Configuration for Modules

Most modules need to offer system administrators and users some means of configuring and controlling them. In some cases, this may even be the primary purpose of a module.

System administrators configure Apache using `httpd.conf`, while end users have more limited control through `.htaccess` files. Modules give control to them by implementing configuration directives that can be used in these files.

This chapter shows how to implement configuration directives in a module, and how to work with directives implemented by other modules.

1 Configuration Basics

From the point of view of a system administrator, there are several kinds of directive. These can be broadly classified according to their scope and validity in the configuration files. That is to say, some directives are valid only for the server as a whole, while others apply within a scope such as `<VirtualHost>` or `<Directory>`.

Conflicting directives may override each other on the basis of specificity. For example, a directive in a `.htaccess` file overrides one outside in `httpd.conf` (provided the system administrator has enabled `.htaccess`). In most cases this applies recursively, although this is controlled by individual modules whose behaviour may differ.

The standard contexts supported by Apache are:

Main Config

Directives appearing in `httpd.conf` but not inside any container apply globally, except where overridden. This is appropriate for setting system defaults such as MIME types, and for once-only setup such as loading modules. Most directives can be used here.

Virtual Host

Each virtual host has its (virtual) server-wide configuration, set within a `<VirtualHost>` container. Directives that are valid in the main config are also valid in a virtual host, and vice versa.

Directory

The `<Directory>`, `<Files>` and `<Location>` directives define a hierarchy within which configuration can be set and overridden at any level. This is the most usual form of configuration, and is orthogonal to the virtual hosts. In the interests of brevity, we'll refer to this collectively as the Directory hierarchy.

.htaccess

`.htaccess` files are an extension of the Directory hierarchy that serves to enable users to set directives for themselves, subject to permissions (`AllowOverride`) set up by the server administrator.

Additionally, modules may themselves implement their own containers: for example, `mod_proxy` implements `<Proxy>`, and `mod_perl` implements `<Perl>`.

2 Configuration Data Structs

As noted above, there are two orthogonal hierarchies of configuration directives: (virtual) hosts and directories. Internally, this is based on having two different data structs: the per-server config and the per-directory config. In fact, every module has its own pointers for implementing each of these structs, although either or both can be unused (NULL), and it is unusual for a module to use both of them.

The per-server config is kept on the `server_rec`, of which there is one for each virtual host, created at server startup. The per-directory config is kept on the `request_rec` and may be computed using the merge function for every request.

3 Managing a Module Configuration

No less than five out of the six (usable) elements of the Apache module struct are concerned with configuration:

```
module my_module = {
  STANDARD20_MODULE_STUFF,
  my_create_dir_conf,           /* Create config rec for Directory */
  my_merge_dir_conf,            /* Merge config rec for Directory */
  my_create_svr_conf,           /* Create config rec for Host */
  my_merge_svr_conf,            /* Merge config rec for Host */
  my_cmds,                      /* Configuration directives */
  my_hooks
} ;
```

It is up to each module whether and how to define each struct. Whenever a struct is defined, the module must implement an appropriate `create` function to allocate and (usually) initialise it:

```
typedef struct {
  ... ;
} my_svr_cfg ;

static void* my_create_svr_conf(apr_pool_t* pool, server_rec* svr) {
  my_svr_cfg* svr = apr_pcalloc(pool, sizeof(my_svr_cfg));
  /* Set up the default values for fields of svr */
  return svr ;
}

typedef struct {
  ... ;
} my_dir_cfg ;

static void* my_create_dir_conf(apr_pool_t* pool, char* x) {
```

```
  my_dir_cfg* dir = apr_pcalloc(pool, sizeof(my_dir_cfg));
  /* Set up the default values for fields of dir */
  return dir ;
}
```

At this point, just allocating and returning a struct of the right size is often sufficient: Apache uses the return value. Now these values can be accessed at any time a `server_rec` or `request_rec`, respectively, is available.

```
my_svr_cfg* svr
      = ap_get_module_config(server->module_config, &my_module) ;
my_dir_cfg* dir
      = ap_get_module_config(request->per_dir_config, &my_module) ;
```

Server and Directory Configuration

So, why does Apache have two separate configurations, how are they related, and which should my module use?

Most directives work in the Directory hierarchy: for example, all the directives from our `mod_choices` and `mod_txt` in Chapters 5 and 6. This offers the greatest flexibility to system administrators to control the configuration, and use different configurations in different areas of their server, with `<Directory>`, `<Files>`, `<Location>`, and pattern-matching versions `<DirectoryMatch>`, etc. It can also be extended via the `AllowOverride` directive and `.htaccess` files to allow users to set their own configurations (although this carries a significant performance penalty: wherever `AllowOverride` is not `None`, Apache re-reads `.htaccess` every hit, and has to look not only in the directory it is serving from, but in every parent directory that could contain one). When in doubt, implementing a directive in Directory configuration is unlikely to be wrong.

There is, however, a subtle pitfall. Where a directive is allowed to appear at top-level in `httpd.conf` (i.e. outside any `<Directory>`/etc. container), it is also syntactically valid inside a `<VirtualHost>`. But the `<VirtualHost>` section has no meaning in the Directory hierarchy. So we cannot set different configurations for different hosts simply by putting directory configuration inside different `<VirtualHost>` containers: the hosts will simply override each other. We need an additional `<Directory>` or `<Location>` for each host where the configuration differs.

The server hierarchy is simpler. There is no nesting, and only two levels (top-level or inside a `<VirtualHost>`). It is appropriate for, among other things:

- directives explicitly concerned with virtual host configuration
- situations where the Directory hierarchy is meaningless or irrelevant – such as in a proxy configuration
- managing a persistent resource such as a database connection pool or a cache.

Configuration directives on the server heirarchy should always use `RSRC_CONF`. That makes them syntactically invalid in a `<Directory>` context, so there is no risk of the

confusion that can affect Directory configuration.

4 Implementing Configuration Directives

`my_cmds` in `my_module` is a null-terminated array containing the commands implemented by the module. Normally they are defined using macros defined in `http_config.h`. For example,

```
static const cmd_rec my_cmds[] = {
  AP_INIT_TAKE1("MyFirstDirective", my_first_cmd_func, my_ptr, OR_ALL,
        "This is My First Directive"),
  /* more directives as applicable */
  { NULL }
} ;
```

`AP_INIT_TAKE1` is one of many such macros, all having the same prototype (more later). The arguments to it are:

1. Directive Name.
2. Function implementing the directive.
3. Data pointer (often NULL).
4. Where this directive is allowed.
5. A brief "Help" message for the directive.

Configuration Functions

An essential component of every directive is the function implementing it. Normally the function serves to set some data field(s) in one of the config structs. The function prototype for `AP_INIT_TAKE1` is the same, regardless of whether we're setting per-server or per-directory config:

```
const char* my_first_cmd_func(cmd_parms* cmd, void* cfg,
   const char* arg)
```

`cmd` is a `cmd_parms_struct` comprising a number of fields used internally by Apache and available to modules. Fields likely to be of interest in modules include:

- `void* info` - contains `my_ptr` from the command declaration
- `apr_pool_t* pool` - pool for permanent resource allocation
- `apr_pool_t* temp_pool` - pool for temporary resource allocation
- `server_rec* server` - the server rec

`cfg` is the directory config rec, and **`arg`** is an argument to the directive set in the configuration file we are processing. Because we specified `AP_INIT_TAKE1`, there is

exactly one argument.

Thus, if we are setting per-directory configuration, we just cast the `cfg` argument, whereas if we are setting per-server configuration we need to retrieve it from the `server_rec` in the `cmd_parms`.

We are now in a position to implement a simple example. Our `mod_txt` in Chapter 6 needs a user-defined header and footer, each of which is a file. Let's go ahead and implement the configuration for it. We would like to be able to specify different headers and footers at will, so that a user can apply different looks-and-feels to different areas of a site, so we need to implement these directives in the Directory hierarchy.

```
typedef struct txt_cfg {
  const char* header ;
  const char* footer ;
}

static const cmd_rec txt_cmds[] = {
  AP_INIT_TAKE1("TextHeader", txt_set_header, NULL, OR_ALL,
        "Header for prettified text files"),
  AP_INIT_TAKE1("TextFooter", txt_set_footer, NULL, OR_ALL,
        "Footer for prettified text files"),
  { NULL }
} ;
```

Now we just need to implement the functions to set the header and footer. Just for a moment, we'll simply set it, and ignore checking that they're really files, and are accessible to the server, and that displaying them in a web page won't be a security risk.

```
static const char* txt_set_header(cmd_parms* cmd, void* cfg,
      const char* val) {
  ((txt_cfg*)cfg)->header = val ;
  return NULL ;
}
static const char* txt_set_footer(cmd_parms* cmd, void* cfg,
      const char* val) {
  ((txt_cfg*)cfg)->footer = val ;
  return NULL ;
}
```

UserData in Configuration Functions

In the above, we implemented two essentially-identical functions to set different fields of the configuration. We can consolidate them into a single function by passing it a context variable in `cmd->info`. Apache (APR) provides a handy macro for passing a pointer to individual fields of a configuration struct, so we can just set its contents:

```
static const cmd_rec txt_cmds[] = {
  AP_INIT_TAKE1("TextHeader", txt_set_var,
      (void*)APR_OFFSETOF(txt_cfg, header),
      OR_ALL, "Header for prettified text files"),
  AP_INIT_TAKE1("TextFooter", txt_set_var,
      (void*)APR_OFFSETOF(txt_cfg, footer),
```

```
      OR_ALL, "Footer for prettified text files"),
  { NULL }
} ;
static const char* txt_set_var(cmd_parms* cmd, void* cfg,
      const char* val) {
  int offset = (int)(long)cmd->info;
  *(const char **)((char *)cfg + offset) = arg;
  return NULL ;
}
```

Pre-Packaged Configuration Functions

In general, as above, we write our own function to implement a directive. But this is not always necessary. In the common case of a directive that simply sets a field in the directory config, we can use one of the pre-packaged functions: `ap_set_string_slot`, `ap_set_string_slot_lower`, `ap_set_int_slot`, `ap_set_flag_slot`, `ap_set_file_slot` to set a field, according to the type of the field to be set.

Our function `txt_set_var` above is in fact a direct copy of `ap_set_string_slot`. Since the fields we are setting are actually filenames, we should instead use `ap_set_file_slot`: this means that the user can specify either absolute or relative pathnames for the file, and Apache will resolve these correctly according to the underlying filesystem and the `server_root`. So we can reduce our `mod_txt` configuration to:

```
static const cmd_rec txt_cmds[] = {
  AP_INIT_TAKE1("TextHeader", ap_set_file_slot,
      (void*)APR_OFFSETOF(txt_cfg, header),
      OR_ALL, "Header for prettified text files"),
  AP_INIT_TAKE1("TextFooter", ap_set_file_slot,
      (void*)APR_OFFSETOF(txt_cfg, footer),
      OR_ALL, "Footer for prettified text files"),
  { NULL }
} ;
```

and we've improved our configuration without writing any configuration functions at all.

These functions are provided for directives in the Directory hierarchy. There are no equivalent functions for implementing configuration directives in the server hierarchy.

Scope of Configuration

The above example used `OR_ALL`, to say that `TxtHeader/TxtFooter` can be used anywhere in `httpd.conf` or in any `.htaccess` file (provided htaccess is enabled on the server). Other options we could have used include:

- `RSRC_CONF` - `httpd.conf` at top level or in a VirtualHost context. All directives using server config should use this, as other contexts are meaningless for a server config.

- ACCESS_CONF - httpd.conf in a Directory context. This is appropriate to per-dir config directives for a server administrator only, and is often combined (using OR) with RSRC_CONF to allow its use anywhere within httpd.conf.
- OR_LIMIT, OR_OPTIONS, OR_FILEINFO, OR_AUTHCFG, OR_INDEXES – extend ACCESS_CONF to allow use of the directive in .htaccess according to the AllowOverride setting.

Configuration Function Types

The above example used the AP_INIT_TAKE1 macro, which defines a function having a single string argument. This is one of several such macros defined in http_config.h:

- AP_INIT_NO_ARGS (no arguments)
- AP_INIT_FLAG (a single On/Off argument)
- AP_INIT_TAKE1 (a single string argument)
- AP_INIT_TAKE2, AP_INIT_TAKE3, AP_INIT_TAKE12, etc. - directives taking different numbers of string arguments
- AP_INIT_ITERATE (function will be called repeatedly with each of an unspecified number of arguments)
- AP_INIT_ITERATE2 (function will be called repeatedly with two arguments)
- AP_INIT_RAW_ARGS (function will be called with arguments unprocessed)

This gives module authors a choice of simple prototypes, together with the hands-on RAW_ARGS for modules to do their own parsing. Modules using RAW_ARGS should retrieve the arguments using the function ap_getword_conf repeatedly until it returns NULL.

Let's look at some examples. We've already seen a TAKE1 case. The other AP_INIT_TAKE* functions are similar but have different numbers of arguments (those with variable numbers of arguments simply work by passing NULL values where no argument was specified in the configuration).

AP_INIT_FLAG

In the Directory hierarchy, this can generally be dealt with using ap_set_flag_slot. For example, in our mod_choices from Chapter 5, we need to implement the directive Choices On|Off. Recollect we have a per-directory configuration record:

```
typedef struct choices_cfg {
  int choices ;              /* flag to turn this module on/off */
  apr_hash_t* transforms ;   /* table of "extensions" known to
                              * this server.
                              */
} choices_cfg ;
```

So all we need to implement the directive is:

```
  AP_INIT_FLAG("Choices", ap_set_flag_slot,
```

```
        (void*)APR_OFFSETOF(choices_cfg, choices),
        RSRC_CONF|ACCESS_CONF,
        "Enable different document formats according to .extension" )
```

In the Server heirarchy, you would treat it as equivalent to a restricted `AP_INIT_TAKE1`.

AP_INIT_ITERATE

The function is called once for each argument. So this is suitable for directives having variable arguments all having the same significance. `AP_INIT_ITERATE2` provides an additional case for where the first argument has some syntactically different purpose and is passed to every call.

There are several examples in `mod_proxy`, where you can supply a list of addresses to which a proxy is or isn't allowed to connect.

```
    AP_INIT_ITERATE("AllowCONNECT", set_allowed_ports, NULL, RSRC_CONF,
     "A list of ports which CONNECT may connect to")
```

Here's the function: it's very simple because it only ever has to deal with one argument at a time. Note that this is also an example of a directive in the Server hierarchy (directories are meaningless in a proxy context, but virtual hosts are important)!

```
/*
 * Set the ports CONNECT can use
 */
static const char *
    set_allowed_ports(cmd_parms *parms, void *dummy, const char *arg)
{
    server_rec *s = parms->server;
    proxy_server_conf *conf =
        ap_get_module_config(s->module_config, &proxy_module);
    int *New;

    if (!apr_isdigit(arg[0]))
        return "AllowCONNECT: port number must be numeric";

    New = apr_array_push(conf->allowed_connect_ports);
    *New = atoi(arg);
    return NULL;
}
```

AP_INIT_RAW_ARGS

Raw arguments are needed where a directive's syntax is highly variable and needs to be fully parsed in the configuration function. One such is `mod_publisher`'s **`MLMacro`**, which directs the publisher filter to treat an XML or HTML element as a macro to transform in some manner. It is defined as follows:

```
MLMacro element hide|insert|replace [start|end] [string|var|file] [val]
```

There are two fixed arguments: the name of the element to rewrite, and the action required. There are three optional arguments.

Here's how we define and parse it. There are two variants on the directive: one is permitted in `.htaccess`, the other is restricted to `httpd.conf`, because it allows users to display the contents of any file accessible to Apache, and so might be a security issue if permitted to untrusted users. We flag this by setting `cmd->info` to a non-null value.

```
  AP_INIT_RAW_ARGS("MLMacro", set_special, NULL, OR_ALL,
        "Define processing for an element")
  AP_INIT_RAW_ARGS("MLMacroPath", set_special, set_special,
        RSRC_CONF|ACCESS_CONF, "Define processing for an element") ,

static const char* set_special(cmd_parms* cmd,
           void* CFG, const char* args) {
  html_conf* cfg = CFG;
  int is_new = 0;
  /* It's easy to get the two fixed args */
  const char* elt = ap_getword_conf(cmd->temp_pool, &args) ;
  const char* op = ap_getword_conf(cmd->temp_pool, &args) ;
  const char* where ;
  const char* type ;
  const char* value ;
  const char* dummy = NULL ;
  insertion* var ;
  special_elt* special ;

  /* Message user will get if a syntax error is detected */
  const char* errmsg = "MLMacro: element hide|insert|replace [start|
end] [string|var|file|path|url] [value]" ;

  /* If we didn't get at least the first two args it's a syntax error */
  if ( !*elt || !*op )
    return errmsg ;

  /* Check if we already have macro definitions for this element
   * (we can define things both at the start and end) */

  special = apr_hash_get(cfg->special, elt, APR_HASH_KEY_STRING) ;
  if ( ! special ) {
    special = apr_palloc(cmd->pool, sizeof(special_elt)) ;
    is_new = 1 ;
  }

  /* Check for the allowed actions.  It's a syntax error if the
   * action argument isn't one of them. */
  if ( !strcasecmp(op, "insert") ) {
    special->etype = ELEM_INSERT ;
  } else if ( !strcasecmp(op, "replace") ) {
    special->etype = ELEM_REPLACE ;
  } else if ( !strcasecmp(op, "hide") ) {
    special->etype = ELEM_HIDE ;
  } else {
    return errmsg ;
  }
```

```
switch ( special->etype ) {
  case ELEM_INSERT:
  case ELEM_REPLACE:
    /* Insert and Replace happen at the start (default) or end of
     * the element, and we need to define what's being added. */
    where = ap_getword_conf(cmd->tmp_pool, &args) ;
    type = ap_getword_conf(cmd->tmp_pool, &args) ;
    value = ap_getword_conf(cmd->tmp_pool, &args) ;

    if ( !strcasecmp(where, "start") )
      var = &special->at_start ;
    else if ( !strcasecmp(where, "end") )
      var = &special->at_end ;
    else {
      /* If the argument was neither “start” nor “end”, it was
       * omitted, so we default to at_start */
      var = &special->at_start ;
      dummy = value ;
      value = type ;
      type = where ;
    }
    /* Check what kind of thing we are substituting */
    if ( !strcasecmp(type, "var") )
      var->t = INSERT_VAR ;       /* a variable */
    else if ( !strcasecmp(type, "string") )
      var->t = INSERT_DATA ;      /* a literal string */
    else if ( !strcasecmp(type, "file") )
      var->t = INSERT_FILE ;      /* a file under DOCUMENT_ROOT */
    else if ( !strcasecmp(type, "path") && (cmd->info != NULL) )
      var->t = INSERT_PATH ;      /* a filepath anywhere */
    else if ( !strcasecmp(type, "url") )
      var->t = INSERT_URI ;       /* a HTTP URL */
    else {
      /* If it was non of those, we default to a literal string,
       * and this is its value */
      var->t = INSERT_DATA ;
      dummy = value ;
      value = type ;
    }
    /* and here's what we're setting it to */
    var->what = value ;
    break ;
  case ELEM_HIDE:
    /* if the action is to hide this element and any contents,
     * we don't have any more arguments */
  default:
    break ;
}
/* Now just check there aren't any (bogus) extra args */
if ( ! dummy )
  dummy = ap_getword_conf(cmd->tmp_pool, &args) ;
if ( *dummy )
  return errmsg ;
```

```
  /* and store it, if it wasn't already stored */
  if ( is_new )
    apr_hash_set(cfg->special, elt, sizeof(special_elt), special) ;

  return NULL ;
}
```

5 The Configuration Hierarchy

We have now dealt with creating the configuration structures and populating them using configuration directives. The next topic we need to understand is managing the configuration hierarchy: how directives set at different levels interact with each other. This is the purpose of the merge functions in the module struct.

A merge function is called whenever there are directives at more than one level in a hierarchy, starting at the top level of `httpd.conf`. In the case of the per-directory config there may be several levels and thus several calls to a merge function, incorporating `htaccess` files (if applicable) as well as sections in `httpd.conf`.

A merge function may also be NULL. In that case, all directives in the less-specific container are discarded, so incremental configuration is not possible. Nevertheless, it is perfectly adequate for some modules.

More typically, we want the merge function to honour directives set in the more specific container, but inherit values that are not explicitly set. This is where we need a merge function. Consider the following example:

```
typedef struct {
  int a , b , c :
} my_dir_cfg;
```

with directives to set each of these, and a configuration

```
<Location />
        SetMyA  123
        SetMyC  321
</Location>
<Location /somewhere/>
        SetMyB  456
</Location>
<Location /somewhere/else/again/>
        SetMyC  789
</Location>
```

Here the most specific section is `/somewhere/else/again/`, so in the absence of a directory merge function, `c` will be set to 789 but the values of `a` and `b` are unset. We need a merge function, which takes the generic form:

```
static void* my_merge_dir_conf(apr_pool_t* pool, void* BASE, void* ADD)
{
    my_dir_cfg* base = BASE ;
```

```
    my_dir_cfg* add = ADD ;
    my_dir_cfg* conf = apr_palloc(pool, sizeof(my_dir_cfg)) ;
    conf->a = ( add->a == UNSET ) ? base->a : add->a ;
    conf->b = ( add->b == UNSET ) ? base->b : add->b ;
    conf->c = ( add->c == UNSET ) ? base->c : add->c ;
    return conf ;
}
```

To make this effective, we define the value of `UNSET` to some value that won't be used (e.g. -1 if our integers will always be positive), and initialise them to that in our `create_config` function. Now our configuration is processed as follows:

1. At the top level, `a` is set to 123 and `c` to 321 while b is unset.
2. The first merge sets `b` to 456. Since `a` and `c` are not set (overridden) at this level, the previous values are inherited in the merge.
3. There are no configuration directives at `/somewhere/else/`, so this level simply inherits from `/somewhere/` without any need for a merge.
4. The second merge sets the value of `c` overriding the previous setting, while inheriting the previous values of `a` and b. Now we have `a`=123, `b`=456, `c`=789.

This is obviously a trivial merge function. Often we may need to do something a little more interesting: for example to merge non-trivial structures, or to deal with cases where there is no meaningful `UNSET` value to test. When merging structures involving pointers, it is important to take care about modifying the originals: it's usually safer to make a copy unless using a standard APR datatype with its merge functions. We just have to deal with each case on its merits.

6 Dealing with Variables

The configuration structures should normally be treated as read-only outside of the functions discussed above. A few limited exceptions may be appropriate, usually on the server config, where it is used to manage, for example, a pool or cache of resources whose contents might change at any time. This can safely be done in a `post_config` or `child_init` hook. But at any later point - when processing a connection or request - this gives rise to a race condition. Any such operations must therefore use an appropriate lock: usually an `apr_thread_mutex` (which must itself be set up during module initialisation).

Request and Connection Variables

It is not appropriate to use the configuration structs for variables used in processing a Request or Connection. However, similar structs are provided for these, and can be allocated on the request or connection pools with the lifetime of the request or connection.

```
typedef struct {
    ....
```

```
} my_request_vars ;
```

We can now set this in some hook:

```
my_request_vars* vars = apr_palloc(r->pool, sizeof(my_request_vars)) ;
/* store stuff in vars */
ap_set_module_config(r->request_config, &my_module, vars) ;
```

and retrieve what we set later in the request:

```
my_request_vars* vars
      = ap_get_module_config(r->request_config, &my_module) ;
```

The `conn_rec` has an analogous `conn_config` field. Apache provides other contexts that may be useful for some applications: each filter and namespace has, as described in Chapters 6 and 7, its own context.