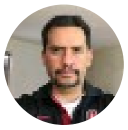

Iván Mejía Follow
Full stack Software Engineer, C++, C#, Java, Swift and Objective-C developer, Backend development as well as iOS apps, Progressive metal fan, Swim enthusiast.
Jan 4, 2017 · 7 min read

Modern C++ micro-service implementation + REST API

Part II of this article

Nowdays there is a lot of hype about node.js in the micro-service field due to its productivity and the great community, and indeed node.js is really a powerful platform for cloud computing development, but don't forget that all its productivity relays on various layers of abstraction which come with a penalty in performance and a huge stack of components to make it work. But I do not want to start here a polemic about which one is best, this article is about using modern C++ to implement a microservice where the idea is using the modern syntax to reach the level of productivity that node.js provides but with the performance benefits of C++, and if you are like me that loves C++ and its great performance thanks to its zero levels of abstractions, then this article is for you.

To implement the C++ micro-service with the less amount of code we will make use of a great library from Microsoft called the C++ REST SDK, this is a cross-platform library meaning it runs on Linux, macOS and Windows, this library uses modern C++11 syntax and provides all the threading and networking I/O goodies that we require to implement a full-fledge micro-service that exposes a REST API interface.

To write and debug the code we'll use Visual Studio Code (VSC), but you can use Xcode on macOS or Visual Studio on Windows, vim, Emacs, etc. whatever you fill confortable with because the project does not depend on any IDE. But if you don't know how to use VSC but are curious about it check out my article about it.

Pre-requisites

- I am working on macOS and I use homebrew to install most of my dependencies so the following command will suffice:

```
$ brew update
$ brew install cmake openssl boost
```

- After *brew install* you should see the following directories (be aware that the versions shown are the latests available at the time of this article):

```
/usr/local/Cellar/boost/1.60.0_1
/usr/local/Cellar/cmake/3.4.2
/usr/local/Cellar/openssl/1.0.2f
```

Build the sample code

Clone the micro-service repository from github and follow there the instructions to build the sample code, then please come back here for further explanation of the code.

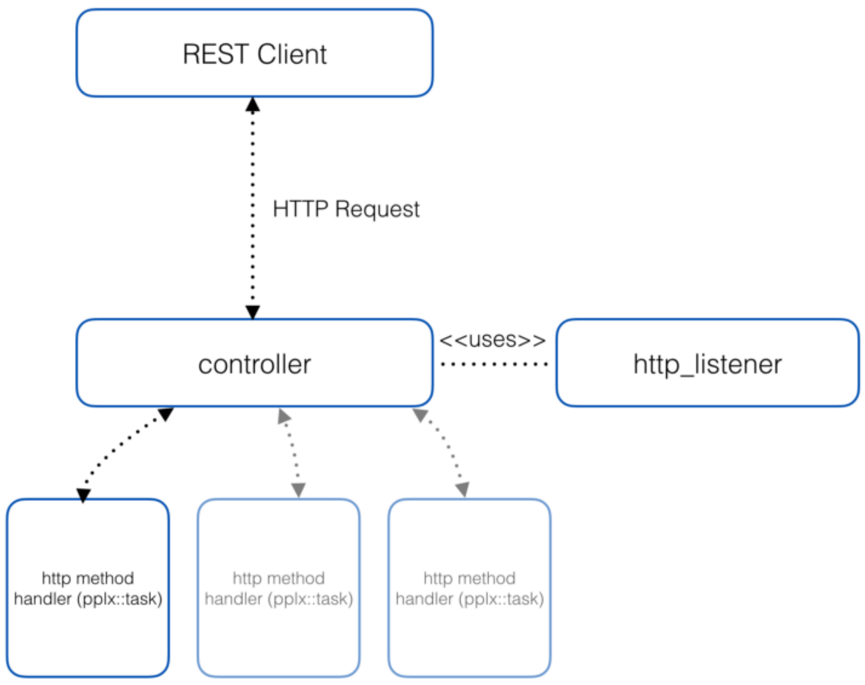
About C++ REST SDK

This is a cross-plaform open source library from Microsoft that allows you to create asynchronous REST clients on top of the Concurrency Runtime under the namespace **pplx** but it enables you to build REST servers as well. This library is adaptive which means that depending on the operating system you are compiling it, it takes care of the native implementations for example the **http_listener** class is implemented on top of IO Completion Ports (IOCP), which on **macOS** this is accomplished through the **kqueue** system call, on Linux this is accomplished through the **epoll** system call and on Windows through **CreateIoCompletionPort** but you don't need to know all about those IO APIs due to C++ REST SDK takes care of it becasue it additionally relies heavily on boost asio library all throught a single class called **http_listener**, but that is just an example of the adaptiveness of the library, the same applied for threads, sockets, character encoding, etc. elegantly wrapped on modern C++11 classes.

We are using C++ REST SDK to create a multi-threaded server that exposes a REST API and that can perfectly work as the basis of a full-fledge micro-service, capable to be used on any serious micro-serivce architecture backend implementation, specially on mission critical and high performance micro-services.

Micro-serivce implementation

The service has the following basic architecture:


REST Client
HTTP Request
controller
<<uses>>
http_listener
http method handler (pplx::task)
http method handler (pplx::task)
http method handler (pplx::task)

where a controller basically provides the handlers for each of the HTTP methods:

- GET
- POST
- PUT
- DELETE
- HEAD
- OPTIONS
- TRACE
- CONNECT
- MERGE

- PATCH

for more information about each method please take a look here.

The controller internally makes use of the class *web::http::experimental::listener::***http_listener** which basically enables the service to listen for HTTP requests.

Once a request is received the **http_listener** verifies is a valid HTTP request and that the HTTP method is supported by the service, if that is the case then handles the request to the **controller** for further processing in a thread different than the one demultiplexing the original request, this allow you to perform the request's logic in a separate thread of execution laving the **http_listener** ready to **accept** more request(s).

In order for the **http_listener** to know which handler to invoke on the *Controller* you need to register your handlers with the **http_listener**. For that we have the class *MicroserviceController* which derives from *BasicController* and *Controller* where *BasicController* provides a wrapper around **http_listener** and some convenient methods to read the endpoint string and decide if the service should figure out the host's IP address whether it is IP4 or IP6, or you can pass a fixed domain name or IP address; and the *Controller* provides an interface which the concrete *Controller* in this case *MicroserviceController* implements for each HTTP method, so open the file *microsvc_controller.hpp* and you will see the following code:

```
#pragma once

#include <basic_controller.hpp>

using namespace cfx;

class MicroserviceController : public BasicController,
Controller {
public:
 MicroserviceController() : BasicController() {}
 ~MicroserviceController() {}
 void handleGet(http_request message) override;
 void handlePut(http_request message) override;
 void handlePost(http_request message) override;
 void handlePatch(http_request message) override;
 void handleDelete(http_request message) override;
 void initRestOpHandlers() override;
};
```

you can see in the code above that *MicroserviceController* overrides the method *BasicController::initRestOpHandlers(),* and all the methods from the *Controller* interface. Now if you open the correspoding *.cpp file you'll see the code that registers the handlers on the *Controller* interface with the **http_listener** represented by the member variable *_listener*:

```
void MicroserviceController::initRestOpHandlers() {
 _listener.support(methods::GET,
std::bind(&MicroserviceController::handleGet, this,
std::placeholders::_1));
 _listener.support(methods::PUT,
std::bind(&MicroserviceController::handlePut, this,
std::placeholders::_1));
 _listener.support(methods::POST,
std::bind(&MicroserviceController::handlePost, this,
std::placeholders::_1));
 _listener.support(methods::DEL,
std::bind(&MicroserviceController::handleDelete, this,
std::placeholders::_1));
 _listener.support(methods::PATCH,
std::bind(&MicroserviceController::handlePatch, this,
std::placeholders::_1));
}
```

the binding magic of handlers is performed by the *std::bind* function template that generates a forwarding call wrapper for a non-static member function, where for all member functions only one argument is required, hence the *std::placeholders::_1* argument at the end of each declaration.

Once properly registered our handlers further in the code on the same file *microscv_controller.cpp* you can see the implementation of each method:

```
void MicroserviceController::handleGet(http_request message)
{
  auto path = requestPath(message);
  if (!path.empty()) {
     if (path[0] == “service” && path[1] == “test”) {
        auto response = json::value::object();
        response["version"] = json::value::string("0.1.1");
        response["status"] = json::value::string("ready!");
        message.reply(status_codes::OK, response);
     }
  }
  message.reply(status_codes::NotFound);
}
```

in the code above we can see the implementation of the handler for the HTTP method GET. Each time a GET request is issued to the service the http_listener will call this class method on our *MicroserviceController* class, and the same for each an every handler we register with http_listener.

The code above is passing the input *message* through the class method *requestPath* which basically extracts the request path from the message and returns a *std::vector<string>* that contains each section of the path so we can evaluate the REST API requested and its input parameters, in this case the request should be something like this:

```
curl -k -i -X GET -H "Content-Type:application/json"
http://<host_ip>:6502/v1/ivmero/api/service/test
```

> *Be aware that the <host_ip> is just a place holder, later in this article I'll talk about how to specify the endpoint string so the micro-service can automatically resolve the host's IP.*

and returns a JSON response like this, if the request is successful:

```
{
 "status" : "ready!",
 "version" : "0.1.1"
}
```

or an HTTP status code 404 (Not Found) if the request path does not match with any REST API supported by the service:

```
HTTP/1.1 404 Not Found


Content-Length: 0
```

The response is sent back to the caller via the input *message* instance of type **http_request** which has a reference to the caller.

Initializing the service

The initialization is located on the *main.cpp* file so let's walk through the code:

```
#include <iostream>


#include <usr_interrupt_handler.hpp>
#include <runtime_utils.hpp>


#include "microsvc_controller.hpp"


using namespace web;
using namespace cfx;


int main(int argc, const char * argv[]) {
    InterruptHandler::hookSIGINT();


    MicroserviceController server;

server.setEndpoint("http://host_auto_ip4:6502/v1/ivmero/api"
);

    try {
       // wait for server initialization…
       server.accept().wait();
       std::cout << "Copyright © ivmeroLabs 2016. ... now
listening for requests at: " << server.endpoint() << '\n';

    InterruptHandler::waitForUserInterrupt();


       server.shutdown().wait();
    }
    catch(std::exception & e) {
       std::cerr << "somehitng wrong happen! :(" << '\n';
    }
    catch(…) {
       RuntimeUtils::printStackTrace();
    }
    return 0;
}
```

there are a couple of header files *usr_interrupt_handler.hpp* which contains code to handle when the user types Ctrl+C on the terminal where the service is running and *runtime_utils.hpp* that contains code to print the stack trace when an unexpected exception ocurrs. Those files can be found under */micro-service/foundation/include.*

In the first line of code we hook on the SIGINT interrupt to handle the Ctrl+C.

```
InterruptHandler::hookSIGINT();
```

Then we create and initialize the service with the endpoint where it will be listening for requests:

```
MicroserviceController server;
server.setEndpoint("http://host_auto_ip4:6502/v1/ivmero/api"
);
```

if you pass *host_auto_ip4* or *host_auto_ip6* means the service should auto detect the IP address of the host, if you have more than one network cards it will use the first one it finds in the list of available IP addresses. Then comes the port number and finally the base address of the REST API in this case is */v1/ivmero/api.*

Then we initialize the thread pool that is used by pplx concurrency runtime and start accepting network connections:

```
server.accept().wait();
```

after that call if you are runnin on the debugger you'll see about 40 or so threads on the service process, that constitutes the thread pool hosted by the pplx scheduler. Those threads are pre-spawned to avoid delays when a thread is required and they are re-used during the service life time. So nothing to be scared of. :)

IMPORTANT: the *accept()* method returns a *pplx::task,* but in order to let it perform the creation of the thread pool and initialize the acceptor we have to wait for the task to finish hence the call to the *wait()* method.

As soon as the *wait()* call returns we need to setup a barrier to avoid the process to finish leaving the acceptor to die before any request can be processed, so we call:

```
InterruptHandler::waitForUserInterrupt();
```

which basically is a blocking call that handles the SIGINT and terminates as soon as the user types Ctrl + C on the terminal.

Finally if the user terminates the process we just shutdown the service and wait for all thread to finish:

```
server.shutdown().wait();
```

Conclusion

As you can see the code is quite simple to implement a C++ micro-service, as simple as a node.js server that exposes a REST interface, in subsequent articles I will continue talking about how to extract the body from a request in JSON format and how to add SSL support.