Un **serveur** REST Web API en C++

Christophe PICHAUD
Consultant sur les technologies Microsoft
christophepichaud@hotmail.com
www.windowscpp.net

Imaginez que vous puissiez développer le back-end de votre solution logicielle sur un serveur avec le roi des langages qu'est le C++ ? Sur Linux ? Oui. Sur Windows ? Oui. Un seul code portable ? Oui. Le tout sans complexité délirante, juste avec des classes et des méthodes de traitement… Et on retourne du JSON pour un front-end en JS ? Chiche ! Suivez-moi, je vous explique…

Casablanca alias C++ REST SDK

Pour développer une Web API, beaucoup d'entre nous ne connaissent que ASP.NET et C#… Depuis MFC ISAPI et ATL Server, Visual C++ ne fournit plus de templates de projets pour des développements serveur. Un SDK reprend le flambeau : le REST SDK, nom de code « Casablanca ». Ce SDK écrit en C++ moderne (C++ 11) propose de réaliser du code serveur et client en utilisant les derniers patterns, à savoir, le support du http et du JSON.
Disponible sur Github à l'adresse suivante : https://github.com/Microsoft/cpprestsdk . Ce SDK propose aussi des exemples de code pour commencer en douceur.

Un serveur http REST simple

Examinons le code minimum pour créer un serveur Web API qui retourne du JSON. Premièrement, il faut définir une classe qui contient un **http_listener**.

```
using namespace web;
using namespace http;
using namespace utility;
using namespace http::experimental::listener;

class MyServer
{
public:
    MyServer() {}
    MyServer(utility::string_t url);

    pplx::task<void> open() { return m_listener.open(); }
    pplx::task<void> close() { return m_listener.close(); }

private:

    static void handle_get(http_request message);
    static void handle_put(http_request message);
    static void handle_post(http_request message);
    static void handle_delete(http_request message);

    http_listener m_listener;
};
```

Ensuite, il faut configurer ce listener en lui fournissant un URI sur laquelle on se met en écoute ainsi que les handlers pour les opérations GET, PUT, POST et DELETE. Mais avant tout, il faut déclarer le serveur comme étant une variable globale comme suit :

```
std::unique_ptr<MyServer> g_http;
```

Ensuite, il faut préparer les éléments indispensables au serveur, à savoir son URI d'écoute :

```
utility::string_t port = U("34568");
utility::string_t address = U("http://localhost:");
address.append(port);

uri_builder uri(port);
uri.append_path(U("MyServer/Action/"));

auto addr = uri.to_uri().to_string();
g_http = std::unique_ptr<MyServer>(new MyServer(addr));
g_http->open().wait();

ucout << utility::string_t(U("Listening for requests at: ")) << addr << std::endl;
```

L'appel à open() active l'écoute. Reste à découvrir le code du constructeur de la classe MyServer, car il faut enregistrer les handlers sur les méthodes HTTP :

```
MyServer::MyServer(utility::string_t url) : m_listener(url)
{
    std::function<void(http_request)> fnGet = &MyServer::handle_get;
    m_listener.support(methods::GET, fnGet);

    std::function<void(http_request)> fnPut = &MyServer::handle_put;
    m_listener.support(methods::PUT, fnPut);

    std::function<void(http_request)> fnPost = &MyServer::handle_post;
    m_listener.support(methods::POST, fnPost);

    std::function<void(http_request)> fnDel = &MyServer::handle_delete;
    m_listener.support(methods::DEL, fnDel);
}
```

OK, le corps du serveur http est là… Regardons le corps des méthodes HTTP. Nous allons en coder une seule mais le système est simple.

```
void MyServer::handle_post(http_request message)
{
    ucout <<  message.to_string() << endl;
    message.reply(status_codes::OK);
};

void MyServer::handle_delete(http_request message)
{
    ucout << message.to_string() << endl;
    message.reply(status_codes::OK);
}

void MyServer::handle_put(http_request message)
{
    ucout << message.to_string() << endl;
    message.reply(status_codes::OK);
};
```

Ces méthodes ne sont pas implémentées mais patience. Voici ce que

nous allons coder. Sur la méthode GET si on nous envoie « get_developers », alors on dump une structure de données en JSON.

Le corps du handler GET

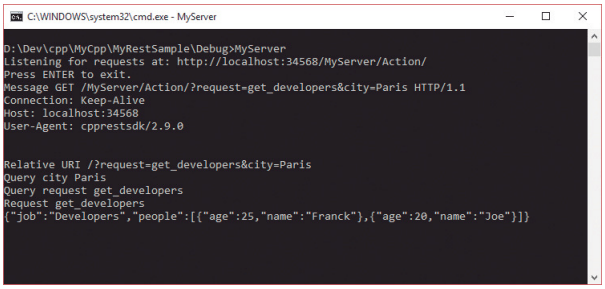
La structure de données JSON retournée est définie comme suit :

```
{"job":"Developers","people":[{"age":25,"name":"Franck"},{"age":20,"name":"Joe"}]}
```

Voyons le corps de cette méthode GET :

```
void MyServer::handle_get(http_request message)
{
  ucout << U("Message") << U(" ")
     << message.to_string() << endl;

  ucout << U("Relative URI") << U(" ")
     << message.relative_uri().to_string() << endl;

  auto paths = uri::split_path(uri::decode(message.relative_uri().path()));
  for (auto it1 = paths.begin(); it1 != paths.end(); it1++)
  {
     ucout << U("Path") << U(" ")
        << *it1 << endl;
  }

  auto query = uri::split_query(uri::decode(message.relative_uri().query()));
  for (auto it2 = query.begin(); it2 != query.end(); it2++)
  {
     ucout << U("Query") << U(" ")
        << it2->first << U(" ") << it2->second << endl;
  }

  auto queryItr = query.find(U("request"));
  utility::string_t request = queryItr->second;
  ucout << U("Request") << U(" ") << request << endl;

  if (request == U("get_developers"))
  {
     Data data;
     data.job = U("Developers");
     People p1;
     p1.age = 25;
     p1.name = U("Franck");
     data.peoples.push_back(p1);
     People p2;
     p2.age = 20;
     p2.name = U("Joe");
     data.peoples.push_back(p2);

     utility::string_t response = data.AsJSON().serialize();
     ucout << response << endl;

     message.reply(status_codes::OK, data.AsJSON());
     return;
  }

  message.reply(status_codes::OK);
};
```

On commence par dumper les éléments reçus en entrée ? On affiche la requête, l'url demandée et les paramètres. Vous remarquerez que uri::split_query retourne une map de string pour récupérer les paramètres de la requête. Si la requête est « get_developers », alors on déclare une structure de données et on génère du JSON. [1]



La génération du JSON

Voyons le code qui permet de générer du JSON à partir d'une structure de données. Il n'y a rien de magique, regardez :

```
struct People
{
  utility::string_t name;
  double age;

  static People FromJSON(const web::json::object & object)
  {
     People result;
     result.name = object.at(U("name")).as_string();
     result.age = object.at(U("age")).as_integer();
     return result;
  }

  web::json::value AsJSON() const
  {
     web::json::value result = web::json::value::object();
     result[U("name")] = web::json::value::string(name);
     result[U("age")] = web::json::value::number(age);
     return result;
  }
};
```

Cette structure représente un développeur. Il y a 2 propriétés : name et age. Le mécanisme de transformation d'une chaîne se fait avec les méthodes object.at().as_string() et web::json::value::string(). Pour un entier, c'est avec object.at().as_integer() et web::json::value::number(). C'est assez simple. Pour la structure suivante, qui met en œuvre un tableau (vector), il faut utiliser d'autres méthodes :

```
struct Data
{
  std::vector<People> peoples;
  utility::string_t job;

  Data() {}

  void Clear() { peoples.clear(); }
```

```
static Data FromJSON(const web::json::object &object)
{
    Data res;

    web::json::value job = object.at(U("job"));
    res.job = job.as_string();

    web::json::value p = object.at(U("people"));
    for (auto iter = p.as_array().begin(); iter != p.as_array().end(); ++iter)
    {
        if (!iter->is_null())
        {
            People people;
            people = People::FromJSON(iter->as_object());
            res.peoples.push_back(people);
        }
    }

    return res;
}

web::json::value AsJSON() const
{
    web::json::value res = web::json::value::object();
    res[U("job")] = web::json::value::string(job);

    web::json::value jPeoples = web::json::value::array(peoples.size());

    int idx = 0;
    for (auto iter = peoples.begin(); iter != peoples.end(); iter++)
    {
        jPeoples[idx++] = iter->AsJSON();
    }

    res[U("people")] = jPeoples;
    return res;
}
};
```

Pour générer un tableau en JSON, il faut utiliser la fonction web::json::value::array. Ensuite on itère sur le vector pour associer les éléments du tableau un par un.

La partie client

Le côté client peut être développé avec n'importe quelle technologie à partir du moment où il est permis de déclencher des requêtes http et de lire du JSON. Ici, le client est écrit aussi en C++ pour vous montrer comme c'est facile d'utiliser le REST SDK :

```
#ifdef _WIN32
# define iequals(x, y) (_stricmp((x), (y))==0)
#else
# define iequals(x, y) boost::iequals((x), (y))
#endif

int wmain(int argc, wchar_t *argv[])
{
    utility::string_t port = U("34568");
    if(argc == 2)
    {
        port = argv[1];
    }

    utility::string_t address = U("http://localhost:");
    address.append(port);
    http::uri uri = http::uri(address);

    http_client client(http::uri_builder(uri)
        .append_path(U("/MyServer/Action/")).to_uri());

    while (true)
    {
        std::string method;
        ucout << "Enter method:";
        cin >> method;

        http_response response;

        if (iequals(method.c_str(), "get_developers"))
        {
            utility::ostringstream_t buf;
            buf << U("?request=")
                << utility::conversions::to_string_t(method)
                << U("&city=Paris");

            response = client.request(methods::GET, buf.str()).get();

            ucout << U("Response ") << response.to_string() << endl;

            json::value jdata = json::value::array();
            jdata = response.extract_json().get();
            Data data = Data::FromJSON(jdata.as_object());
        }
        else
        {
            ucout << utility::conversions::to_string_t(method)
                  << ": not understood\n";
        }
    }

    return 0;
}
```

CONCLUSION

Le C++ REST SDK est facile à mettre en œuvre et permet d'avoir des fonctionnalités serveur sans difficultés particulières. Le support du JSON n'est pas trop pénible. Les opérations sont simples et la performance est au rendez-vous, et ceci en standard. L'avantage de la solution proposée c'est qu'elle ne dépend pas de IIS. Le module serveur est `stand-alone` Pour finaliser l'application, il faut lui ajouter une couche de service Windows et là, c'est la fête !