Writing a Content Generator > The HelloWorld Module

Chapter 5. Writing a Content Generator

In principle, one can do anything with the Common Gateway Interface (CGI). But the range of problems for which CGI provides a good solution is much smaller! The same is true of a content generator in Apache. It lies at the heart of processing a request and of building a web application. Indeed, it can be extended to do anything that the underlying system permits the webserver to do. The content generator is the most basic kind of module in Apache.

All of the major traditional applications normally work as content generators. For example, CGI, PHP, and application servers proxied by Apache are content generators.

5.1. The HelloWorld Module

In this chapter, we will develop a simple content generator. The customary HelloWorld example demonstrates the basic concepts of module programming, including the complete module structure, and use of the handler callback and request_rec.

By the end of the chapter, we will have extended our HelloWorld module to report the full details of the request and response headers, the environment variables, and any data posted to the server, and we will be equipped to write content generator modules in situations where we might otherwise have used a CGI script or comparable extension.

5.1.1. The Module Skeleton

Every Apache module works by exporting a module data structure. In general, an Apache 2.x module takes the following form:

```c
module AP_MODULE_DECLARE_DATA some_module = {
    STANDARDS20_MODULE_STUFF,
    some_dir_cfg,     /* create per-directory config struct */
    some_dir_merge,   /* merge per-directory config struct */
    some_svr_cfg,     /* create per-host config struct */
    some_svr_merge,   /* merge per-host config struct */
    some_cmds,        /* configuration directives for this module */
    some_hooks,       /* register module's hooks/etc. with the core */
};
```

The STANDARDS20_MODULE_STUFF macro expands to provide version information that ensures the compiled module will load into a server build only when it is fully binary compatible, together with the filename and reserved fields. Most of the remaining fields are concerned with module configuration; they will be discussed in detail in Chapter 9. For the purposes of our HelloWorld module, we need only the hooks:

```c
module AP_MODULE_DECLARE_DATA helloworld_module = {
    STANDARDS20_MODULE_STUFF,
    NULL,
    NULL,
    NULL,
    NULL,
    helloworld_hooks
};
```

Having declared the module structure, we now need to instantiate the hooks function. Apache will run this function at server start-up. Its purpose is to register our module's processing functions with the server core, so that our module's functions will subsequently be invoked whenever they are appropriate. In the case of HelloWorld, we just need to register a simple content generator, or handler, which is one of many
kinds of functions we can insert here.

Chapter 10 explains the meaning of registering the function with a hook.

```c
static void helloworld_hooks(apr_pool_t *pool)
{
    ap_hook_handler(helloworld_handler, NULL, NULL, APR_HOOK_MIDDLE);
}
```

Finally, we need to implement `helloworld_handler`. This is a callback function that will be called by Apache at the appropriate point in processing an HTTP request. It may choose to handle or ignore a request. If it handles a request, the function is responsible for sending a valid HTTP response to the client and for ensuring that any data coming from the client are read (or discarded). This is very similar to the responsibilities of a CGI script—or, indeed, the responsibilities of the webserver as a whole.

Here's our simplest handler:

```c
static int helloworld_handler(request_rec *r)
{
    if (!r->handler || (strcmp(r->handler, "helloworld") != 0)) {
        return DECLINED;
    }
    if (r->method_number != M_GET) {
        return HTTP_METHOD_NOT_ALLOWED;
    }
    ap_set_content_type(r, "text/html;charset=ascii");
    ap_rputs("<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">\n", r);
    ap_rputs("<html><head><title>Apache HelloWorld Module</title></head>", r);
    ap_rputs("<body><h1>Hello World!</h1>", r);
    ap_rputs("<p>This is the Apache HelloWorld module!</p>", r);
    ap_rputs("</body></html>", r);
    return OK;
}
```

This callback function starts with a couple of basic sanity checks. First, we check `r->handler` to determine whether the request is for us. If the request is not for us, we ignore it by returning `DECLINED`. Apache will then pass control to the next handler.

Second, we want to support only the HTTP GET and HEAD methods. We check for those cases and, if appropriate, return an HTTP error code indicating that the method is not allowed. Returning an error code here will cause Apache to return an error page to the client. Note that the HTTP standard (see Appendix C) defines HEAD as being identical to GET except for the response body, which is omitted in HEAD. Both methods are included in Apache's `M_GET`, and content generator functions should treat them as identical.

The order in which these checks are performed is important. If we reversed them, our module might cause Apache to return an error page in cases such as POST requests intended for another handler, such as a CGI script that accepts them.

Once we are satisfied that the request is acceptable and is meant for this handler, we generate the actual response—in this case, a trivial HTML page. Having done so, we return OK to tell Apache that we have dealt with this request and that it should not call any other handler.

### 5.1.2. Return Values

Even this trivial handler has three possible return values. In general, handlers provided by modules can return

- **OK**, to indicate that the handler has fully and successfully dealt with the request. No further processing is necessary.
- **DECLINED**, to indicate that the handler takes no interest in the request and declines to process it. Apache will then try the next handler. The default handler, which simply returns a file from the local disk (or an error page if that fails), never returns `DECLINED`, so requests are always handled by some function.
- An HTTP status code, to indicate an error. The handler has taken responsibility for the request, but was unable or unwilling to complete it.

An HTTP status code diverts the entire processing chain within Apache. Normal processing of the request is aborted, and Apache sets up an internal redirect to an error document, which may either be one of Apache's predefined defaults or be a document or handler specified by the `ErrorDocument` directive in the server configuration. Note that this diversion works only if Apache hasn't already started to send the response down the wire to the client—this can be an important design consideration in handling errors. To ensure correct behavior, any such diversion must take place before writing any data (the first `ap_rputs` statements in our case).

Where possible, it is good practice to deal with errors earlier in the request processing...
5.1.3. The Handler Field

Having to check \( r->\text{handler} \) may seem counterintuitive, but this step is generally necessary in all content generators. Apache will call all content generators registered by any module until one of them returns either OK or an HTTP status code. Thus it's up to each module to check \( r->\text{handler} \), which tells the module whether it should process the request.

This scheme is made necessary by the implementation of Apache's hooks, which are designed to enable any number of functions (or nothing) to run on a hook. The content generator is unique among Apache's hooks in that exactly one content generator function must take responsibility for every request. Other hooks that share the implementation have different semantics, as we will see in Chapters 6 and 10.

5.1.4. The Complete Module

Putting it all together and adding the required headers, we have a complete mod_helloworld.c source file:

```c
#include <httpd.h>
#include <http_protocol.h>
#include <http_config.h>

static int helloworld_handler(request_rec *r)
{
    if (!r->handler || strcmp(r->handler, "helloworld")) {
        return DECLINED;
    }
    if (r->method_number != M_GET) {
        return HTTP_METHOD_NOT_ALLOWED;
    }
    ap_set_content_type(r, "text/html;charset=ascii");
    ap_rputs("<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"">\n", r);
    ap_rputs("<html><head><title>Apache HelloWorld Module</title></head>", r);
    ap_rputs("<body><h1>Hello World!</h1>", r);
    ap_rputs("<p>This is the Apache HelloWorld module!</p>", r);
    ap_rputs("</body></html>", r);
    return OK;
}

static void helloworld_hooks(apr_pool_t *pool)
{
    ap_hook_handler(helloworld_handler, NULL, NULL, APR_HOOK_MIDDLE);
}

module AP_MODULE_DECLARE_DATA helloworld_module = {
    STANDARD20_MODULE_STUFF,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    helloworld_hooks
};
```

And that's all we need! Now we can build the module and insert it into Apache. We use the apxs utility, which is bundled with Apache and serves to ensure the compilation flags and paths are correct:

Compile the module

```
$ apxs -c mod_helloworld.c
```

and (working as root) install it:

```
# apxs -i mod_helloworld.la
```

Now configure it as a handler in httpd.conf:

```
LoadModule helloworld_module modules/mod_helloworld.so
<Location /helloworld>
    SetHandler helloworld
</Location>
```
This code causes any request to /helloworld on our server to invoke this module as its handler.

Note that the helloworld_hooks and helloworld_handler functions are both declared as static. This practice is typical—though not quite universal—in Apache modules. In general, only the module symbol is exported, and everything else remains private to the module itself. As a consequence, it is good practice to declare all functions as static. Exceptions may arise when a module exports a service or API for other modules, as discussed in Chapter 10. Another case arises when a module is implemented in multiple source files and needs some symbols to be common to those files. A naming convention should be adopted in such cases, to avoid symbol space pollution.

5.1.5. Using the request_rec Object

As we have just seen, the single argument to our handler function is the request_rec object. The same argument is used for all hooks involved in request processing.

The request_rec object is a large data structure that represents an HTTP request and provides access to all data involved in processing a request. It is also an argument to many lower-level API calls. For example, in helloworld_handler, it serves as an argument to ap_set_content_type and as an I/O descriptor-like argument to ap_rputs.

Let's look at another example. Suppose we want to serve a file from the local filesystem instead of a fixed HTML page. To do so, we would use the r->filename argument to identify the file. But we can also use file stat information to optimize the process of sending the file. Instead of reading the file and sending its contents with ap_rwrite, we can send the file itself, allowing APR to take advantage of available system optimizations:

```c
static int helloworld_handler(request_rec *r)
{
    apr_file_t *fd;
    apr_size_t sz;
    apr_status_t rv;

    /* "Is it for us?" checks omitted for brevity */

    /* It's an error if r->filename and finfo haven't been set for us. */
    /* We could omit this check if we make certain assumptions concerning */
    /* use of our module, but if 'normal' processing is prevented by */
    /* some other module, then r->filename might be null, and we don't */
    /* want to risk a segfault! */
    if (r->filename == NULL) {
        ap_log_rerror(APLOG_MARK, APLOG_ERR, 0, r,
                       "Incomplete request_rec!");
        return HTTP_INTERNAL_SERVER_ERROR;
    }

    apr_set_content_type(r, "text/html;charset=ascii");

    /* Now we can usefully set some additional headers from file info */
    /* (1) Content-Length */
    /* (2) Last-Modified */
    /* ap_set_content_length(r, r->finfo.size); */
    if ((r->finfo.mtime) {
        char *datestring = apr_palloc(r->pool, APR_RFC822_DATE_LEN);
        apr_rfc822_date(datestring, r->finfo.mtime);
        apr_table_setn(r->headers_out, "Last-Modified", datestring);
    }

    rv = apr_file_open(&fd, r->filename,
                      APR_READ|APR_SHARELOCK|APR_SENDFILE_ENABLED,
                      APR_OS_DEFAULT, r->pool);
    if (rv != APR_SUCCESS) {
        ap_log_rerror(APLOG_MARK, APLOG_ERR, 0, r,
                      "can't open %s", r->filename);
        return HTTP_NOT_FOUND;
    }

    ap_send_fd(fd, r, 0, r->finfo.size, &sz);
    /* file close here is purely optional. If we omit it, APR will close */
    /* the file for us when r is destroyed, because apr_file_open */
    /* registered a close on r->pool. */
    apr_file_close(fd);
    return OK;
}
```
## Additional Reading

Safari has identified sections in other books that relate directly to this selection using Self-Organizing Maps (SOM), a type of neural network algorithm. SOM enables us to deliver related sections with higher quality results than traditional query-based approaches allow.

1. Beyond XHTML with CGI  
   From Platinum Edition Using XHTML™, XML, and Java™ 2 by Eric Ladd; Jim O'Donnell; Mike Morgan; Andrew H. Watt

2. The Common Gateway Interface  
   From Special Edition Using HTML 4, Sixth Edition by Molly E. Holzschlag

3. CGI Overview  
   From Webmaster in a Nutshell, 3rd Edition by Robert Eckstein; Stephen Spainhour

4. How CGI Scripting Works  
   From How the Internet Works, Eighth Edition by Preston Gralla; Michael Troller

5. Understanding How the CGI Works  
   From Special Edition Using HTML 4, Sixth Edition by Molly E. Holzschlag

6. CGI::Apache  
   From Perl in a Nutshell, 2nd Edition by Nathan Patwardhan; Ellen Siever; Stephen Spainhour

7. CGI  
   From Perl in a Nutshell, 2nd Edition by Nathan Patwardhan; Ellen Siever; Stephen Spainhour

8. The Future of CGI Scripting  
   From Platinum Edition Using XHTML™, XML, and Java™ 2 by Eric Ladd; Jim O'Donnell; Mike Morgan; Andrew H. Watt

9. CGI  
   From Perl in a Nutshell, 2nd Edition by Nathan Patwardhan; Ellen Siever; Stephen Spainhour

10. CGI and the World Wide Web  
    From Platinum Edition Using XHTML™, XML, and Java™ 2 by Eric Ladd; Jim O'Donnell; Mike Morgan; Andrew H. Watt

**URL** [http://proquest.safaribooksonline.com/0132409674/ch05](http://proquest.safaribooksonline.com/0132409674/ch05)