PageCare Apache Module

Jose San Leandro

February 15, 2017

Contents

1 Introduction **1**
- 1.1 Setting up Docker . . . 2
- 1.2 Added Apache development libraries to the Docker image . . 3
- 1.3 First attempt . . . 6
- 1.4 Second attempt . . . 11
- 1.5 Enabling automatic reload . . . 16
- 1.6 Learning about Apache modules . . . 17
- 1.7 The sample page from mod_learn.c . . . 18

2 Designing our Apache module **21**
- 2.1 Match every request . . . 23

3 Troubleshooting **23**
- 3.1 The Docker container does not start . . . 23
- 3.2 Apache does not load my module . . . 23

Contents

1 Introduction

This document the process of building an Apache module, based on the Apache guide.

However we'll use a custom Docker image, with an Apache instance, and sharing our module through host volumes.

1.1 Setting up Docker

We'll extend our `base` image as usual, and start with a simple *apache/Dockerfile.template*:

```
FROM pagecare/base:latest              .
MAINTAINER ${MAINTAINER}               .

RUN \
  ${APTGET_INSTALL} -u apache && \
  ${APTGET_CLEANUP}

VOLUME [ "/var/www", "/usr/local/src/pagecare" ]
EXPOSE 80
@include("copy-metadata")
@include("symlinks")
@include("instructions")
```

We're using set-square to build our images, so, among other things (providing information about the maintain.r and so on), we need to include a `README` file to help users fi.. out how to run our image.

```
= PageCare Apache module

Docker image to develop PageCare's Apache module.

== Examples
 - Run PageCare's Apache module from sources:
  docker run -d --name pagecare -v [workspace]:/usr/local/src/pagecare pagecare/apache-
```

Building it is simple as well:

```
./build.sh apache
```

Simple and wrong :)

```
Package apache is not available, but is referred to by another package.
This may mean that the package is missing, has been obsoleted, or
is only available from another source

E: Package 'apache' has no installation candidate
```

1.2 Added Apache development libraries to the Docker image

We don't remember the name of the package for Apache in Ubuntu, so the best approach is to run *base* image and find out by ourselves.

```
> docker run -it --rm --entrypoint=/bin/bash pagecare/apache
root@[container-id]:/# apt-cache search apache
[..]
apache2 - Apache HTTP Server
[..]
apache2-dev - Apache HTTP Server (development headers)
[..]
apache2-utils - Apache HTTP Server (utility programs for web servers)
[..]
```

Let's change our Dockerfile template accordingly:

```
[..]
RUN \
  ${APTGET_INSTALL} -u apache2 apache2-dev apache2-utils gcc && \
  ${APTGET_CLEANUP}
[..]
```

And try again. Now it succeeds. Our purpose is to find out where to place the files after compiling our module, and make sure Apache finds them.

Let's run our new image to find out.

```
> docker run -it --rm --entrypoint=/bin/bash pagecare/apache
root@[container-id]:/# cd /etc/apache2
root@[container-id]:/etc/apache2# less apache2.conf
[..]
# It is split into several files forming the configuration hierarchy outlined
# below, all located in the /etc/apache2/ directory:
#
#       /etc/apache2/
#       |-- apache2.conf
#       |       `--  ports.conf
#       |-- mods-enabled
#       |       |-- *.load
#       |       `-- *.conf
```

```
#       |-- conf-enabled
#       |       `-- *.conf
#       `-- sites-enabled
#               `-- *.conf
[..]
root@[container-id]:/etc/apache2# cat /etc/apache2/mods-enabled/autoindex.load
LoadModule autoindex_module /usr/lib/apache2/modules/mod_autoindex.so
```

After reviewing the folder structure, Apache should be able to find our module, provided we create a proper symbolic link to our files, in a host volume.

Let's try it out. Our Dockerfile template now reads:

```
FROM pagecare/base:latest
MAINTAINER ${MAINTAINER}

RUN \
  ${APTGET_INSTALL} -u apache2 apache2-dev apache2-utils gcc && \
  ${APTGET_CLEANUP} && \
  mkdir /usr/local/lib/apache2 && \
  cd /etc/apache2/mods-available && \
  ln -s /usr/local/lib/apache2/learn.conf learn.conf && \
  echo "LoadModule learn_module /usr/local/lib/apache2/mod_learn.so" > learn.load && \
  a2enmod learn

VOLUME ["/var/www", "/usr/local/lib/apache2"]

WORKDIR /usr/local/lib/apache2

EXPOSE 80
@include("copy-metadata")
@include("symlinks")
@include("instructions")
```

So far so good. However, we're using Phusion-baseimage, so we must provide either a `rc.local` file, or implement a new `service` that will be responsible of running our Apache instance.

Both approaches are similar, but for simplicity we'll start writing a `rc.local` file to launch Apache.

```
#!/bin/bash
```

```
service apache2 start
```

Finally, our Dockerfile gets

```
FROM pagecare/base:latest
MAINTAINER ${MAINTAINER}

COPY rc.local /etc/rc.local

RUN \
  ${APTGET_INSTALL} -u apache2 apache2-dev apache2-utils gcc && \
  ${APTGET_CLEANUP} && \
  mkdir /usr/local/lib/apache2 && \
  cd /etc/apache2/mods-available && \
  ln -s /usr/local/lib/apache2/learn.conf learn.conf && \
  echo "LoadModule learn_module /usr/local/lib/apache2/mod_learn.so" > learn.load && \
  a2enmod learn && \
  chmod +x /etc/rc.local

VOLUME ["/var/www", "/usr/local/lib/apache2"]

EXPOSE 80
@include("copy-metadata")
@include("symlinks")
@include("instructions")
```

If we run it,

```
docker run -d -p 8888:80 pagecare/apache
```

we can check the Apache process is not running. Within the container, we don't have to guess what the error is. It doesn't find our module, and, since we have enabled it, it refuses to start.

```
> docker exec -it [container-id] /bin/bash
root@[container-id]:/# /etc/init.d/apache2 status
 * apache2 is not running
root@[container-id]:/# /etc/init.d/apache2 start
 * Starting web server apache2
 *
```

```
 * The apache2 configtest failed.
Output of config test was:
apache2: Syntax error on line 140 of /etc/apache2/apache2.conf: \
Syntax error on line 1 of /etc/apache2/mods-enabled/learn.load: \
Cannot load /usr/local/lib/apache2/mod_learn.so into server: \
/usr/local/lib/apache2/mod_learn.so: cannot open shared object \
file: No such file or directory
Action 'configtest' failed.
The Apache error log may have more information.
```

1.3 First attempt

Now that we have set up the Docker image, we can start following the Apache guide to build a sample module.

We first create a new folder somewhere:

```
> mkdir workspace
> cd workspace
```

Then, define the `AP_MODULE_DECLARE_DATA` module, in a new `mod\textunderscore{}learn.c` file

```
/* Include the required headers from httpd */
#include "httpd.h"
#include "http_core.h"
#include "http_protocol.h"
#include "http_request.h"

/* Define prototypes of our functions in this module */
static void register_hooks(apr_pool_t *pool);
static int learn_handler(request_rec *r);

/* Define our module as an entity and assign a function for registering hooks  */

module AP_MODULE_DECLARE_DATA learn_module =
{
    STANDARD20_MODULE_STUFF,
    NULL,            // Per-directory configuration handler
    NULL,            // Merge handler for per-directory configurations
    NULL,            // Per-server configuration handler
    NULL,            // Merge handler for per-server configurations
```

```
    NULL,            // Any directives we may have for httpd
    register_hooks   // Our hook registering function
};

/* register_hooks: Adds a hook to the httpd process */
static void register_hooks(apr_pool_t *pool)
{
    /* Hook the request handler */
    ap_hook_handler(learn_handler, NULL, NULL, APR_HOOK_LAST);
}

/* The handler function for our module.
 * This is where all the fun happens!
 */
static int learn_handler(request_rec *r)
{
    /* First off, we need to check if this is a call for the "learn" handler.
     * If it is, we accept it and do our things, it not, we simply return DECLINED,
     * and Apache will try somewhere else.
     */
    if (!r->handler || strcmp(r->handler, "learn")) return (DECLINED);

    // The first thing we will do is write a simple "Hello, world!" back to the client
    ap_rputs("Hello, world!", r);
    return OK;
}
```

However, it doesn't compile:

```
docker run -it --rm --entrypoint=/bin/bash -v workspace:/usr/local/lib/apache2 pagecare
$ apxs -i -a -c mod_learn.c
Use of uninitialized value $ENV{"LDFLAGS"} in concatenation (.) or string at /usr/bin/a
/usr/share/build-1/libtool --silent --mode=compile x86_64-pc-linux-gnu-gcc -prefer-pic
 -march=native -O2 -pipe -mmmx -msse -msse2 -mssse3 -msse4.1 -msse4.2  -DLINUX \
 -D_REENTRANT -D_GNU_SOURCE -pthread -I/usr/include/apache2  -I/usr/include/apr-1 \
 -I/usr/include/apr-1 -I/usr/include/db4.8  -c -o mod_learn.lo mod_learn.c && \
touch mod_learn.slo
mod_learn.c:13:1: error: unknown type name 'module'
 module AP_MODULE_DECLARE_DATA learn_module =
 ^
```

```
mod_learn.c:15:5: error: 'STANDARD20_MODULE_STUFF' undeclared here (not in a function)
     STANDARD20_MODULE_STUFF,
     ^
mod_learn.c:16:5: warning: excess elements in scalar initializer
     NULL,            // Per-directory configuration handler
     ^
mod_learn.c:16:5: warning: (near initialization for 'learn_module')
mod_learn.c:17:5: warning: excess elements in scalar initializer
     NULL,            // Merge handler for per-directory configurations
     ^
mod_learn.c:17:5: warning: (near initialization for 'learn_module')
mod_learn.c:18:5: warning: excess elements in scalar initializer
     NULL,            // Per-server configuration handler
     ^
mod_learn.c:18:5: warning: (near initialization for 'learn_module')
mod_learn.c:19:5: warning: excess elements in scalar initializer
     NULL,            // Merge handler for per-server configurations
     ^
mod_learn.c:19:5: warning: (near initialization for 'learn_module')
mod_learn.c:20:5: warning: excess elements in scalar initializer
     NULL,            // Any directives we may have for httpd
     ^
mod_learn.c:20:5: warning: (near initialization for 'learn_module')
mod_learn.c:22:1: warning: excess elements in scalar initializer
 };
 ^
mod_learn.c:22:1: warning: (near initialization for 'learn_module')
apxs:Error: Command failed with rc=65536
.
```

After googling this, the solution is simple: add a new *include* directive at the end.

```
6  [..]
7  #include "http_request.h"
8  #include "http_config.h"
9  [..]
```

Now it compiles, but cannot copy the file to a destination location which is not what we need.

```
> apxs -i -a -c mod_learn.c
Use of uninitialized value $ENV{"LDFLAGS"} in concatenation (.) \
or string at /usr/bin/apxs line 423.
/usr/share/build-1/libtool --silent --mode=compile x86_64-pc-linux-gnu-gcc \
 -prefer-pic -march=native -O2 -pipe -mmmx -msse -msse2 -mssse3 -msse4.1 -msse4.2 \
 -DLINUX -D_REENTRANT -D_GNU_SOURCE -pthread -I/usr/include/apache2 \
 -I/usr/include/apr-1   -I/usr/include/apr-1 -I/usr/include/db4.8 \
 -c -o mod_learn.lo mod_learn.c && touch mod_learn.slo
/usr/share/build-1/libtool --silent --mode=link x86_64-pc-linux-gnu-gcc \
 -o mod_learn.la   -rpath /usr/lib64/apache2/modules -module -avoid-version \
   mod_learn.lo
/usr/lib64/apache2/build/instdso.sh SH_LIBTOOL='/usr/share/build-1/libtool' \
 mod_learn.la /usr/lib64/apache2/modules
/usr/share/build-1/libtool --mode=install cp mod_learn.la /usr/lib64/apache2/modules/
libtool: install: cp .libs/mod_learn.so /usr/lib64/apache2/modules/mod_learn.so
cp: cannot create regular file '/usr/lib64/apache2/modules/mod_learn.so': Permission de
apxs:Error: Command failed with rc=65536
.
```

We want it to create the `mod\textunderscore{}_learn.so` file therein. `apxs` allows working with *template modules*, so let's check it out:

```
apxs -g -n learn
```

This creates a `learn` folder with the following files:

- `Makefile`: rules to build the module;
- `modules.mk`: additional rules included in the Makefile (indirectly via `/usr/lib64/apache2/build/special.mk`);
- `mod\textunderscore{}learn.c`: a sample module;
- `.deps`: an empty file.

However, in order to customize where the final `.so` file gets created, we'd need to copy some files (`instdso.sh`, `config_vars.mk`, `rules.mk`, `special.mk`) from Apache (`/usr/lib64/apache2/build`) to our folder, and perform some changes in some internal variables used when compiling. Some of the changes require us to use absolute paths, which is something we should avoid.

Anyway, here are the required changes:

- `rules.mk`

```
19c19
< include  $(top_builddir)/config_vars.mk
---
> include  $(top_builddir)/build/config_vars.mk
```

- `instdso.sh`: copy it from `/usr/lib64/apache2/build`.
- `config_vars.mk`

```
5,6c5
< #exp_libexecdir = /usr/lib64/apache2/modules
< exp_libexecdir = .
---
> exp_libexecdir = /usr/lib64/apache2/modules
10,11c9
< #exp_installbuilddir = /usr/lib64/apache2/build
< exp_installbuilddir = .
---
> exp_installbuilddir = /usr/lib64/apache2/build
45,46c43
< #libexecdir = /usr/lib64/apache2/modules
< libexecdir = [our-working-directory]
---
> libexecdir = /usr/lib64/apache2/modules
53,54c50
< #installbuilddir = /usr/lib64/apache2/build
< installbuilddir = .
---
> installbuilddir = /usr/lib64/apache2/build
```

- `special.mk`

```
27c27
< include $(top_builddir)/rules.mk
---
> include $(top_builddir)/build/rules.mk
32c32
<         $(top_srcdir)/instdso.sh SH_LIBTOOL='$(SH_LIBTOOL)' $$i $(DESTDIR)$(libexecdi
---
>         $(top_srcdir)/build/instdso.sh SH_LIBTOOL='$(SH_LIBTOOL)' $$i $(DESTDIR)$(lib
```

After these changes, running

```
> make
```

generates our beloved `mod\textunderscore{}learn.dso` module. However, our Docker container doesn't accept it.

```
root@[container-id]:/# /etc/init.d/apache2 start
 * Starting web server apache2
 *
 * The apache2 configtest failed.
Output of config test was:
apache2: Syntax error on line 140 of /etc/apache2/apache2.conf: \
 Syntax error on line 1 of /etc/apache2/mods-enabled/learn.load: \
 Cannot load /usr/local/lib/apache2/mod_learn.so into server: \
mod_learn.so: undefined symbol: ap_rputs
Action 'configtest' failed.
The Apache error log may have more information.
```

The cause is a mismatch between the `apxs` tool I used to compile the module, and the Apache which is trying to use it.

1.4 Second attempt

If we compile and build in the same environment as we work, things should work fine.

Let's start over. We need to install `libtool` package in our Docker image. And we'd like also to avoid coupling the image to the name of our Apache modules.

The Dockerfile is now:

```
FROM pagecare/base:latest
MAINTAINER ${MAINTAINER}

RUN \
  ${APTGET_INSTALL} -u apache2 apache2-dev apache2-utils libtool gcc && \
  mkdir /usr/local/lib/apache2

COPY rc.local /etc/rc.local

WORKDIR /usr/local/lib/apache2

VOLUME ["/var/www", "/usr/local/lib/apache2"]
```

```
EXPOSE 80
@include("copy-metadata")
@include("symlinks")
@include("instructions")
```

And the biggest changes are in the `rc.local` startup script, since it now looks for any modules in the host volume, so that Apache can see them.

```
#!/bin/bash

cd /etc/apache2/mods-available;

for d in $(find /usr/local/lib/apache2/ -maxdepth 1 -type d | grep -v -e '^/usr/local/l
  for ext in load conf; do
    ln -s ${d}/$(basename ${d}).${ext} $(basename ${d}).${ext};
  done
  a2enmod $(basename ${d});
done

service apache2 start
```

We'll compile our code inside the container from now on.

Let's start with the default sample module generated by `apxs`.

```
root@[container-id]:/usr/local/lib/apache2# rm -rf learn
root@[container-id]:/usr/local/lib/apache2# apxs -g -n learn
Creating [DIR]  learn
Creating [FILE] learn/Makefile
Creating [FILE] learn/modules.mk
Creating [FILE] learn/mod_learn.c
Creating [FILE] learn/.deps
root@[container-id]:/usr/local/lib/apache2# cd learn
root@[container-id]:/usr/local/lib/apache2/learn# make
[..]
root@[container-id]:/usr/local/lib/apache2/learn# make install
[..]
```

To test if it works, we have to create two files: one to load our module, and another one to bind it to the Apache flow.

```
root@[container-id]:/usr/local/lib/apache2/learn# cat <<EOF > learn.load
LoadModule learn_module /usr/lib/apache2/modules/mod_learn.so
EOF
root@[container-id]:/usr/local/lib/apache2/learn# cat <<EOF > learn.conf
<IfModule mod_learn.c>
  <Location "/learn">
    SetHandler learn
  </Location>
</IfModule>
EOF
root@[container-id]:/usr/local/lib/apache2/learn# a2enmod learn
root@[container-id]:/usr/local/lib/apache2/learn# service apache2 restart
```

When we visit now `http://localhost:8888/learn`, we can see the following text:

```
Hello, world
```

To be confident we can change our module and check those changes quickly, let's modify the sample text.

To do that, we first have to change the permissions of the files, since we created them inside the container, as root.

```
> chmod a+w *.c
```

Additionally, we'd like to automate the process of compiling the source files, installing the module, and restarting Apache, when we change anything.

We can use a simple script for that, adapted from a serverfault answer, `watch_module_changes.sh`:

```
#!/bin/bash

function compile() {
    make > /dev/null && \
    make install > /dev/null && \
    apxs -i -a -c ${FILE}
}

FILE="${1}"
cd "${2}"
LAST=$(md5sum "$FILE")
```

```
compile
service apache2 restart  > /dev/null 3>&1 2>&1 > /dev/null
while true; do
  sleep 1
  NEW=$(md5sum "$FILE")
  if [ "$NEW" != "$LAST" ]; then
    LAST="$NEW"
    compile && \
    service apache2 restart > /dev/null 3>&1 2>&1 > /dev/null && \
    echo "Apache restarted as ${FILE} changed"
  fi
done
```

We have to run this script when the container starts, so we'll add it to our `rc.local` script. To avoid messing up file permissions in `/usr/local/lib/apache2/`, we'll use `run-as.sh` script. We need also to include the $service_{user}$ template and the `build-settings.sh` file. We're changing the shell of *www-data* since it's the user Apache runs as, and we need it to be able to run *sudo* commands. It's not a typical use-case, but it fits our purposes. Keep in mind this is a Docker image used only for developing Apache modules.

```
defineEnvVar SERVICE_USER "The service user" "www-data";
defineEnvVar SERVICE_GROUP "The service group" "www-data";
defineEnvVar SERVICE_USER_SHELL "The shell of the service account" "/bin/bash";
defineEnvVar SERVICE_USER_HOME "The home of the service account" "/var/www/";
```

I'm using `run-as.sh` to avoid messing up the permissions of the folders that get shared as host volumes. It changes the *id* of the user so that it matches the *id* of the owner of the folder, and thus all permissions work just fine inside and outside the container.

```
#!/bin/bash

for d in $(find /usr/local/lib/apache2/ -maxdepth 1 -type d | grep -v -e '^/usr/local/l
  cd /etc/apache2/mods-available;
  for ext in load conf; do
    ln -s ${d}/$(basename ${d}).${ext} $(basename ${d}).${ext};
  done
  cd ${d};
  for f in $(find . -maxdepth 1 -name '*.c' | grep -v '#'); do
    # We don't need the ${d} parameter
```

```
    # but it makes easier to find out
    # which folder is being monitored
    # when inspecting processes via ps -ef
    /usr/local/bin/run-as.sh -U www-data -G www-data /usr/local/lib/apache2 -- /usr/loc
  done
  a2enmod $(basename ${d});
done

# To prevent issues with invalid modules
# when starting up, we let the container
# launch even if Apache initially doesn't.
service apache2 restart &

exit 0
```

The Dockerfile needs to include the new script. Since all the logic to enable the modules is launched upon container startup, we can omit enabling our specific module in the Dockerfile.

```
FROM pagecare/base:latest
MAINTAINER ${MAINTAINER}

@include("service_user")

COPY rc.local /etc/rc.local
COPY watch_module_changes.sh /usr/local/bin/watch_module_changes.sh

RUN \
  ${APTGET_INSTALL} -u apache2 apache2-dev apache2-utils libtool gcc && \
  ${APTGET_CLEANUP} && \
  mkdir /usr/local/lib/apache2 && \
  chmod +x /etc/rc.local /usr/local/bin/watch_module_changes.sh && \
  chsh -s ${SERVICE_USER_SHELL} ${SERVICE_USER} && \
  echo '${SERVICE_USER} ALL=NOPASSWD: ALL' >> /etc/sudoers

WORKDIR /usr/local/lib/apache2
VOLUME ["/var/www", "/usr/local/lib/apache2"]

EXPOSE 80
```

1.5 Enabling automatic reload

To make our changes immediately visible, we can setup LiveReloadX to receive notifications from our `watch_module_changes.sh`, and refresh the page for us.

We need to pay attention LiveReloadX uses port 35729 to notify changes to the browser. Afterwards, we just need to call it from within our `watch_module_changes.sh`

```
#!/bin/bash
function compile() {
  make clean && \
  make && \
  sudo make install && \
  sudo apxs -i -a -c ${FILE}
}

FILE="${1}"
cd "${2}"
LAST=$(md5sum "$FILE")
compile
sudo service apache2 restart > /dev/null 3>&1 2>&1 > /dev/null
livereloadx --include 'apache2.pid' /var/run/apache2/ &
while true; do
  sleep 1
  NEW=$(md5sum "$FILE")
  if [ "$NEW" != "$LAST" ]; then
      LAST="$NEW"
      compile && \
      sudo service apache2 restart > /dev/null 3>&1 2>&1 > /dev/null && \
      echo "Apache restarted as ${FILE} changed"
  fi
done
```

```
FROM pagecare/base:latest
MAINTAINER ${MAINTAINER}

@include("service_user")
@include("nodejs")

COPY rc.local /etc/rc.local
COPY watch_module_changes.sh /usr/local/bin/watch_module_changes.sh
```

```
RUN \
  ${APTGET_INSTALL} -u apache2 apache2-dev apache2-utils libtool gcc && \
  ${APTGET_CLEANUP} && \
  mkdir /usr/local/lib/apache2 && \
  chmod +x /etc/rc.local /usr/local/bin/watch_module_changes.sh && \
  chsh -s ${SERVICE_USER_SHELL} ${SERVICE_USER} && \
  echo '${SERVICE_USER} ALL=NOPASSWD: ALL' >> /etc/sudoers && \
  npm install -g livereloadx

WORKDIR /usr/local/lib/apache2
VOLUME ["/var/www", "/usr/local/lib/apache2"]

EXPOSE 80
EXPOSE 35729
```

Then, add the Firefox extension, and setup a new reload rule:

- url: `http://localhost:8888/learn`
- File: **[module-folder]/mod_learn.c**
- Execute action: **Force reload document**

We can now run our Apache container with:

```
docker run -it --rm -p 8888:80 -p 35729:35729 -v ${PWD}/workspace:/usr/local/lib/apache
```

1.6 Learning about Apache modules

Now that we have a proper environment, we can start learning Apache's internal API.

Let's print what information we have access to.

```
/*
  mod_learn.c -- Apache sample learn module
  [Autogenerated via ``apxs -n learn -g'']
  To play with this sample module first compile it into a
  DSO file and install it into Apache's modules directory
  by running:
    $ apxs -c -i mod_learn.c
  Then activate it in Apache's apache2.conf file for instance
```

```
  for the URL /learn in as follows:
    #   apache2.conf
    LoadModule learn_module modules/mod_learn.so
    <Location /learn>
    SetHandler learn
    </Location>
  Then after restarting Apache via
    $ apachectl restart
  you immediately can request the URL /learn and watch for the
  output of this module. This can be achieved for instance via:
    $ lynx -mime_header http://localhost/learn
  The output should be similar to the following one:
    HTTP/1.1 200 OK
    Date: Tue, 31 Mar 1998 14:42:22 GMT
    Server: Apache/1.3.4 (Unix)
    Connection: close
    Content-Type: text/html
```

1.7 The sample page from mod_learn.c

```
#include "httpd.h"
#include "http_config.h"
#include "http_protocol.h"
#include "ap_config.h"

static void print_string(const char *name, char *value, request_rec *r) {
    ap_rputs("<dt>", r);
    ap_rputs(name, r);
    ap_rputs("</dt>\n", r);
    ap_rputs("<dd>", r);
    if (value == NULL) {
        ap_rputs("null", r);
    } else {
        ap_rputs(value, r);
    }
    ap_rputs("</dd>\n", r);
}

static void print_pool(request_rec *r) {
    print_string("r->pool", "TODO", r);
```

```
}

static void print_connection(request_rec *r) {
    print_string("r->connection", "TODO", r);
}

static void print_server(request_rec *r) {
    print_string("r->server", "TODO", r);
}

static void print_next(request_rec *r) {
    print_string("r->next", "TODO", r);
}

static void print_prev(request_rec *r) {
    print_string("r->prev", "TODO", r);
}

static void print_main(request_rec *r) {
    print_string("r->main", "TODO", r);
}

static void print_request_time(request_rec *r) {
    print_string("r->request_time", "TODO", r);
}

static int learn_handler(request_rec *r)
{
    if (strcmp(r->handler, "learn")) {
        return DECLINED;
    }

    r->content_type = "text/html";

    if (!r->header_only) {
        ap_rputs("<html><head><title>Learn module</title></head><body>", r);
        ap_rputs("<h1>Learn module</h1>\n", r);
        ap_rputs("<dl>\n", r);
        print_pool(r);
        print_connection(r);
```

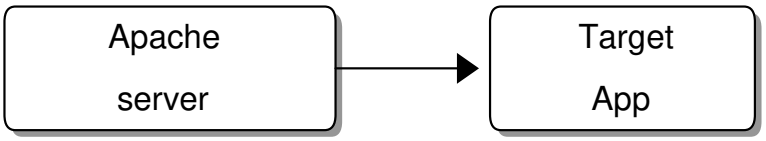
```
        print_server(r);
        print_next(r);
        print_prev(r);
        print_main(r);
        print_string("r->the_request", r->the_request, r);
        print_string("r->handler", r->handler, r);
        print_string("r->protocol", r->protocol, r);
        print_string("r->hostname", r->hostname, r);
        print_request_time(r);
        print_string("r->status_line", r->status_line, r);
        print_string("r->method", r->method, r);
        print_string("r->range", r->range, r);
        print_string("r->content_type", r->content_type, r);
        print_string("r->content_encoding", r->content_encoding, r);
        print_string("r->vlist_validator", r->vlist_validator, r);
        print_string("r->user", r->user, r);
        print_string("r->ap_auth_type", r->ap_auth_type, r);
        print_string("r->unparsed_uri", r->unparsed_uri, r);
        print_string("r->uri", r->uri, r);
        print_string("r->filename", r->filename, r);
        print_string("r->canonical_filename", r->canonical_filename, r);
        print_string("r->path_info", r->path_info, r);
        print_string("r->args", r->args, r);
        print_string("r->log_id", r->log_id, r);
        print_string("r->useragent_ip", r->useragent_ip, r);
        ap_rputs("</dl></body></html>\n", r);
    }
    return OK;
}

static void learn_register_hooks(apr_pool_t *p)
{
    ap_hook_handler(learn_handler, NULL, NULL, APR_HOOK_MIDDLE);
}

/* Dispatch list for API hooks */
module AP_MODULE_DECLARE_DATA learn_module = {
    STANDARD20_MODULE_STUFF,
    NULL,                  /* create per-dir    config structures */
    NULL,                  /* merge  per-dir    config structures */
```

```
    NULL,                   /* create per-server config structures */
    NULL,                   /* merge  per-server config structures */
    NULL,                   /* table of config file commands       */
    learn_register_hooks  /* register hooks                      */
};
```

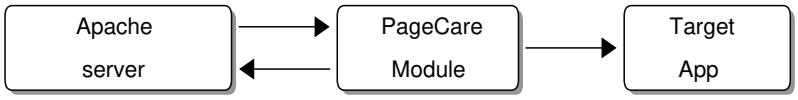
2 Designing our Apache module

We now have a working environment in which we can start implementing our custom Apache module. It's time to think what we're trying to achieve, and how to accomplish it once step at a time.

We start by thinking of a web application using Apache as web server (or reverse proxy). The details won't be affecting us. We want our module to support any web application served by Apache.



With our module in place, the scenario would be different. Our module would act as a *man-in-the-middle*, for the target application.



We want out module to intercept every request, check if that request satisfies some conditions, and depending on that let it pass, or provide a response directly ourselves.

Let's first start by copying the `mod_learn` module to `mod_pagecare`, and replace all occurrences of *learn* with *pagecare*.

The PageCare module is now as simple as this:

```
#include "httpd.h"
#include "http_config.h"
#include "http_protocol.h"
#include "ap_config.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int pagecare_handler(request_rec *r) {
    if (strcmp(r->handler, "pagecare")) {
        return DECLINED;
    }

    r->content_type = "text/html";

    if (!r->header_only) {
        ap_rputs("<html><head><title>PageCare module</title></head><body>\n", r);
        ap_rputs("<h1>Intercepted!</h1>", r);
        ap_rputs("</body></html>\n", r);
    }
    return OK;
}

static void pagecare_register_hooks(apr_pool_t *p) {
    ap_hook_handler(pagecare_handler, NULL, NULL, APR_HOOK_MIDDLE);
}

/* Dispatch list for API hooks */
module AP_MODULE_DECLARE_DATA pagecare_module = {
    STANDARD20_MODULE_STUFF,
    NULL,                   /* create per-dir    config structures */
    NULL,                   /* merge  per-dir    config structures */
    NULL,                   /* create per-server config structures */
    NULL,                   /* merge  per-server config structures */
    NULL,                   /* table of config file commands       */
    pagecare_register_hooks  /* register hooks                      */
};
```

2.1 Match every request

Let's configure our module to match all requests.

```
<IfModule mod_pagecare.c>
  <LocationMatch "/*">
    SetHandler pagecare
  </LocationMatch>
</IfModule>
```

Running our Docker image as usual yields the expected results.

3 Troubleshooting

3.1 The Docker container does not start

If after launching your docker container, it dies immediately (it's not listed in `docker ps`), run it without the `-d` flag.

```
docker run -it --rm -p 8888:80 -v $PWD/workspace:/usr/local/lib/apache2 pagecare/apache
```

You'll be able to inspect the problem, as it will be displayed in the console.

3.2 Apache does not load my module

Things to check:

- There's a `[module].conf` file in your working directory.
- Such file is linked from `/etc/apache2/mods-available` within the Docker container.
- The module is enabled (`mod2enmod [module]`).
- The module's `[module].load` file exists in your working directory,

and its contents point to `/usr/lib/apache2/modules/[module].so`.