1 Introducing mod_perl Handlers

1.1 Description

This chapter provides an introduction into mod_perl handlers.

1.2 What are Handlers?

Apache distinguishes between numerous phases for which it provides hooks (because the C functions are called *ap_hook_<phase_name>*) where modules can plug various callbacks to extend and alter the default behavior of the webserver. mod_perl provides a Perl interface for most of the available hooks, so mod_perl modules writers can change the Apache behavior in Perl. These callbacks are usually referred to as *handlers* and therefore the configuration directives for the mod_perl handlers look like: `PerlFooHandler`, where `Foo` is one of the handler names. For example `PerlResponseHandler` configures the response callback.

A typical handler is simply a perl package with a *handler* subroutine. For example:

```
file:MyApache2/CurrentTime.pm
-----------------------------
package MyApache2::CurrentTime;

use strict;
use warnings;

use Apache2::RequestRec ();
use Apache2::RequestIO ();

use Apache2::Const -compile => qw(OK);

sub handler {
    my $r = shift;

    $r->content_type('text/plain');
    $r->print("Now is: " . scalar(localtime) . "\n");

    return Apache2::Const::OK;
}
1;
```

This handler simply returns the current date and time as a response.

Since this is a response handler, we configure it as a such in *httpd.conf*:

```
PerlResponseHandler MyApache2::CurrentTime
```

Since the response handler should be configured for a specific location, let's write a complete configuration section:

```
PerlModule MyApache2::CurrentTime
<Location /time>
    SetHandler modperl
    PerlResponseHandler MyApache2::CurrentTime
</Location>
```

Now when a request is issued to *http://localhost/time* this response handler is executed and a response that includes the current time is returned to the client.

1.3 Handler Return Values

Different handler groups are supposed to return different values.

Make sure that you **always** explicitly return a wanted value and don't rely on the result of last expression to be used as the return value -- things will change in the future and you won't know why things aren't working anymore.

The only value that can be returned by all handlers is `Apache2::Const::OK`, which tells Apache that the handler has successfully finished its execution.

`Apache2::Const::DECLINED` is another return value that indicates success, but it's only relevant for phases of type `RUN_FIRST`.

HTTP handlers may also return `Apache2::Const::DONE` which tells Apache to stop the normal HTTP request cycle and fast forward to the `PerlLogHandler`, followed by `PerlCleanupHandler`. HTTP handlers may return any HTTP status, which similarly to `Apache2::Const::DONE` will cause an abort of the request cycle, by also will be interpreted as an error. Therefore you don't want to return `Apache2::Const::HTTP_OK` from your HTTP response handler, but `Apache2::Const::OK` and Apache will send the `200 OK` status by itself.

Filter handlers return `Apache2::Const::OK` to indicate that the filter has successfully finished. If the return value is `Apache2::Const::DECLINED`, mod_perl will read and forward the data on behalf of the filter. Please notice that this feature is specific to mod_perl. If there is some problem with obtaining or sending the bucket brigades, or the buckets in it, filters need to return the error returned by the method that tried to manipulate the bucket brigade or the bucket. Normally it'd be an `APR::` constant.

Protocol handler return values aren't really handled by Apache, the handler is supposed to take care of any errors by itself. The only special case is the `PerlPreConnectionHandler` handler, which, if returning anything but `Apache2::Const::OK` or `Apache2::Const::DONE`, will prevent from `PerlConnectionHandler` to be run. `PerlPreConnectionHandler` handlers should always return `Apache2::Const::OK`.

1.4 mod_perl Handlers Categories

The mod_perl handlers can be divided by their application scope in several categories:

- **Server life cycle**
  - **`PerlOpenLogsHandler`**
  - **`PerlPostConfigHandler`**
  - **`PerlChildInitHandler`**
  - **`PerlChildExitHandler`**
- **Protocols**

    - **PerlPreConnectionHandler**
    - **PerlProcessConnectionHandler**
- **Filters**
    - **PerlInputFilterHandler**
    - **PerlOutputFilterHandler**
- **HTTP Protocol**
    - **PerlPostReadRequestHandler**
    - **PerlTransHandler**
    - **PerlMapToStorageHandler**
    - **PerlInitHandler**
    - **PerlHeaderParserHandler**
    - **PerlAccessHandler**
    - **PerlAuthenHandler**
    - **PerlAuthzHandler**
    - **PerlTypeHandler**
    - **PerlFixupHandler**
    - **PerlResponseHandler**
    - **PerlLogHandler**
    - **PerlCleanupHandler**

1.5 Stacked Handlers

For each phase there can be more than one handler assigned (also known as *hooks*, because the C functions are called *ap_hook_<phase_name>*). Phases' behavior varies when there is more then one handler registered to run for the same phase. The following table specifies each handler's behavior in this situation:

```
  Directive                    Type
-------------------------------------
PerlOpenLogsHandler          RUN_ALL
PerlPostConfigHandler        RUN_ALL
PerlChildInitHandler         VOID
PerlChildExitHandler         VOID

PerlPreConnectionHandler     RUN_ALL
PerlProcessConnectionHandler RUN_FIRST

PerlPostReadRequestHandler   RUN_ALL
PerlTransHandler             RUN_FIRST
PerlMapToStorageHandler      RUN_FIRST
PerlInitHandler              RUN_ALL
PerlHeaderParserHandler      RUN_ALL
PerlAccessHandler            RUN_ALL
PerlAuthenHandler            RUN_FIRST
PerlAuthzHandler             RUN_FIRST
PerlTypeHandler              RUN_FIRST
PerlFixupHandler             RUN_ALL
PerlResponseHandler          RUN_FIRST
PerlLogHandler               RUN_ALL
```

```
PerlCleanupHandler          RUN_ALL

PerlInputFilterHandler      VOID
PerlOutputFilterHandler     VOID
```

Note: `PerlChildExitHandler` and `PerlCleanupHandler` are not real Apache hooks, but to mod_perl users they behave as all other hooks.

And here is the description of the possible types:

1.5.1 VOID

Handlers of the type `VOID` will be *all* executed in the order they have been registered disregarding their return values. Though in mod_perl they are expected to return `Apache2::Const::OK`.

1.5.2 RUN_FIRST

Handlers of the type `RUN_FIRST` will be executed in the order they have been registered until the first handler that returns something other than `Apache2::Const::DECLINED`. If the return value is `Apache2::Const::DECLINED`, the next handler in the chain will be run. If the return value is `Apache2::Const::OK` the next phase will start. In all other cases the execution will be aborted.

1.5.3 RUN_ALL

Handlers of the type `RUN_ALL` will be executed in the order they have been registered until the first handler that returns something other than `Apache2::Const::OK` or `Apache2::Const::DECLINED`.

For C API declarations see *include/ap_config.h*, which includes other types which aren't exposed by mod_perl handlers.

Also see mod_perl Directives Argument Types and Allowed Location

1.6 Hook Ordering (Position)

The following constants specify how the new hooks (handlers) are inserted into the list of hooks when there is at least one hook already registered for the same phase.

META: Not working yet.

META: need to verify the following:

- **`APR::Const::HOOK_REALLY_FIRST`**

  run this hook first, before ANYTHING.

- **`APR::Const::HOOK_FIRST`**

  run this hook first.

- **`APR::Const::HOOK_MIDDLE`**

  run this hook somewhere.

- **`APR::Const::HOOK_LAST`**

  run this hook after every other hook which is defined.

- **`APR::Const::HOOK_REALLY_LAST`**

  run this hook last, after EVERYTHING.

META: more information in mod_example.c talking about position/predecessors, etc.

1.7 Bucket Brigades

Apache 2.0 allows multiple modules to filter both the request and the response. Now one module can pipe its output as an input to another module as if another module was receiving the data directly from the TCP stream. The same mechanism works with the generated response.

With I/O filtering in place, simple filters, like data compression and decompression, can be easily implemented and complex filters, like SSL, are now possible without needing to modify the the server code which was the case with Apache 1.3.

In order to make the filtering mechanism efficient and avoid unnecessary copying, while keeping the data abstracted, the *Bucket Brigades* technology was introduced. It's also used in protocol handlers.

A bucket represents a chunk of data. Buckets linked together comprise a brigade. Each bucket in a brigade can be modified, removed and replaced with another bucket. The goal is to minimize the data copying where possible. Buckets come in different types, such as files, data blocks, end of stream indicators, pools, etc. To manipulate a bucket one doesn't need to know its internal representation.

The stream of data is represented by bucket brigades. When a filter is called it gets passed the brigade that was the output of the previous filter. This brigade is then manipulated by the filter (e.g., by modifying some buckets) and passed to the next filter in the stack.

The following figure depicts an imaginary bucket brigade:

bucket brigades

The figure tries to show that after the presented bucket brigade has passed through several filters some buckets were removed, some modified and some added. Of course the handler that gets the brigade cannot tell the history of the brigade, it can only see the existing buckets in the brigade.

Bucket brigades are discussed in detail in the protocol handlers and I/O filtering chapters.

1.8 Maintainers

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

- Stas Bekman [http://stason.org/]

1.9 Authors

-

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

Table of Contents: