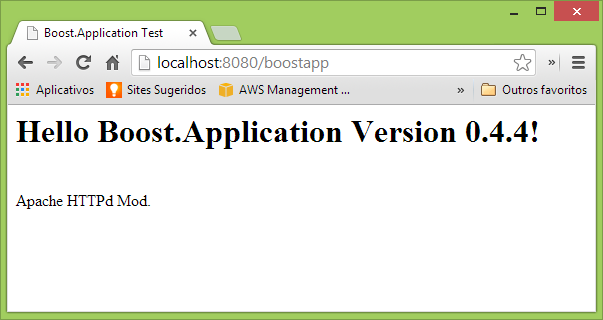
**Creating a new application mode for Boost.Application (0.4) Library.**



**Introduction**

This article presents an introduction to new version of proposed ‘Application’ library to boost.org. This new version is based on ‘aspect’ concept and is labeled as 0.4.x version.

**Warning!**

Some time ago, I wrote 2 articles based on version 0.3:

* Creating a Work Queue

<http://www.codeproject.com/Articles/664709/Creating-a-Work-Queue-Thread-Pool-Application-Usin>

Note that an updated sample of this code is present on new version of library (0.4) on example/uuid\_client\_server folder.

* Create a Windows Service Application Using the Boost.Application Library

<http://www.codeproject.com/Articles/662221/Create-a-Windows-Service-Application-Using-the-Boo>

Note that an updated sample of this code is present on new version of library (0.4) on example/work\_queue folder.

## Note that version 0.3 is no longer maintained! Version 0.4, is now maintained and receives regular updates.

**Feedback**

If you are boost user, and use Boost Mailing Lists, please provide your feedback about the library, directly on list. (Specify if you think if the library should be accepted as part of boost.org).  
<http://www.boost.org/community/groups.html>

If you are Code Project user, please provide your feedback about library direct in this page.

**Bugs**

If you find any BUG, please, send it to me at: [re.tf@acm.org](mailto:re.tf@acm.org)

|  |
| --- |
| **Caution** |
| The Boost.Application is not yet an official Boost C++ library. It wasn't reviewed and can't be downloaded from www.boost.org. This beta is available to boost community to know real interest and get comments for refinement. The intention is to submit library to formal review, if community think that it is interesting! |

**What is Boost.Application**

Boost.Application provides an application environment, or start point to any people that want a basic infrastructure to build a system application on Windows or Unix Variants (e.g. Linux, MacOS).

Boost.Application allows user extends library functionality too, e.g. in this article an Apache Httpd Module will be constructed as an “Boost Application Mode”.

An important concept of Boost.Application is the 'aspects', that was proposed by 'Vicente J. Botet Escriba'. An “aspect concept” allows easy extension and customization of library components.

Boost.Application provides many useful ready-to-use features, e.g:

* Run application as Windows Service;
* Run application as UNIX/POSIX Daemon;
* Plugin extension system;
* Process(executable) Single instance Instantiation support;
* Application SIGNAL/Callbacks customization;
* Windows Service Setup feature;
* Application Mode extension; (that will be presented in this article)
* And many others.

**Boost.Application Resources**

You can download library from GitHub at: <https://github.com/retf/Boost.Application>

An online documentation (under construction) is available at: <http://www.dokfile.com/appbeta4/docs/libs/application/doc/html/index.html>

**Library Official ‘Application Modes’**

An ‘application mode’ is a ready to use feature that are offered by library, in this version we have 2 official application modes (types):

         Common Application

This kind of application is a usual Interactive Terminal/Console Application.

         Server Application

This kind of application generates a [Service](http://en.wikipedia.org/wiki/Windows_service) (Windows), or a background process/[Daemon](http://en.wikipedia.org/wiki/Daemon_(computing)) (Unix).

This article will present how to extend library modes, it shows how to create a new simple ‘application mode’ to be used on Apache Web Server. With this ‘mode’ ready, the user can use it to generate any Apache Content Generator Module.

This is not official “library mode”, it is only a sample that illustrate how extend library functionality. The resultant mode is a very simple and don’t exposes all Apache Httpd Server API functionality. We will only implement a simple ‘Content Generator’ that supports the http GET VERB!

**0. Basic Setup**

This library is not an official Boost.org library yet, ant it uses other 2 not official libraries too, then the installations is not too easy.

1. The first step is: download the last version of boost and build it.

The build process of boost is not very easy too, but a good documentation is provided, refer to: <http://www.boost.org/doc/libs/1_54_0/more/getting_started/windows.html>

Or you can check this good article to help with build: <http://www.codeproject.com/Articles/11597/Building-Boost-libraries-for-Visual-Studio>

1. The second step is download :
   1. Boost.Application (<https://github.com/retf/Boost.Application/zipball/master>)
   2. Boost.TypeIndex (<https://github.com/apolukhin/type_index/zipball/master>)
   3. Boost.Singularity (<https://github.com/cppmaven/Singularity/zipball/master>)

You can unzip each of it on your ‘c:\’ or other directory. We will configure Visual Studio Project to find these directories later!

Note that Boost.org is moving to git, thus the folder structure of libraries can change in future, refer to boost.org to know more!

**1. Apache Http Web Server**

The Apache Web Server is one of the most deployed Web Server today. This article presents how to use Boost.Application library to create a way to extend Apache Web Server.

The Apache Server comprises a relative small core, and a lot of modules. Modules commonly held on a /modules directory and are loaded at runtime. This article shows how to build a new application mode for “Boost.Application”, that can be used to generate new ‘Apache Content Generator Module’.

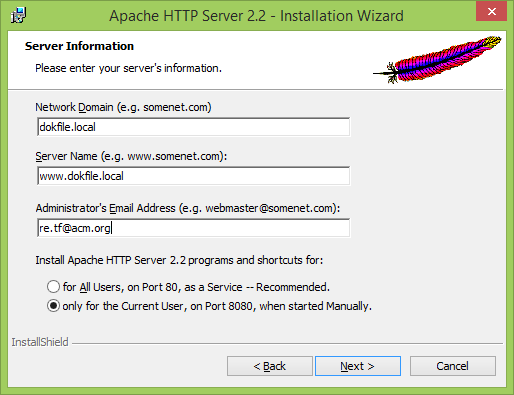
**1.1 Install Apache**

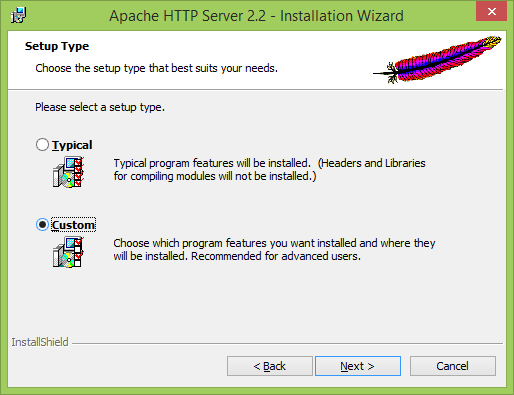
You need install Apache with include files.

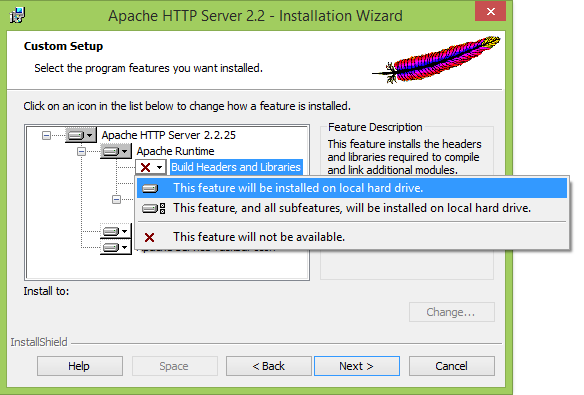
You can download a Apache (2.3) for windows here:

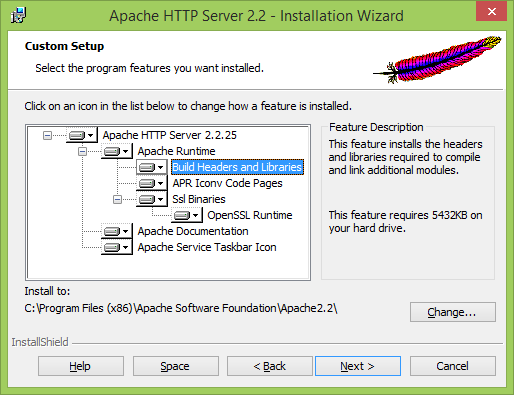
[**http://ftp.unicamp.br/pub/apache//httpd/binaries/win32/httpd-2.2.25-win32-x86-openssl-0.9.8y.msi**](http://ftp.unicamp.br/pub/apache//httpd/binaries/win32/httpd-2.2.25-win32-x86-openssl-0.9.8y.msi)

**The screen below shows how to install Apache:**





****



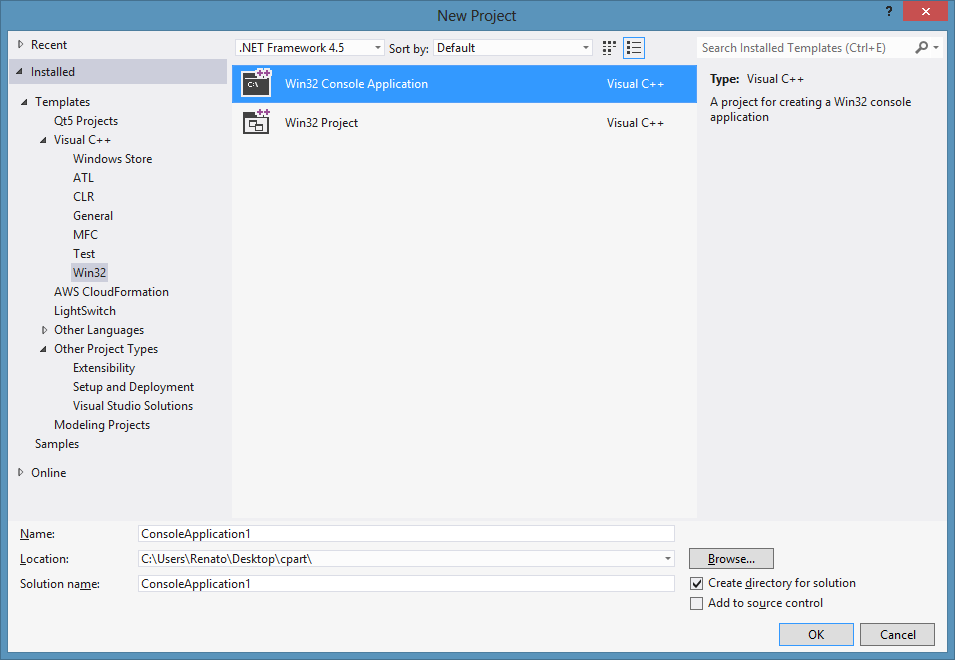
**Important Note: In this article the focus platform are Windows, but if you are UNIX/LINUX user you can adapt this it to your platform. “Boost.Application” is Multiplatform, like any Boost library.**

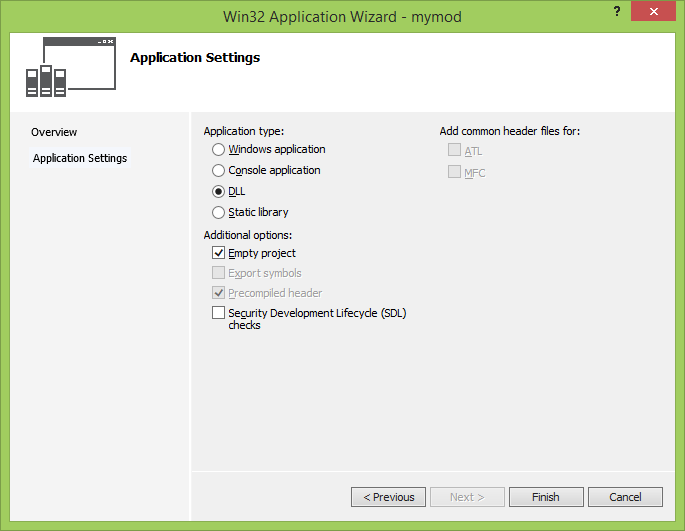
**2. Setup a Visual Studio Project**

The Visual Studio 2012 will be used to build the project, and all provided samples are for this version only. All code and projects are provided; check the download link on top of page.

**2.1 Create Visual Studio Project skeletons to Apache Module.**

Open Visual Studio, and create an Win32 Project. On ‘Application Settings’ select ‘DLL and ‘Empty Project’’.

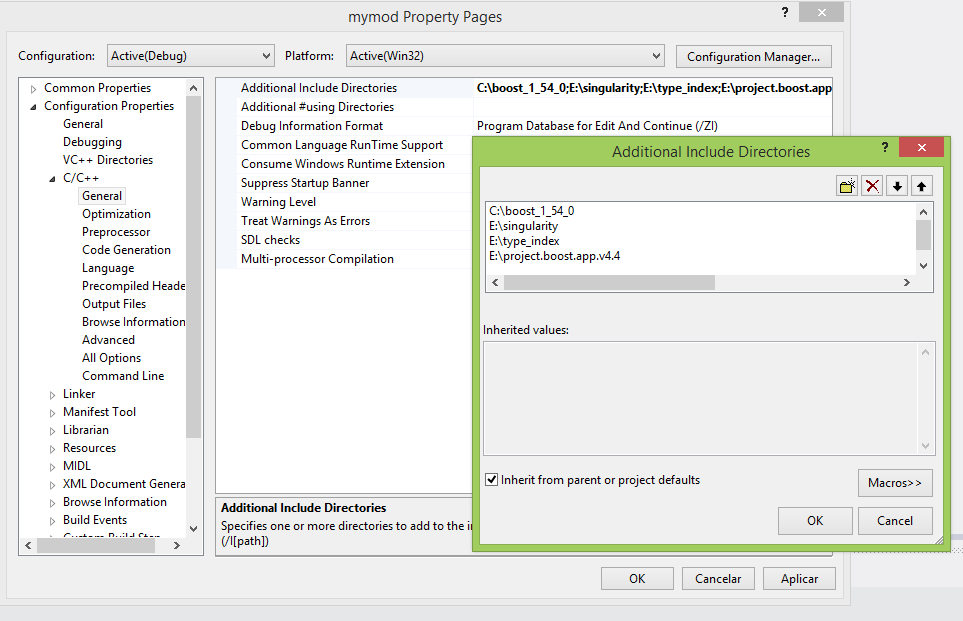




**2.2 On project, open properties window and add needed include directories to “C/C++ -> General -> Additional Include Directories”**

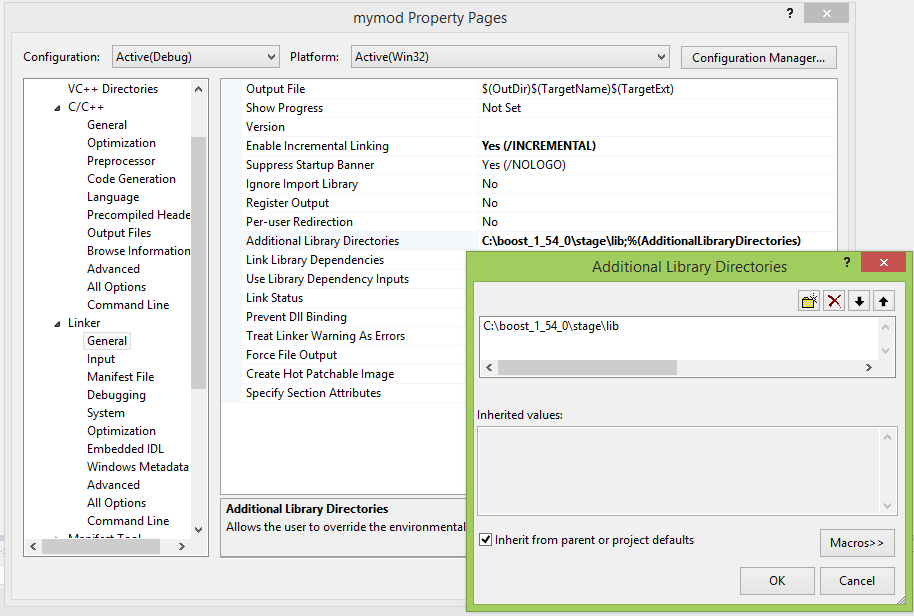
The needed directories are:

|  |  |
| --- | --- |
| **Name** | **Directories (example)** |
|  | |
| Your Main Boost | **C:\boost\_1\_54\_0** |
| Your Boost.Application | **E:\project.boost.app.v4.4** |
| Your Boost.TypeIndex | **E:\type\_index** |
| Your Boost.Singularity | **E:\singularity** |
| Your Apache API/APR | **C:\Program Files (x86)\Apache Software Foundation\Apache2.2\include** |
| *Note that your directories may have different names and locations.* | |

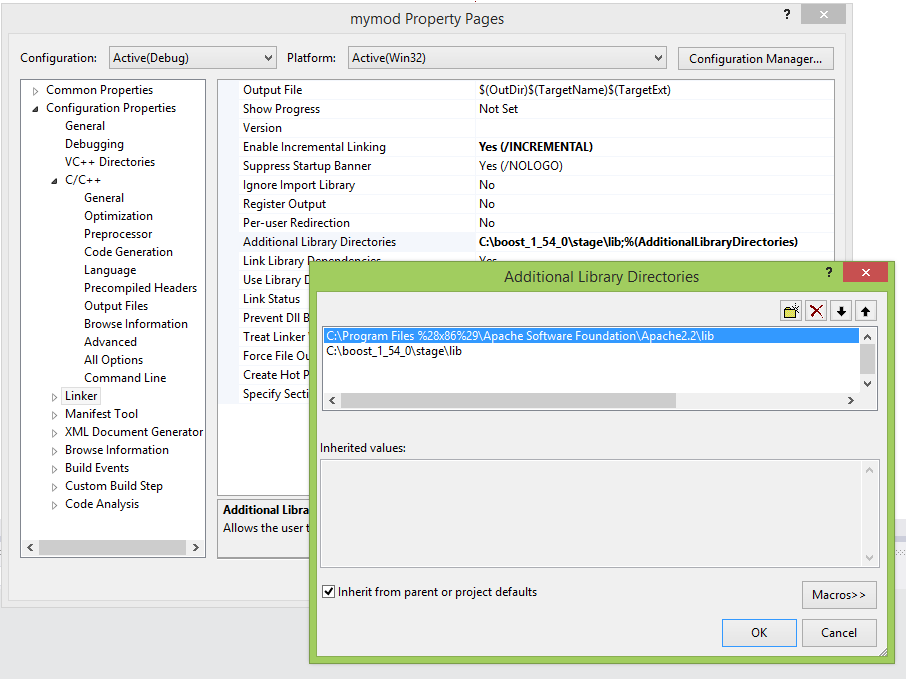


**2.3 On project, open properties window again, and add boost libs directories to “Linker -> General -> Additional Library Directories”**

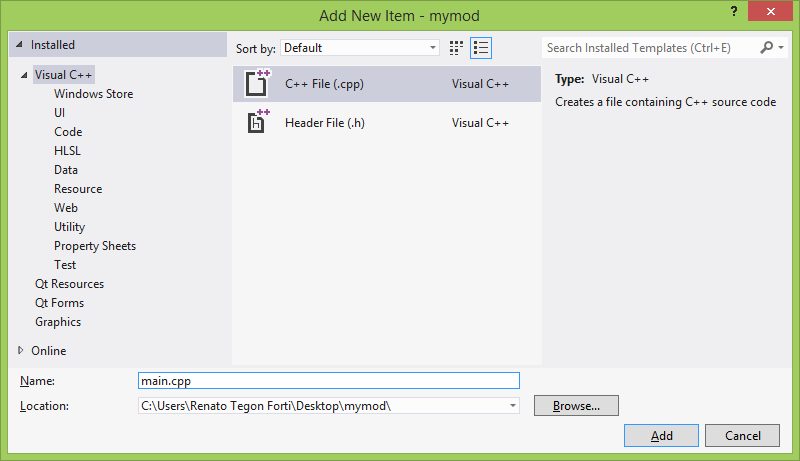
Note that when you build boost, the common place to libs are: e.g.: C:\boost\_1\_54\_0\stage\lib

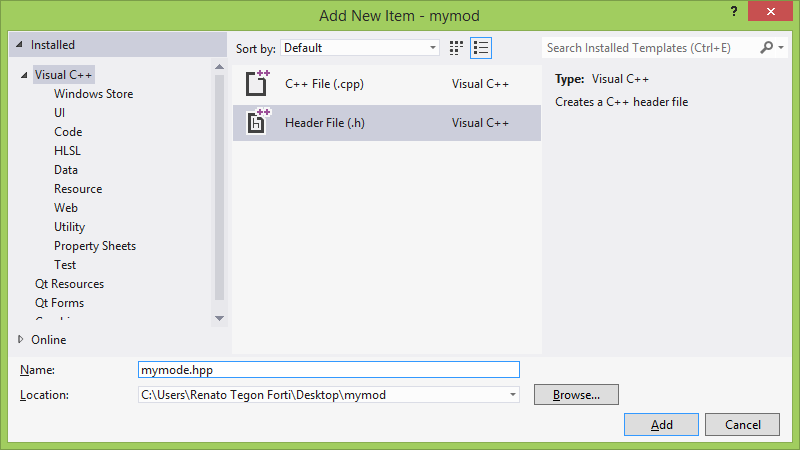


**2.4 We need do something to Apache “Linker -> General -> Additional Library Directories”**



**2.5 On project, add a new source file called “main.cpp”, and a header called “mymode.hpp”**





**3. Boost.Application -> application mode**

As previously discussed, by default (at this first version) Boost.Application provides two ready to use application modes: common and server application modes. In general way, user will use one of these modes (types) to build a server or a terminal application. If provided modes don’t accomplish what user desire, the user need extend the library mode.

The ‘application mode’ extension is the most advanced feature of Boost.Application, and a new mode in general way, needs work with some 3rdpart API, e.g. on Windows, the ‘server’ mode encapsulate and abstract to final user the Windows Service API.

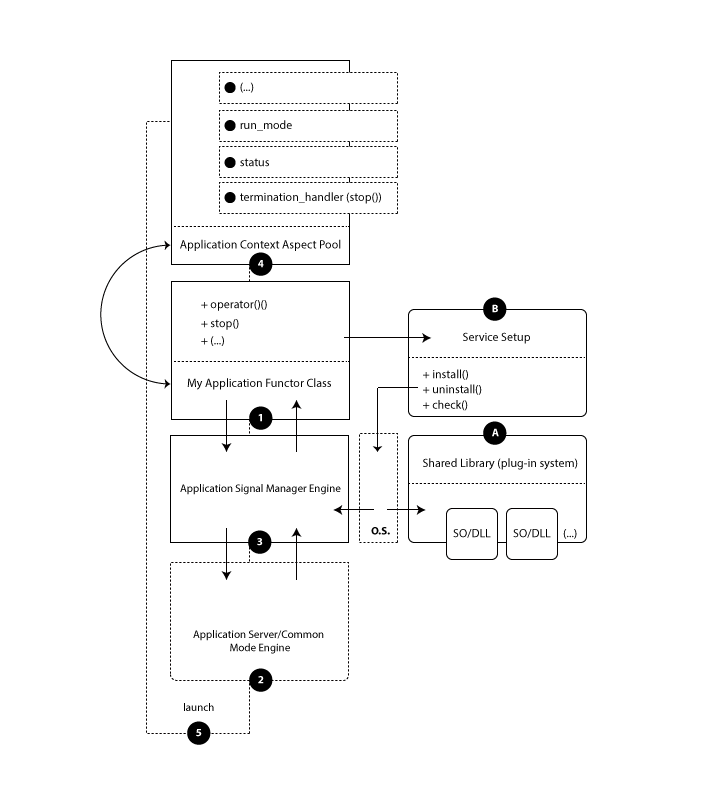
In our case, we will work (encapsulate on the new mode) the Apache API we will use Apache Portable Runtime (APR) inside our mode.

In the diagram below we have a schematic diagram of the components of “Boost.Application”, is highlighted in red which we modify/add.

We will don’t use “Signal Manager”, “Service Setup” or “Shared Library” in our Apache Mode.

We will create a new mode: apache2\_httpd\_mode.

We will create a new aspect to our new mode.



For complete reference, refer to: <http://www.dokfile.com/appbeta4/docs/libs/application/doc/html/boost_application/design_overview.html>

**4. Starting our application mode**

On ‘main.cpp’ we will have our client code, and on “mymode.hpp” we will have our mode implementation.

At this point is good you take a look at the rodmap to learn how an simple application is made:

<http://www.dokfile.com/appbeta4/docs/libs/application/doc/html/boost_application/roadmap.html>

We will create new mode to be used in step “H”.

**4.1 Application Mode Introduction**

Basically an “Application Module” is composed by one ‘mode class’, which need have a defined structure (constructor, and methods) and a collection of ‘aspects’.

An ‘aspect’ can be any class, but “Boost.Application”, provides some base class that allows we create a special type of ‘aspects’, e.g. a callback (handler) aspect.

To know more, refer to <http://www.dokfile.com/appbeta4/docs/libs/application/doc/html/boost_application/customization.html#boost_application.customization.customize_handlers>

In our case we will have one aspect to handle ‘apache log’, one to “web application name”, one for “content type” and the last will be our handler that will be called when a GET request arrives.

**5. Our aspects (Number 4 on diagram)**

Here I will present implementation of all aspects to out mode.

**5.1 Log Aspect**

This aspect will provide log functionality to final user. Take a look on:

C:\Program Files (x86)\Apache Software Foundation\Apache2.2\logs

Inside this folder we have many logs. Our “log aspect” will provide to user a way to use this log!

|  |
| --- |
| httpd.exe: Could not reliably determine the server's fully qualified domain name, using 192.168.0.9 for ServerName  [Wed Dec 11 09:38:05 2013] [notice] Apache/2.2.25 (Win32) configured -- resuming normal operations  [Wed Dec 11 09:38:05 2013] [notice] Server built: Jul 10 2013 01:52:12  [Wed Dec 11 09:38:05 2013] [notice] Parent: Created child process 6836  [Wed Dec 11 09:38:12 2013] [warn] Page requested! Boost.Application! |
| **Apache log sample** |

|  |
| --- |
| class apache\_log  {  friend class apache2\_httpd\_mod;  public:  apache\_log(request\_rec \*r)  : r\_(r) { }  void error(const std::string& msg)  {  ap\_log\_error(APLOG\_MARK, APLOG\_NOERRNO | APLOG\_ERR, 0, r\_->server, msg.c\_str());  }  void information(const std::string& msg)  {  ap\_log\_error(APLOG\_MARK, APLOG\_NOERRNO | APLOG\_INFO, 0, r\_->server, msg.c\_str());  }  void warning(const std::string& msg)  {  ap\_log\_error(APLOG\_MARK, APLOG\_NOERRNO | APLOG\_WARNING, 0, r\_->server, msg.c\_str());  }  private:  request\_rec \*r\_;  }; |
| **The apache\_log Aspect** |

**5.2 Web Application Name Aspect**

This aspect is used to identify our application on Apache, thus Apache can know if it handles our request or decline.

e.g.: <http://localhost:8080/boostapp>

The ‘[boostapp](http://localhost:8080/boostapp)’ is our web application name.

|  |
| --- |
| class web\_app\_name  {  friend class apache2\_httpd\_mod;  public:  web\_app\_name(const std::string& web\_app\_name)  : web\_app\_name\_ (web\_app\_name)  {}  private:  std::string web\_app\_name\_;  }; |
| **The web\_app\_name** **Aspect** |

**5.3 Content Type Aspect**

This aspect is used to identify our content that will be pushed to browser client

e.g.: "text/html;charset=ascii"

|  |
| --- |
| class content\_type  {  friend class apache2\_httpd\_mod;  public:  content\_type(const std::string& my\_content\_type)  : content\_type\_ (my\_content\_type)  {}  private:  std::string content\_type\_;  }; |
| **The content\_type Aspect** |

**5.4 GET VERB handler Aspect**

This aspect is the most important, it will allow to final user of our mode to tie a custom ‘content generator’ handler to HTTP GET VERB.

|  |
| --- |
| class http\_get\_verb\_handler : public handler<std::string>  {  public:  http\_get\_verb\_handler(const parameter\_callback& callback)  : handler<std::string>(callback) {}  http\_get\_verb\_handler(const singleton\_callback& callback)  : handler<std::string>(callback) {}  }; |
| **The http\_get\_verb\_handler** **Aspect** |

Note that we inheritance from ‘boost::application::handler<std::string>’.

To know more about provided handlers, refer to:

<http://www.dokfile.com/appbeta4/docs/libs/application/doc/html/boost_application/using_application_handlers.html>

**6. Implement our aspects (Number 2 on diagram)**

The ‘Application Mode’ is a class that will be initiated by ‘boost::application::launch’ function. On our mode design, each request that arrives will launch a new apache2\_httpd\_mod! Here the mode designer is free to do anything.

|  |
| --- |
| class apache2\_httpd\_mod  {  public:  static int mode()  {  static int id = new\_run\_mode<int>();  return id;  }  template <typename Application, typename RequestRec>  apache2\_httpd\_mod(Application& myapp, RequestRec &rr,  context &cxt, boost::system::error\_code& ec)  : error\_(OK)  {  }  template <typename Application, typename RequestRec>  apache2\_httpd\_mod(Application& myapp, RequestRec &rr,  boost::singularity<context> &cxt, boost::system::error\_code& ec)  : error\_(OK)  {  }  int run() { return error\_; }  protected:  // ...  }; |
| **The apache2\_httpd\_mod mode skeleton** |

Note that all ‘modes’ need respect these signatures.

The ‘mode’ method is used to identify a mode, and we have 2 constructor, one that support singularity, and other that will receives a ‘context’ as parameter, and ‘run’ method that in our case return one status code to client.

After that the designer of the module is free to add any other method that is necessary. Let’s do this now:

|  |
| --- |
| class apache2\_httpd\_mod  {  public:  static int mode()  {  static int id = new\_run\_mode<int>();  return id;  }  template <typename Application, typename RequestRec>  apache2\_httpd\_mod(Application& myapp, RequestRec &rr,  context &cxt, boost::system::error\_code& ec)  : error\_(OK)  {  handle\_request(myapp, rr, cxt);  }  template <typename Application, typename RequestRec>  apache2\_httpd\_mod(Application& myapp, RequestRec &rr,  boost::singularity<context> &cxt, boost::system::error\_code& ec)  : error\_(OK)  {  handle\_request(myapp, rr, cxt.get\_global());  }  int run() { return error\_; }  protected:  template <typename Application, typename RequestRec>  void handle\_request(Application& myapp, RequestRec &rr, context &cxt)  {  // default impl aspects  if(!cxt.find<run\_mode>())  {  cxt.insert<run\_mode>(  csbl::make\_shared<run\_mode>(mode()));  }  if(!cxt.find<status>())  {  cxt.insert<status>(  csbl::make\_shared<status>(status::running));  }  csbl::shared\_ptr<web\_app\_name> appname = cxt.find<web\_app\_name>();  if(!appname)  {  error\_ = DECLINED; return;  }  if (strcmp(rr.handler, appname->web\_app\_name\_.c\_str()))  {  error\_ = DECLINED; return;  }  // we allow only GET    // Add other http verbs  // ...  if(rr.method\_number != M\_GET)  {  error\_ = HTTP\_METHOD\_NOT\_ALLOWED; return;  }    // GET  csbl::shared\_ptr<http\_get\_verb\_handler> http\_get\_verb =  cxt.find<http\_get\_verb\_handler>();    if(http\_get\_verb)  {  // apache log  cxt.insert<apache\_log>(csbl::make\_shared<apache\_log>(&rr));  csbl::shared\_ptr<content\_type> contenttype =  cxt.find<content\_type>();  if(contenttype)  ap\_set\_content\_type(&rr, contenttype->content\_type\_.c\_str());  else  ap\_set\_content\_type(&rr, "text/html;charset=ascii");  // check if we have any callback to call  handler<std::string>::parameter\_callback\* parameter = 0;  if(http\_get\_verb->callback(parameter))  {  ap\_rputs((\*parameter)(cxt).c\_str(), &rr); return;  }  handler<std::string>::singleton\_callback\* singleton = 0;  if(http\_get\_verb->callback(singleton))  {  ap\_rputs((\*singleton)().c\_str(), &rr); return;  }  }  // we need set application\_state to stop  cxt.find<status>()->state(status::stoped);  // we cant find any handler, generate apache error  error\_ = HTTP\_INTERNAL\_SERVER\_ERROR;  }  private:  int error\_;  }; |
| **Full apache2\_httpd\_mod mode that’s use Apache APR** |

**6.1 Export all to Apache be aware of our mode.**

To do this we will implement a MACRO, that final user will need add on “.cpp” file.

|  |
| --- |
| #define **BOOST\_APPLICATION\_APACHE\_REGISTER\_TEST\_MY\_MODE**(h, m) \  extern "C" { \  void **boost\_application\_register\_hooks**(apr\_pool\_t \*p) \  { \  **ap\_hook\_handler**(h, NULL, NULL, APR\_HOOK\_MIDDLE); \  } \  \  module AP\_MODULE\_DECLARE\_DATA m = { \  STANDARD20\_MODULE\_STUFF, \  NULL, \  NULL, \  NULL, \  NULL, \  NULL, \  boost\_application\_register\_hooks \  }; } |
| **Export Macro** |

**6.2 Mode Conclusion**

Now we have our mode ready to use. Now we our role is not more as ‘Mode Designer’, now our role will be ‘Mode User’ (client/final user)

**7. Using the new ‘Mode’**

Here I will present how the filal/client user can use our new mode.

**7.1 Create an Application Functor Class**

On our ‘main.cpp’ we will create our functor class.

|  |
| --- |
| class my\_apache2\_httpd\_content\_generator\_mod  {  public:  int operator()(application::context& context)  {  return 0;  }  std::string get(application::context& context)  {  context.insert<content\_type>(  boost::make\_shared<content\_type>("text/html;charset=ascii"));  std::stringstream htm;  htm.str("");  htm << "<html>"  << " <head>"  << " <title>Boost.Application Test</title>"  << " </head>"  << ""  << " <body>"  << " <h1> Hello Boost.Application Version "  << application::library\_version\_string() << "!"  << " </h1>"  << " <br/>"  << " Apache HTTPd Mod."  << " <br/>"  << " </body>"  << "</html>"  ;  boost::shared\_ptr<apache\_log> apachelog = context.find<apache\_log>();  if(apachelog)  {  // log something on apache log file  apachelog->warning("Page requested!");  }  return htm.str();  }  }; |
| **Application Functor Class** |

In ‘get’ method we will generate our content, in this case the content will be a simple html page. We will use our ‘log aspect’ tool!

**7.2 Implement “main” function**

This function will be called by apache on a request, and we use it to setup our application mode ate run-time.

|  |
| --- |
| // an application will be launched to handle each request that arrives.  extern "C" int myhandle(request\_rec \*r)  {  my\_apache2\_httpd\_content\_generator\_mod app;  application::context app\_context;  app\_context.insert<web\_app\_name>(  boost::make\_shared<web\_app\_name>("boostapp"));  handler<std::string>::parameter\_callback my\_http\_get\_verb  = boost::bind<std::string>(  &my\_apache2\_httpd\_content\_generator\_mod::get, &app, \_1);  app\_context.insert<http\_get\_verb\_handler>(  boost::make\_shared<  http\_get\_verb\_handler>(my\_http\_get\_verb));  return application::launch<apache2\_httpd\_mod>(app, \*r, app\_context);  } |
| **Our main function (myhandler)** |

Apache Web Server calls this function passing an ‘request\_rec’, that we will pass to our mode.

Then we create our application functor class, our context, that will hold all of our aspects. To know more about context, refer to: <http://www.dokfile.com/appbeta4/docs/libs/application/doc/html/boost_application/context.html>

After that we use the mode aspects that we just created. Note that we bind our ‘get’ method to our ‘http\_get\_verb\_handler’ aspect.

The last step is tie all together and run the ‘content generator’, we do this using “application::launch<apache2\_httpd\_mod>” function that’s receives our mode as template parameter.

**7.3 Register our ‘main (myhandler)’ function**

The final step is register our main function, using our MAACRO.

|  |
| --- |
| // register request function and mod on apache server  BOOST\_APPLICATION\_APACHE\_REGISTER\_TEST\_MY\_MODE(myhandle, my\_boost\_app\_mod) |
| **Register myhandler** |

**8. Configure httpd.cong**

Now is time to configure our ‘httpd.cong’ file.

This file is located on ‘/conf’, in my case:

C:\Program Files (x86)\Apache Software Foundation\Apache2.2\conf

**8.1 Add our module on httpd.cong**

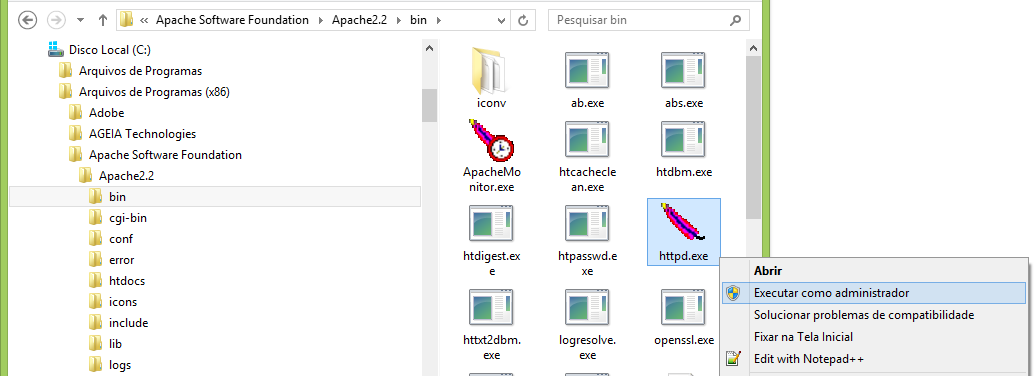
Open file and add this:

|  |
| --- |
| // file continues ...  #LoadModule version\_module modules/mod\_version.so  #LoadModule vhost\_alias\_module modules/mod\_vhost\_alias.so  LoadModule my\_boost\_app\_mod "C:\Users\Renato Tegon Forti\Desktop\mymod\Debug\mymod.dll"  <Location /boostapp>  SetHandler boostapp  </Location>    // file continues ... |
| **Httpd Configurarion** |

Change "*C:\Users\Renato Tegon Forti\Desktop\mymod\Debug\mymod.dll*" to your path.

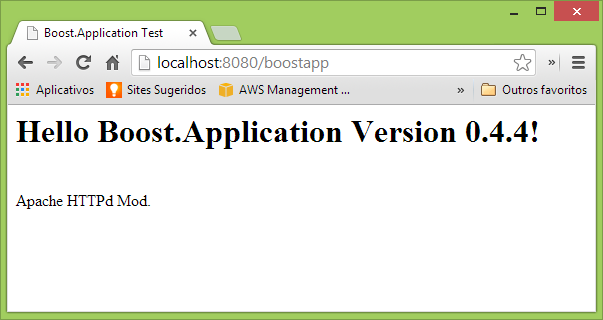
**9. Test**

Go to your Apache folder, and start Apache as Admin.



Run As Admin

**9.1 Open your Browser and request page, and you need see:**



**10. Conclusion**

Here I show how user can extend the Library, many others ready to use features are available, one that I like a lot is plug-in system. Take a look on documentation.

And, please send your comments, if you liked or not, suggestions and bugs .