1 Apache Server Configuration Customization in Perl

1.1 Description

This chapter explains how to create custom Apache configuration directives in Perl.

1.2 Incentives

mod_perl provides several ways to pass custom configuration information to the modules.

The simplest way to pass custom information from the configuration file to the Perl module is to use the `PerlSetVar` and `PerlAddVar` directives. For example:

```
PerlSetVar Secret "Matrix is us"
```

and in the mod_perl code this value can be retrieved as:

```
my $secret = $r->dir_config("Secret");
```

Another alternative is to add custom configuration directives. There are several reasons for choosing this approach:

- When the expected value is not a simple argument, but must be supplied using a certain syntax, Apache can verify at startup time that this syntax is valid and abort the server start up if the syntax is invalid.

- Custom configuration directives are faster because their values are parsed at the startup time, whereas `PerlSetVar` and `PerlAddVar` values are parsed at the request time.

- It's possible that some other modules have accidentally chosen to use the same key names but for absolutely different needs. So the two now can't be used together. Of course this collision can be avoided if a unique to your module prefix is used in the key names. For example:

  ```
  PerlSetVar ApacheFooSecret "Matrix is us"
  ```

Finally, modules can be configured in pure Perl using `<Perl> Sections` or a startup file, by simply modifying the global variables in the module's package. This approach could be undesirable because it requires a use of globals, which we all try to reduce. A bigger problem with this approach is that you can't have different settings for different sections of the site (since there is only one version of a global variable), something that the previous two approaches easily achieve.

1.3 Creating and Using Custom Configuration Directives

In mod_perl 2.0, adding new configuration directives is a piece of cake, because it requires no XS code and *Makefile.PL*, needed in case of mod_perl 1.0. In mod_perl 2.0, custom directives are implemented in pure Perl.

Here is a very basic module that declares two new configuration directives: `MyParameter`, which accepts one or more arguments, and `MyOtherParameter` which accepts a single argument.

```
#file:MyApache/MyParameters.pm
#------------------------------
package MyApache::MyParameters;

use strict;
use warnings FATAL => 'all';

use Apache::Test;
use Apache::TestUtil;

use Apache::Const -compile => qw(OR_ALL ITERATE);

use Apache::CmdParms ();
use Apache::Module ();

our @APACHE_MODULE_COMMANDS = (
    {
     name         => 'MyParameter',
     func         => __PACKAGE__ . '::MyParameter',
     req_override => Apache::OR_ALL,
     args_how     => Apache::ITERATE,
     errmsg       => 'MyParameter Entry1 [Entry2 ... [EntryN]]',
    },
    {
     name         => 'MyOtherParameter',
    },
);

sub MyParameter {
    my($self, $parms, @args) = @_;
    $self->{MyParameter} = \@args;
}
1;
```

And here is how to use it in *httpd.conf*:

```
# first load the module so Apache will recognize the new directives
PerlLoadModule MyApache::MyParameters

MyParameter one two three
MyOtherParameter Foo
<Location /perl>
  MyParameter eleven twenty
  MyOtherParameter Bar
</Location>
```

The following sections discuss this and more advanced modules in detail.

A minimal configuration module is comprised of two groups of elements:

- **A global array `@APACHE_MODULE_COMMANDS` for declaring the new directives and their behavior.**
- **A subroutine per each new directive, which is called when the directive is seen**

1.3.1 @APACHE_MODULE_COMMANDS

`@APACHE_MODULE_COMMANDS` is a global array of hash references. Each hash represents a separate new configuration directive. In our example we had:

```
our @APACHE_MODULE_COMMANDS = (
    {
     name         => 'MyParameter',
     func         => __PACKAGE__ . '::MyParameter',
     req_override => Apache::OR_ALL,
     args_how     => Apache::ITERATE,
     errmsg       => 'MyParameter Entry1 [Entry2 ... [EntryN]]',
    },
    {
     name         => 'MyOtherParameter',
    },
);
```

This structure declares two new directives: `MyParameter` and `MyOtherParameter`. You have to declare at least the name of the new directive, which is how we have declared the `MyOtherParameter` directive. mod_perl will fill in the rest of the configuration using the defaults described next.

These are the attributes that can be used to define the directives behavior: *name*, *func*, *args_how*, *req_override* and *errmsg*. They are discussed in the following sections.

1.3.1.1 `name`

This is the only required attribute. And it declares the name of the new directive as it'll be used in *httpd.conf*.

1.3.1.2 `func`

The *func* attribute expects a reference to a function or a function name. This function is called by httpd every time it encounters the directive that is described by this entry while parsing the configuration file. Therefore it's invoked once for every instance of the directive at the server startup, and once per request per instance in the *.htaccess* file.

This function accepts two or more arguments, depending on the *args_how* attribute's value.

This attribute is optional. If not supplied, mod_perl will try to use a function in the current package whose name is the same as of the directive in question. In our example with `MyOtherParameter`, mod_perl will use:

```
__PACKAGE__ . '::MyOtherParameter'
```

as a name of a subroutine and it anticipates that it exists in that package.

1.3.1.3 req_override

The attribute defines the valid scope in which this directive can appear. There are several constants which map onto the corresponding Apache macros. These constants should be imported from the `Apache::Const` package.

For example, to use the `OR_ALL` constant, which allows directives to be defined anywhere, first, it needs to be imported:

```
use Apache::Const -compile => qw(OR_ALL);
```

and then assigned to the *req_override* attribute:

```
req_override => Apache::OR_ALL,
```

It's possible to combine several options using the unary operators. For example, the following setting:

```
req_override => Apache::RSRC_CONF | Apache::ACCESS_CONF
```

will allow the directive to appear anywhere in *httpd.conf*, but forbid it from ever being used in *.htaccess* files:

This attribute is optional. If not supplied, the default value of `Apache::OR_ALL` is used.

1.3.1.4 args_how

Directives can receive zero, one or many arguments. In order to help Apache validate that the number of arguments is valid, the *args_how* attribute should be set to the desired value. Similar to the *req_override* attribute, the `Apache::Const` package provides special constants which map to the corresponding Apache macros. There are several constants to choose from.

In our example, the directive `MyParameter` accepts one or more arguments, therefore we have the `Apache::ITERATE` constant:

```
args_how => Apache::ITERATE,
```

This attribute is optional. If not supplied, the default value of `Apache::TAKE1` is used.

META: the default may change to use a constant corresponding to the *func* prototype.

1.3.1.5 errmsg

The *errmsg* attribute provides a short but succinct usage statement that summarizes the arguments that the directive takes. It's used by Apache to generate a descriptive error message, when the directive is configured with a wrong number of arguments.

In our example, the directive `MyParameter` accepts one or more arguments, therefore we have chosen the following usage string:

```
errmsg => 'MyParameter Entry1 [Entry2 ... [EntryN]]',
```

This attribute is optional. If not supplied, the default value of will be a string based on the directive's *name* and *args_how* attributes.

1.3.1.6 `cmd_data`

Sometimes it is useful to pass information back to the directive handler callback. For instance, if you use the *func* parameter to specify the same callback for two different directives you might want to know which directive is being called currently. To do this, you can use the *cmd_data* parameter, which allows you to store arbitrary strings for later retrieval from your directive handler. For instance:

```
our @APACHE_MODULE_COMMANDS = (
    {
     name         => '<Location',
     # func defaults to Redirect()
     req_override => Apache::RSRC_CONF,
     args_how     => Apache::RAW_ARGS,
    },
    {
     name         => '<LocationMatch',
     func         => Redirect,
     req_override => Apache::RSRC_CONF,
     args_how     => Apache::RAW_ARGS,
     cmd_data     => '1',
    },
);
```

Here, we are using the `Location()` function to process both the `Location` and `LocationMatch` directives. In the `Location()` callback we can check the data in the *cmd_data* slot to see whether the directive being processed is `LocationMatch` and alter our logic accordingly. How? Through the `info()` method exposed by the `Apache::CmdParms` class.

```
use Apache::CmdParms ();

sub Location {

  my ($cfg, $parms, $data) = @_;

  # see if we were called via LocationMatch
  my $regex = $parms->info;

  # continue along
}
```

In case you are wondering, `Location` and `LocationMatch` were chosen for a reason - this is exactly how httpd core handles these two directives.

1.3.2 Directive Scope Definition Constants

The *req_override* attribute specifies the configuration scope in which it's valid to use a given configuration directive. This attribute's value can be any of or a combination of the following constants:

(these constants are declared in *httpd-2.0/include/http_config.h*.)

1.3.2.1 `Apache::OR_NONE`

The directive cannot be overridden by any of the `AllowOverride` options.

1.3.2.2 `Apache::OR_LIMIT`

The directive can appear within directory sections, but not outside them. It is also allowed within *.htaccess* files, provided that `AllowOverride Limit` is set for the current directory.

1.3.2.3 `Apache::OR_OPTIONS`

The directive can appear anywhere within *httpd.conf*, as well as within *.htaccess* files provided that `AllowOverride Options` is set for the current directory.

1.3.2.4 `Apache::OR_FILEINFO`

The directive can appear anywhere within *httpd.conf*, as well as within *.htaccess* files provided that `AllowOverride FileInfo` is set for the current directory.

1.3.2.5 `Apache::OR_AUTHCFG`

The directive can appear within directory sections, but not outside them. It is also allowed within *.htaccess* files, provided that `AllowOverride AuthConfig` is set for the current directory.

1.3.2.6 `Apache::OR_INDEXES`

The directive can appear anywhere within *httpd.conf*, as well as within *.htaccess* files provided that `AllowOverride Indexes` is set for the current directory.

1.3.2.7 `Apache::OR_UNSET`

META: details? "unset a directive (in Allow)"

1.3.2.8 `Apache::ACCESS_CONF`

The directive can appear within directory sections. The directive is not allowed in *.htaccess* files.

1.3.2.9 Apache::RSRC_CONF

The directive can appear in *httpd.conf* outside a directory section (<Directory>, <Location> or <Files>; also <FilesMatch> and kin). The directive is not allowed in *.htaccess* files.

1.3.2.10 Apache::OR_EXEC_ON_READ

Force directive to execute a command which would modify the configuration (like including another file, or IFModule).

Normally, Apache first parses the configuration tree and then executes the directives it has encountered (e.g., SetEnv). But there are directives that must be executed during the initial parsing, either because they affect the configuration tree (e.g., Include may load extra configuration) or because they tell Apache about new directives (e.g., IfModule or PerlLoadModule, may load a module, which installs handlers for new directives). These directives must have the Apache::OR_EXEC_ON_READ turned on.

1.3.2.11 Apache::OR_ALL

The directive can appear anywhere. It is not limited in any way.

1.3.3 Directive Callback Subroutine

Depending on the value of the *args_how* attribute the callback subroutine, specified with the *func* attribute, will be called with two or more arguments. The first two arguments are always $self and $parms. A typical callback function which expects a single value (Apache::TAKE1) might look like the following:

```
sub MyParam {
    my($self, $parms, $arg) = @_;
    $self->{MyParam} = $arg;
}
```

In this function we store the passed single value in the configuration object, using the directive's name (assuming that it was MyParam) as the key.

Let's look at the subroutine arguments in detail:

1. $self is the current container's configuration object.

   This configuration object is a reference to a hash, in which you can store arbitrary key/value pairs. When the directive callback function is invoked it may already include several key/value pairs inserted by other directive callbacks or during the SERVER_CREATE and DIR_CREATE functions, which will be explained later.

   Usually the callback function stores the passed argument(s), which later will be read by SERVER_MERGE and DIR_MERGE, which will be explained later, and of course at request time.

The convention is use the name of the directive as the hash key, where the received values are stored. The value can be a simple scalar, or a reference to a more complex structure. So for example you can store a reference to an array, if there is more than one value to store.

This object can be later retrieved at request time via:

```
my $dir_cfg = $self->get_config($s, $r->per_dir_config);
```

You can retrieve the server configuration object via:

```
my $srv_cfg = $self->get_config($s);
```

if invoked inside the virtual host, the virtual host's configuration object will be returned.

2. `$parms` is an `Apache::CmdParms` object from which you can retrieve various other information about the configuration. For example to retrieve the server object:

   ```
   my $s = $parms->server;
   ```

   See `Apache::CmdParms` for more information.

3. The rest of the arguments whose number depends on the *args_how*'s value are covered in the next section.

1.3.4 Directive Syntax Definition Constants

The following values of the *args_how* attribute define how many arguments and what kind of arguments directives can accept. These values are constants that can be imported from the `Apache::Const` package. For example:

```
use Apache::Const -compile => qw(TAKE1 TAKE23);
```

1.3.4.1 `Apache::NO_ARGS`

The directive takes no arguments. The callback will be invoked once each time the directive is encountered. For example:

```
sub MyParameter {
     my($self, $parms) = @_;
     $self->{MyParameter}++;
}
```

1.3.4.2 `Apache::TAKE1`

The directive takes a single argument. The callback will be invoked once each time the directive is encountered, and its argument will be passed as the third argument. For example:

```
sub MyParameter {
   my($self, $parms, $arg) = @_;
   $self->{MyParameter} = $arg;
}
```

1.3.4.3 Apache::TAKE2

The directive takes two arguments. They are passed to the callback as the third and fourth arguments. For example:

```
sub MyParameter {
   my($self, $parms, $arg1, $arg2) = @_;
   $self->{MyParameter} = {$arg1 => $arg2};
}
```

1.3.4.4 Apache::TAKE3

This is like `Apache::TAKE1` and `Apache::TAKE2`, but the directive takes three mandatory arguments. For example:

```
sub MyParameter {
   my($self, $parms, @args) = @_;
   $self->{MyParameter} = \@args;
}
```

1.3.4.5 Apache::TAKE12

This directive takes one mandatory argument, and a second optional one. This can be used when the second argument has a default value that the user may want to override. For example:

```
sub MyParameter {
  my($self, $parms, $arg1, $arg2) = @_;
  $self->{MyParameter} = {$arg1 => $arg2||'default'};
}
```

1.3.4.6 Apache::TAKE23

`Apache::TAKE23` is just like `Apache::TAKE12`, except now there are two mandatory arguments and an optional third one.

1.3.4.7 Apache::TAKE123

In the `Apache::TAKE123` variant, the first argument is mandatory and the other two are optional. This is useful for providing defaults for two arguments.

1.3.4.8 Apache::ITERATE

`Apache::ITERATE` is used when a directive can take an unlimited number of arguments. The callback is invoked repeatedly with a single argument, once for each argument in the list. It's done this way for interoperability with the C API, which doesn't have the flexible argument passing that Perl provides. For example:

```
sub MyParameter {
   my($self, $parms, $args) = @_;
   push @{ $self->{MyParameter} }, $arg;
}
```

1.3.4.9 Apache::ITERATE2

Apache::ITERATE2 is used for directives that take a mandatory first argument followed by a list of arguments to be applied to the first. A familiar example is the AddType directive, in which a series of file extensions are applied to a single MIME type:

```
AddType image/jpeg JPG JPEG JFIF jfif
```

Apache will invoke your callback once for each item in the list. Each time Apache runs your callback, it passes the routine the constant first argument (*"image/jpeg"* in the example above), and the current item in the list (*"JPG"* the first time around, *"JPEG"* the second time, and so on). In the example above, the configuration processing routine will be run a total of four times.

For example:

```
sub MyParameter {
   my($self, $parms, $key, $val) = @_;
   push @{ $self->{MyParameter}{$key} }, $val;
}
```

1.3.4.10 Apache::RAW_ARGS

An *args_how* of Apache::RAW_ARGS instructs Apache to turn off parsing altogether. Instead it simply passes your callback function the line of text following the directive. Leading and trailing whitespace is stripped from the text, but it is not otherwise processed. Your callback can then do whatever processing it wishes to perform.

This callback receives three arguments (similar to Apache::TAKE1), the third of which is a string-valued scalar containing the text following the directive.

```
sub MyParameter {
   my($self, $parms, $val) = @_;
   # process $val
}
```

If this mode is used to implement a custom "container" directive, the attribute *req_override* needs to OR Apache::OR_EXEC_ON_READ. e.g.:

```
req_override => Apache::OR_ALL | Apache::OR_EXEC_ON_READ,
```

META: complete the details, which are new to 2.0.

There is one other trick to making configuration containers work. In order to be recognized as a valid directive, the *name* attribute must contain the leading <. This token will be stripped by the code that handles the custom directive callbacks to Apache. For example:

```
name     => '<MyContainer',
```

One other trick that is not required, but can provide some more user friendliness is to provide a handler for the container end token. In our example, the Apache configuration gears will never see the `</MyContainer>` token, as our `Apache::RAW_ARGS` handler will read in that line and stop reading when it is seen. However in order to catch cases in which the `</MyContainer>` text appears without a preceding `<MyContainer>` opening section, we need to turn the end token into a directive that simply reports an error and exits. For example:

```
{
  name         => '</MyContainer>',
  func         => __PACKAGE__ . "::MyContainer_END",
  errmsg       => 'end of MyContainer without beginning?',
  args_how     => Apache::NO_ARGS,
  req_override => Apache::OR_ALL,
},
...
my $EndToken = "</MyContainer>";
sub MyContainer_END {
    die "$EndToken outside a <MyContainer> container\n";
}
```

Now, should the server administrator misplace the container end token, the server will not start, complaining with this error message:

```
Syntax error on line 54 of httpd.conf:
</MyContainer> outside a <MyContainer> container
```

1.3.4.11 `Apache::FLAG`

When `Apache::FLAG` is used, Apache will only allow the argument to be one of two values, `On` or `Off`. This string value will be converted into an integer, `1` if the flag is `On`, `0` if it is `Off`. If the configuration argument is anything other than `On` or `Off`, Apache will complain:

```
Syntax error on line 73 of httpd.conf:
MyFlag must be On or Off
```

For example:

```
sub MyFlag {
    my($self, $parms, $arg) = @_;
    $self->{MyFlag} = $arg; # 1 or 0
}
```

1.3.5 Enabling the New Configuration Directives

As seen in the first example, the module needs to be loaded before the new directives can be used. A special directive `PerlLoadModule` is used for this purpose. For example:

```
PerlLoadModule MyApache::MyParameters
```

This directive is similar to `PerlModule`, but it require()'s the Perl module immediately, causing an early mod_perl startup. After loading the module it let's Apache know of the new directives and installs the callbacks to be called when the corresponding directives are encountered.

1.3.6 Creating and Merging Configuration Objects

By default mod_perl creates a simple hash to store each container's configuration values, which are populated by directive callbacks, invoked when the *httpd.conf* and the *.htaccess* files are parsed and the corresponding directive are encountered. It's possible to pre-populate the hash entries when the data structure is created, e.g., to provide reasonable default values for cases where they weren't set in the configuration file. To accomplish that the optional `SERVER_CREATE` and `DIR_CREATE` functions can be supplied.

When a request is mapped to a container, Apache checks if that container has any ancestor containers. If that's the case, it allows mod_perl to call special merging functions, which decide whether configurations in the parent containers should be inherited, appended or overridden in the child container. The custom configuration module can supply custom merging functions `SERVER_MERGE` and `DIR_MERGE`, which can override the default behavior. If these functions are not supplied the following default behavior takes place: The child container inherits its parent configuration, unless it specifies its own and then it overrides its parent configuration.

1.3.6.1 SERVER_CREATE

`SERVER_CREATE` is called once for the main server, and once more for each virtual host defined in *httpd.conf*. It's called with two arguments: `$class`, the package name it was created in and `$parms` the already familiar `Apache::CmdParms` object. The object is expected to return a reference to a blessed hash, which will be used by configuration directives callbacks to set the values assigned in the configuration file. But it's possible to preset some values here:

For example, in the following example the object assigns a default value, which can be overridden during merge if a the directive was used to assign a custom value:

```
package MyApache::MyParameters;
...
use Apache::Module ();
use Apache::CmdParms ();
our @APACHE_MODULE_COMMANDS = (...);
...
sub SERVER_CREATE {
    my($class, $parms) = @_;
    return bless {
        name => __PACKAGE__,
    }, $class;
}
```

To retrieve that value later, you can use:

```
use Apache::Module ();
...
my $srv_cfg = Apache::Module->get_config('MyApache::MyParameters', $s);
print $srv_cfg->{name};
```

If a request is made to a resource inside a virtual host, `$srv_cfg` will contain the object of the virtual host's server. To reach the main server's configuration object use:

```
use Apache::Module ();
use Apache::Server ();
use Apache::ServerUtil ();
...
if ($s->is_virtual) {
    my $base_srv_cfg = Apache::Module->get_config('MyApache::MyParameters',
                                                  Apache->server);
    print $base_srv_cfg->{name};
}
```

If the function `SERVER_CREATE` is not supplied by the module, a function that returns a blessed into the current package reference to a hash is used.

1.3.6.2 SERVER_MERGE

During the configuration parsing virtual hosts are given a chance to inherit the configuration from the main host, append to or override it. The `SERVER_MERGE` subroutine can be supplied to override the default behavior, which simply overrides the main server's configuration.

The custom subroutine accepts two arguments: `$base`, a blessed reference to the main server configuration object, and `$add`, a blessed reference to a virtual host configuration object. It's expected to return a blessed object after performing the merge of the two objects it has received. Here is the skeleton of a merging function:

```
sub merge {
    my($base, $add) = @_;
    my %mrg = ();
    # code to merge %$base and %$add
    return bless \%mrg, ref($base);
}
```

The section Merging at Work provides an extensive example of a merging function.

1.3.6.3 DIR_CREATE

Similarly to `SERVER_CREATE`, this optional function, is used to create an object for the directory resource. If the function is not supplied mod_perl will use an empty hash variable as an object.

Just like `SERVER_CREATE`, it's called once for the main server and one more time for each virtual host. In addition it'll be called once more for each resource (`<Location>`, `<Directory>` and others). All this happens during the startup. At request time it might be called for each parsed *.htaccess* file and for each resource defined in it.

The `DIR_CREATE` function's skeleton is identical to `SERVER_CREATE`. Here is an example:

```
package MyApache::MyParameters;
...
use Apache::Module ();
use Apache::CmdParms ();
```

```
our @APACHE_MODULE_COMMANDS = (...);
...
sub DIR_CREATE {
    my($class, $parms) = @_;
    return bless {
        foo => 'bar',
    }, $class;
}
```

To retrieve that value later, you can use:

```
use Apache::Module ();
...
my $dir_cfg = Apache::Module->get_config('MyApache::MyParameters',
                                         $s, $r->per_dir_config);
print $dir_cfg->{foo};
```

The only difference in the retrieving the directory configuration object. Here the third argument `$r->per_dir_config` tells `Apache::Module` to get the directory configuration object.

1.3.6.4 DIR_MERGE

Similarly to `SERVER_MERGE`, `DIR_MERGE` merges the ancestor and the current node's directory configuration objects. At the server startup `DIR_MERGE` is called once for each virtual host. At request time, the merging of the objects of resources, their sub-resources and the virtual host/main server merge happens. Apache caches the products of merges, so you may see certain merges happening only once.

The section Merging Order Consequences discusses in detail the merging order.

The section Merging at Work provides an extensive example of a merging function.

1.4 Examples

1.4.1 Merging at Work

In the following example we are going to demonstrate in details how merging works, by showing various merging techniques.

Here is an example Perl module, which, when loaded, installs four custom directives into Apache.

```
#file:MyApache/CustomDirectives.pm
#---------------------------------
package MyApache::CustomDirectives;

use strict;
use warnings FATAL => 'all';

use Apache::CmdParms ();
use Apache::Module ();
use Apache::ServerUtil ();
```

```
use Apache::Const -compile => qw(OK);

our @APACHE_MODULE_COMMANDS = (
    { name => 'MyPlus' },
    { name => 'MyList' },
    { name => 'MyAppend' },
    { name => 'MyOverride' },
);

sub MyPlus     { set_val('MyPlus',     @_) }
sub MyAppend   { set_val('MyAppend',   @_) }
sub MyOverride { set_val('MyOverride', @_) }
sub MyList     { push_val('MyList',    @_) }

sub DIR_MERGE    { merge(@_) }
sub SERVER_MERGE { merge(@_) }

sub set_val {
    my($key, $self, $parms, $arg) = @_;
    $self->{$key} = $arg;
    unless ($parms->path) {
        my $srv_cfg = Apache::Module->get_config($self,
                                                 $parms->server);
        $srv_cfg->{$key} = $arg;
    }
}

sub push_val {
    my($key, $self, $parms, $arg) = @_;

    push @{ $self->{$key} }, $arg;
    unless ($parms->path) {
        my $srv_cfg = Apache::Module->get_config($self,
                                                 $parms->server);
        push @{ $srv_cfg->{$key} }, $arg;
    }
}

sub merge {
    my($base, $add) = @_;

    my %mrg = ();
    for my $key (keys %$base, %$add) {
        next if exists $mrg{$key};
        if ($key eq 'MyPlus') {
            $mrg{$key} = ($base->{$key}||0) + ($add->{$key}||0);
        }
        elsif ($key eq 'MyList') {
            push @{ $mrg{$key} },
                @{ $base->{$key}||[] }, @{ $add->{$key}||[] };
        }
        elsif ($key eq 'MyAppend') {
            $mrg{$key} = join " ", grep defined, $base->{$key},
                                                 $add->{$key};
        }
        else {
            # override mode
```

```
            $mrg{$key} = $base->{$key} if exists $base->{$key};
            $mrg{$key} = $add->{$key}  if exists $add->{$key};
        }
    }

    return bless \%mrg, ref($base);
}

1;
__END__
```

It's probably a good idea to specify all the attributes for the `@APACHE_MODULE_COMMANDS` entries, but here for simplicity we have only assigned to the *name* directive, which is a must. Since all our directives take a single argument, `Apache::TAKE1`, the default *args_how*, is what we need. We also allow the directives to appear anywhere, so `Apache::OR_ALL`, the default for *req_override*, is good for us as well.

We use the same callback for the directives `MyPlus`, `MyAppend` and `MyOverride`, which simply assigns the specified value to the hash entry with the key of the same name as the directive.

The `MyList` directive's callback stores the value in the list, a reference to which is stored in the hash, again using the name of the directive as the key. This approach is usually used when the directive is of type `Apache::ITERATE`, so you may have more than one value of the same kind inside a single container. But in our example we choose to have it of the type `Apache::TAKE1`.

In both callbacks in addition to storing the value in the current *directory* configuration, if the value is configured in the main server or the virtual host (which is when `$parms->path` is false), we also store the data in the same way in the server configuration object. This is done in order to be able to query the values assigned at the server and virtual host levels, when the request is made to one of the sub-resources. We will show how to access that information in a moment.

Finally we use the same merge function for merging directory and server configuration objects. For the key `MyPlus` (remember we have used the same key name as the name of the directive), the merging function performs, the obvious, summation of the ancestor's merged value (base) and the current resource's value (add). `MyAppend` joins the values into a string, `MyList` joins the lists and finally `MyOverride` (the default) overrides the value with the current one if any. Notice that all four merging methods take into account that the values in the ancestor or the current configuration object might be unset, which is the case when the directive wasn't used by all ancestors or for the current resource.

At the end of the merging, a blessed reference to the merged hash is returned. The reference is blessed into the same class, as the base or the add objects, which is `MyApache::CustomDirectives` in our example. That hash is used as the merged ancestor's object for a sub-resource of the resource that has just undergone merging.

Next we supply the following *httpd.conf* configuration section, so we can demonstrate the features of this example:

```
PerlLoadModule MyApache::CustomDirectives
MyPlus 5
MyList     "MainServer"
MyAppend   "MainServer"
MyOverride "MainServer"
Listen 8081
<VirtualHost _default_:8081>
    MyPlus 2
    MyList     "VHost"
    MyAppend   "VHost"
    MyOverride "VHost"
    <Location /custom_directives_test>
        MyPlus 3
        MyList     "Dir"
        MyAppend   "Dir"
        MyOverride "Dir"
        SetHandler modperl
        PerlResponseHandler MyApache::CustomDirectivesTest
    </Location>
    <Location /custom_directives_test/subdir>
        MyPlus 1
        MyList     "SubDir"
        MyAppend   "SubDir"
        MyOverride "SubDir"
    </Location>
</VirtualHost>
<Location /custom_directives_test>
    SetHandler modperl
    PerlResponseHandler MyApache::CustomDirectivesTest
</Location>
```

`PerlLoadModule` loads the Perl module `MyApache::CustomDirectives` and then installs a new Apache module named `MyApache::CustomDirectives`, using the callbacks provided by the Perl module. In our example functions `SERVER_CREATE` and `DIR_CREATE` aren't provided, so by default an empty hash will be created to represent the configuration object for the merging functions. If we don't provide merging functions, Apache will simply skip the merging. Though you must provide a callback function for each directive you add.

After installing the new module, we add a virtual host container, containing two resources (which at other times called locations, directories, sections, etc.), one being a sub-resource of the other, plus one another resource which resides in the main server.

We assign different values in all four containers, but the last one. Here we refer to the four containers as *MainServer*, *VHost*, *Dir* and *SubDir*, and use these names as values for all configuration directives, but `MyPlus`, to make it easier understand the outcome of various merging methods and the merging order. In the last container used by `<Location /custom_directives_test>`, we don't specify any directives so we can verify that all the values are inherited from the main server.

For all three resources we are going to use the same response handler, which will dump the values of configuration objects that in its reach. As we will see that different resources will see see certain things identically, while others differently. So here it the handler:

```
#file:MyApache/CustomDirectivesTest.pm
#-------------------------------------
package MyApache::CustomDirectivesTest;

use strict;
use warnings FATAL => 'all';

use Apache::RequestRec ();
use Apache::RequestIO ();
use Apache::Server ();
use Apache::ServerUtil ();
use Apache::Module ();

use Apache::Const -compile => qw(OK);

sub get_config {
    Apache::Module->get_config('MyApache::CustomDirectives', @_);
}

sub handler {
    my($r) = @_;
    my %secs = ();

    $r->content_type('text/plain');

    my $s = $r->server;
    my $dir_cfg = get_config($s, $r->per_dir_config);
    my $srv_cfg = get_config($s);

    if ($s->is_virtual) {
        $secs{"1: Main Server"}  = get_config(Apache->server);
        $secs{"2: Virtual Host"} = $srv_cfg;
        $secs{"3: Location"}     = $dir_cfg;
    }
    else {
        $secs{"1: Main Server"}  = $srv_cfg;
        $secs{"2: Location"}     = $dir_cfg;
     }

    $r->printf("Processing by %s.\n",

        $s->is_virtual ? "virtual host" : "main server");

    for my $sec (sort keys %secs) {
        $r->print("\nSection $sec\n");
        for my $k (sort keys %{ $secs{$sec}||{} }) {
            my $v = exists $secs{$sec}->{$k}
                ? $secs{$sec}->{$k}
                : 'UNSET';
            $v = '[' . (join ", ", map {qq{"$_"}} @$v) . ']'
                if ref($v) eq 'ARRAY';
            $r->printf("%-10s : %s\n", $k, $v);
        }
    }

    return Apache::OK;
```

```
}

1;
__END__
```

The handler is relatively simple. It retrieves the current resource (directory) and the server's configuration objects. If the server is a virtual host, it also retrieves the main server's configuration object. Once these objects are retrieved, we simply dump the contents of these objects, so we can verify that our merging worked correctly. Of course we nicely format the data that we print, taking a special care of array references, which we know is the case with the key *MyList*, but we use a generic code, since Perl tells us when a reference is a list.

It's a show time. First we issue a request to a resource residing in the main server:

```
% GET http://localhost:8002/custom_directives_test/

Processing by main server.

Section 1: Main Server
MyAppend   : MainServer
MyList     : ["MainServer"]
MyOverride : MainServer
MyPlus     : 5

Section 2: Location
MyAppend   : MainServer
MyList     : ["MainServer"]
MyOverride : MainServer
MyPlus     : 5
```

Since we didn't have any directives in that resource's configuration, we confirm that our merge worked correctly and the directory configuration object contains the same data as its ancestor, the main server. In this case the merge has simply inherited the values from its ancestor.

The next request is for the resource residing in the virtual host:

```
% GET http://localhost:8081/custom_directives_test/

Processing by virtual host.

Section 1: Main Server
MyAppend   : MainServer
MyList     : ["MainServer"]
MyOverride : MainServer
MyPlus     : 5

Section 2: Virtual Host
MyAppend   : MainServer VHost
MyList     : ["MainServer", "VHost"]
MyOverride : VHost
MyPlus     : 7

Section 3: Location
```

```
MyAppend   : MainServer VHost Dir
MyList     : ["MainServer", "VHost", "Dir"]
MyOverride : Dir
MyPlus     : 10
```

That's where the real fun starts. We can see that the merge worked correctly in the virtual host, and so it did inside the `<Location>` resource. It's easy to see that `MyAppend` and `MyList` are correct, the same for `MyOverride`. For `MyPlus`, we have to work harder and perform some math. Inside the virtual host we have main(5)+vhost(2)=7, and inside the first resource vhost_merged(7)+resource(3)=10.

So far so good, the last request is made to the sub-resource of the resource we have requested previously:

```
% GET http://localhost:8081/custom_directives_test/subdir/

Processing by virtual host.

Section 1: Main Server
MyAppend   : MainServer
MyList     : ["MainServer"]
MyOverride : MainServer
MyPlus     : 5

Section 2: Virtual Host
MyAppend   : MainServer VHost
MyList     : ["MainServer", "VHost"]
MyOverride : VHost
MyPlus     : 7

Section 3: Location
MyAppend   : MainServer VHost Dir SubDir
MyList     : ["MainServer", "VHost", "Dir", "SubDir"]
MyOverride : SubDir
MyPlus     : 11
```

No surprises here. By comparing the configuration sections and the outcome, it's clear that the merging is correct for most directives. The only harder verification is for `MyPlus`, all we need to do is to add 1 to 10, which was the result we saw in the previous request, or to do it from scratch, summing up all the ancestors of this sub-resource: 5+2+3+1=11.

1.4.1.1 Merging Entries Whose Values Are References

When merging entries whose values are references and not scalars, it's important to make a deep copy and not a shallow copy, when the references gets copied. In our example we merged two references to lists, by explicitly extracting the values of each list:

```
push @{ $mrg{$key} },
    @{ $base->{$key}||[] }, @{ $add->{$key}||[] };
```

While seemingly the following snippet is doing the same:

```
$mrg{$key} = $base->{$key};
push @{ $mrg{$key} }, @{ $add->{$key}||[] };
```

it won't do what you expect if the same merge (with the same `$base` and `$add` arguments) is called more than once, which is the case in certain cases. What happens in the latter implementation, is that the first line makes both `$mrg{$key}` and `$base->{$key}` point to the same reference. When the second line expands the `@{ $mrg{$key} }`, it also affects `@{ $base->{$key} }`. Therefore when the same merge is called second time, the `$base` argument is not the same anymore.

Certainly we could workaround this problem in the mod_perl core, by freezing the arguments before the merge call and restoring them afterwards, but this will incur a performance hit. One simply has to remember that the arguments and the references they point to, should stay unmodified through the function call, and then the right code can be supplied.

1.4.1.2 Merging Order Consequences

Sometimes the merging logic can be influenced by the order of merging. It's desirable that the logic will work properly regardless of the merging order.

In Apache 1.3 the merging was happening in the following order:

```
(((base_srv -> vhost) -> section) -> subsection)
```

Whereas as of this writing Apache 2.0 performs:

```
((base_srv -> vhost) -> (section -> subsection))
```

A product of subsections merge (which happen during the request) is merged with the product of the server and virtual host merge (which happens at the startup time). This change was done to improve the configuration merging performance.

So for example, if you implement a directive `MyExp` which performs the exponential: `$mrg=$base**$add`, and let's say there directive is used four times in *httpd.conf*:

```
MyExp 5
<VirtualHost _default_:8001>
     MyExp 4
     <Location /section>
         MyExp 3
     </Location>
     <Location /section/subsection>
         MyExp 2
     </Location>
```

The merged configuration for a request *http://localhost:8001/section/subsection* will see:

```
(5 ** 4) ** (3 ** 2)  = 1.45519152283669e+25
```

under Apache 2.0, whereas under Apache 1.3 the result would be:

```
( (5 ** 4) ** 3) ** 2 = 5.96046447753906e+16
```

which is not quite the same.

Chances are that your merging rules work identically, regardless of the merging order. But you should be aware of this behavior.

1.5 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman <stas (at) stason.org>

1.6 Authors

- Stas Bekman <stas (at) stason.org>

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents: