The Apache Platform and Architecture

Apache runs as a permanent background task: a daemon (Unix) or service (Windows). Startup is a slow and expensive operation, so for an operational server, it is usual for Apache to start at system boot and remain permanently up. Early versions of Apache had documented support for an inetd mode, but this was only ever of academic interest.

1 Two-Phase Operation

Apache operation has two main phases: startup and normal operation. System startup takes place as root, and includes parsing the configuration file(s), loading modules, and initialising system resources such as log files, shared memory segments, and database connections. For normal operation, Apache relinquishes its system privileges and runs as an unprivileged user before accepting and processing connections from clients over the network. This is a basic security measure, which helps to prevent a simple bug in Apache (or a module) becoming a devastating system vulnerability, like those exploited by malware such as 'Code Red' and 'Nimda' in MS IIS. There is no shutdown phase as such: when Apache stops, it runs registered cleanups for each resource that requires them, as described in the next chapter.

This two-stage operation has some implications for applications architecture. Firstly, anything that requires system privileges must be run at system startup. Secondly, it is good practice to run as much initialisation as possible at startup, to minimise the processing required to service each hit; later chapters will describe some standard techniques for this. Conversely, because so many slow/expensive operations are concentrated in system startup, it would be hugely inefficient to try to run Apache from a generic server such as inetd or tcpserver.

2 Basic Concepts and Structures

To work with Apache as a development platform, we need an overview of the basic units of webserver operation and the core objects that represent them within Apache. The most important are the Server, the TCP Connection, and the HTTP Request. A fourth basic Apache object, the Process, is a unit of operating system rather than application architecture. Each of these basic units is represented by a core data structure defined in the header file httpd.h.

Before describing these core data structures, we need to introduce some further concepts used throughout Apache, and closely tied to the architecture:

- APR Pools (`apr_pool_t`) are the core of resource management in Apache. Whenever a resource is allocated dynamically, a cleanup is registered with a pool, ensuring that system resources are freed when no longer required. Pools are used to tie resources to the lifetime of one of the the core objects. We will describe pools in depth in the APR chapter.
- Configuration Records are used by each module to tie its own data to one of the core objects. The core data structs include configuration vectors (ap_conf_vector_t), with each module having its own entry in the vector. They

are used in two ways: to set and retrieve permanent configuration data, and to store temporary data associated with a transient object. They are often essential to avoid use of unsafe static or global data in a module, and will be described in Chapter 8.

Having introduced pools and configuration records, we are now ready to look at the Apache core objects. In order of importance to most modules, they are

- `request_rec`
- `server_rec`
- `conn_rec`
- `process_rec`

The first two are by far the most commonly encountered in application development.

request_rec

A `request_rec` object is created whenever Apache accepts an HTTP request from a client, and is destroyed as soon as Apache finishes processing the request. The request_rec is the object passed to every handler implemented by any module in the course of processing a request (Chapters 4 and 5). It holds all the internal data relevant to processing an HTTP request, and comprises, as well as a number of fields used internally to maintain state and client information by Apache:

- A request pool, for management of objects having the lifetime of the Request. This is used to manage resources allocated whilst processing the request.
- A vector of configuration record for static request configuration (per-directory data specified in `httpd.conf` or `.htaccess`).
- A configuration record for transient data used in processing.
- Tables of HTTP Input, Output and Error headers.
- Table of Apache Environment variables (the environment as seen in scripting extensions such as SSI, CGI, mod_rewrite, and PHP), and a similar "notes" table for request data that should not be seen by scripts.
- Pointers to all the other relevant objects, including the Connection, the Server, and any related request objects..
- Pointers to the Input and Output filter chains (Chapter 6).
- The URI requested, and the internal parsed representation of it, including handler (Chapter 4) and filesystem mapping (Chapter 5).

Here is the full definition, from `httpd.h`:

```
/** A structure that represents the current request */
struct request_rec {
    /** The pool associated with the request */
    apr_pool_t *pool;
```

```
/** The connection to the client */
conn_rec *connection;
/** The virtual host for this request */
server_rec *server;

/** Pointer to the redirected request if this is an external
  redirect */
request_rec *next;
/** Pointer to the previous request if this is an internal redirect
  */
request_rec *prev;

/** Pointer to the main request if this is a sub-request
 * (see http_request.h) */
request_rec *main;

/* Info about the request itself... we begin with stuff that only
 * protocol.c should ever touch...
 */
/** First line of request */
char *the_request;
/** HTTP/0.9, "simple" request (e.g. GET /foo\n w/no headers) */
int assbackwards;
/** A proxy request (calculated during
        post_read_request/translate_name)
 *  possible values PROXYREQ_NONE, PROXYREQ_PROXY, PROXYREQ_REVERSE,
 *                  PROXYREQ_RESPONSE
 */
int proxyreq;
/** HEAD request, as opposed to GET */
int header_only;
/** Protocol string, as given to us, or HTTP/0.9 */
char *protocol;
/** Protocol version number of protocol; 1.1 = 1001 */
int proto_num;
/** Host, as set by full URI or Host: */
const char *hostname;

/** Time when the request started */
apr_time_t request_time;

/** Status line, if set by script */
const char *status_line;
/** Status line */
int status;

/* Request method, two ways; also, protocol, etc..  Outside of
        protocol.c,
 * look, but don't touch.
 */

/** Request method (eg. GET, HEAD, POST, etc.) */
const char *method;
/** M_GET, M_POST, etc. */
```

```
int method_number;

/**
 *  'allowed' is a bitvector of the allowed methods.
 *
 *  A handler must ensure that the request method is one that
 *  it is capable of handling.  Generally modules should DECLINE
 *  any request methods they do not handle.  Prior to aborting the
 *  handler like this the handler should set r->allowed to the list
 *  of methods that it is willing to handle.  This bitvector is used
 *  to construct the "Allow:" header required for OPTIONS requests,
 *  and HTTP_METHOD_NOT_ALLOWED and HTTP_NOT_IMPLEMENTED status
       codes.
 *
 *  Since the default_handler deals with OPTIONS, all modules can
 *  usually decline to deal with OPTIONS.  TRACE is always allowed,
 *  modules don't need to set it explicitly.
 *
 *  Since the default_handler will always handle a GET, a
 *  module which does *not* implement GET should probably return
 *  HTTP_METHOD_NOT_ALLOWED.  Unfortunately this means that a
       Script GET
 *  handler can't be installed by mod_actions.
 */
apr_int64_t allowed;
/** Array of extension methods */
apr_array_header_t *allowed_xmethods;
/** List of allowed methods */
ap_method_list_t *allowed_methods;

/** byte count in stream is for body */
apr_off_t sent_bodyct;
/** body byte count, for easy access */
apr_off_t bytes_sent;
/** Last modified time of the requested resource */
apr_time_t mtime;

/* HTTP/1.1 connection-level features */

/** sending chunked transfer-coding */
int chunked;
/** The Range: header */
const char *range;
/** The "real" content length */
apr_off_t clength;

/** Remaining bytes left to read from the request body */
apr_off_t remaining;
/** Number of bytes that have been read  from the request body */
apr_off_t read_length;
/** Method for reading the request body
 * (eg. REQUEST_CHUNKED_ERROR, REQUEST_NO_BODY,
 *  REQUEST_CHUNKED_DECHUNK, etc...) */
int read_body;
```

```
    /** reading chunked transfer-coding */
    int read_chunked;
    /** is client waiting for a 100 response? */
    unsigned expecting_100;

    /* 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).
     *
     * The 'notes' apr_table_t is for notes from one module to another,
           with no
     * other set purpose in mind...
     */

    /** MIME header environment from the request */
    apr_table_t *headers_in;
    /** MIME header environment for the response */
    apr_table_t *headers_out;
    /** MIME header environment for the response, printed even on
           errors and
     * persist across internal redirects */
    apr_table_t *err_headers_out;
    /** Array of environment variables to be used for sub processes */
    apr_table_t *subprocess_env;
    /** Notes from one module to another */
    apr_table_t *notes;

   /* content_type, handler, content_encoding, and all content_languages
     * MUST be lowercased strings.  They may be pointers to static
           strings;
     * they should not be modified in place.
     */
    /** The content-type for the current request */
    const char *content_type;   /* Break these out --- we dispatch on
      'em */
    /** The handler string that we use to call a handler function */
    const char *handler;        /* What we *really* dispatch on */

    /** How to encode the data */
    const char *content_encoding;
    /** Array of strings representing the content languages */
    apr_array_header_t *content_languages;

    /** variant list validator (if negotiated) */
    char *vlist_validator;

    /** If an authentication check was made, this gets set to the user
```

```
        name. */
char *user;
/** If an authentication check was made, this gets set to the auth
        type. */
char *ap_auth_type;

/** This response can not be cached */
int no_cache;
/** There is no local copy of this response */
int no_local_copy;

/* What object is being requested (either directly, or via include
 * or content-negotiation mapping).
 */
/** The URI without any parsing performed */
char *unparsed_uri;
/** The path portion of the URI */
char *uri;
/** The filename on disk corresponding to this response */
char *filename;
/* XXX: What does this mean? Please define "canonicalize" -aaron */
/** The true filename, we canonicalize r->filename if these don't
        match */
char *canonical_filename;
/** The PATH_INFO extracted from this request */
char *path_info;
/** The QUERY_ARGS extracted from this request */
char *args;
/**  finfo.protection (st_mode) set to zero if no such file */
apr_finfo_t finfo;
/** A struct containing the components of URI */
apr_uri_t parsed_uri;

/**
 * Flag for the handler to accept or reject path_info on
 * the current request.  All modules should respect the
 * AP_REQ_ACCEPT_PATH_INFO and AP_REQ_REJECT_PATH_INFO
 * values, while AP_REQ_DEFAULT_PATH_INFO indicates they
 * may follow existing conventions.  This is set to the
 * user's preference upon HOOK_VERY_FIRST of the fixups.
 */
int used_path_info;

/* 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).
 */

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

/**
```

```
     * A linked list of the .htaccess configuration directives
     * accessed by this request.
     * N.B. always add to the head of the list, _never_ to the end.
     * that way, a sub request's list can (temporarily) point to a
           parent's list
     */
    const struct htaccess_result *htaccess;

    /** A list of output filters to be used for this request */
    struct ap_filter_t *output_filters;
    /** A list of input filters to be used for this request */
    struct ap_filter_t *input_filters;

    /** A list of protocol level output filters to be used for this
     *  request */
    struct ap_filter_t *proto_output_filters;
    /** A list of protocol level input filters to be used for this
     *  request */
    struct ap_filter_t *proto_input_filters;

    /** A flag to determine if the eos bucket has been sent yet */
    int eos_sent;

/* Things placed at the end of the record to avoid breaking binary
 * compatibility.  It would be nice to remember to reorder the entire
 * record to improve 64bit alignment the next time we need to break
 * binary compatibility for some other reason.
 */
};
```

server_rec

The `server_rec` defines a logical webserver. If virtual hosts are in use[1], each virtualhost has its own `server_rec`, defining it independently of the other hosts. The `server_rec` is created at server startup, and never dies unless the entire httpd is shut down. The `server_rec` does not have its own pool: server resources need to be allocated from the process pool, which is shared by all servers. It does have a configuration vector, as well as server resources including the server name and definition, resources and limits, and logging info.

The `server_rec` is the second most important structure to programmers, after the `request_rec`. It will feature throughout our discussion of module programming, particularly in Chapter 7.

Here is the full definition, from `httpd.h`:

```
/** A structure to store information for each virtual server */
struct server_rec {
    /** The process this server is running in */
    process_rec *process;
    /** The next server in the list */
```

1 Mass virtual hosting configurations use a single server_rec for all vhosts, which is why they don't have the flexibility of normal vhosts.

```
server_rec *next;

/** The name of the server */
const char *defn_name;
/** The line of the config file that the server was defined on */
unsigned defn_line_number;

/* Contact information */

/** The admin's contact information */
char *server_admin;
/** The server hostname */
char *server_hostname;
/** for redirects, etc. */
apr_port_t port;

/* Log files --- note that transfer log is now in the modules... */

/** The name of the error log */
char *error_fname;
/** A file descriptor that references the error log */
apr_file_t *error_log;
/** The log level for this server */
int loglevel;

/* Module-specific configuration for server, and defaults... */

/** true if this is the virtual server */
int is_virtual;
/** Config vector containing pointers to modules' per-server config
 *  structures. */
struct ap_conf_vector_t *module_config;
/** MIME type info, etc., before we start checking per-directory
        info */
struct ap_conf_vector_t *lookup_defaults;

/* Transaction handling */

/** I haven't got a clue */
server_addr_rec *addrs;
/** Timeout, as an apr interval, before we give up */
apr_interval_time_t timeout;
/** The apr interval we will wait for another request */
apr_interval_time_t keep_alive_timeout;
/** Maximum requests per connection */
int keep_alive_max;
/** Use persistent connections? */
int keep_alive;

/** Pathname for ServerPath */
const char *path;
/** Length of path */
int pathlen;
```

```
    /** Normal names for ServerAlias servers */
    apr_array_header_t *names;
    /** Wildcarded names for ServerAlias servers */
    apr_array_header_t *wild_names;

    /** limit on size of the HTTP request line    */
    int limit_req_line;
    /** limit on size of any request header field */
    int limit_req_fieldsize;
    /** limit on number of request header fields  */
    int limit_req_fields;
};
```

conn_rec

The `conn_rec` is Apache's internal representation of a TCP connection. It is created when Apache accepts a connection from a client, and destroyed when the connection is closed. The usual reason for a connection to be made is to serve one or more HTTP requests, so one or more `request_rec` will be instantiated from each `conn_rec`. Most applications will focus on the request and ignore the `conn_rec`, but protocol modules and connection-level filters will need to use the `conn_rec`, and modules may sometimes use it in tasks such as optimising the use of resources over the lifetime of an HTTP Keepalive (persistent connection).

The `conn_rec` has no configuration info, but has a configuration vector for transient data associated with a connection, and a pool for connection resources. It also has connection input and output filter chains, and data describing the TCP connection.

It is important to distinguish clearly between the request and the connection: the former is always a subcomponent of the latter. Apache cleanly represents each as a separate object, with one important exception, which we will deal with in discussing connection filters in Chapter 6.

Here is the full definition from `httpd.h`:

```
/** Structure to store things which are per connection */
struct conn_rec {
    /** Pool associated with this connection */
    apr_pool_t *pool;
    /** Physical vhost this conn came in on */
    server_rec *base_server;
    /** used by http_vhost.c */
    void *vhost_lookup_data;

    /* Information about the connection itself */
    /** local address */
    apr_sockaddr_t *local_addr;
    /** remote address */
    apr_sockaddr_t *remote_addr;

    /** Client's IP address */
    char *remote_ip;
    /** Client's DNS name, if known.  NULL if DNS hasn't been checked,
```

```
     *  "" if it has and no address was found.  N.B. Only access this
            though
     * get_remote_host() */
    char *remote_host;
    /** Only ever set if doing rfc1413 lookups.  N.B. Only access this
            through
     *  get_remote_logname() */
    char *remote_logname;

    /** Are we still talking? */
    unsigned aborted:1;

    /** Are we going to keep the connection alive for another request?
     * @see ap_conn_keepalive_e */
    ap_conn_keepalive_e keepalive;

    /** have we done double-reverse DNS? -1 yes/failure, 0 not yet,
     *  1 yes/success */
    signed int double_reverse:2;

    /** How many times have we used it? */
    int keepalives;
    /** server IP address */
    char *local_ip;
    /** used for ap_get_server_name when UseCanonicalName is set to DNS
     *  (ignores setting of HostnameLookups) */
    char *local_host;

    /** ID of this connection; unique at any point in time */
    long id;
    /** Config vector containing pointers to connections per-server
     *  config structures. */
    struct ap_conf_vector_t *conn_config;
    /** Notes on *this* connection: send note from one module to
     *  another. must remain valid for all requests on this conn */
    apr_table_t *notes;
    /** A list of input filters to be used for this connection */
    struct ap_filter_t *input_filters;
    /** A list of output filters to be used for this connection */
    struct ap_filter_t *output_filters;
    /** handle to scoreboard information for this connection */
    void *sbh;
    /** The bucket allocator to use for all bucket/brigade creations */
    struct apr_bucket_alloc_t *bucket_alloc;
    /** The current state of this connection */
    conn_state_t *cs;
    /** Is there data pending in the input filters? */
    int data_in_input_filters;
};
```

process_rec

Unlike the other core objects discussed above, the `process_rec` is an operating system object rather than a web architecture object. The only time applications need concern

themselves with it is when working with resources having the lifetime of the server, when the process pool serves for all the `server_recs` (and is accessed from a `server_rec s` as `s->process->pool`). The definition is in `httpd.h`, but is not reproduced here.

3 Apache Configuration Basics

Apache configuration is mostly determined at startup, when the server reads `httpd.conf` (and any included files). Configuration data, including resources derived from it by a module (e.g. by opening a file), are stored on each module's configuration records.

Each module has two configuration records:

- The per-server configuration is stored directly on the `server_rec`, so there is one instance per virtual host. The scope of per-server directives is controlled by `<VirtualHost>` containers in `httpd.conf`, but other containers such as `<Location>`, `<Directory>` or `<Files>` will be ignored.
- The per-directory configuration is stored indirectly, and is available to modules via the `request_rec` in the course of processing a request. It is the opposite of per-server configuration: `<Virtualhost>` containers are ignored, but other containers including `<Location>`, `<Directory>` or `<Files>` define the scope of a directive.

To implement a configuration directive, a module has to supply a function that will recognise the directive and set a field in one of the configuration records at system startup time. After startup, the configuration is set and should not be changed. In particular, the configuration records should generally be treated as read-only while processing requests (or connections). Changing configuration data during request processing violates thread-safety (requiring use of programming techniques such as locking), and runs a high risk of other bugs due to the increased complexity. Apache provides a separate configuration record on each `conn_rec` and `request_rec` for transient data.

Chapter 8 describes working with configuration records and data.

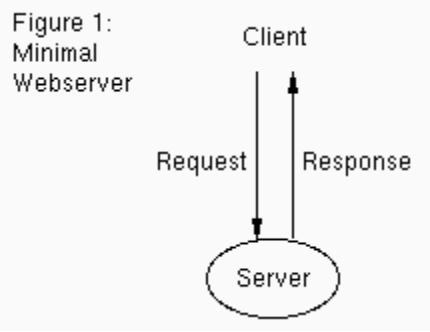
4 Request Processing in Apache

Most, though by no means all, modules are concerned with some aspect of processing an HTTP request. But there is rarely if ever a reason for a module to concern itself with every aspect of HTTP: that is the business of the httpd. The advantage of a modular approach is that a module can easily focus on a particular task but ignore aspects of HTTP that are not relevant to it.

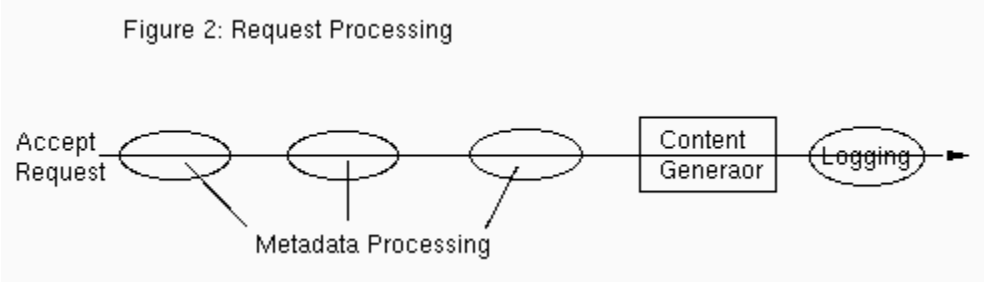
Content Generation

The simplest possible formulation of a webserver is a program that listens for HTTP requests and returns a response when it receives one. In Apache, this is fundamentally the business of a content generator, the core of the webserver.

Exactly one content generator must be run for every HTTP request. Any module may register content generators, normally by defining a function referenced by a handler that can be configured using the `SetHandler` or `AddHandler` directives in `httpd.conf`. The default generator, which is used when no specific generator is defined by any module, simply returns a file mapped directly from the request to the filesystem. Modules that implement content generators are sometimes known as 'content generator' or 'handler' modules.



Request Processing Phases



In principle, a content generator can handle all the functions of a webserver: for example, a CGI program gets the request and produces the response, and can take full control of what happens between them. But in common with other webservers, Apache splits the request into different phases. So, for example, it checks whether the user is authorised to do something before the content generator does that thing.

There are several request phases before the content generator. These serve to examine and perhaps manipulate the request headers, and determine what to do with the request. For example:

- The request URL will be matched against the configuration, to determine what content generator should be used.
- The request URL will be normally mapped to the filesystem. The mapping may be to a static file, a CGI script, or whatever else the content generator may use.
- If content negotiation is enabled, `mod_negotiation` will find the version of the resource that best matches the browser's preference. For example, the Apache manual pages are served in the language requested by the browser.
- Access and Authentication modules will enforce the servers access rules, and determine whether the user is permitted what has been requested.
- `mod_alias` or `mod_rewrite` may change the effective URL in the request.

There is also a request logging phase, which comes after the content generator has sent a reply to the browser.

Processing Hooks

The mechanism by which a module can influence or take charge of some aspect of processing in Apache is through a sequence of hooks. The usual hooks for processing a request in Apache 2.0 are:

post_read_request

The first hook available to modules in normal request processing. This is available to modules that need to hook very early into processing a request.

translate_name

Apache maps the request URL to the filesystem. A module can insert a hook here to substitute its own logic: for example, `mod_alias`.

map_to_storage

Since the URL has been mapped to the filesystem, we are now in a position to apply per-directory configuration (`<Directory>` and `<Files>` sections and their variants, including any relevant `.htaccess` files if enabled). This hook is for Apache to determine the configuration options that apply to this request. This applies normal configuration directives for all active modules, and few modules should ever need to apply hooks here. The only standard module to do so is `mod_proxy`.

header_parser

A hook for inspecting the request headers. Not often used, as modules can do that at any point in the request processing, and they usually do so in the context of another hook. `mod_setenvif` is a standard module that uses a `header_parser`, to set internal environment variables according to the request headers.

access_checker

Apache checks whether access to the requested resource is permitted according to the server configuration (`httpd.conf`). A module can add to or replace Apache's standard logic, which implements the Allow/Deny From directives in `mod_access` (httpd 1.x and 2.0) or `mod_authz_host` (httpd 2.1).

check_user_id

If any authentication method is in use, Apache will apply the relevant authentication and set r->user. A module may implement an authentication method with this hook.

auth_checker

Checks whether the requested operation is permitted to the authenticated user.

type_checker

Applies rules related to the MIME type (where applicable) of the requested resource, and determines the content handler to use (if not already set). Standard modules implementing this hook include `mod_negotiation` (selection of a resource based on HTTP content negotiation) and `mod_mime` (setting MIME type and handler information according to standard configuration directives and conventions such as filename "extensions").

fixups

This is a general-purpose hook for modules to run any necessary processing after the preceding hooks but before the content generator. Like **`post_read_request`** it is something of a catch-all, and is one of the most commonly used hooks:

handler

This is the content generator hook. It is responsible for sending an appropriate response to the client. If there is input data, the `handler` is also responsible for reading it. Unlike other hooks, where zero or many may be involved in processing a request, every request is processed by exactly one `handler`.

log_transaction

Logs the transaction after the response has been returned to the client. A module may modify or replace Apache's standard logging.

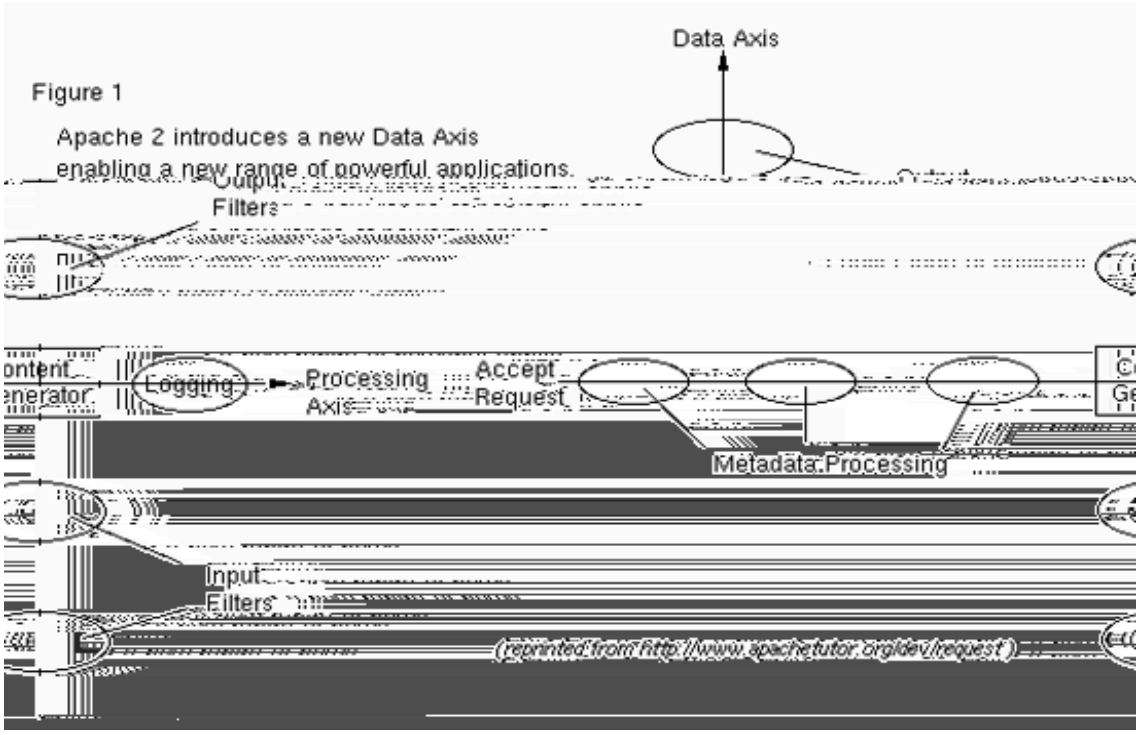
A module may hook its own handlers into any of these processing hooks. Modules that concern themselves with the phases before content generation are known as metadata modules, and will be described in detail in Chapter 5. Those that deal with logging are known as logging modules.

Note that this list may be augmented by modules implementing additional hooks. This will be described in Chapter 7.

The Data Axis and Filters

What we have described so far is essentially similar to the architecture of every general-purpose webserver. There are of course differences in the detail, but the request processing [metadata-> generator->logger] phases are common.

The major innovation in Apache 2, which transforms it from a 'mere' webserver (like Apache 1.3 and others) into a powerful applications platform, is the filter chain. This can be represented as a data axis, orthogonal to the request-processing axis. The request data may be processed by input filters before reaching the content generator, and the response may be processed by output filters before being sent to the client. Filters enable a far cleaner and more efficient implementation of data processing than was possible in the past, as well as separating it from content generation.



Handler or Filter?

Many applications can be implemented either as a handler or a filter. Sometimes it may be clear that one of these solutions is appropriate and the other would be nonsensical, but between these, there is a grey area. How does one decide whether to write a handler or a filter?

There are several questions to consider:

- Feasibility: can it be made to work in both cases? If not, there's an instant decision.
- Utility: is the functionality it provides more useful in one case than the other? Filters are often far more useful than handlers, because they can be re-used with different content generators, and chained both with generators and other filters. But every request has to be processed by some handler, even if it does nothing!
- Complexity: is one version substantially more complex than the other? Will it take more time and effort to develop, and/or run slower? Filter modules are usually more complex than an equivalent handler. This is because a handler is in full control of its data and can read or write at will, whereas a filter has to implement a callback that may get called several times with partial data, which it must treat as unstructured chunks. We will discuss this in detail in Chapter 6.

For example, Apache 1.3 users can do XSLT transformation by building it into handlers, such as CGI or PHP. Or they can use an XSLT module, but this is very slow and cumbersome (this author tried an XSLT module for Apache 1.3, but found it many hundreds of times slower than running XSLT in a CGI script operating on tempfiles).

Running XSLT in a handler works, but loses modularity and reusability: any non-trivial application that needs it has to reinvent that wheel, using whatever libraries are available for the programming or scripting language used and often resorting to ugly hacks such as tempfiles.

Apache 2, by contrast, allows one to run XSLT in a filter. Content handlers requiring XSLT can simply output the XML as-is, and leave the transformation to Apache. The first XSLT module for Apache 2, written by Phillip Dunkel and released while Apache 2.0 was still in beta testing, was at first incomplete, but already worked well. Unlike any of the 1.3 options, it was already fit for operational use. It is now further improved, and is one of a choice of XSLT modules. This author developed another XSLT module.

More generally, if a module has both data inputs and outputs, and if it may be used in more than one application, then it is a strong candidate for implementation as a filter.

Content Generator Examples:

- The default handler simply sends a file from the local disc under the `DocumentRoot`. Although a filter could do that, there's nothing to be gained.
- CGI, the generic API for server-side programming, is a handler. Since CGI scripts expect the central position in the webserver architecture, it has to be a handler. However, a somewhat similar framework for external filters is also provided by `mod_ext_filter`.
- The Apache proxy is a handler that fetches contents from a backend server.
- Any form-processing application will normally be implemented as a handler, particularly those that accept POST data, or other operations that can alter the state of the server itself. Likewise, applications that generate a report from any backend are usually implemented as handlers. However, when the handler is based on HTML or XML pages with embedded programming elements, it can usefully be implemented as a filter.

Filter Examples

- `mod_includes` implements server side includes, a simple scripting language embedded in pages. It is implemented as a filter, so it can post-process contents from any content generator, as discussed with reference to XSLT above.
- `mod_ssl` implements secure transport as a connection-level filter, thereby enabling all normal processing in the server to work with unencrypted data. This is a major advance over Apache 1.x, where secure transport was complex and required a lot of work to combine with other applications.
- Markup parsing modules are used to post-process and transform XML or HTML in more sophisticated ways, from simple link rewriting[2] through XSLT and

2 Http://apache.webthing.com/mod_proxy_html/

Xinclude processing[3] to a complete API for markup filtering[4], to a security filter that blocks attempts to attack vulnerable applications such as PHP scripts[5]. Examples will be introduced in Chapter 6.

- Image processing can take place in a filter. This author developed a custom proxy for a developer of mobile phone browsers. Since the browser tells the proxy its capabilities, images can be reduced to fit within the screen space and, where appropriate, to greyscale, thereby reducing the volume of data sent and accelerating browsing over slow connections.
- Form-processing modules need to decode data sent from a web browser. Input filter modules, such as `mod_form` and `mod_upload`[6], spare applications from reinventing that wheel.
- Data compression and decompression is implemented in `mod_deflate`. The filter architecture allows this to be much simpler than `mod_gzip` (the 1.3 equivalent), and to dispense with the use of tempfiles.

Smart Filtering

The original filtering framework in Apache 2.0 supports a wide range of applications, but leaves problems with configuring the server in some cases. Directives such as `SetOutputFilter` and `AddOutputFilter` work well when one can determine the configuration fully in `httpd.conf`. But what about dynamic or proxied contents? If a proxy inserts an HTML filter, how does it prevent proxied images also being processed as markup? The proxy only knows what the contents will be after it has started to fetch the contents from the backend; but by that time, the filter chain is already set up.

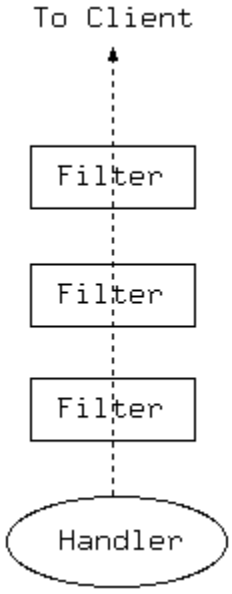


Illustration 1Apache 2.0 Filter Chain

In Apache 2.0, every filter has to take responsibility for this itself if it is to work with dynamic content. The server administrator then has to configure every possible filter, and those that are not relevant uninsert themselves. The `AddOutputFilterByType` directive provides a partial solution, but that is problematic because it is tied into the `ap_set_content_type` function (which may be called several times during a request), and because it cannot deal adequately with situations such as when a filter follows after another filter (e.g. XSLT) that itself affects the type of the data.

From Apache 2.1, a new smart filtering framework is available in `mod_filter`, to enable filters to be configured dynamically according to contents. Filters can now be inserted when and only when some aspect of a request (HTTP request or response headers, and/or

3 Http://www.outoforder.cc/projects/apache/mod_transform
4 Http://apache.webthing.com/xmlns.html
5 Http://modsecurity.org/
6 Http://apache.webthing.com/

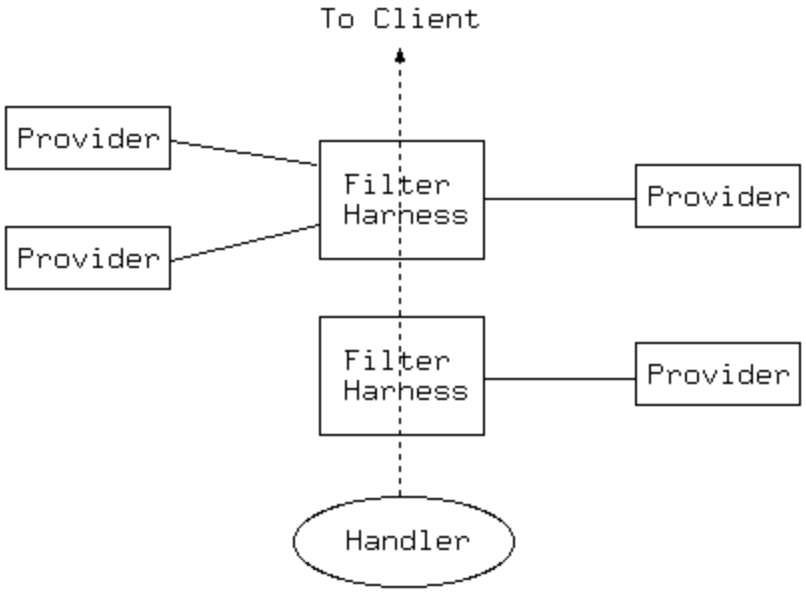


Illustration 2Smart Filtering in Apache 2.1+

internal Apache variables set using some other means such as `mod_rewrite`) suggests that they are appropriate. The decision to insert a filter takes place exactly when the first chunk of data is available for filtering, so uncertainties are dealt with no matter where they arise.

Filters written for Apache 2.0 can be used as *providers* in the new framework: they will require recompilation, but not source changes. New filters for Apache 2.1/2.2 can also be simplified if back-compatibility with 2.0 is not required. We will discuss this further in Chapter 6.

Order of Processing

Before going on to discuss how a module hooks itself into any of the stages of processing a request/data, we should pause to clear up a matter that often causes confusion , namely, the order of processing.

The request processing axis is straightforward: phases happen strictly in order. But confusion arises in the data axis. For maximum efficiency, this is pipelined, so the content generator and filters do not run in a deterministic order. So, for example, you cannot in general set something in an input filter and expect it to apply in the generator or output filters.

The order of processing is in fact centred on the content generator, which is responsible for pulling data from the input filter stack and pushing data to the output filters (where applicable, in both cases). When a generator or filter needs to set something affecting the request as a whole, it must do so before passing any data down the chain (generator and output filters), or before returning data to the caller (input filters).

Processing Hooks

Now that we have an overview of request processing in Apache, we can proceed to show how a module hooks into it to play a part.

The Apache module structure declares several (optional) data and function members:

```
module AP_MODULE_DECLARE_DATA my_module = {
  STANDARD20_MODULE_STUFF,
  my_dir_conf,
  my_dir_merge,
  my_server_conf,
  my_server_merge,
  my_cmds,
  my_hooks
} ;
```

The relevant function for the module to create request processing hooks is the final member:

```
static void my_hooks(apr_pool_t* pool) {
  /* create request processing hooks as required */
}
```

What hooks we need to create here depend on what part or parts of the request our module is interested in. For example, a module that implements a content generator (handler) will need a handler hook, looking something like:

```
  ap_hook_handler(my_handler, NULL, NULL, APR_HOOK_MIDDLE) ;
```

Now `my_handler` will be called when a request reaches the content generation phase. Hooks for other request phases are similar.

The prototype for a handler for any of these phases is:

```
static int my_handler(request_rec* r) {
  /* do something with the request */
}
```

Details and implementation are discussed in Chapters 4 and 5.

5 MPMs

The only thing an applications developer needs to know about MPMs is this: ignore them!

The old NCSA server, and Apache 1, grew up in a Unix environment. It was a multi-process server, where each client would be serviced by one server instance. If there were more concurrent clients than server processes, Apache would fork additional server processes to deal with them. Under normal operation, Apache would maintain a pool of

available server processes to deal with incoming requests.

Whereas this works well on Unix-family systems, it is an inefficient solution on platforms such as Windows, where forking a process is an expensive operation. So making Apache truly cross-platform required another solution. The approach adopted for Apache 2 is to make the core processing into a pluggable module – the Multi-Processing Model (MPM) – which can be optimised for different environments. But the MPM architecture also allows different Apache models to coexist even within a single operating system, thus providing users with options for different usages.

In practice, only Unix-family operating systems offer a useful choice: other systems, such as Windows, Netware and BeOS, each have a single MPM optimised for them. Unix has two production-quality MPMs available as standard, a third that is under active development and may be declared stable in Apache 2.2, and several experimental options unsuitable for production use:

- The **Prefork** MPM is a non-threaded model essentially similar to Apache 1.x. It is a safe option in all cases, and for servers running non-threadsafe software such as PHP it is the only safe option. For some applications, including many of those popular with Apache 1.3 (simple static pages, CGI scripts) it may be as good as anything.
- The **Worker** MPM is a threaded model, with advantages including lower memory usage (important on busy servers), and enabling much greater scalability than Prefork in certain types of application. We will discuss some such cases later in this book when we introduce SQL support and `mod_dbd`.
- Both of the stable MPMs suffer from a limitation that affects very busy servers. Whereas HTTP Keepalive is necessary to reduce TCP connection and network overhead, it ties up a server process or thread while the keepalive is active. This means that a very busy server may run out of available threads. The **Event** MPM is a new model that deals with the problem by decoupling the server thread from the connection. Cases where the Event MPM may prove most useful are servers with extremely high hit rates but where the server processing is fast, so that the number of available threads is a critical resource limitation. A busy server with Worker MPM may sustain tens of thousands of hits per second (as happens, for example, with popular news outlets at peak times), but the Event MPM might help to handle high loads more easily.
- There are also several experimental MPMs for Unix that are not, at the time of writing, under active development, and may or may not ever be completed. Amongst these, the **Perchild** MPM, which runs servers for different virtual hosts under different user ids, would have many real-life applications. There are a number of alternatives: the third-party Metux MPM[7], `mod_ruid`[8] (for Linux only) will fulfil a similar function, and for running external programs, other options include `fastcgi/mod_fcgid`[9], and `suexec` (CGI). The author does not have

7 Http://www.metux.de/mpm/
8 http://websupport.sk/~stanojr/projects/mod_ruid/
9 http://fastcgi.coremail.cn/

personal knowledge of any of the third-party solutions, and so cannot make recommendations.

STOP PRESS: the perchild MPM project may be revived as part of the Google "Summer of Code", 2005!

Applications developed for Apache should normally be MPM-agnostic. Since MPM internals are not exposed in the API, this is basically straightforward, provided programmers observe basic rules of good practice (mainly: write thread-safe, re-entrant code). This is in fact closely related to the broader question of developing platform-independent code: indeed, it is sometimes useful to regard the MPM, rather than the operating system, as the applications platform.

Sometimes an application is naturally better-suited to some MPMs than others: for example, database-driven or load-balancing applications benefit substantially from connection pooling (discussed later in this book) and therefore from a threaded MPM, whereas forking a child process (the original CGI implementation or `mod_ext_filter`) is a substantial overhead in a threaded program, and therefore works best with the Prefork MPM. Nevertheless, an application should work even when used with a suboptimal MPM, unless there are strong reasons to limit it..

6 The Apache Portable Runtime (APR) Library

The APR library provides a range of data types and functions used by the Apache core and available to modules. Its most important functions are to support cross-platform, cross MPM development, and to provide capabilities needed by Apache and Apache-based applications. Working with the APR is a fundamental to programming in C with Apache, and is the subject of the next chapter.