Title: Build a server application using Application Library

Author: Renato Tegon Forti

Email: re.tf@acm.org

Language: C++

Platform: Windows, POSIX

Technology: Service, Daemon, Plug-in System

Level: Intermediate

Description: This article presents the new version of proposed library ‘Application’ to boost.org, and show how you can build a long time (e.g. Server Application) running applications  
Section Platforms, Frameworks & Libraries

SubSection Cross Platform

License: CPOL

* [Download demo project - 18,4 KB](source.zip)

**Introduction**

This article presents the new version of proposed library ‘Application’ to boost.org (0.4.6), and show how you can build a long time running applications (e.g. Server Application).

**Basic Setup**

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

The build process of boost is not very easy, but a good documentation is provided, refer to [this link](http://www.boost.org/doc/libs/1_54_0/more/getting_started/windows.html).

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

2) The second step is to download:

Boost.Application (<https://github.com/retf/Boost.Application/zipball/master>)

Boost.TypeIndex (<https://github.com/apolukhin/type_index/zipball/master>)

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

**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 the real interest and get comments for refinement. The intention is to submit the library for formal review, if the community thinks that it is interesting.

**Boost.Application Resources**

You can download the library from GitHub [here](https://github.com/retf/Boost.Application).

An online documentation (under construction) is available [here](http://www.dokfile.com/appbeta4/docs/libs/application/doc/html/index.html).

**Motivation**

Sometimes we need run application for long time (e.g. Server Application), we need a mechanism to extend this application on runtime, and we need some facilities to access paths, ensure single instance instantiation on system, manage and catch signals and so on.

This work is recurrent each time that we need build and deploy an application for particular system, and the way to do this, change a lot on each of these systems.

For instance on Windows side we have ‘services’ and on Unix (POSIX) side we have daemons that are used to build long-running executable applications (e.g. server) and these two APIs have no similarity.

Thus, in this scenarios, is so much difficult to developer have your application running on Windows or on POSIX as server without a lot of work. The work is harder if we need run same application in both systems.

Other problem raise when user want provide a way to extend your application using a plug-in mechanism. Like Service/Daemon, the shared modules (DSO) manipulation changes a lot on Windows and POSIX.

Obtain simple thing like paths, arguments, manipulate signal, can be annoying, since it also don’t has a common interface to do this on both systems.

The 'Application' library aims to make significantly easier for the developer has your application runs in cross-platform (POSIX/Windows) environment.

**Application**

The ‘application’ library provides a cross-platform C++ way to you do a service/daemon, provides a mechanism to help with a plug-in and provide a way to do annoying things using ‘aspects’ and other things.

**Our Sample**

Sometimes is a common task executes or check something at specified intervals of time (e.g. unix cron). This can be knows as time-based job (task) scheduler.

We will use ‘application’ to build a server application that process a timer event as tasks on plugi-ns (our jobs). Thus our application will have a timer that will expire and trigger tasks that are on a plug-in

At the end, if you are on Windows you will have a service that load a ‘.dll’ on runtime, and if you are on POSIX you will have a Daemon that loads a ‘.so’ on runtime and trigger a task of it.

When our timer (asio) expires we will scan a folder (filesystem), and look for plug-ins (application) on it, and then execute the job inside of the plugi-ns.

**Sample Business Logic and Used Boost Libraries**

Or sample will contact a web service (api.openweathermap.org) to get weather information.

We are build a server for windows, then we will use Boot.Program\_Options to handle service installation, see details below on ‘Project Sample and Development environment’.

The Web API returns a ‘json’ response, then we will use Boost.Property\_Tree and Boost. Foreach to handle this response.

We will compare the “humidity” value that Web API returns and log it on text file if it is less than 80%.

The plugin uses a ‘sync’ Boost.Asio sample client to connect on web API. User can implement ‘async‘ plugin as exercise, if desired.

We will need to use Boost.Filesystem, Boost.Bind , Boost.Function, Boost.System to do some things.

So, in our sample we will use the following Boost Libraries:

* Boost.Asio
* Boost.Property\_Tree
* Boost.Foreach
* Boost.Program\_Options
* Boost.Filesystem
* Boost.Bind
* Boost.Function
* Boost.System

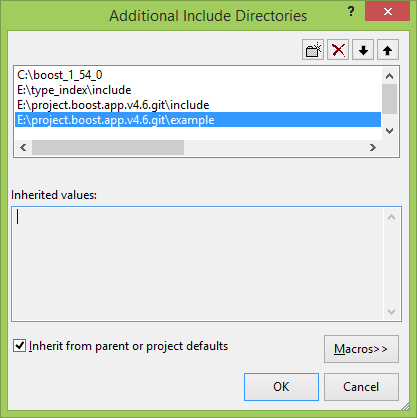
**Project Sample and Development environment**

I will use Windows and Visual Studio as a development platform due to its popularity. But as I said earlier, the code compiles on POSIX generating a daemon without modifications.

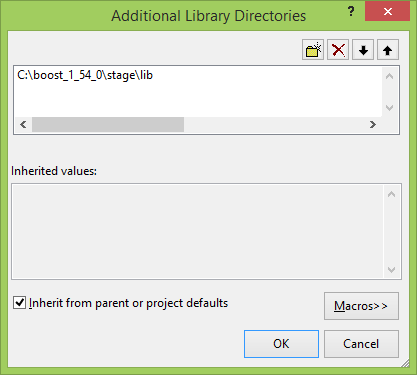
In download link you will find a Visual Studio 2013 project. If you use POSIX, you will need create your own make/jam file to compile the project. If you have any difficulties in this process, please enter in contact then I can help you.

Note that you need change the paths of project to reflect your environment; in my case I have this:

*C/C++ | General*



*Linker | General*



**1.Basic**

Boost.Application requires a ‘Functor’ class that will be used to model the concept of an “application”, thus, before your ‘main’ function you need add your ‘functor’ class, like this:

// our server class

class time\_based\_job : boost::noncopyable

{

public:

int operator()()

{

return 0;

}

};

// main

int main(int argc, char \*argv[])

{

time\_based\_job app;

return 0;

}

**2. Context and aspects**

Boost.Application has a concept of ‘aspects’, to allow easy extension, customization and make things more flexible.

Any c++ class can be an ‘aspect’, and you will use an ‘application context’ to have access to applications ‘aspects’, thus this takes us to another concept: the ‘application context’.

Note that Boost.Application provides a lot of ready-to-use aspects, and we will use some of these ‘aspects’ in our sample.

The ‘application context’ is like a container that you can add your ‘application aspects’. The ‘application context’ can be accessed using 2 ways:

1. As function parameter
2. As a global object (singleton)

User is free to choose the best option to your application design. Here we will use ‘global\_context’, so let’s we add it to our source file.

// we will use global\_context for our sample, we can access global\_context in

// any place of our application

inline global\_context\_ptr this\_application\_cxt() {

return global\_context::get();

}

// our server class

class time\_based\_job : boost::noncopyable

{

public:

int operator()()

{

return 0;

}

};

// main

int main(int argc, char \*argv[])

{

time\_based\_job app;

// create a global context for application (our aspect\_map),

// from this point it is available to be used on any part of our application.

global\_context::create();

boost::shared\_ptr<void> global\_context\_destroy(nullptr, [&](void\*) {

// this code will be executed at the scope exit

// destroy our global\_context and ends our application

global\_context::destroy();

});

// 1

return 0;

}

The ‘global\_context::create’ will create our context, and we use a trick to delete our context when main ends (exit of scope). To do this we use ‘shared\_ptr’.

**3. Ready to use aspects**

Now we will add 2 ‘ready to use’ aspects that we will need to our ‘application context’. The first aspect will help us to handle ‘args’ and another is to help us with ‘paths’.

Add after: ‘// 1’ ;

this\_application\_cxt()->insert<args>(  
 boost::make\_shared<args>(argc, argv));

this\_application\_cxt()->insert<path>(

boost::make\_shared<path\_default\_behaviour>(argc, argv));

// 2

**4. Launch application**

Boost.Application supports basically 2 flavors of applications that can be:

* **Common Application**

This kind of application is a usual Interactive Terminal application.

* **Server Application**

This kind of application generates a Service (Windows), or a background process/Daemon (Unix).

Note that others kinds of application can be added be user. Refer to this article that’s extend Boost.Application to support ‘Apache Http Module’ application mode.

<http://www.codeproject.com/Articles/695937/Creating-a-New-Application-Mode>

Now we will launch an ‘Common Application’. Add after: ‘// 2’ ;

boost::system::error\_code ec;

return launch<common>(app, this\_application\_cxt(), ec);

Note that Boost.Application can handle error setting ‘boost::system::error\_code’ or use 'exception version' that’s throws an exception of type boost::system::system\_error.

At this point we have a complete skeleton of application.

**5. Buid an aspect**

Our application will process a timer event as tasks, thus now we will model the behavior on one aspect that we will add on our application context.

// our timer aspect

template< typename Handler >

class timer\_job : boost::noncopyable

{

typedef handler<bool>::global\_context\_callback job\_callback;

public:

timer\_job(Handler h, unsigned int timeout)

: jtimer\_(io\_service\_), timeout\_(timeout), job\_(h)

{

trigger();

boost::thread(

boost::bind(&boost::asio::io\_service::run,

boost::ref(io\_service\_))).detach();

}

void stop()

{

io\_service\_.stop();

}

protected:

bool job()

{

if(job\_()) // retur true from job to stop

return true;

trigger(); // trigger next timeout event

return false;

}

void trigger()

{

jtimer\_.expires\_from\_now(boost::posix\_time::seconds(timeout\_));

jtimer\_.async\_wait(

[this](const boost::system::error\_code& ec) -> void {

if(!ec) {

if(this->job()) {

io\_service\_.stop(); // error

}

}

else

io\_service\_.stop(); // 'ec' is true

}

);

}

private:

unsigned int timeout\_;

job\_callback job\_;

boost::asio::io\_service io\_service\_;

boost::asio::deadline\_timer jtimer\_;

};

typedef timer\_job< handler<bool>::global\_context\_callback > timer\_callback;

In this class (aspect), we use ‘asio::deadline\_timer’ to schedule and call a callback method that will do our required action.

**6. Add our handmade aspect to context**

To do this, we need add our callback handler to our functor class, like this:

// our server class

class time\_based\_job : boost::noncopyable

{

public:

int operator()()

{

return 0;

}

bool doit()

{

return false;

}

};

And add the below code after ‘path’ aspect on main.

this\_application\_cxt()->insert<

timer\_callback >(boost::make\_shared<

timer\_callback >(

handler<bool>::make\_global\_callback(app, &time\_based\_job::doit), 5));

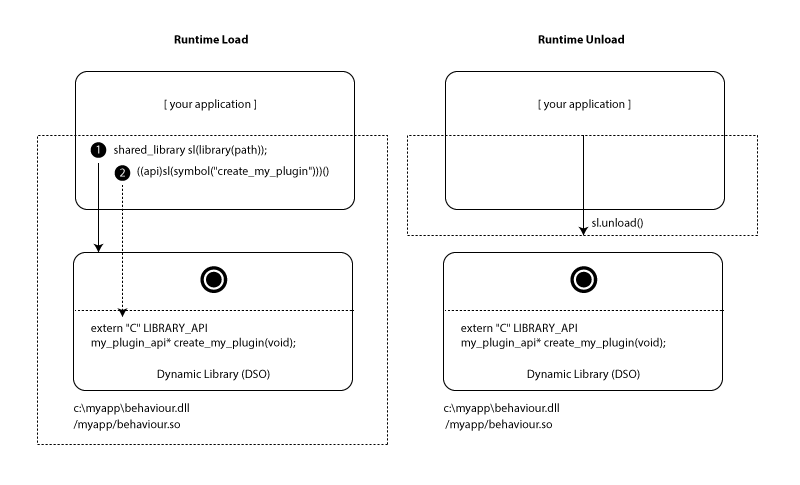
Boost Application has a class that we can use to make a callback (handler<bool>::make\_global\_callback), here we use it to bind our timer event to ‘doit’ on our functor class.

Thus, when time event expire, it will call “doit” on functor class.

Note that the time event will expire on 5 seconds.

**7. Make our plug-in**

A plugin is a shared library that exports some functions (C function) that gives access to our behavior, see :



The first thing is create our plugin project, refer to sample file to details. Basically is a “**Win32 Console Application (DLL)” Project type.**

Now we will create a file called “*job\_plugin\_api.hpp*”, this represents a plugin API interface. In this case we will build a plugin that will “call a web service” using a sync Boost.Asio interface.

class my\_plugin\_api

{

public:

virtual ~my\_plugin\_api(){};

virtual float version() = 0;

virtual bool do\_the\_job(

const std::string& query, const std::string& resource,

std::string& result, std::string& err) = 0;

};

The implementation goes on “job\_plugin\_library.cpp”, here the code is based on Boost.Asio sample:

<http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio/example/cpp03/http/client/sync_client.cpp>

I will comment only the plugin access interface, which is that:

my\_plugin\_api\* create\_my\_plugin(void)

{

my\_plugin\_api \*myplugin = new my\_job\_plugin();

return myplugin;

}

void delete\_my\_plugin(my\_plugin\_api\* myplugin)

{

delete myplugin;

}

If we respect this interface, we can have others plug-ins, e.g.: here we use Boost.Asio “sync” sample, but we can make another plugin that use Boost.Asio “async” sample:

<http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio/example/cpp03/http/client/async_client.cpp>

**8. Load our plug-in using Boost.Application "Shared Library Class"**

Now we will modify our “doit” method like this:

bool doit()

{

boost::shared\_ptr<path> pt

= this\_application\_cxt()->find<path>();

boost::filesystem::path scan\_dir(pt->executable\_path());

std::vector<boost::filesystem::path> v; // store paths

std::copy(boost::filesystem::directory\_iterator(scan\_dir),

boost::filesystem::directory\_iterator(), back\_inserter(v));

int count\_synced\_files = 0;

for (std::vector<boost::filesystem::path>::const\_iterator it

(v.begin()); it != v.end(); ++it)

{

boost::filesystem::path p(\*it);

std::string base = boost::to\_lower\_copy(p.extension().string());

if( base == shared\_library::suffix() ) // dll or so

{

std::cout << "our plugin: " << p.string() << std::endl;

my\_plugin\_api\* plugin = NULL;

shared\_library sl(library(p.string()));

if(sl.search\_symbol(symbol("create\_my\_plugin")))

{

plugin = ((pluginapi\_create)sl(symbol("create\_my\_plugin")))();

}

if(plugin != NULL)

{

boost::shared\_ptr<void> delete\_plugin(nullptr, [&](void\*) {

// this code will be executed at the scope exit

((pluginapi\_delete) sl(symbol("delete\_my\_plugin")))(plugin);

});

std::cout << "Plugin Version: " << plugin->version() << std::endl;

std::string query = "api.openweathermap.org", resource = "/data/2.5/find?q=Americana,br&mode=json", aswer, err;

if(plugin->do\_the\_job(query, resource, aswer, err))

std::cout << "Error from plugin: " << err << std::endl;

else

{

/\*

{

"message": "accurate",

"cod": "200",

"count": 1,

"list": [

{

"id": 3472343,

"name": "Americana",

"coord": {

"lon": -47.33139,

"lat": -22.73917

},

"main": {

"temp": 302.15,

"pressure": 1018,

"humidity": 48,

"temp\_min": 302.15,

"temp\_max": 302.15

},

"dt": 1396634400,

"wind": {

"speed": 6.7,

"deg": 130

},

"sys": {

"country": "BR"

},

"clouds": {

"all": 40

},

"weather": [

{

"id": 802,

"main": "Clouds",

"description": "scattered clouds",

"icon": "03d"

}

]

}

]

}

\*/

pt::ptree json\_reponse; std::istringstream is(aswer);

read\_json(is, json\_reponse);

BOOST\_FOREACH(boost::property\_tree::ptree::value\_type& v, json\_reponse.get\_child("list"))

{

if(v.second.get<int>("main.humidity") < 80)

{

// record

weather\_log\_ << v.second.get<std::string>("dt") << ":" << v.second.get<std::string>("main.humidity") << std::endl;

}

}

}

}

}

}

return false;

}

Well, here we have a lot of code.

Here we use “boost::filesystem” to scan folder to get all shared modules. To do that we use the aspect:

boost::shared\_ptr<path> pt

= this\_application\_cxt()->find<path>();

boost::filesystem::path scan\_dir(pt->executable\_path());

That’s gives us the path of our executable module.

The “shared\_library::suffix()” return a string ‘dll’ on windows and ‘so’ on unix.

Then we use “application::shared\_library” to load a plugin, check if specific simbol is provided, and call it.

shared\_library sl(library(p.string()));

if(sl.search\_symbol(symbol("create\_my\_plugin")))

{

plugin = ((pluginapi\_create)sl(symbol("create\_my\_plugin")))();

}

// ...

The plugin response is a json string that we ‘read’ using Boost.Property\_Tree:

pt::ptree json\_reponse; std::istringstream is(aswer);

read\_json(is, json\_reponse);

BOOST\_FOREACH(boost::property\_tree::ptree::value\_type& v, json\_reponse.get\_child("list"))

{

if(v.second.get<int>("main.humidity") < 80)

{

// record

weather\_log\_ << v.second.get<std::string>("dt") << ":" << v.second.get<std::string>("main.humidity") << std::endl;

}

}

**9. Instantiate application as Service or Daemon**

To do this we need change a little our main function. We will provide a way to user choose if he want start the applications as service or as common application.

std::vector<std::string> arg\_list

= this\_application\_cxt()->find<args>()->arg\_vector();

int ret = 0;

if(std::find(arg\_list.begin(), arg\_list.end(), "--common") != arg\_list.end())

// launch our application as a common app

ret = launch<common>(app, this\_application\_cxt(), ec);

else

{

// launch our application as a server app (service / daemon)

ret = launch<server>(app, this\_application\_cxt(), ec);

}

**10. Installation**

In this version of lib, we don’t provide official support for "service or daemon" installation. This is because we need maintain symmetry between functionality provided on all platforms that the lib support, and on POSIX side, the installation process varies a lot. Thus we provide installation functionality for major system as sample on examples folder. In future we have plan to bring these functionality to core lib.

Note: For POSIX you need use some shell script to control a daemon, refer to:

\example\setup\posix\ubuntu

That has some samples from Ubuntu.

On Window we will use this sample code:

\example\setup\windows\setup\service\_setup.hpp

Now is time to code our setup method for Windows.

// my setup code for windows service

bool setup()

{

boost::shared\_ptr<args> myargs

= this\_application\_cxt()->find<args>();

boost::shared\_ptr<path> mypath

= this\_application\_cxt()->find<path>();

// provide setup for windows service

#if defined(BOOST\_WINDOWS\_API)

// get our executable path name

boost::filesystem::path executable\_path\_name = mypath->executable\_path\_name();

std::string exename = mypath->executable\_name().stem().string();

// define our simple installation schema options

po::options\_description install("service options");

install.add\_options()

("help", "produce a help message")

(",i", "install service")

(",u", "unistall service")

("user", po::value<std::string>()->default\_value(""),

"user logon (optional, installation only)")

("pass", po::value<std::string>()->default\_value(""),

"user password (optional, installation only)")

("name", po::value<std::string>()->default\_value(exename),

"service name")

("display", po::value<std::string>()->default\_value(""),

"service display name (optional, installation only)")

("description", po::value<std::string>()->default\_value(""),

"service description (optional, installation only)")

;

po::parsed\_options parsed =

po::command\_line\_parser(myargs->argc(), myargs->argv()

).options(install).allow\_unregistered().run();

po::variables\_map vm;

po::store(parsed, vm);

boost::system::error\_code ec;

if (vm.count("help"))

{

std::cout << install << std::cout;

return true;

}

if (vm.count("-i"))

{

example::install\_windows\_service(

setup\_arg(vm["name"].as<std::string>()),

setup\_arg(vm["display"].as<std::string>()),

setup\_arg(vm["description"].as<std::string>()),

setup\_arg(executable\_path\_name),

setup\_arg(vm["user"].as<std::string>()),

setup\_arg(vm["pass"].as<std::string>())).install(ec);

std::cout << ec.message() << std::endl;

return true;

}

if (vm.count("-u"))

{

example::uninstall\_windows\_service(

setup\_arg(vm["name"].as<std::string>()),

setup\_arg(executable\_path\_name)).uninstall(ec);

std::cout << ec.message() << std::endl;

return true;

}

#endif

return false;

}

Here we use Boost.Program\_Options to know user desired options.

Now we have a command line interface on our server to install and uninstall if the service is installed. When the user requests installation, e.g.: ‘time\_based\_plugin\_job.exe -i', the installation requisition is identified by Boost.Program\_options and the code on ‘-i if' is executed, installing the service.

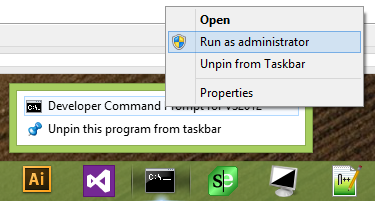
Note that you can provide “user login” to install the service for a specific Windows User.

The command line options are:

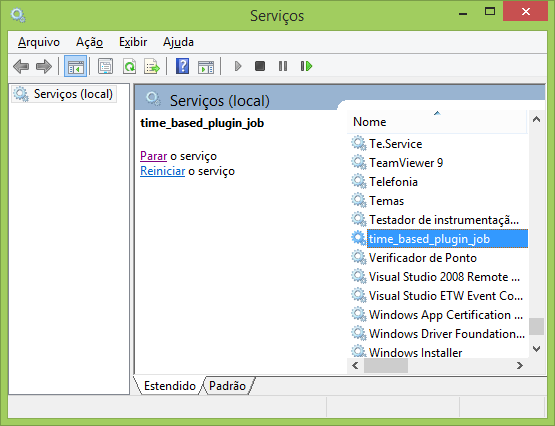
C:\Users\Renato Tegon Forti\Desktop\time\_based\_plugin\_job\Debug>time\_based\_plugin\_job.exe –i  
 [I] Setup changed the current configuration.

C:\Users\Renato Tegon Forti\Desktop\time\_based\_plugin\_job\Debug>time\_based\_plugin\_job.exe –u  
[I] Setup changed the current configuration.

To install service you need have permission to do that. An easy way is execute installation as Admin like:



After that you will see on SCM our application as Service.



**11. Other handlers**

Boost.Application allow you add some handlers, e.g. code some action when user click on stop link (windows) or send CTRL-C signal on POSIX.

We will add code to stop. First we need add ‘stop’ method to our functor class, like this:

bool stop()

{

if (weather\_log\_.is\_open())

{

weather\_log\_.close();

}

this\_application\_cxt()->find<timer\_callback>()->stop();

return true;

}

Here we close our log file and stop our timer engine.

Now we need add handler to ‘stop’ on main in your application context, like this:

this\_application\_cxt()->insert<termination\_handler>(

boost::make\_shared<termination\_handler\_default\_behaviour>(

handler<bool>::make\_global\_callback(app, &time\_based\_job::stop)));

Now when user clicks stop or send ctrl-c signal to application the method ‘stop’ will be called!

Note if you return false from ‘stop’ the application engine will ignore action, and the application will continue.

You have other handles available, like pause, or limit single instance. Check documentation to more detail.

**Conclusion**

Boost.Application can save time when the task is to build a cross platform server; otherwise the developer must work directly with a complex API provided for that. Furthermore Boost.Application provides an extension mechanism for an application (based on plugins) simple and efficient, and many other ready-to-use features.

**License**

This article, along with any associated source code and files, is licensed under [The Code Project Open License (CPOL)](http://www.codeproject.com/info/cpol10.aspx)

**Feedback**

If you are a Boost user, and use Boost Mailing Lists, please provide your feedback about the library, directly on list. (Specify if you think the library should be accepted as part of boost.org.)

<http://www.boost.org/community/groups.html>

If you are a CodeProject user, please provide your feedback about the library directly on this page.

**Use**

If you intend to use ‘Application’ on your application, please send-me your name and project. I am looking for create a list of users of ‘Application’

[re.tf@acm.or](mailto:re.tf@acm.or) (Renato Tegon Forti)