you allocate a scratch string, and just forget about it); also, for memory allocation, `palloc` is generally faster than `malloc`.

We begin here by describing how memory is allocated to pools, and then discuss how other resources are tracked by the resource pool machinery.

6.3.1 Allocation of memory in pools

Memory is allocated to pools by calling the function `palloc`, which takes two arguments, one being a pointer to a resource pool structure, and the other being the amount of memory to allocate (in `char`s). Within handlers for handling requests, the most common way of getting a resource pool structure is by looking at the `pool` slot of the relevant `request_rec`; hence the repeated appearance of the following idiom in module code:

```
int my_handler(request_rec *r)
{
    struct my_structure *foo;
    ...

    foo = (foo *)palloc (r->pool, sizeof(my_structure));
}
```

Note that *there is no* `pfree` ––– `palloc`ed memory is freed only when the associated resource pool is cleared. This means that `palloc` does not have to do as much accounting as `malloc()`; all it does in the typical case is to round up the size, bump a pointer, and do a range check.

(It also raises the possibility that heavy use of `palloc` could cause a server process to grow excessively large. There are two ways to deal with this, which are dealt with below; briefly, you can use `malloc`, and try to be sure that all of the memory gets explicitly `freed`, or you can allocate a sub–pool of the main pool, allocate your memory in the sub–pool, and clear it out periodically. The latter technique is discussed in the section on sub–pools below, and is used in the directory–indexing code, in order to avoid excessive storage allocation when listing directories with thousands of files).

6.3.2 Allocating initialized memory

There are functions which allocate initialized memory, and are frequently useful. The function `pcalloc` has the same interface as `palloc`, but clears out the memory it allocates before it returns it. The function `pstrdup` takes a resource pool and a `char *` as arguments, and allocates memory for a copy of the string the pointer points to, returning a pointer to the copy. Finally `pstrcat` is a varargs–style function, which takes a pointer to a resource pool, and at least two `char *` arguments, the last of which must be `NULL`. It allocates enough memory to fit copies of each of the strings, as a unit; for instance:

```
pstrcat (r->pool, "foo", "/", "bar", NULL);
```

returns a pointer to 8 bytes worth of memory, initialized to `"foo/bar"`.

6.3.3 Tracking open files, etc.

As indicated above, resource pools are also used to track other sorts of resources besides memory. The most common are open files. The routine which is typically used for this is `pfopen`, which takes a resource pool and two strings as arguments; the strings are the same as the typical arguments to `fopen`, e.g.,

```
...
FILE *f = pfopen (r->pool, r->filename, "r");

if (f == NULL) { ... } else { ... }
```

There is also a `popenf` routine, which parallels the lower–level `open` system call. Both of these routines arrange for the file to be closed when the resource pool in question is cleared.

Unlike the case for memory, there *are* functions to close files allocated with `pfopen`, and `popenf`, namely `pfclose` and `pclosef`. (This is because, on many systems, the number of files which a single process can have open is quite limited). It is important to use these functions to close files allocated with `pfopen` and `popenf`, since to do otherwise could cause fatal errors on systems such as Linux, which react badly if the same `FILE*` is closed more than once.

(Using the `close` functions is not mandatory, since the file will eventually be closed regardless, but you should consider it in cases where your module is opening, or could open, a lot of files).

6.3.4 Other sorts of resources — cleanup functions

More text goes here. Describe the the cleanup primitives in terms of which the file stuff is implemented; also, `spawn_process`.

6.3.5 Fine control — creating and dealing with sub-pools, with a note on sub-requests

On rare occasions, too–free use of `palloc()` and the associated primitives may result in undesirably profligate resource allocation. You can deal with such a case by creating a *sub-pool*, allocating within the sub–pool rather than the main pool, and clearing or destroying the sub–pool, which releases the resources which were associated with it. (This really *is* a rare situation; the only case in which it comes up in the standard module set is in case of listing directories, and then only with *very* large directories. Unnecessary use of the primitives discussed here can hair up your code quite a bit, with very little gain).

The primitive for creating a sub–pool is `make_sub_pool`, which takes another pool (the parent pool) as an argument. When the main pool is cleared, the sub–pool will be destroyed. The sub–pool may also be cleared or destroyed at any time, by calling the functions `clear_pool` and `destroy_pool`, respectively. (The difference is that `clear_pool` frees resources associated with the pool, while `destroy_pool` also deallocates the pool itself. In the former case, you can allocate new resources within the pool, and clear it again, and so forth; in the latter case, it is simply gone).

One final note ––– sub–requests have their own resource pools, which are sub–pools of the resource pool for the main request. The polite way to reclaim the resources associated with a sub request which you have allocated (using the `sub_req_lookup_...` functions) is `destroy_sub_request`, which frees the resource pool. Before calling this function, be sure to copy anything that you care about which might be allocated in the sub–request's resource pool into someplace a little less volatile (for instance, the filename in its `request_rec` structure).

(Again, under most circumstances, you shouldn't feel obliged to call this function; only 2K of memory or so are allocated for a typical sub request, and it will be freed anyway when the main request pool is cleared. It is only when you are allocating many, many sub–requests for a single main request that you should seriously consider the `destroy...` functions).

6.4 Configuration, commands and the like

One of the design goals for this server was to maintain external compatibility with the NCSA 1.3 server ––– that is, to read the same configuration files, to process all the directives therein correctly, and in general to be a drop–in replacement for NCSA. On the other hand, another design goal was to move as much of the server's functionality into modules which have as little as possible to do with the monolithic server core. The only way to reconcile these goals is to move the handling of most commands from the central server into the modules.

However, just giving the modules command tables is not enough to divorce them completely from the server core. The server has to remember the commands in order to act on them later. That involves maintaining data which is private to the modules, and which can be either per–server, or per–directory. Most things are per–directory, including in particular access control and authorization information, but also information on how to determine file types from suffixes, which can be modified by `AddType` and `DefaultType` directives, and so forth. In general, the governing philosophy is that anything which *can* be made configurable by directory should be; per–server information is generally used in the standard set of modules for information like `Aliases` and `Redirects` which come into play before the request is tied to a particular place in the underlying file system.

Another requirement for emulating the NCSA server is being able to handle the per–directory configuration files, generally called `.htaccess` files, though even in the NCSA server they can contain directives which have nothing at all to do with access control. Accordingly, after URI –> filename translation, but before performing any other phase, the server walks down the directory hierarchy of the underlying filesystem, following the translated pathname, to read any .htaccess files which might be present. The information which is read in then has to be *merged* with the applicable information from the server's own config files (either from the <Directory> sections in access.conf, or from defaults in srm.conf, which actually behaves for most purposes almost exactly like <Directory />).

Finally, after having served a request which involved reading .htaccess files, we need to discard the storage allocated for handling them. That is solved the same way it is solved wherever else similar problems come up, by tying those structures to the per–transaction resource pool.

6.4.1 Per-directory configuration structures

Let's look out how all of this plays out in mod_mime.c, which defines the file typing handler which emulates the NCSA server's behavior of determining file types from suffixes. What we'll be looking at, here, is the code which implements the AddType and AddEncoding commands. These commands can appear in .htaccess files, so they must be handled in the module's private per–directory data, which in fact, consists of two separate tables for MIME types and encoding information, and is declared as follows:

```
typedef struct {
    table *forced_types;      /* Additional AddTyped stuff */
    table *encoding_types;    /* Added with AddEncoding... */
} mime_dir_config;
```

When the server is reading a configuration file, or <Directory> section, which includes one of the MIME module's commands, it needs to create a mime_dir_config structure, so those commands have something to act on. It does this by invoking the function it finds in the module's 'create per–dir config slot', with two arguments: the name of the directory to which this configuration information applies (or NULL for srm.conf), and a pointer to a resource pool in which the allocation should happen.

(If we are reading a .htaccess file, that resource pool is the per–request resource pool for the request; otherwise it is a resource pool which is used for configuration data, and cleared on restarts. Either way, it is important for the structure being created to vanish when the pool is cleared, by registering a cleanup on the pool if necessary).

For the MIME module, the per–dir config creation function just pallocs the structure above, and a creates a couple of tables to fill it. That looks like this:

```
void *create_mime_dir_config (pool *p, char *dummy)
{
    mime_dir_config *new =
      (mime_dir_config *) palloc (p, sizeof(mime_dir_config));

    new->forced_types = make_table (p, 4);
    new->encoding_types = make_table (p, 4);

    return new;
}
```

Now, suppose we've just read in a .htaccess file. We already have the per–directory configuration structure for the next directory up in the hierarchy. If the .htaccess file we just read in didn't have any AddType or

AddEncoding commands, its per–directory config structure for the MIME module is still valid, and we can just use it. Otherwise, we need to merge the two structures somehow.

To do that, the server invokes the module's per–directory config merge function, if one is present. That function takes three arguments: the two structures being merged, and a resource pool in which to allocate the result. For the MIME module, all that needs to be done is overlay the tables from the new per–directory config structure with those from the parent:

```
void *merge_mime_dir_configs (pool *p, void *parent_dirv, void *subdirv)
{
    mime_dir_config *parent_dir = (mime_dir_config *)parent_dirv;
    mime_dir_config *subdir = (mime_dir_config *)subdirv;
    mime_dir_config *new =
      (mime_dir_config *)palloc (p, sizeof(mime_dir_config));

    new->forced_types = overlay_tables (p, subdir->forced_types,
                                        parent_dir->forced_types);
    new->encoding_types = overlay_tables (p, subdir->encoding_types,
                                          parent_dir->encoding_types);

    return new;
}
```

As a note — if there is no per–directory merge function present, the server will just use the subdirectory's configuration info, and ignore the parent's. For some modules, that works just fine (e.g., for the includes module, whose per–directory configuration information consists solely of the state of the XBITHACK), and for those modules, you can just not declare one, and leave the corresponding structure slot in the module itself NULL.

6.4.2 Command handling

Now that we have these structures, we need to be able to figure out how to fill them. That involves processing the actual AddType and AddEncoding commands. To find commands, the server looks in the module's command table. That table contains information on how many arguments the commands take, and in what formats, where it is permitted, and so forth. That information is sufficient to allow the server to invoke most command–handling functions with pre–parsed arguments. Without further ado, let's look at the AddType command handler, which looks like this (the AddEncoding command looks basically the same, and won't be shown here):

```
char *add_type(cmd_parms *cmd, mime_dir_config *m, char *ct, char *ext)
{
    if (*ext == '.') ++ext;
    table_set (m->forced_types, ext, ct);
    return NULL;
}
```

This command handler is unusually simple. As you can see, it takes four arguments, two of which are pre–parsed arguments, the third being the per–directory configuration structure for the module in question, and the fourth being a pointer to a cmd_parms structure. That structure contains a bunch of arguments which are frequently of use to some, but not all, commands, including a resource pool (from which memory can be allocated, and to which cleanups should be tied), and the (virtual) server being configured, from which the module's per–server configuration data can be obtained if required.

Another way in which this particular command handler is unusually simple is that there are no error conditions which it can encounter. If there were, it could return an error message instead of `NULL`; this causes an error to be printed out on the server's `stderr`, followed by a quick exit, if it is in the main config files; for a `.htaccess` file, the syntax error is logged in the server error log (along with an indication of where it came from), and the request is bounced with a server error response (HTTP error status, code 500).

The MIME module's command table has entries for these commands, which look like this:

```
command_rec mime_cmds[] = {
{ "AddType", add_type, NULL, OR_FILEINFO, TAKE2,
    "a mime type followed by a file extension" },
{ "AddEncoding", add_encoding, NULL, OR_FILEINFO, TAKE2,
    "an encoding (e.g., gzip), followed by a file extension" },
{ NULL }
};
```

The entries in these tables are:

- The name of the command
- The function which handles it
- a `(void *)` pointer, which is passed in the `cmd_parms` structure to the command handler ––– this is useful in case many similar commands are handled by the same function.
- A bit mask indicating where the command may appear. There are mask bits corresponding to each `AllowOverride` option, and an additional mask bit, `RSRC_CONF`, indicating that the command may appear in the server's own config files, but *not* in any `.htaccess` file.
- A flag indicating how many arguments the command handler wants pre–parsed, and how they should be passed in. `TAKE2` indicates two pre–parsed arguments. Other options are `TAKE1`, which indicates one pre–parsed argument, `FLAG`, which indicates that the argument should be `On` or `Off`, and is passed in as a boolean flag, `RAW_ARGS`, which causes the server to give the command the raw, unparsed arguments (everything but the command name itself). There is also `ITERATE`, which means that the handler looks the same as `TAKE1`, but that if multiple arguments are present, it should be called multiple times, and finally `ITERATE2`, which indicates that the command handler looks like a `TAKE2`, but if more arguments are present, then it should be called multiple times, holding the first argument constant.
- Finally, we have a string which describes the arguments that should be present. If the arguments in the actual config file are not as required, this string will be used to help give a more specific error message. (You can safely leave this `NULL`).

Finally, having set this all up, we have to use it. This is ultimately done in the module's handlers, specifically for its file–typing handler, which looks more or less like this; note that the per–directory configuration structure is extracted from the `request_rec`'s per–directory configuration vector by using the `get_module_config` function.

```
int find_ct(request_rec *r)
{
    int i;
    char *fn = pstrdup (r->pool, r->filename);
    mime_dir_config *conf = (mime_dir_config *)
             get_module_config(r->per_dir_config, &mime_module);
    char *type;

    if (S_ISDIR(r->finfo.st_mode)) {
        r->content_type = DIR_MAGIC_TYPE;
        return OK;
    }

    if((i=rind(fn,'.')) < 0) return DECLINED;
    ++i;

    if ((type = table_get (conf->encoding_types, &fn[i])))
    {
        r->content_encoding = type;

        /* go back to previous extension to try to use it as a type */

        fn[i-1] = '\0';
        if((i=rind(fn,'.')) < 0) return OK;
        ++i;
    }

    if ((type = table_get (conf->forced_types, &fn[i])))
    {
        r->content_type = type;
    }

    return OK;
}
```

6.4.3 Side notes — per-server configuration, virtual servers, etc.

The basic ideas behind per–server module configuration are basically the same as those for per–directory configuration; there is a creation function and a merge function, the latter being invoked where a virtual server has partially overridden the base server configuration, and a combined structure must be computed. (As with per–directory configuration, the default if no merge function is specified, and a module is configured in some virtual server, is that the base configuration is simply ignored).

The only substantial difference is that when a command needs to configure the per–server private module data, it needs to go to the `cmd_parms` data to get at it. Here's an example, from the alias module, which also indicates how a syntax error can be returned (note that the per–directory configuration argument to the command handler is declared as a dummy, since the module doesn't actually have per–directory config data):

```
char *add_redirect(cmd_parms *cmd, void *dummy, char *f, char *url)
{
```

```
    server_rec *s = cmd->server;
    alias_server_conf *conf = (alias_server_conf *)
            get_module_config(s->module_config,&alias_module);
    alias_entry *new = push_array (conf->redirects);

    if (!is_url (url)) return "Redirect to non-URL";

    new->fake = f; new->real = url;
    return NULL;
}
```

Index