Writing Apache modules in Free Pascal

Michaël Van Canneyt

April 1, 2007

Abstract

Apache has support for writing custom modules. It offers an extensive C API which can be used inside these modules. With the support of the translated headers, Free Pascal can be used to write apache modules as well. In this article, it will be demonstrated how this can be done in less than 50 lines of code with the aid of FPC or Lazarus.

1 Introduction

Apache modules can be used whenever a scripting language or CGI program would be too slow or put a too big strain on the web server. Typical cases could include applications which need a database connection that has a big overhead when setting up a connection. But this is not all, of course, modules are much more versatile: a custom authentication mechanism could for instance be set up, to handle HTTP authentication with a non-standard database back-end. Writing apache modules is a broad topic, and it is not the intention of this article to explain all possibilities. Instead, the idea is to show that not only it is possible to write modules for Apache in Free Pascal or Lazarus, in fact, it is very easy to do so, making it an attractive alternative to writing CGI programs.

The various Apache versions define different APIs for interaction with a custom made module: Currently, Free Pascal contains bindings for Apache 1.3, 2.0 and 2.2. When coding manually, the version number of Apache should be taken into account; The Lazarus support works with all versions of Apache transparently, i.e., the Lazarus support takes care of the different bindings and presents a uniform interface to the user.

The Lazarus support is geared towards serving web-pages, not towards creating a new authentication mechanism or other specific modules: for this, using the raw apache headers would be necessary. Since the architecture is rather different between version 1.3 and the 2.X versions, only the 2.2 version will be discussed here. The changes for supporting version 2.0 are rather minute. To support the older 1.3 version of apache would take slightly more work.

The headers are translated in a unit called httpd. For 2.x versions of Apache, there is an extra unit `apr`, which

2 Architecture - the manual way

An Apache module is a dynamically loadable library. As such, writing an Apache module means writing a library, not a program. The Apache server loads this library, and expects to find a variable with a specific name in the library: This variable describes the module to apache, and registers one or more callbacks which will be called during the lifetime of the module.

The structure that describes the module is aptly named `module`, and looks like this:

```
module = record
  version: Integer;
  minor_version: Integer;
  module_index: Integer;
  name: PChar;
  dynamic_load_handle: Pointer;
  next: Pmodule_struct;
  magic: Cardinal;
  rewrite_args: procedure(process: Pprocess_rec); cdecl;
  create_dir_config:
    function(p: Papr_pool_t; dir: PChar): Pointer; cdecl;
  merge_dir_config:
    function(p: Papr_pool_t; base,new: Pointer): Pointer;cdecl;
  create_server_config:
    function(p: Papr_pool_t; s: Pserver_rec): Pointer;cdecl;
  merge_server_config:
    function(p: Papr_pool_t; base, new: Pointer): Pointer;cdecl;
  cmds: Pcommand_rec;
  register_hooks:
    procedure(p: Papr_pool_t); cdecl;
end;
```

There are many fields in this record, however, only a few fields are of interest:

name This is the name of the module as it is declared to Apache, and should match the name as declared in the apache configuration files. The name is case sensitive. It is the same name as appears in the `LoadModule` statement in the apache configuration files:

```
LoadModule hello_module mod_hello.so
```

The 'hello_module' is the name of the module.

register_hooks This is a callback which can be used to register command-hooks. It needs to be of the specified type, and have the 'cdecl' callback type.

These 2 fields should always be used. All the other fields need to be initialized with a special call: `STANDARD20_MODULE_STUFF`. This call (actually the translation of a C macro) will fill several fields with standard values:

version Will be filled with the version number in the Apache header translation. It should match the actual apache version, or Apache will refuse to load the module.

minor_version will also be filled with a minor version number.

module_index will be filled with -1. Apache will fill it with a unique number.

dynamic_load_module will be set to Nil, and Apache will fill it with a handle.

next will be set to Nil. Apache will file it with the address of the next module.

magic a magic cookie

rewrite_args will be set to Nil.

It's best to zero out the record prior to calling `STANDARD20_MODULE_STUFF`. After this call, the `name` and `register_hooks` fields can be set. This should be done in the startup code of the library, before Apache attempts to read the content of the module record.

The `register_hooks` callback needs to point to a procedure that will register the various command-handlers that this handler provides. The register procedure needs to call the `ap_hook_handler` routine to register one or more command handlers. This routine is defined as follows:

```
procedure ap_hook_handler(pf: ap_HOOK_handler_t;
                          const aPre: PPChar;
                          const aSucc: PPChar;
                          nOrder: Integer);
```

As can be seen, it accepts 4 parameters:

pf This is the actual function that will be called when the hook is needed.

aPre A pointer to a nil-termined list of names of modules that should be run before this module is run. It can, and in fact mostly will, be `Nil`.

aSucc A pointer to a nil-termined list of names of modules that should be run after this module is run. It can be `Nil`.

nOrder This can be one of the constants `HOOK_FIRST`, `HOOK_MIDDLE` and `HOOK_LAST`: a rough grouping of handlers for a request. When processing a request, Apache first runs it through the handlers that are registered with `HOOK_FIRST`, then considers those with `HOOK_MIDDLE` and finally the ones with `HOOK_LAST`. The usual value for this argument is `HOOK_MIDDLE`.

The hook handler should be of the following type:

```
Function Handler(R: Prequest_rec): Integer; cdecl;
```

and it should return one of the values

OK which means the module has handled this stage of the request.

DONE which means the module has completely handled the request, and the server should do no more processing.

DECLINED if the module should not handle this request, in that case no more modules will get a chance to handle the request.

The `R` parameter is a pointer to a record which describes the request. This is a huge record, with many fields. The main fields of interest are:

handler The name of the request handler. This should match the handler name as specified in the `SetHandler` directive of the apache configuration.

method the HTTP method.

unparsed_uri unparsed request from the client.

args arguments to request (QUERY_ARGS)

path_info the request path from the client

filename the file part of the request from the client.

header_only This is nonzero if only headers should be returned to the client.

headers_in the request headers passed by the client.

content_type content type of response. This should always be set.

headers_out headers to be sent to the client. They can be actually sent by the `ap_send_http_header` call.

Sending the content back to a client needs to be done with the `ap_rputs` or `ap_rwrite` calls:

```
function ap_rputs(const str: PChar;
                  r: Prequest_rec): Integer;
function ap_rwrite(const buf: Pointer;
                   nbyte: Integer;
                   r: Prequest_rec): Integer;
```

The first one sends a string `str` to the client. It scans for a null-terminator, and sends as many bytes as necessary. The second one copies `nbyte` bytes from the memory location-buf to the client. Both accept the `r` parameter, which describes the request. Obviously, the second call is faster than the first, but the first is more safe against buffer overflows.

3 The minimal 'Hello, world' example

Armed with all this knowledge, it's now quite simple to write a simple apache module. The first part is the request handler. In the example, it simply writes a page to the output. This is done in the `DefaultHandler` routine:

```
uses SysUtils, httpd, apr;

Const
  HTMLPage = '<HTML>'+LineEnding+
             '<HEAD>'+LineEnding+
             '<TITLE>Hello There</TITLE>'+LineEnding+
             '</HEAD>'+LineEnding+
             '<BODY BGCOLOR="#FFFFFF">'+LineEnding+
             '<H1>Hello world</H1>'+LineEnding+
             'Hello world from the FPC apache module !'+LineEnding+
             '</BODY></HTML>' + LineEnding;


function DefaultHandler(r: Prequest_rec): Integer; cdecl;

begin
  if not SameText(strpas(R^.handler), 'hello-handler') then
    Exit(DECLINED);
  ap_set_content_type(r,'text/html');
  if (R^.header_only<>0) then
    Exit(OK);
  ap_rputs(HtmlPage, r);
  Result := OK;
end;
```

The first thing that should be done, is check whether the module should handle the request. This is done by comparing the 'handler' field of the request with the name of the request handler. If the module should not handle the request, the handler can exit with a status of `DECLINED`.

If the handler should handle the request, the first thing to do is to set the content-type of the response. The `ap_set_content_type` call should be used for this. If the request was only for headers, the handler simply exits after that. If not, the `ap_rputs` call is used to return a simple HTML page to the client.

With the handler coded, we can register the handler in the registration hook:

```
procedure RegisterHooks(p: Papr_pool_t); cdecl;
begin
  ap_hook_handler(@DefaultHandler, nil, nil, APR_HOOK_MIDDLE);
end;
```

Lastly, the module record describing our module must be set up:

```
var
 hello_module: module;
  {$ifdef unix} public name 'hello_module';{$endif}

{$ifdef WINDOWS}
exports hello_module name 'hello_module';
{$endif}

begin
  FillChar(hello_module, sizeof(hello_module), 0);
  STANDARD20_MODULE_STUFF(hello_module);
  with hello_module do
    begin
    name := 'mod_hello.so';
    register_hooks := @RegisterHooks;
    end;
end.
```

The `public` modifier makes sure the module record is exported from the library on Unix-type systems. On windows, the `exports` statement takes care of this. Currently, the modifier is necessary because FPC does not allow to export variables on Unix systems. This limitation will be removed from future versions.

That's it. The above code is enough to create an apache module. All that needs to be done is to

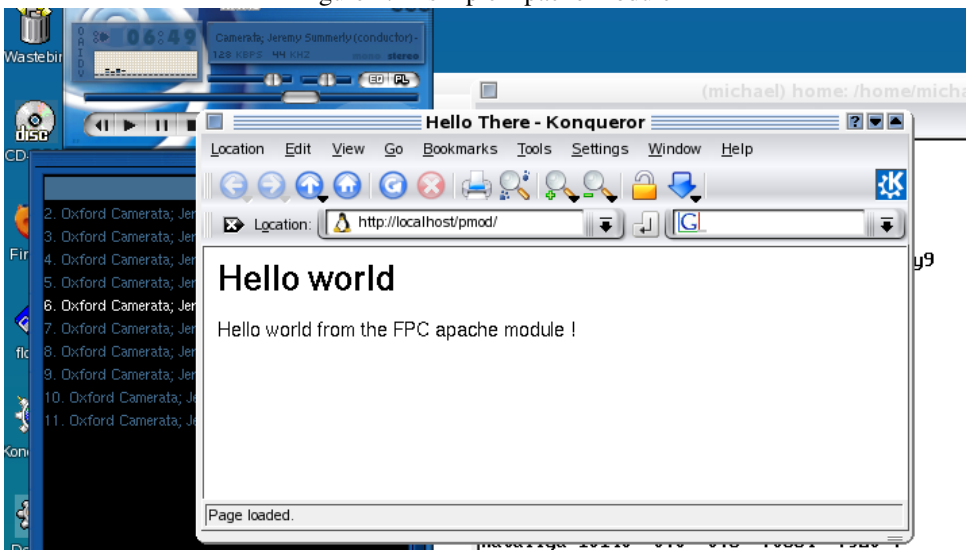
1. compile the module, and copy the module to the apache modules directory. This directory is dependent on the installation.
2. Tell apache to load the module. To do this, a statement like the following should be added to the apache configuration file:

   ```
   loadmodule hello_module mod_hello.so
   ```

   on unix, or

   ```
   loadmodule hello_module mod_hello.dll
   ```

Figure 1: A simple Apache module



on windows.

3. Tell apache to use the module. For this, a statement like the following can be used:

```
<Location /pmod>
  SetHandler hello-handler
</Location>
```

The above configuration line will make sure that a URL like

```
http://yourhost/pmod
```

will be handled by the mod_hello module.

After the following was done, apache must be restarted. Once restarted, the configuration can be tested, using the URL `http://localhost/pmod`, and something similar to the figure 1 on page 6 should appear.

4 Lazarus support for Apache modules

The previous section has shown that creating a simple apache module is not really difficult. When creating more elaborate modules, the ease of use of the Lazarus IDE and it's various RAD capabilities can make developing modules really easy. In earlier contributions, it was shown what support Lazarus offers for CGI programming: the Apache module support allows to re-use the modules – introduced for CGI support – for handling requests in apache modules.

The 'weblaz' package (which should be included with the Lazarus IDE by the time this article goes in print) can be installed in the Lazarus IDE. Next to the Lazarus CGI support, it will also install support for Apache Modules. The actual unit with Apache support (fpapache) is distributed with Free Pascal.

When installing the support for Apache modules, 2 things should be taken into account:

1. Free Pascal comes with support for Apache 1.2, 2.0 and 2.2. The compiler path should point to the directory for the Apache version that the module should be installed in. The lazarus support currently works only with version 2.0 or 2.2, so the directory httpd-2.0 or httpd-2.2 should be added to the unit patch. Or the unused directories may be archived and deleted to avoid mistakes.

2. When compiling on Linux 64-bit systems (x86_64 systems), the LCL must be recompiled with PIC support (Position Independent Code). This is currently not done by default, so when compiling lazarus the option '-Cg' should be added to the compiler options. When compiling using the Makefiles, the following cmmand can be used:

   ```
   make clean all OPT=-Cg
   ```

   If Lazarus is rebuilt from within the IDE, the var-Cg option can be added in the 'Configure "Build Lazarus"' dialog, available under the 'Tools' menu.

   This is not necessary on Windows or i386 systems.

Failure to observe the second point will result in linker errors when compiling the Apache module.

Once the 'Weblaz' package is installed, the support for Apache modules becomes visible in the Lazarus 'New item' dialog, where 'Apache module' item can be chosen.

Choosing the 'Apache module' option will create a main file (a library) and 1 web module. The web module will function in much the same way as it worked for CGI programs, and multiple modules can be made.

The main library file contains code looking like this (comments stripped):

```
Library mod_demo;

{$mode objfpc}{$H+}

Uses
  fpWeb,lazweb,httpd,fpApache,dmdemo;

Const
  ModuleName='demo_module';
  HandlerName=ModuleName;

Var
  DefaultModule : module;
   {$ifdef unix} public name ModuleName;{$endif unix}

{$ifdef windows}
Exports defaultmodule name ModuleName;
{$endif windows}

begin
  Application.Title:='mod_demo';
  Application.ModuleName:=ModuleName;
  Application.HandlerName:=HandlerName;
  Application.SetModuleRecord(DefaultModule);
  Application.Initialize;
end.
```

The meaning of most of this should be clear:

- The `ModuleName` constant is the name by which the module record is exported from the library, it should be the same as the name in the `LoadModule` statement in the config file of Apache.
- The `HandlerName` constant is the name of the handler for which the apache module will accept requests, and which should be equal to the name used in the `SetHandler` statement in the config file. By default, the handler name is the same as the module name.
- The `SetModuleRecord` statement is used to indicate to the apache support where the module record is.
- The `Initialize` call will fill the `module` record with the appropriate content.

Note that there is no 'Run' statement for the `Application` instance. This is logical, since it will be the Apache webserver which will call the appropriate callbacks.

The `Application` instance, of type `TApacheApplication`, will register 1 hook with Apache: all requests are guided through this hook. By default, the hook is registered with the `HandlerPriority` property of the Application instance. This is an enumerated property of type `THandlerPriority` and can have one of the following 3 values:

hpFirst corresponds to `HOOK_FIRST` in `apr_hook_handler`

hpMiddle the default value, corresponding to `HOOK_MIDDLE`

hpLast corresponds to `HOOK_LAST`.

The `BeforeModules` and `AfterModules` properties of the `TApacheApplication` class are `TStrings` and can be used to specify to apache when the module should be loaded: the names of the modules can simply be added to the lists, one module per line.

The handler that is registered with apache will only handle a request if the `Handler` field in the Apache `request_rec` record matches the `HandlerName` property of the Application instance. This is not always desirable, and therefore a `BeforeRequest` handler is provided. This handler is of the following type:

```
Procedure(Sender : TObject;
          Const AHandler : String;
          Var AllowRequest : Boolean) of object;
```

The `AllowRequest` will be initialized with `True` if the ruquested handler matches the `HandlerName` property. If, on return of the event handler, `AllowRequest` is `False`, the request will not be handled, and a `DECLINED` value will be reported to Apache.

After checking the handler, the application object must decide which web module (there can be more than one) will handle the request. The algorithm used to determine which module will handle the request is identical to the one used in the CGI application. That is, it examines the `PathInfo` property of the request. For a URL which looks like this:

```
http://www.myserver.com/mymodule/myaction?ID=23&What=Add
```

the `PathInfo` property will contain

```
mymodule/myaction
```

Now, the first part of this path will be used to determine the module wich should handle the request. In the above case, this will be `mymodule`, and a web module which has the name 'mymodule' will be searched and instantiated, and the request will be passed on to it.

This behaviour is not always desirable. Supposing a website exists which serves both statical pages and which contains also a module (`demo_module`) to handle special, interactive requests. This (pascal) module contains 2 webmodules: `mymodule` and `myothermodule`. In that case, the configuration file of the website might contain something like

```
<Location /mylocation>
  SetHandler demo_module
</Location>
```

All requests that do not start with /mylocation will be treated as regular files (or some other handler may exist for it). But, all requests to apache that do start with /mylocation, will be routed through handler `demo_module`. The idea is that urls like

```
http://www.myserver.com/mylocation/mymodule/myfirstaction
http://www.myserver.com/mylocation/myothermodule/myotheraction
```

are handled by the first and second webmodule, respectively. But, when the request arrives in the Pascal module, the request handler examines the first component of the path, and would therefor look for a module called `mylocation`, which will of course not be found.

To cater for this situation, the `BaseLocation` property is introduced in `TApacheApplication`. The value of this property is first removed from the `PathInfo` property before determining what module to call. In the above case, this would mean adding a line

```
  Application.BaseLocation:='/mylocation';
```

to make sure that the second part of the path is used to determine the name of the module that should handle the request.

5 A simple Lazarus demo

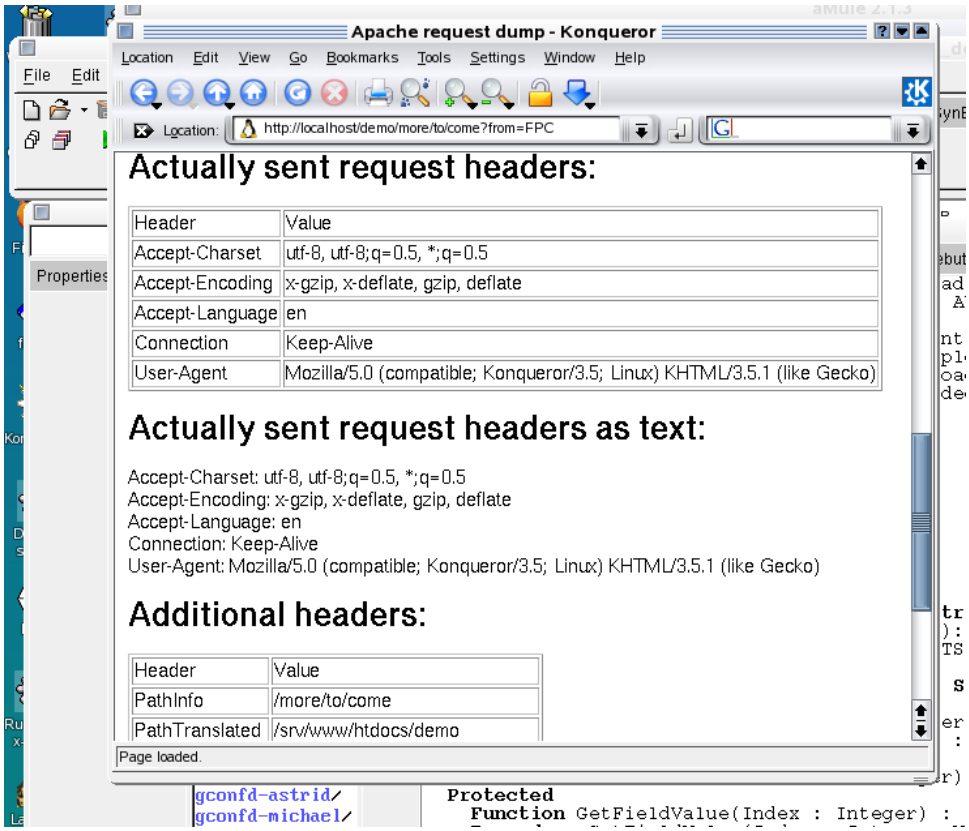
To demonstrate all this, a small demo is written which will demonstrate the concept. After starting Lazarus, the 'Apache module' item is chosen in the 'File - New' dialog. The value of the `ModuleName` constant in the project source file is changed to `module_demo`.

The name of the generated webmodule is changed to 'DemoWebModule' and saved in a file dmdemo The registration statement in the initialization section of the unit is changed to

```
initialization
  {$i dmdemo.lrs}
  RegisterHTTPModule('demo',TDemoWebModule);
end.
```

The `OnRequest` event is coded with the following:

```
procedure TDemoWebModule.FPWebModule1Request(Sender: TObject;
  ARequest: TRequest; AResponse: TResponse; var Handled: Boolean);
begin
  AResponse.ContentType:='text/html';
  AResponse.Contents.Add('<HTML><HEAD><TITLE>Apache request dump</TITLE></HEAD>')
```




```
  AResponse.Contents.Add('<BODY>');
  DumpRequest(Arequest,AResponse.Contents);
  AResponse.Contents.Add('</BODY></HTML>');
  Aresponse.SendContent;
  Handled:=True;
end;
```

The `DumpRequest` routine is in the standard `webutil` unit and will simply produce some tables which list all the request variables passed by Apache.

The module is ready to be compiled. To register it with apache, it should first be copied to the apache module directory, after which the following statements can be added to the apache configuration files:

```
LoadModule demo_module mod_demo.so

<Location /demo>
  SetHandler demo_module
</Location>
```

After a restart of Apache, entering the following URL:

```
http://localhost/demo/more/to/come?from=FPC
```

will result in a page looking like figure 5 on page 10.

6 Conclusion

In this article, it has been shown that creating an Apache module in Free Pascal is not only possible, but actually quite easy. Depending on ones preference, the module can be coded with the rather low-level Apache API, but it can equally easy be created with the full RAD power of lazarus. The architecture is done in such a way that the same code (the webmodules) can be re-used in both CGI applications (when apache is not an option) or in real Apache modules. In time, if someone takes the trouble to implement the IIS interfaceAPI, the webmodules should also be usable in Microsofts webserver.

The main problem in coding a module is understanding how requests are handled by apache, and configuring both the module and Apache so the requests are sent correctly to the webmodule that should handle it. Creating content is the easy part: for this, FPC also has some components, but the discussion of these will be left for a future contribution.