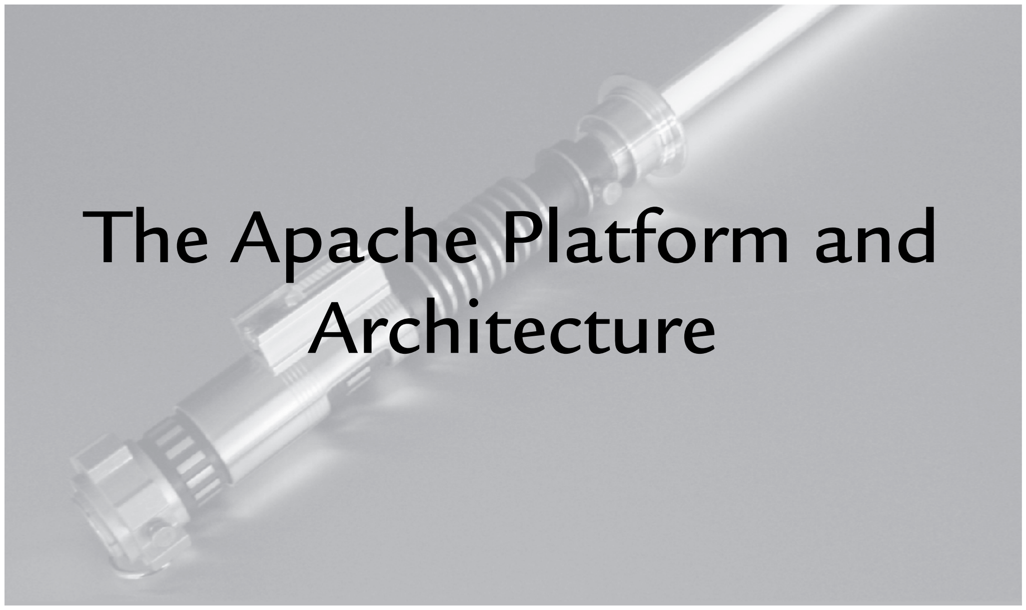
Kew_CH02.qxd 12/19/06 9:19 AM Page 21

Chapter 2



The Apache Platform and Architecture

Apache runs as a permanent background task: a daemon (UNIX) or service (Windows). Start-up 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 (run from a generic superserver for every incoming request), but this mode was never appropriate for operational use.

2.1 Overview

The Apache HTTP Server comprises a relatively small core, together with a number of modules (Figure 2-1). Modules may be compiled statically into the server or, more commonly, held in a `/modules/` or `/libexec/` directory and loaded dynamically at runtime. In addition, the server relies on the Apache Portable Runtime (APR) libraries, which provide a cross-platform operating system layer and utilities,

Kew_CH02.qxd 12/19/06 9:19 AM Page 22

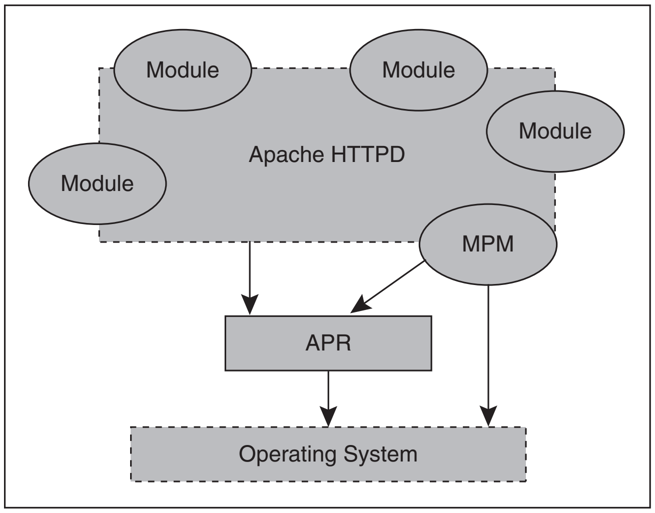


FIGURE 2-1
Apache architecture

so that modules don't have to rely on non-portable operating system calls. A special-purpose module, the Multi-Processing Module (MPM), serves to optimize Apache for the underlying operating system. The MPM should normally be the only module to access the operating system other than through the APR.

2.2 Two-Phase Operation

Apache operation proceeds in two phases: start-up and operational. System start-up takes place as `root`, and includes parsing the configuration file(s), loading modules, and initializing 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 basic security measure helps to prevent a simple bug in Apache (or a module or script) from becoming a devastating system vulnerability, like those exploited by malware such as "Code Red" and "Nimda" in MS IIS.

This two-stage operation has some implications for applications architecture. First, anything that requires system privileges must be run at system start-up. Second, it is

Kew_CH02.qxd 12/19/06 9:19 AM Page 23

good practice to run as much initialization as possible at start-up, so as to minimize the processing required to service each request. Conversely, because so many slow and expensive operations are concentrated in system start-up, it would be hugely inefficient to try to run Apache from a generic server such as `inetd` or `tcpserver`.

One non-intuitive quirk of the architecture is that the configuration code is, in fact, executed twice at start-up (although not at restart). The first time through checks that the configuration is valid (at least to the point that Apache *can* successfully start); the second pass is "live" and leads into the operational phase. Most modules can ignore this behavior (standard use of APR pools ensures that it doesn't cause a resource leak), but it may have implications for some modules. For example, a module that dynamically loads new code at start-up may want to do so just once and, therefore, must use a technique such as setting and checking a static flag to ensure that critical initialization takes place just once.

2.2.1 Start-up Phase

The purpose of Apache's start-up phase is to read the configuration, load modules and libraries, and initialize required resources. Each module may have its own resources, and has the opportunity to initialize those resources. At start-up, Apache runs as a single-process, single-thread program and has full system privileges.

2.2.1.1 Configuration

Apache's main configuration file is normally called `httpd.conf`. However, this nomenclature is just a convention, and third-party Apache distributions such as those provided as `.rpm` or `.deb` packages may use a different naming scheme. In addition, `httpd.conf` may be a single file, or it may be distributed over several files using the `Include` directive to include different configuration files. Some distributions have highly intricate configurations. For example, Debian GNU/Linux ships an Apache configuration that relies heavily on familiarity with Debian, rather than with Apache. It is not the purpose of this book to discuss the merits of different layouts, so we'll simply call this configuration file `httpd.conf`.

The `httpd.conf` configuration file is a plain text file and is parsed line-by-line at server start-up. The contents of `httpd.conf` comprise directives, containers, and comments. Blank lines and leading whitespace are also allowed, but will be ignored.

Kew_CH02.qxd 12/19/06 9:19 AM Page 24

Directives

Most of the contents of `httpd.conf` are directives. A directive may have zero or more arguments, separated by whitespace. Each directive determines its own syntax, so different directives may permit different numbers of arguments, and different argument types (e.g., string, numeric, enumerated, Boolean on/off, or filename). Each directive is implemented by some module or the core, as described in Chapter 9.

For example:

```
LoadModule foo_module modules/mod_foo.so
```

This directive is implemented by `mod_so` and tells it to load a module. The first argument is the module name (string, alphanumeric). The second argument is a filename, which may be absolute or relative to the server root.

```
DocumentRoot /usr/local/apache/htdocs
```

This directive is implemented by the core, and sets the directory that is the root of the main document tree visible from the Web.

```
SetEnv hello "Hello, World!"
```

This directive is implemented by `mod_env` and sets an environment variable. Note that because the second argument contains a space, we must surround it with quotation marks.

```
Choices On
```

This directive is implemented by `mod_choices` (Chapter 6) and activates that module's options.

Containers

A container is a special form of directive, characterized by a syntax that superficially resembles markup, using angle brackets. Containers differ semantically from other directives in that they comprise a start and an end on separate lines, and they affect directives falling between the start and the end of the container. For example, the `<VirtualHost>` container is implemented by the core and defines a virtual host:

```
<VirtualHost 10.31.2.139>
  ServerName www.example.com
  DocumentRoot /usr/www/example
  ServerAdmin webmaster@example.com
  CustomLog /var/log/www/example.log
</VirtualHost>
```

Kew_CH02.qxd 12/19/06 9:19 AM Page 25

The container provides a *context* for the directives within it. In this case, the directives apply to requests to `www.example.com`, but not to requests to any other names this server responds to. Containers can be nested unless a module explicitly prevents it. Directives, including containers, may be context sensitive, so they are valid only in some specified type of context.

Comments

Any line whose first character is a hash is read as a comment.

```
# This line is a comment
```

A hash within a directive doesn't in general make a comment, unless the module implementing the directive explicitly supports it.

If a module is not loaded, directives that it implements are not recognized, and Apache will stop with a syntax error when it encounters them. Therefore `mod_so` must be statically linked to load other modules. This is pretty much essential whenever you're developing new modules, as without `LoadModule` you'd have to rebuild the entire server every time you change your module!

2.2.2 Operational Phase

At the end of the start-up phase, control passes to the Multi-Processing Module (see Section 2.3). The MPM is responsible for managing Apache's operation at a systems level. It typically does so by maintaining a pool of worker processes and/or threads, as appropriate to the operating system and other applicable constraints (such as optimization for a particular usage scenario). The original process remains as "master," maintaining a pool of worker children. These workers are responsible for servicing incoming connections, while the parent process deals with creating new children, removing surplus ones as necessary, and communicating signals such as "shut down" or "restart."

Because of the MPM architecture, it is not possible to describe the operational phase in definite terms. Whereas the standard MPMs use worker children in some manner, they are not constrained to work in only one way. Thus another MPM could, in principle, implement an entirely different server architecture at the system level.

Kew_CH02.qxd 12/19/06 9:19 AM Page 26

2.2.3 Shutdown

There is no shutdown phase as such. Instead, anything that needs be done on shutdown is registered as a cleanup, as described in Chapter 3. When Apache stops, all registered cleanups are run.

2.3 Multi-Processing Modules

At the end of the start-up phase, after the configuration has been read, overall control of Apache passes to a Multi-Processing Module. The MPM provides the interface between the running Apache server and the underlying operating system. Its primary role is to optimize Apache for each platform, while ensuring the server runs efficiently and securely.

As indicated by the name, the MPM is itself a module. But the MPM is uniquely a *systems-level* module (so developing an MPM falls outside the scope of a book on *applications* development). Also uniquely, every Apache instance must contain exactly one MPM, which is selected at build-time.

2.3.1 Why MPMs?

The old NCSA server, and Apache 1, grew up in a UNIX environment. It was a multiprocess 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 scheme works well on UNIX-family[1] 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 turn the core processing into a pluggable module, the MPM, which can be optimized for different environments. The MPM architecture also allows different Apache models to coexist even within a single operating system, thus providing users with options for different usages.

1. Here and elsewhere in this book, terms such as "UNIX-family" imply both UNIX itself and other POSIX-centered operating systems such as Linux and MacOSX.

Kew_CH02.qxd 12/19/06 9:19 AM Page 27

In practice, only UNIX-family operating systems offer a useful[2] choice: Other supported platforms (Windows, Netware, OS/2, BeOS) have a single MPM optimized for each platform. UNIX has two production-quality MPMs (Prefork and Worker) available as standard, a third (Event) that is thought to be stable for non-SSL uses in Apache 2.2, and several experimental options unsuitable for production use. Third-party MPMs are also available.

2.3.2 The UNIX-Family MPMs

- The **Prefork** MPM is a nonthreaded model essentially similar to Apache 1.x. It is a safe option in all cases, and for servers running non-thread-safe software such as PHP, it is the only safe option. For some applications, including many of those popular with Apache 1.3 (e.g., simple static pages, CGI scripts), this MPM may be as good as anything.[3]
- The **Worker** MPM is a threaded model, whose advantages include lower memory usage (important on busy servers) and much greater scalability than that provided by Prefork in certain types of applications. We will discuss some of these cases later when we introduce SQL database 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. As a consequence, a very busy server may run out of available threads. The **Event** MPM is a new model that deals with this 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 for which the server processing is fast, so that the number of available threads is a critical resource limitation. A busy server with the 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. Note that the Event MPM will *not* work with secure HTTP (HTTPS).

2. MPMs are not necessarily tied to an operating system (most systems have some kind of POSIX support and might be able to use it to run Prefork, for instance). But if you try to build Apache with a "foreign" MPM, you're on your own!

3. This depends on the platform. On Linux versions without NPTL, Prefork is commonly reported to be as fast as Worker. On Solaris, Worker is reported to be much faster than Prefork. Your mileage may vary.

Kew_CH02.qxd 12/19/06 9:19 AM Page 28

- There are also several experimental MPMs for UNIX that are not, at the time of this book's writing, under active development; they may or may not ever be completed. The **Perchild** MPM promised a much-requested feature: It runs servers for different virtual hosts under different user IDs. Several alternatives offer similar features, including the third-party **Metux**[4] and **Peruser**[5] MPMs, and (for Linux only) **`mod_ruid`**.[6] For running external programs, other options include **`fastcgi/mod_fcgid`**[7] and **`suexec`** (CGI). The author does not have personal knowledge of these third-party solutions and so cannot make recommendations about them.

2.3.3 Working with MPMs and Operating Systems

The one-sentence summary: MPMs are invisible to applications and should be ignored!

Applications developed for Apache should normally be MPM-agnostic. Given that MPM internals are not part of the API, this is basically straightforward, provided programmers observe basic rules of good practice (namely, write thread-safe, cross-process-safe, reentrant code), as briefly discussed in Chapter 4. This issue is 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. In contrast, forking a child process (the original CGI implementation or `mod_ext_filter`) creates greater 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 compelling reasons to limit it.

If you wish to run Apache on an operating system that is not yet supported, the main task is to add support for your target platform to the APR, which provides the operating system layer. A custom MPM may or may not be necessary, but is likely to deliver better performance than an existing one. From the point of view of

4. `http://www.metux.de/mpm/`
5. `http://www.telana.com/peruser.php`
6. `http://websupport.sk/~stanojr/projects/mod_ruid/`
7. `http://fastcgi.coremail.cn/`

Kew_CH02.qxd 12/19/06 9:19 AM Page 29

Apache, this is a systems programming task, and hence it falls outside the scope of an applications development book.

2.4 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 the operating system rather than the application architecture. Each of these basic units is represented by a core data structure defined in the header file `httpd.h` and, like other core objects we encounter in applications development, is completely independent of the MPM in use.

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 they are no longer required. Pools tie resources to the lifetime of one of the core objects. We will describe pools in depth in Chapter 3.
- Configuration records are used by each module to tie its own data to one of the core objects. The core data structures 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, as discussed in Chapters 4 and 9.

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.

Kew_CH02.qxd 12/19/06 9:19 AM Page 30

2.4.1 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` object is passed to every handler implemented by any module in the course of processing a request (as discussed in Chapters 5 and 6). It holds all of the internal data relevant to processing an HTTP request. It also includes 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. It is used to manage resources allocated while processing the request.
- A vector of configuration records for static request configuration (per-directory data specified in `httpd.conf` or `.htaccess`).
- A vector of configuration records for transient data used in processing.
- Tables of HTTP input, output, and error headers.
- A 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 other relevant objects, including the connection, the server, and any related request objects.
- Pointers to the input and output filter chains (discussed in Chapter 8).
- The URI requested, and the internal parsed representation of it, including the handler (see Chapter 5) and filesystem mapping (see Chapter 6).

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;
```

Kew_CH02.qxd 12/19/06 9:19 AM Page 31

```
/** 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 (e.g., GET, HEAD, POST, etc.) */
const char *method;
/** M_GET, M_POST, etc. */
int method_number;

/**
 *  'allowed' is a bit-vector 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
```

Kew_CH02.qxd 12/19/06 9:19 AM Page 32

```
     *  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
     * (e.g., 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.
     *
```

Kew_CH02.qxd 12/19/06 9:19 AM Page 33

```
     * 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 subprocesses */
    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 cannot 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 */
```

Kew_CH02.qxd 12/19/06 9:19 AM Page 34

```
char *unparsed_uri;
/** The path portion of the URI */
char *uri;
/** The filename on disk corresponding to this response */
char *filename;
/** 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 */
```

Kew_CH02.qxd 12/19/06 9:19 AM Page 35

```
    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 64-bit alignment the next time we need to break
 * binary compatibility for some other reason.
 */
};
```

2.4.2 server_rec

The server_rec defines a logical webserver. If virtual hosts are in use,[8] each virtual host has its own server_rec, defining it independently of the other hosts. The server_rec is created at server start-up, and it never dies unless the entire httpd is shut down. The server_rec does not have its own pool; instead, 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 information.

The server_rec is the second most important structure to programmers, after the request_rec. It will feature prominently throughout our discussion of module programming.

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 */
    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;
```

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

Kew_CH02.qxd 12/19/06 9:19 AM Page 36

```
/* 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;
```

Kew_CH02.qxd 12/19/06 9:19 AM Page 37

```
    /** 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;
};
```

2.4.3 conn_rec

The `conn_rec` object is Apache's internal representation of a TCP connection. It is created when Apache accepts a connection from a client, and later it is 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` structures 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 optimizing the use of resources over the lifetime of an HTTP Keepalive (persistent connection).

The `conn_rec` has no configuration information, but has a configuration vector for transient data associated with a connection as well as a pool for connection resources. It also has connection input and output filter chains, plus 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 8.

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;
```

Kew_CH02.qxd 12/19/06 9:19 AM Page 38

```
    /** 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 through
     * 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;
};
```

Kew_CH02.qxd 12/19/06 9:19 AM Page 39

2.4.4 process_rec

Unlike the other core objects discussed earlier, 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 they are working with resources having the lifetime of the server, when the process pool serves all of the `server_rec` objects (and is accessed from a `server_rec` as `s->process->pool`). The definition appears in `httpd.h`, but is not reproduced here.

2.5 Other Key API Components

The header file `httpd.h` that defines these core structures is but one of many API header files that the applications developer will need to use. These fall into several loosely bounded categories that can be identified by naming conventions:

- **`ap_`** header files generally define low-level API elements and are usually (though not always) accessed indirectly by inclusion in other headers.
- **`http_`** header files define most of the key APIs likely to be of interest to application developers. They are also exposed in scripting languages through modules such as `mod_perl` and `mod_python`.
- **`util_`** header files define API elements at a higher level than **`ap_`**, but are rarely used directly by application modules. Two exceptions to that rule are `util_script.h` and `util_filter.h`, which define scripting and filtering APIs, respectively, and are commonly accessed by modules.
- **`mod_`** header files define APIs implemented by modules that are optional. Using these APIs may create dependencies. Best practice is discussed in Chapter 10.
- **`apr_`** header files define the APR APIs. The APR libraries are external but essential to the webserver, and the APR is required (directly or indirectly) by any nontrivial module. The APR is discussed in Chapter 3.
- Other header files generally define system-level APIs only.
- Third-party APIs may follow similar conventions (e.g., a **`mod_`** header file) or adopt their own.

Kew_CH02.qxd 12/19/06 9:19 AM Page 40

As noted earlier, the primary APIs for application modules are the **`http_*`** header files.

- **`http_config.h`**—Defines the configuration API, including the configuration data structures, the configuration vectors, any associated accessors, and, in particular, the main APIs presented in Chapter 9. It also defines the module data structure itself and associated accessors, and the handler (content generator) hook. It is required by most modules.
- **`http_connection.h`**—Defines the (small) TCP connection API, including connection-level hooks. Most modules will access the connection through the `conn_rec` object, so this API is seldom required by application modules.
- **`http_core.h`**—Defines miscellaneous APIs exported by the Apache core, such as accessor functions for the `request_rec` object. It includes APIs exported for particular modules, such as to support `mod_perl`'s configuration sections. This header file is rarely required by application modules.
- **`http_log.h`**—Defines the error logging API and piped logs. Modules will need it for the error reporting functions and associated macros.
- **`http_main.h`**—Defines APIs for server start-up. It is unlikely to be of interest to modules.
- **`http_protocol.h`**—Contains high-level functional APIs for performing a number of important operations, including all normal I/O to the client, and for dealing with aspects of the HTTP protocol such as generating the correct response headers. It also exports request processing hooks that fall outside the scope of `http_request`. Many modules will require this header file—for example, content generators (unless you use only the lower-level APIs) and authentication modules.
- **`http_request.h`**—Defines the main APIs discussed in Chapter 6. It exports most of the request processing hooks, and the subrequest and internal redirect APIs. It is required by some, but not all, modules.
- **`http_vhost.h`**—Contains APIs for managing virtual hosts. It is rarely needed by modules except those concerned with virtual host configuration.
- **`httpd.h`**—Contains Apache's core API, which is required by (probably) all modules. It defines a lot of system constants, some of them derived from local build parameters, and various APIs such as HTTP status codes and methods. Most importantly, it defines the core objects mentioned earlier in this chapter.

Kew_CH02.qxd 12/19/06 9:19 AM Page 41

Other important API headers we will encounter include the following files:

- **`util_filter.h`**—The filter API, required by all filter modules (Chapter 8)
- **`ap_provider.h`**—The provider API (Chapter 10)
- **`mod_dbd.h`**—The DBD framework (Chapters 10 and 11)

Other API headers likely to be of interest to application developers include the following files:

- **`util_ldap.h`**—The LDAP API
- **`util_script.h`**—A scripting environment that originally supported CGI, but is also used by other modules that use CGI environment variables (e.g., `mod_rewrite`, `mod_perl`, `mod_php`) or that generate responses using CGI rules (e.g., `mod_asis`)

2.6 Apache Configuration Basics

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

Each module has two configuration records, either or both of which may be null (unused):

- 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>`, and `<Files>` will be ignored.
- The per-directory configuration is stored indirectly and is available to modules via the `request_rec` object in the course of processing a request. It is the opposite of per-server configuration: Its scope is defined by containers such as `<Location>`, `<Directory>`, and `<Files>`.

To implement a configuration directive, a module must supply a function that will recognize the directive and set a field in one of the configuration records at system start-up time. After start-up, 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 introducing other bugs due to the increased complexity. Apache provides a separate configuration record on each `conn_rec` and `request_rec` for transient data.

Chapter 9 describes working with configuration records and data.

2.7 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.

2.7.1 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 (Figure 2-2). In Apache, this job is fundamentally the business of a content generator, the core of the webserver.

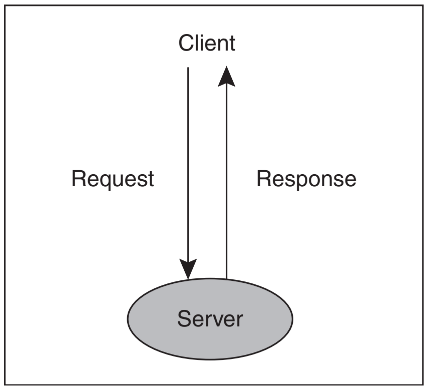


FIGURE 2-2
Minimal webserver

Kew_CH02.qxd 12/19/06 9:19 AM Page 43

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.

2.7.2 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 it can take full control of what happens between them. Like other webservers, Apache splits the request into different phases. For example, it checks whether the user is authorized to do something before the content generator does that thing.

Several request phases precede the content generator (Figure 2-3). These serve to examine and perhaps manipulate the request headers, and to determine what to do with the request. For example:

- The request URL will be matched against the configuration, to determine which content generator should be used.
- The request URL will normally be 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 server's 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.

Kew_CH02.qxd 12/19/06 9:19 AM Page 44

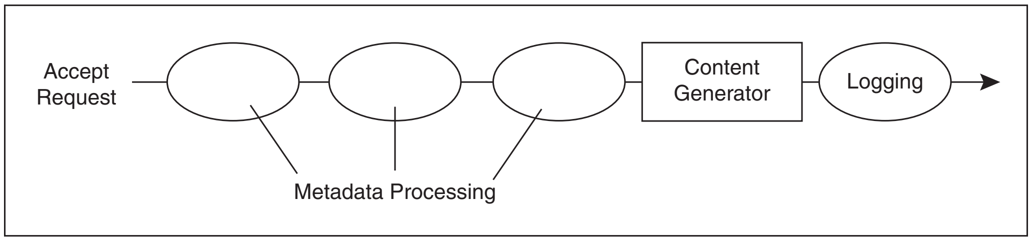


FIGURE 2-3
Request processing in Apache

2.7.2.1 Nonstandard Request Processing

Request processing may sometimes be diverted from the standard processing axis described here, for a variety of reasons:

- A module may divert processing into a new request or error document at any point before the response has been sent (Chapter 6).
- A module may define additional phases and enable other modules to hook their own processing in (Chapter 10).
- There is a `quick_handler` hook that bypasses normal processing, used by `mod_cache` (not discussed in this book).

2.7.3 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 described next.

`post_read_request`—This is the first hook available to modules in normal request processing. It 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 enables Apache to determine the configuration options that apply to this

Kew_CH02.qxd 12/19/06 9:19 AM Page 45

request. It applies normal configuration directives for all active modules, so few modules should ever need to apply hooks here. The only standard module to do so is `mod_proxy`.

`header_parser`—This hook inspects the request headers. It is rarely used, as modules can perform that task at any point in the request processing, and they usually do so within 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 2.2).

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

`auth_checker`—This hook checks whether the requested operation is permitted to the authenticated user.

`type_checker`—This hook 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 the MIME type and handler information according to standard configuration directives and conventions such as filename "extensions").

`fixups`—This general-purpose hook enables 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 are input data, the `handler` is also responsible for reading them. Unlike the other hooks, where zero or many functions may be involved in processing a request, every request is processed by exactly one handler.

`log_transaction`—This hook logs the transaction after the response has been returned to the client. A module may modify or replace Apache's standard logging.

Kew_CH02.qxd 12/19/06 9:19 AM Page 46

A module may hook its own handlers into any of these processing phases. The module provides a callback function and hooks it in, and Apache calls the function during the appropriate processing phase. Modules that concern themselves with the phases before content generation are sometimes known as metadata modules; they are described in detail in Chapter 6. Modules that deal with logging are known as logging modules. In addition to using the standard hooks, modules may define further processing hooks, as described in Chapter 10.

2.7.4 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 details, 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. The filter chain can be represented as a data axis, orthogonal to the request-processing axis (Figure 2-4). 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 content generation from its transformation and aggregation.

2.7.4.1 Handler or Filter?

Many applications can be implemented as either 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 extremes lies a gray area. How does one decide whether to write a handler or a filter?

When making this decision, 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 reused 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!

Kew_CH02.qxd 12/19/06 9:19 AM Page 47

- Complexity: Is one version substantially more complex than the other? Will it take more time and effort to develop, and/or run more slowly? Filter modules are usually more complex than the equivalent handler, 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 be called several times with partial data, which it must treat as unstructured chunks. We will discuss this issue in detail in Chapter 8.

For example, Apache 1.3 users can do an XSLT transformation by building it into handlers, such as CGI or PHP. Alternatively, 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 temporary files). Running XSLT in a handler works, but loses modularity and reusability. Any nontrivial application that needs it has to reinvent that

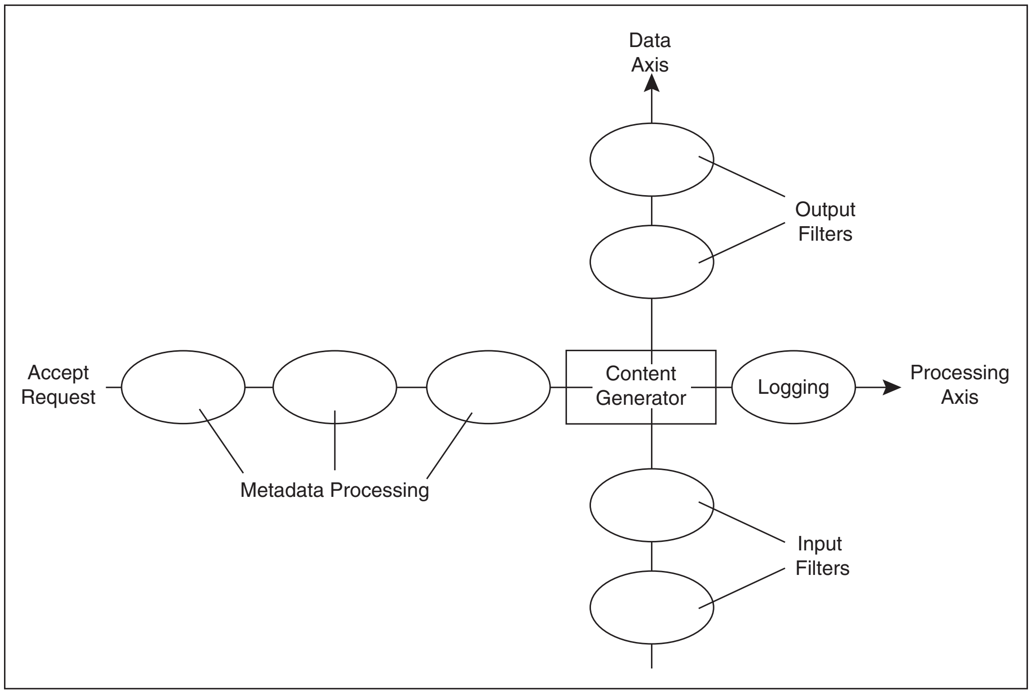


FIGURE 2-4
Apache 2 introduces a new data axis enabling a new range of powerful applications

Kew_CH02.qxd 12/19/06 9:19 AM Page 48

wheel, using whatever libraries are available for the programming or scripting language used and often resorting to ugly hacks such as temporary files.

Apache 2, by contrast, allows us 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 initially incomplete, but already worked far better than XSLT in Apache 1.3. It is now further improved, and is one of a choice of XSLT modules. This book's 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.

2.7.4.2 Content Generator Examples

- The default handler sends a file from the local disk 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. Because 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 back-end 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 back end 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.

2.7.4.3 Filter Examples

- `mod_include` implements server-side includes, a simple scripting language embedded in pages. It is implemented as a filter, so it can post-process content from any content generator, as discussed earlier with reference to XSLT.

Kew_CH02.qxd 12/19/06 9:19 AM Page 49

- `mod_ssl` implements secure transport as a connection-level filter, thereby enabling all normal processing in the server to work with unencrypted data. This represents a major advance over Apache 1.x, where secure transport was complex and required a lot of work to combine it with other applications.
- Markup parsing modules are used to post-process and transform XML or HTML in more sophisticated ways, from simple link rewriting[9] through XSLT and Xinclude processing,[10] to a complete API for markup filtering,[11] to a security filter that blocks attempts to attack vulnerable applications such as PHP scripts.[12] Examples will be introduced in Chapter 8.
- Image processing can take place in a filter. This author developed a custom proxy for a developer of mobile phone browsers. Because the browser tells the proxy its capabilities, images can be reduced to fit within the screen space and, where appropriate, translated to gray scale, 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`,[13] spare applications from reinventing that wheel.
- Data compression and decompression are implemented in `mod_deflate`. The filter architecture allows this module to be much simpler than `mod_gzip` (an Apache 1.3 compression module) and to dispense with any use of temporary files.

2.7.5 Order of Processing

Before moving 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.

9. `http://apache.webthing.com/mod_proxy_html/`
10. `http://www.outoforder.cc/projects/apache/mod_transform`
11. `http://apache.webthing.com/xmlns.html`
12. `http://modsecurity.org/`
13. `http://apache.webthing.com/`

Kew_CH02.qxd 12/19/06 9:19 AM Page 50

The request processing axis is straightforward, with phases happening strictly in order. But confusion arises in the data axis. For maximum efficiency, this axis is pipelined, so the content generator and filters do not run in a deterministic order. 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 centers on the content generator, which is responsible for pulling data from the input filter stack and pushing data onto 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).

2.7.6 Processing Hooks

Now that we have an overview of request processing in Apache, we can 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,  /* macro to ensure version consistency */
    my_dir_conf,              /* create per-directory configuration record */
    my_dir_merge,             /* merge per-directory configuration records */
    my_server_conf,           /* create per-server configuration record */
    my_server_merge,          /* merge per-server configuration records */
    my_cmds,                  /* configuration directives */
    my_hooks                  /* register modules functions with the core */
};
```

The configuration directives are presented as an array; the remaining module entries are functions. 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 */
}
```

Which hooks we need to create here depend on which 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 this:

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

Kew_CH02.qxd 12/19/06 9:19 AM Page 51

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

The following prototype applies to a handler for any of these phases:

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

Details and implementation of this prototype are discussed in Chapters 5 and 6.

2.8 Summary

This basic introduction to the Apache platform and architecture sets the scene for the following chapters. We have now looked at the following aspects of Apache:

- The Apache architecture, and its relationship to the operating system
- The roles of the principal components: MPMs, APR, and modules
- The separation of tasks into initialization and operation
- The fundamental Apache objects and (briefly) the API header files
- Configuration basics
- The request processing cycle
- The data axis and filter architecture

Nothing in this general overview is specific to C programming, so Chapter 2 should be equally relevant to scripting languages. Together with the next two chapters (on the APR and programming techniques, respectively), it provides the essential basis for understanding the core information and advanced topics covered in Chapters 5–11. In those chapters, the concepts introduced here are examined in more detail, and demonstrated in the context of developing real applications.