

# Writing Modules for Apache 1.3

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

*Presented at the August 1999 O'Reilly Open Source Conference in Monterey, California*

## **Topics:**

- Definitions
- The Apache API
- Module Interaction with the Server
- Operating Environments
- Configuration Hooks
- Server Config Hooks
- *Per-Directory* Config Hooks
- Config Processing Phases
- Accessing Config Records
- API Structures
- Request Processing
- Module Order
- Hook Return Values
- Request Processing Phases
- Notes about Module Processing

You can find the latest version of this handout online at  
<<http://Web.Golux.Com/coar/slides/>>.

## Some Definitions

### *Config File:*

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

### *Container:*

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

### *Directive:*

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

### *DocumentRoot:*

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

### *Internet Media Type:*

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

### *MIME Type:*

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

### *Overrides:*

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

### *Scope:*

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

### *ServerRoot:*

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

### *Stanza:*

See *container*.

### *Virtual Hosting:*

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

## What's an Apache Module?

Probably the first question should be “What’s Apache?” However, since this is a fairly advanced session, that’s really not appropriate. If you don’t know the answer to this question, you probably shouldn’t proceed any further until you learn more. This is definitely a coding session for geeks.

An Apache *module* is a piece of compiled code (usually C) that is either built into the server executable image when the entire Apache package is constructed, or else an external fragment of code, called a *DSO* (‘dynamic shared object’) or *DLL* (‘dynamically loaded library’), that is activated by the server at run time.

**Note:** Although it’s tempting to apply the overused term *plug-in* to Apache modules, you really should resist. For one thing, modules are pieces of the server, and ‘plug-in’ frequently refers to client-side functionality. For another, no-one will know what you’re talking about until you’re all on the same wavelength and using the same terminology.

The Apache Web server architecture allows you to augment the basic functionality and get your code involved at various stages of Web request processing. The extremes of the enhancement spectrum are modifying the base code itself, and using something like CGI, which is wholly outside the server. Modules fall into the middle ground.

In order to build an Apache module, you need the rest of the Apache source and build environment. This isn’t as scary as it sounds; I don’t mean a full ADE like Visual J++. Rather, you just need the Apache source kit, an ANSI C compiler, and some basic tools (*make*, *awk*, *sed*, *et cetera*). If you’re building on Windows, you need Microsoft’s Visual Studio and Visual C++ V5.

**Note:** There’s nothing stopping you from using some other development environment on Windows – but if you do, you’re going to have to port the existing stuff to use it. The Apache developers made the decision to use the Microsoft tools, so that’s how the scripts and makefiles are set up.

You **don’t** need any development stuff at all in order to simply use a binary DSO or DLL module. However, since this is about writing modules, you can’t get off that easily.

## Why a Module?

With the full source available, and an external access mechanism like CGI supported, why would anyone want to write a module for Apache? Some reasons include:

- You want to make the functionality available on other systems without having to recompile the whole server on each one;
- You want the functionality to be portable across multiple systems without having to worry about whether add-on tools like Perl are installed (or where);
- You want better performance than you can get from something like a CGI script.

Modules allow you to address these and other issues – but they are **not** a universal panacea. In some cases, writing a module might end up being like using a sledge hammer to drive a push-pin.

Before you give serious consideration to writing a module, ask yourself the following question: “*What is it supposed to do?*” Due to the phased processing model of the Apache architecture, there are clear types of functionality that are suited to realisation as modules. Other things just don’t fit into the model, and should be accomplished some other way.

These are the basic types of operations that the Apache module architecture is well-suited to handling:

- Securing access to Web resources
- Manipulating URLs
- Logging server activity
- Providing/generating content

A single module can perform more than one of these functions, though it's not very common.

## Examples of Apache Modules

Here are some examples of Apache modules and what they do. These are all distributed as part of the base Apache package:

### **mod\_auth**

Simple text-based user/password access control; user credentials are looked up in a server-local database to see if they're valid and grant access to the requested resource

### **mod\_speling**

If no resource can be found that exactly matches the request, this module looks around to see if there are any that are similarly spelt. If found, it will either automatically redirect the browser to the correctly-spelt document, or if there are multiple near-misses, will give the browser a list from which to choose.

### **mod\_rewrite**

This is without doubt one of the most powerful and flexible modules in the base Apache package. It allows you to select documents or redirect to other URLs – even on other servers – based upon a dizzying array of possible criteria.

A large list of other modules people have written for Apache, but which aren't part of the base distribution for whatever reason, can be found at <http://modules.apache.org/>. You should definitely give careful scrutiny to existing modules before embarking on the project of writing your own – unless you're just doing it for fun.

## The API

A module has access to the Apache server core functions through the Application Programming Interface or *API*. The API is actually a pretty nebulous thing, comprising both the functions and structures available to the module and the mechanism by which the server activates the module. The primary structures in which a module may be interested are described later

The apache developers are generally interested in writing code, rather than documenting it, but a rudimentary set of documentation pages about the majority of the API structures and functions can be found at the developer Web site:

<http://dev.apache.org/apidoc/>

## Module Involvement

In order for the server to be able to take advantage of a module, it needs to know it's available. In the case of a dynamically loaded module, this is done with the `LoadModule` directive. This calls the operating system's dynamic loader to pull the specified file into memory, and links the module into the list of those available. In the case of a statically loaded module, the list was constructed at compile-time, and so no special run-time directives are necessary. The list of modules can be stripped down to just the core Apache processing module with the `ClearModuleList` directive, and reassembled module by module with the `ActivateModule` directive. This is useful if you want to change the ordering of the modules in the list; this order can be important for reasons to be described shortly.

In both the dynamic and the static cases, though, one thing that *is* necessary is the name of the `module` structure, which defines to Apache how the module is to be accessed and in what phases of processing the module is interested. The structure must be globally declared with a name that won't conflict with any other modules' structure name; the convention for a module named `mod_foo_bar` is to name the `module` structure "`foo_bar_module`" for easy remembering and location.

The `module` structure informs the main Apache code of during which request-processing phases the module should be invoked, and includes pointers to lists of directives and content-handler names that the module provides.

Module operation is controlled through the use of module-specific configuration directives. You may have encountered an error message in your server error log that looked something like this:

```
Invalid command 'foo', perhaps mis-spelled or defined by a module
not included in the server configuration
```

Exactly as it says, this is the sort of error message you'll get if you use the `'foo'` directive and your server's module list doesn't include a module that defines the directive.

**Note:** You can avoid this type of error, and have the directive processed if and only if the appropriate module is loaded, by using the `<IfModule>` directive container.

Even though Apache provides for overloading of directives (*i.e.*, for multiple modules to use the same directive name), modules need to treat such possibly shared directives specially – and most don't. So the safest thing is to try to ensure that your module's directive names are unique to it.

As the server progresses through processing its configuration or a request from a client, it will invoke the appropriate functions in your module, according to the indications of interest defined in your `module` structure. For example, if the server comes to the point in processing a request at which it's checking for access restrictions based on the client's IP address, and your `module` structure included a function pointer in the appropriate slot, the server will invoke your function so your module can do whatever access checking it wants.

The different phases and conditions in which you can ask the server to involve your module are described later.

## Operating Environments

There are two basic types of operations in which the server will involve your module:

1. Configuration, and

## 2. Request processing.

Each of these is broken down into further situations. Configuration, for example, involves server-wide directive handling, *per*-virtual host directives, and *per*-directory directives. Request processing likewise involves up to nearly a dozen different stages of processing.

Your module's functions can tell what the situation is from two factors: which of your routines is being invoked, and the arguments being passed to it.

The configuration callbacks are invoked whenever it's appropriate – which may mean only when the server is being started up, or when it's in the process of locating a particular file on disk in the processing of an actual Web request.

I've already listed the two different types of configuration environments: server and *per*-directory. Each of those, however, breaks down into still further situations.

When dealing with server-wide configuration (which your module will only do if you've explicitly indicated that you want it to), there are two subcategories to the environment being configured. Either it's what's called the *global* or *default* server environment, which applies to all virtual hosts and situations in which a virtual host isn't involved, or it's handling the configuration for a specific virtual host.

It's quite possible that your configuration functions will be called multiple times, such as in the case of a server with more than one virtual host.

Server configuraton hooks are only invoked during server startup; unlike the *per*-directory configuration process, there are no changes that are made to the server configuration once it's up and running.

*Per*-directory configuration can happen both during server-wide configuration and during the processing of a client request at run-time.

**Note:** None of your module configuration routines should *ever* make assumptions about what other callbacks have been made, or in what order. Each needs to be completely independent of all others. If your module needs to pass some sort of state information between the various callback routines, you need to use the module configuration record mechanism described in the next section.

## Directive Scopes

All occurrences of Apache directives are associated with a *scope*. In some cases, the scope is the entire server (such as the `PidFile` directive); in others, the scope applies to a particular directory on the filesystem. The normal condition is for the scope of a directory-oriented directive to include any sub- or sub-sub-directories as well, so if directive “`Foo On`” applies to “`C:\htdocs\bar`” it also applies to “`C:\htdocs\bar\bletch`” and “`C:\htdocs\bar\frotz`” as well.

This is called *inheritance*, and the inheritance rules are defined on a directive-by-directive basis by the owning module. Of course, it's quite possible that the effect of the “`Foo On`” directive in a parent directory may be overridden by a “`Foo Off`” directive in a subordinate directory – which would then apply to all subdirectories of *that* until overridden again.

The set of environments (typically a directory tree) to which a directive applies is called its *scope*.

## Configuration Hooks

When the server is processing configuration directives and it encounters one defined by your module, it will call from one to three functions listed in your `module` structure.

1. Module configuration record creation routine
2. Directive handler
3. Module configuration record merging routine

All configuration information in Apache is managed in a set of tree structures, containing *module configuration records*. When the server encounters one of your module's directives in its processing of its configuration instructions, it will check to see if it already has a module configuration record for the current scope; if it doesn't, then it calls your module's record-creation routine (if you have declared one). The routine in your module that's responsible for processing the directive (checking to see if it's valid, and saving its meaning for later) typically stores information about the directive in the module record that's passed.

Once Apache has finished processing all of your module's directives in a particular scope, it calls one of your merge functions to handle the inheritance issues. The merge routine is given pointers to the module record for the current scope and for the closest ancestor scope for which there *is* a record. It's up to your merge routine to figure out the correct settings that apply to the current scope, based upon the directives that were processed specific to it and the inheritance of any settings from a broader scope (*e.g.*, from higher up the directory tree). Your merge routine is responsible for creating a *new* module record, filling it in with the merged settings, and returning it to the server, which then links it into the chain.

If the process should happen to occur again at a lower level (a narrower scope), the 'parent' module record that will be passed in will be the one created here.

*Per*-directory module records are *much* more common than those created for server scopes. In fact, when your directive handlers are called, they're given a pointer to the appropriate *per*-directory module record, but not to any *per*-server module record. If your directive handler wants to store information in a server-wide module record, it needs to ask the server for the record explicitly.

*Per*-server and *per*-directory module records can have identical layouts, or be completely different; it's entirely up to your module, since *you* define the structures. But even if they're identical, in *no* case will Apache try to merge them together; they're on two completely separate lists and the server considers them unrelated.

The routines you declare for the purpose of creating your module configuration records are passed some information about the scope to which the record applies. In the case of a routine tasked with creating a record for a server-wide scope, this is in the form of a pointer to the server's control record (its *server\_rec*). For a *per*-directory record creation routine, the argument is a string identifying the directory in question, such as `"/` or `"/usr/local/apache/htdocs/bar"`. What use you make of this information is entirely up to you.

*Per*-directory record creation routines can be passed a 'magic' directory string which means "this applies to *all* directories unless overridden." This magic value is simply `""`, an empty string.

Module configuration records are all chained together in a list attached to the appropriate structure (the *server\_rec* or *request\_rec*). In order to find *your* module's record, you pass this list pointer to the `ap_get_module_config()` routine, along with a pointer to your module's `module` record. (This

overusage of the word ‘module’ is getting rather confusing, isn’t it?) For example, if *r* is a pointer to a *request\_rec* for a request in progress, you can get both your module’s *per*-directory and server-wide records as follows (where *foo\_srec\_t* and *foo\_rrec\_t* are the typedef names for your server and *per*-directory record structures, and *foo\_module* is the label of your module structure):

```
foo_srec_t *srec;
foo_rrec_t *rrec;
rrec = (foo_rrec_t *) ap_get_module_config(r->per_dir_config,
                                           &foo_module);
srec = (foo_srec_t *) ap_get_module_config(r->server->module_config,
                                           &foo_module);
```

## Server Config Hooks

As mentioned earlier, there are two slots in the module structure for dealing with server-wide module records, one for creating a new record and one for merging two existing records into a new combined one.

The creation routine tends to be very simple, along the lines of

```
static void *foo_new_server_rec(pool *p, server_rec *s)
{
    foo_srec_t *srec;

    srec = (foo_srec_t *) ap_pccalloc(p, sizeof(foo_srec_t));
    return (void *) srec;
}
```

It may do additional things like setting default values into the various fields, but the above is the function reduced to its most basic.

Similarly, the *per*-server module record merging routine might look like this:

```
static void *foo_merg_server_rec(pool *p, void *p_srec_p, void *c_srec_p)
{
    foo_srec_t *p_srec, *c_srec;    /* p arent and c urrent records */
    foo_srec_t *new_srec;

    p_srec = (foo_srec_t *) p_srec_p;
    c_srec = (foo_srec_t *) c_srec_p;
    new_srec = (foo_srec_t *) ap_pccalloc(p, sizeof(foo_srec_t));
    /*
     * foo and bar are *always* inherited from the current record.
     */
    new_srec->foo = c_srec->foo;
    new_srec->bar = c_srec->bar;
    /*
     * zed is inherited from the current directory if there's
     * a non-default setting there, otherwise from the parent
     */
    new_srec->zed = (c_srec->zed == 0) ? p_srec->zed : c_srec->zed;
    return (void *) new_srec;
}
```

As you can see, these routines are really quite simple.

*Per*-server module records are not used very often, since they don’t lend themselves to being customisable at different levels and locations in the server’s URI space. Examples of their use include the *Alias*,

Port, ServerName, and MaxRequestsPerChild directives – none of which have anything to do with where the server might end up finding the requested resource.

When it comes to scoping for this type of record, there are only two levels: the global server environment (*i.e.*, anything that isn't inside a <VirtualHost> container block), and <VirtualHost> container definitions.

## Per-directory Config

The *per*-directory callbacks are identical to the *per*-server ones described in the previous section – save that they are used to deal with information about locations in the underlying document filesystem, rather than to server-wide settings. Since this is by far the more common environment for directives, the Apache architecture makes things a little easier, such as by looking up and providing a pointer to the module's *per*-directory configuration record as part of the directive handler argument list.

Examples of functionality that is better suited to *per*-directory scoping than server-wide include

- labeling file types as indicating specific contents (*e.g.*, defining “.html” as meaning HTML)
- applying access control
- controlling whether directory listings are allowed

## Config Processing Phases

In addition to the callback routines used to manage a module's private configuration information, there are a few additional hooks which your `module` structure can declare:

- The **init** hook allows your module to do any one-time initial setup it needs. The only argument to this hook is a pool pointer, however, which makes it very difficult to pass information to other callbacks made later. An example use for this callback might be to set up a connexion to a database server.

**Note:** One thing that serves as an endless source of confusion is that this callback is invoked **twice**. Don't ask why; it's a legacy thing from the early days of Apache. Just be aware of the fact, and don't be surprised by it.

- Similar to the `init` callback, the **child\_init** hook is available to modules that want or need to setup processing of some sort. The difference between this callback and the `init` callback is that this one is invoked *only* when a new child process is being created. In the case of a threaded server or environment (such as Apache 1.3 on Windows), these two callbacks have distinctly different meanings. In strict multi-process Apache environments such as Unix, however, the lines blur.
- The `module` structure includes a slot for a **child\_exit** callback, which will get invoked when the server is preparing to shut down a child process. Use of this callback is discouraged, however; it is preferable to register a cleanup in the appropriate pool as part of the `init` or `child_init` callback routines.

Your module will only be invoked for the phases or functions that you have requested by putting routine pointers into the `module` structure. If you don't put anything in the `child_init` slot, for instance, your module won't be invoked when the server is doing that processing.

## Accessing Config Records

As was mentioned before, in many cases your module configuration record will be handed to your routines as part of their argument list. If you need to locate them yourself, however, you do so with the `ap_get_module_config()` function described earlier; just give it the pointer to the appropriate list of records and a pointer to your `module` structure so it can find your record in amongst the others, and that's it.

The two main list pointers for module records are in the `request_rec` and `server_rec` structures, at `r->per_dir_config` and `s->module_config`, respectively. There are a couple of additional list pointers as well, but you should stay away from using them unless and until you have both a need and a clear understanding of their use. These are at `r->request_config` and `s->lookup_defaults`.

If you end up having to actually add one of your own module configuration record to a list, you can do so with the `ap_set_module_config()` routine. This is the same routine the server itself uses to manage these lists.

## API Structures

The Apache API is full of structures and routines; there are well over three hundred of them. However, for most purposes there are only five API structures that are really important. Some of them are so common and use the same naming convention so consistently that you can easily tell what's going on by looking at the variable names.

Each of these structures are described in more detail later, but here are the Big Five:

- **pool** (usually pointed to by “p”) – a memory management structure
- **server\_rec** (usually pointed to by “s”) – contains information about a server or virtual host
- **request\_rec** (usually pointed to by “r”) – defines the attributes of a client (or internal) request
- **module** – defines the callback hooks for your module and points to lists of directives and content handlers
- **cmd\_parms** – used to convey information to a directive handler about the environment

### *pool* Structure

The *pool* structure is arguably the most important one in the entire API. A pool is a memory management structure with the following attributes:

- It may (or may not) have a parent
- Memory can be allocated using either the `ap_palloc()` (allocate from pool) or `ap_pccalloc()` (allocate from pool and clear) routines
- There is no `free()` function; pools grow until they're destroyed
- When a pool is destroyed, and all sub-pools are recursively destroyed as well
- When a pool is destroyed, any cleanup functions registered in it are invoked

Because of the highly dynamic nature of a Web server, handling a variable load of typically short-lived requests, the lack of a `free()` routine in the pool API is not a problem. Apache will allocate a pool when it begins processing a request, and all modules that deal with the request use that pool for their scratch space. When the request has been completed, the pool is destroyed and everything is automatically cleaned up.

Everything involves pools; other structures (such as the `request_rec` and `server_rec`) are allocated from them, pool-oriented string manipulation routines like `ap_pstrncpy()` and `ap_pstrdup()` are used in preference to their mundane counterparts, and `malloc()` and `free()` is strongly discouraged. Almost every single API routine has access to a pool, either through a pointer to one being explicitly in the argument list or by there being one inside one of the structures that is passed.

Pools include what's called a 'cleanup' mechanism. If you allocate structures from a pool and need to clean them up when the pool is destroyed (such as closing a database connexion or the like), you accomplish this by registering a cleanup in the pool. When the pool destruction code runs, it invokes any and all cleanup hooks it finds registered with it.

The pool structure is completely opaque to modules, meaning that it has no fields or components that you need to access (and can't even if you wanted to).

Two other structures that are closely related to pools are Apache *arrays* and *tables*. Apache arrays are dynamically-sizable sequential lists of arbitrarily-sized elements; tables are key-value pairs (where both the key and the value must be strings). Both structures include a pool pointer, and additions to the array or table automatically results in memory being allocated from that pool at need.

### ***server\_rec Structure***

Each global server and virtual host environment is defined in a `server_rec` structure. This includes things like information about the activity and error logs, timeout values, the server's host name, pointers to lists of module configuration records, and so on. In most cases this structure should be considered "read-only" by modules, and in fact most modules don't even care about the contents of the structure (unless they have directives with server-wide scope, of course).

For more information about this structure, see its definition in the `src/include/httpd.h` header file and the (rudimentary) `<http://dev.apache.org/apidoc/>` online API documentation.

### ***request\_rec Structure***

Each client request for a URI (and each such request that the server may generate internally) is described by a `request_rec` structure. A `request_rec` consists mainly of pointers: a pool pointer, a pointer to a list of environment variables made available to CGI scripts and SSI documents (if that's what the request resolves to), pointers to lists of HTTP request and response header fields, a pointer to the actual file name (if the request resolves to a file), and so on.

Unlike the `server_rec`, the `request_rec` is most definitely a read/write structure from the module writer's point of view. Most of the early request processing phases are transformations or interpretations of the information in the `request_rec`, and the results are likewise stored there for later use by the same or other modules.

Requests may include an Apache concept called a *subrequest*. A subrequest is essentially a subordinate request the server makes internally, in order to answer a question about the current request. For example, in an SSI parsed document, each `#include` directive will result in a subrequest being made for the

specified resource; this allows the server to either insert the content or an error message in the content of the main request it is constructing.

Subrequests differ from normal requests in that they are *not* automatically completed. That is, a module may fire off a subrequest for a file, and only be interested in the filled-in `request_rec` that results (such as the file statistics or the content type).

### ***cmd\_parms* Structure**

As far as module writers are concerned, the `cmd_parms` structure is only used when processing directives. It's passed to the directive handler in the argument list, and include things such as the actual directive name, the full line being processed from the configuration source, the line number, a pool pointer for any structures the directive handler may need to allocation, the applicable limitations and overrides in effect, a pointer to the `server_rec` for the server in whose scope the directive was found, and so on. In most cases, the only thing used from this structure is the pool pointer.

### ***module* Structure**

As described earlier, the `module` structure provides the glue between the Apache server core code and the module itself. The server only knows about what it finds in the `module` structure, and no more.

**Note:** The `MODULE_MAGIC_NUMBER` (described later) is stored in the module structure at compile time.

The module structure includes slots for pointers to all the possible callback hooks, some identification information, and pointers to lists of directives and content handler names.

You basically just set this structure up in your module source and just leave it alone; at run-time it's a read-only structure.

## **Request Processing**

As the Apache server processes a request, it takes it through ten (10) different stages, called *phases*. At each phase, any modules that have registered a callback for that phase are invoked and given an opportunity to work their will upon the request.

The phases in Apache 1.3.9 are, in order:

1. Post-read request
2. URI translation
3. Header field parsing
4. Mandatory access checking
5. Discretionary authentication
6. Discretionary authorisation
7. MIME/IMT type determination or setting
8. Last-chance 'fixups'

9. Content emission

10. Logging

All of these phases are described in the sections that follow.

## Module Order

Before we actually get to the phase descriptions, something should be said about module invocation order.

The list of active modules that Apache keeps internally is just that – a linked list of the `module` structures from all of them. This list is handled as a push-down or LIFO list, meaning that the modules listed earliest are the ones invoked last.

One of the consequences of having a single list is that modules are always called in the same order regardless of the current phase. For involved or complex modules this may require careful placement in the `src/Configuration` and/or `conf/httpd.conf` files (depending upon whether the modules are statically or dynamically loaded); you may want to have the post-read phase of your module called before that of `mod_foobar`, but you want your `fixup` phase called after that of `mod_bletch`.

Any module hook is called at most one time *per* request. The changes it makes to the `request_rec` structure are how it communicates its effects to other modules and later phases. In some cases, not all modules will even get a chance at a request; the most common example is if an earlier module returns an error; the phase processing (and indeed the whole request processing mechanism) aborts at that point.

## Hook Return Values

Each hook or callback returns a value of some kind. The value returned tells the server about the success or failure of the hook's processing. Return values may either be pointers (NULL, a pointer to a string, or a pointer to a structure) or numbers.

- Directive handlers return either NULL (meaning success), a pointer to an error message string, or the magic constant `DECLINE_CMD`, which means that *this* module doesn't want to handle the directive but the server should see if any others do. This is a highly underused facility, but probably for good reason: how can a particular module tell whether the syntax is wrong because it's wrong, or because it's correct but for a different module?
- Config record handlers, like the record creation and merge handlers, return either a pointer to the new structure or NULL.
- All other handlers return either one of the magic constants `OK`, `DECLINED`, or `DONE`, or else an HTTP error status code. `OK` means the handler accepted the challenge and did something with the request; `DECLINED` means that it didn't do anything with it and the server should keep looking for a module to accept it, and `DONE` means the handler did whatever was necessary to complete the request and the server can short-circuit calling any other handlers or phases.

All phases will stop and the request will be aborted if a handler returns an error status. Some phases will end, and processing will advance to the next phase, as soon as any handler returns `OK`; some phases will continue through the list until all modules have been called or an error has been returned. Some phases will abort the request if *no* module returns `OK` (*i.e.*, if no modules will touch the request).

## Request-Processing Phases

The following sections give brief descriptions of each of the phases and their purposes.

### ***Post-read Phase***

This phase is the first, and begins as soon as the server has read the request header from the client. Modules typically use this phase to make notes or decisions about the request (such as by setting environment variables).

Examples

- `mod_setenvif`

### ***URI Translation Phase***

The URI translation phase is typically used to translate a URI into a filesystem location, or possibly into another URI. This is not always appropriate due to its early position in the process, but that's what it's for anyway.

Example modules that use this phase:

- `mod_alias`
- `mod_rewrite`

### ***Header Field Parsing***

The header field parsing phase is called third in the sequence, and was originally intended to be used to make notes or decisions about the contents of the request header. Unfortunately, this phase turns out to occur too *late* for some things, so many of the modules that hooked into it in the past now hook into the post-read phase instead.

### ***Security Callbacks***

The next three callbacks have to do with applying security restrictions, and their interactions are often a source of confusion. The next few paragraphs explain their roles in generic detail (if there is such a thing). If you feel comfortable with the terms *mandatory access control*, *discretionary access control*, *authentication*, and *authorisation*, go ahead and skip ahead.

### ***Understanding Web-based access control***

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

Let's deal with authentication first.

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

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

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

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

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

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

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

## **Security: Access**

Called the *access checking* phase, this one is tasked with examining the characteristics of the request, the client connexion, and so on, and making a decision about whether the request should be denied or not. Since no discretionary credentials are examined – or even available – no handler in this phase can return an 'unauthorised' status which results in a retry; only OK or HTTP\_FORBIDDEN are appropriate.

Example module that uses this phase:

- `mod_access`

## **Security: Authentication**

This is the phase at which any user-supplied credentials are looked up in a server database and verified. The usual responses for a handler in this phase are OK (meaning the credentials are valid), DECLINED (meaning the handler isn't qualified to judge), or HTTP\_UNAUTHORIZED (meaning that there's something wrong with the credentials, such as a password mismatch or a completely invalid username).

This phase will not be invoked unless there is a `Require` directive in the current scope (see the main Apache documentation for details).

## **Security: Authorisation**

Once the client's credentials have been verified as being correct, the next step is to see if they're in the access list for the resource being requested. In Apache terms, the authentication and authorisation phases are typically tightly bound, but that's not necessarily so; different resources may have different access lists, but share the same user list.

The usual returns from a handler in this phase are OK (access granted), DECLINED (not qualified), or HTTP\_UNAUTHORIZED (try again with different credentials).

## **Type Checking/Setting**

This phase gives modules a chance to examine the attributes of the request and the resource and make notes about the type of resource. The usual responses from a handler are OK (with `r->content_type` *et alia* being updated appropriately) and DECLINED (usually because some other module has already done the work).

Example modules that use this phase:

- `mod_mime`
- `mod_mime_magic`

## **Fixups**

This phase is the last one before the server actually starts sending content back to client, so it's the last opportunity modules have to make any tweaks or changes (hence the name of the phase).

Example modules that use this phase:

- `mod_env`

## **Content Handling**

This is the real workhorse phase; it's the one that actually gets the resource content to the client. Content generation and transmission is considerably more involved than the other phases, so it's no wonder that not very many actually contribute to this phase.

Content handling may involve actual content generation, as is done by `mod_example`, `mod_info`, and `mod_status`, or it may involve transforming some existing content before transmission, such as is done by `mod_include` and `mod_php3`.

Only one content handler can be invoked for Apache 1.3; this means that there's no easy way, for instance, to have the output from a CGI script (handled by `mod_cgi`) parsed by `mod_include` for server-side include directives.

Any failure in processing at this or any earlier phase will result in the request being aborted by the Apache core code.

Example modules that hook into this phase:

- `mod_include`
- `mod_cgi`
- `mod_php3`
- `mod_info`
- `mod_status`

## Logging

This phase is invoked after the content has been sent to the client; this is where all the statistics collected during the request processing may be recorded and/or analysed. Any error during this phase is non-fatal, since the request has already been completed as far as the client is concerned.

Example modules that use this phase:

- `mod_log_config`
- `mod_log_referer`
- `mod_log_agent`

## The `MODULE_MAGIC_NUMBER`

Binary compatibility between modules and the main Apache core code is measured using a concept called the `MODULE_MAGIC_NUMBER`. This is a constant value that is defined in the Apache header files and automatically included in the `module` structure when a module is compiled. As the API grows and evolves, the MMN is incremented by the Apache development team, depending upon the compatibility of the change with earlier versions.

At run time, the core code examines the value stored in the `module` structure and compares it to the value with which the core code was built. If they're identical, or indicate compatibility, operation proceeds normally. Otherwise, the core will refuse to include the module in its list and will emit an error message.

This mechanism has many shortcomings, but it at least keeps modules from improperly using API routines whose semantics have changed, or from calling routines that may not be available (if the core code's value is older than the module's).

## Notes about Module Processing

Here are some final notes about common problems and questions about module writing:

- The `request_rec` is the *only* currently-supported way of passing information from phase to phase
- Modules **cannot** assume the same child will handle two consecutive requests from the same client
- Module code should *always* be written to be modular, thread-safe, and register cleanups