# Cadovvl blog

## There is no escape from here

# Организация http сервера с использованием библиотеки Poco

Posted by cadovvl on Февраль 26, 2015
Давненько не страдал фигней… Пришло время. Сегодня мы будем делать нормальный http сервер на C++.

Тут у нас есть множество различных возможностей его реализовать. И если уж я делаю это для себя…

В первую очередь я не хочу самостоятельно разбирать http протокол, поэтому прости, boost::asio, ты мне еще не годишься (несмотря на наличие в твоей документации примера http — сервера, код показался мне ни раза не юзабельным).

Во-вторых я точно хочу писать на C++, не опускаясь до уровня C (да простят меня «тру»), поэтому освоение «самого удобного http сервера» под названием mongoose тоже откладывается до лучших времен.

Из остальных более-менее приемлемых вариантов остановился на библиотеке Poco. Да, я со многим в этой библиотеке несогласен(особенно(!) в плане дизайна API для http), но преимущества в моем случае перевешивают недостатки.

# Введение

Одним из главных недостатков (и причин неиспользования) этой библиотеки является привязка одного ее компонента к другому. Нет, явной привязки нет: можно использовать разные ее компоненты по отдельности… нооооооо… стрелять себе в ногу (https://lurkmore.to/%C2%FB%F1%F2%F0%E5%EB%E8%F2%FC_%F1%E5%E1%E5_%E2_%ED%EE%E3 таким образом могут только истинные мазахисты.

Всязи с чем в первую очередь нам понадобится изучить организацию Application с помощью библиотеки Poco.

В данном случае нам понадобится Poco::Util::ServerApplication, который позволит не особенно напрягаясь превратить приложение при большом желании в демона/сервис. Использование этого класса заключается в наследовании от него и переопределении некоторых функций. Однако на это переопределение накладываются некоторые ограничения, а именно:

- Subsystems must be registered in the constructor.
- All non-trivial initializations must be made in the initialize() method.
- At the end of the main() method, waitForTerminationRequest (http://pocoproject.org/docs/Poco.Util.ServerApplication.html#26964)() should be called.
- New threads must only be created in initialize() or main() or methods called from there, but not in the application class' constructor or in the constructor of instance variables. The reason for this is that fork() will be called in order to create the daemon process, and threads created prior to calling fork() won't be taken over to the daemon process.
- The main(argc, argv) function must look as follows: <и дальше копипаста кода>

Грубо переводя, правила говорят, что общую логику работы класса нарушать не кошерно. Чтож… Не будем.

Итак… Простенький main файл, который позволит проверить, что вы умеете подключать, компилировать и линковать стороннюю библиотеку выглядит следующим образом (перегружать пост своими некрасивыми make-файлами не хочу):

```
1  // Copyright 2015 cadovvl
2  // main.cpp
3  #include <Poco/Util/ServerApplication.h>;
4
5  POCO_SERVER_MAIN(Poco::Util::ServerApplication)
```

Де-факто из-за последнего правила использование поковского серверного аппликейшена удобнее воспользоваться их же макросом (~~НИКОГДА НЕ ИСПОЛЬЗУЙТЕ МАКРОСЫ ВСУЕ!~~), объявление которого в их заголовке выглядит так (писать это не надо, все уже написано. Эти две строчки уже скомпилятся в работающее приложение!):

```
 1  #define POCO_SERVER_MAIN(App) \
 2  int main(int argc, char** argv)     \
 3  {                                   \
 4      try                             \
 5      {                               \
 6          App app;                    \
 7          return app.run(argc, argv); \
 8      }                               \
 9      catch (Poco::Exception& exc)    \
10      {                               \
11          std::cerr << exc.displayText() << std::endl;    \
12          return Poco::Util::Application::EXIT_SOFTWARE;  \
13      }                               \
14  }
```

# Рабочий Application

Отлично. Только теперь нам нужно записать в приложение свою мегалогику. Пока без всяких неймспейсов и все в заголовочном файле(простите за отступы, он их так убого из emacs-а копирует сюда. Сам я по google стайлу пишу. Ну, кроме использования потоков… Честно-честно).

```
1  // Copyright 2015 cadovvl
2  // Listener.h
3  #ifndef SERVER_LISTENER_H_
4  #define SERVER_LISTENER_H_
5
6  #include <Poco/Util/ServerApplication.h>
7
8  #include <vector> // это гуглстайл попросил
9  #include <string> // и это тоже
10 #include <iostream>
11
12 class Listener : public Poco::Util::ServerApplication {
13  virtual int main(const std::vector<std::string>& args) {
14   std::cout << "Use ploho!" << std::endl;
15   waitForTerminationRequest();
16   std::cout << "Ended normally" << std::endl;
17   return 0;
18  }
19 };
20 #endif // SERVER_LISTENER_H_
```

Соответственно, основное приложение меняется следующим образом:

```
1  // Copyright 2015 cadovvl
2  // main.cpp
3  #include <Poco/Util/ServerApplication.h0
4
5  #include "server/Listener.h"
6
7  POCO_SERVER_MAIN(Listener)
```

Теперь по коду. Вместо инстанса обычного серверного приложения мы создаем инстанс нашего серверного приложения. Это происходит в мейне.

В самом листенере мы переопределяем протектед метод «main», который содержит основную логику приложения. В целом он мало чем отличается от стандартного, только аргументы передаются туда в C++ стиле, а так же по правилу использования этого метода в ServerApplication я должен в конце дождаться terminate сигнала(ну, я немного отошел от правила, но на суть это не повлияло…).

Что это нам позволяет сделать. Скомпилировав и запустив это приложение мы увидим следующее:

```
$ ./server.out
Use ploho!
```

Дальше приложение дожидается, пока мы вызовем останавливающий сигнал. В случае с демоном — это остановка от системы. В нашем случае, это аналог kill. Я использую Ctrl+C:

```
$ ./server.out
Use ploho!
^CEnded normally
```

Хей! Да мы круты!!! Теперь вместо использования тупого вывода в консоль можно в другом потоке запустить что-нибудь полезное, а по принятии останавливающего сигнала аккуратненько остановить процесс, дописать логи и т.п. Например, можно запустить примитивный сервер!

# Серверная часть

Немного о серверах. Сервера находятся в Poco::Net и в зависимости от их использования могут стать вам как обузой, так и подспорьем. Для эффективного использования сети с помощью этого дела вам понадобится

1. Набор обработчиков «подсайтов»
2. Генератор обработчиков «подсайтов»
3. Приблуда, устанавливающая соединение, разбирающая http протокол, скармливающая результаты

Начнем с генератора обработчиков. К нему приходят запросы, на основании которых он должен выдать те самые обработчики. В нашем случае мы можем не возвращать их. В этом случае код будет выглядеть следующим образом:

```
1  // Copyright 2015 cadovvl
2  // CommonRequestHandler.h
3  #ifndef SERVER_COMMONREQUESTHANDLER_H_
4  #define SERVER_COMMONREQUESTHANDLER_H_
5
6  #include <Poco/Net/HTTPRequestHandlerFactory.h>
7
8  class CommonRequestHandler : public Poco::Net::HTTPRequestHandlerFactory {
9
10  public:
11  CommonRequestHandler() {
12  }
13
14  virtual Poco::Net::HTTPRequestHandler* createRequestHandler(
15  const Poco::Net::HTTPServerRequest& request) {
16  return nullptr; // Does everybody use C++11?
17  }
18
19 };
20
21 #endif // SERVER_COMMONREQUESTHANDLER_H_
```

Кратко.

```
HTTPRequestHandler* createRequestHandler
```

это чисто-виртуальный метод, в логике которого будет раздача настоящих рабочих папакарл. На данный момент эта штука возвращает пустой указатель на любой запрос. Не самый полезный сервер, но достаточный для того, чтобы оно скомпилировалось.

Конструктор тоже переопределен, и хорошая практика — передавать в него настройки сервера из конфигурационных файлов. Но пока пусть останется пустым. (Ато если так замечтаться, то и папакарл надо грузить из динамических либ. Но не время и не место этим сейчас заниматься).

А запускается вся эта прелесть в нашем листенере. Если мы захардкодим настройки (временно, по спец-необходимости), код будет выглядеть примерно следующим образом (меня тоже бесит как wordpress кастрирует отступы!):

```
// Copyright 2015 cadovvl
// Listener.h
#ifndef SERVER_LISTENER_H_
#define SERVER_LISTENER_H_

#include <Poco/Util/ServerApplication.h>
#include <Poco/Net/HTTPServer.h>

#include <vector>
#include <string>
#include <iostream>

#include "./CommonRequestHandler.h"

class Listener : public Poco::Util::ServerApplication {
 virtual int main(const std::vector<std::string>& args) {
 Poco::Net::HTTPServerParams* params = new Poco::Net::HTTPServerParams();
 params->setMaxQueued(50);
 params->setMaxThreads(4);

 Poco::Net::ServerSocket socket(8765); // argument is a port

 Poco::Net::HTTPServer server(new CommonRequestHandler(),
 socket,
 params);

 server.start();
 waitForTerminationRequest();
 server.stop();

 return 0;
 }
};
#endif // SERVER_LISTENER_H_
```

Первые две переменные — просто параметры. Не стоит обращать на них внимание на первое время. Просто поясню один момент: очищение памяти сервер берет на себя сам.

Третья переменная — сервер. Он берет себе сокет для конкретного порта (всякие там bind, listen делает), принимает параметры соединения и нашу генерилку. Мы запустили его в 4 потока, с пределом в 50 клиентов.

Дальше наш сервер запускается и будет работать до тех пор, пока мы не попытаемся прибить нашу программу. После чего мягенько закроет соединение и почистит за собой.

Эта штука уже способна компилироваться и что-то делать. А именно, запустив теперь приложение, мы можем достучаться до захардкоженного нами порта:

```
$ curl -IL 'localhost:8765'
HTTP/1.1 501 Not Implemented
Connection: Close
```

Он устанавливает соединение, но получает нерабочий обработчик на этот url. Падает с исключением, которое для него означает, что реализации этого функционала еще нет. В этом случае возвращается код 501. Неплохо, дело за малым: написать обработчики!

# Обработчики

На этом этапе код становится достаточно сложным, чтобы содержание его в заголовках не рисковало перерости в проблему. Поэтому в дальнейшем код разбит на заголовки и файлы, с соответствующими изменениями в компиляции и линковке.

Обработчики представляют собой класс, в котором требуется переопредилить одну чисто-виртуальную функцию. Она принимает ссылку на запрос и ссылку на ответ. Соответственно, читаем инфу из одного, пишем в другую. Пример такого хендлера:

```
1  // Copyright 2015 cadovvl
2  // Handlers.h
3  #ifndef SERVER_HANDLERS_H_
4  #define SERVER_HANDLERS_H_
5
6  #include <Poco/Net/HTTPRequestHandler.h>
7
8  #include <iostream>
9
10 class MotherHandler : public Poco::Net::HTTPRequestHandler {
11  public:
12  virtual void handleRequest(Poco::Net::HTTPServerRequest& req,
13  Poco::Net::HTTPServerResponse& res);
14 };
15
16 #endif // SERVER_HANDLERS_H_
```

```
1  // Copyright 2015 cadovvl
2  // Handlers.cpp
3
4  #include "./Handlers.h"
5
6  #include <Poco/Net/HTTPRequestHandler.h>
7  #include <Poco/Net/HTTPServerResponse.h>
8  #include <Poco/Net/HTTPServerRequest.h>
9
10 #include <iostream>
11
12 void MotherHandler::handleRequest(Poco::Net::HTTPServerRequest& req,
13  Poco::Net::HTTPServerResponse& res) {
14
15  std::cout << "Mother request was called" << std::endl;
16
17  std::ostream& reply = res.send();
18  reply << "U have called Mother request" << std::endl;
19  res.setStatus(Poco::Net::HTTPServerResponse::HTTP_OK);
20 }
```

Подводные камни, на которые тут можно наткнуться.

1. В заголовке HTTPRequestHandler реквест и респонс объявлены как частичные классы. Это означает, что дополнительные include-ы для их подгрузки обязательны.
2. Метод res.send() получает поток, в который можно записать данные в ответ на запрос. Однако вызывать его можно **только один раз**!

Далее этот хендлер нужно вставить в наш генератор:

```
// Copyright 2015 cadovvl
// CommonRequestHandler.h
#ifndef SERVER_COMMONREQUESTHANDLER_H_
#define SERVER_COMMONREQUESTHANDLER_H_

#include <Poco/Net/HTTPRequestHandlerFactory.h>

class CommonRequestHandler : public Poco::Net::HTTPRequestHandlerFactory {

 public:
 CommonRequestHandler();

 virtual Poco::Net::HTTPRequestHandler* createRequestHandler(
              const Poco::Net::HTTPServerRequest& request);

};

#endif // SERVER_COMMONREQUESTHANDLER_H_
```

```
// Copyright 2015 cadovvl
// CommonRequestHandler.cpp

#include <Poco/Net/HTTPServerRequest.h>

#include "./Handlers.h"
#include "./CommonRequestHandler.h"

CommonRequestHandler::CommonRequestHandler() {
}

Poco::Net::HTTPRequestHandler* CommonRequestHandler::createRequestHandler(
                      const Poco::Net::HTTPServerRequest& request) {
 if (request.getURI() == "/mother/") {
   return new MotherHandler();
 }
 return nullptr; // Does everybody use C++11?
}
```

Что происходит. При вызове подсайта «mother», запрос передается на обработку нажему обработчику:

```
$ curl -IL 'localhost:8765/mother/'
HTTP/1.1 200 OK
Connection: Close
Date: Thu, 26 Feb 2015 20:22:08 GMT

$ curl 'localhost:8765/mother/'
U have called Mother request
```

При этом его подпути и корень остаются нетронутыми:

```
$ curl -IL 'localhost:8765/'
HTTP/1.1 501 Not Implemented
Connection: Close

$ curl -IL 'localhost:8765/mother/something/'
HTTP/1.1 501 Not Implemented
Connection: Close
```

Дело за малым: научиться принимать чуть более сложные запросы!

# Сложные запросы

Первая вещь, которую надо сделать при разборе сложных запросов — отделить имя запроса от его параметров. Этим занимается объект Poco::URI :

```
1  // Copyright 2015 cadovvl
2  // CommonRequestHandler.cpp
3
4  #include <Poco/Net/HTTPServerRequest.h>
5  #include <Poco/URI.h>
6
7  #include "./Handlers.h"
8  #include "./CommonRequestHandler.h"
9
10 CommonRequestHandler::CommonRequestHandler() {
11 }
12
13 Poco::Net::HTTPRequestHandler* CommonRequestHandler::createRequestHandler(
14  const Poco::Net::HTTPServerRequest& request) {
15  Poco::URI uri(request.getURI());
16
17  if (uri.getPath() == "/mother") {
18  return new MotherHandler();
19  }
20  return nullptr; // Does everybody use C++11?
21 }</pre>
22 <pre>
```

Uri сделает за вас всю грязную работу. Разберется с экранированием в запросе, извлечет путь, запрос, параметр, хост, порт, имя пользователя… Все, что пожелаете. Нас в данный момент интересует только путь. Дальше нужно научится разбирать параметры. Это уже ответственность непосредственно хендлера.

```
1  // Copyright 2015 cadovvl
2  // Handlers.cpp
3
4  #include "./Handlers.h"
5
6  #include <Poco/Net/HTTPRequestHandler.h>
7  #include <Poco/Net/HTTPServerResponse.h>
8  #include <Poco/Net/HTTPServerRequest.h>
9  #include <Poco/Net/HTMLForm.h>
10
11 void MotherHandler::handleRequest(Poco::Net::HTTPServerRequest& req,
12  Poco::Net::HTTPServerResponse& res) {
13
14  Poco::Net::HTMLForm form(req);
15  std::ostream& reply = res.send();
16
17  if (form.has("presented-param")) {
18     reply << "presented param is " << form.get("presented-param") << std::endl
19     // Also works well
20     // reply << "presented param is " << form["presented-param"] << std::endl;
21  }
22
23  reply << "dummy param is "
24     << form.get("dummy-param", "default Dummy Param")
25     << std::endl;
26
27  reply << "***************** All parametrs ***************" << std::endl;
28
29  for (auto i : form) {
30     reply << i.first << "\tis\t" << i.second << std::endl;
31  }
32
33  res.setStatus(Poco::Net::HTTPServerResponse::HTTP_OK);
34 }
```

HTMLForm унаследован от хранилища ключ->множество значений. Поэтому ему подвластны все опреаторы этого хранилища (осторожно! Эксепшены!), т.е. итерации, получение параметров по ключу и т.п.

В результате полученный код будет работать примерно следующим образом:

```
$ curl 'localhost:8765/mother?presented-param=15&multi-param=17&multi-param=28'
presented param is 15
dummy param is default Dummy Param
***************** All parametrs ***************
multi-param is 17
multi-param is 28
presented-param is 15
```

По набору таких параметров уже можно запускать приложение.

# Еще несколько саджестов

Для тех, кто отстал от жизни с самого начала: большинство кода компилится строчкой типа

```
g++ -std=c++11 -Wall -Werror -O2  filename.cpp -lPocoNet -lPocoFoundation -lPocoUtil
```

Понятно, что код еще далек от совершенства. Да, в него уже впилено логгирование (Application::instance().logger()  из любого места), только настрой при запуске путь, куда их складировать.

Хотелось бы настройки сервера видеть не в коде (Poco::Util::XMLConfiguration), а так же, чтобы хендлеры настраивались тоже их конфига, а в идеале просто добрасывались в папочку, откуда их по запросу бы считывали (Poco::ClassLoader), проверку куков (Это прям в реквестах же есть) и многое другое.

~~А мейкфайлы мне просто стыдно показывать…~~

Но тем не менее, с этой версией уже достаточно неплохо можно работать.

Что мне в этом не понравилось:

1. Тошнит от голых указателей. Нет, внутри все довольно умно организовано, но глаза они режут. При том, что «умные указатели» в поко есть даже свои.
2. Не нравится, что с возможностями C++ из него в этой части библиотеки сделали откровенную джаву. Хоть на питон снова пересаживайся… С другой стороны — библиотеке 10 лет. Тогда о C++11 еще даже не мечтали.
3. Недопиленная механика работы Uri со странностями (вещи типа «не различают запросы со слешом и без него на конце» или отсутствие некоторых конструкторов, которые прямо просятся…)

Что понравилось:

1. ООП как оно должно быть.
2. Ловля всех исключений и их трактовка.
3. Куча побочных вкусняшек типа того же логгера.
4. **ДОКУМЕНТАЦИЯ!!!!**

Ну и на последок: есть еще один сервер, который я не упомянул: у facebook. Если у кого остались от него хорошие впечатления — дайте знать. Может приобщусь к прекрасному…

P.S. wordpress трижды пытался добавить экранирование в код, могло остаться. Буду благодарен, если сообщие об опечатках.

Запись опубликована в рубрике Свободные библиотеки с метками BSD, C++, Poco, programming. Добавьте в закладки постоянную ссылку.

# 5 responses to “*Организация http сервера с использованием библиотеки Poco*”

*Владимир*:
Январь 29, 2019 в 14:26
Спасибо за статью! Сейчас поставил https сервер, который был в примерах у POCO. Пока не могу написать клиента. РНа просторах интернета лишь отрывки кода с попытками. Может быть делали? Не подскажите как его сделать?

Ответить

*cadovvl*:
Январь 29, 2019 в 14:54
Делал когда-то. Давно это было, помнится, тогда было два способа это сделать.

Первый — создаешь SocketStream И пишешь туда собственноручно написанный запрос вместе с протоколом.
Второй — создаешь HttpStream, и делаешь то же самое. Пользовался первым способом: во втором тогда чего-то не хватало.

У них общий принцип, схожий с std::iostream. Есть данные, которые ты передаешь, и манипуляторы. Например, в std::ostram можно передать минимальную ширину слова как-то так:
std::ostream out;
out << std::setw(4) << 42;

Примерно так же работаешь и с поковскими стримами. С синтаксисом сейчас могу наврать, но оно должно быть как-то так:
HTTPSession session;
HTTPOutputStream out(session);
HTTPHeaderOutputStream header(session);

out << "MyMegaRequest";
header << "OAuth" << "MyToken";

Дальше передать в сессию сокет и через что-то там отправить.

Скорее всего, в поко давно уже внедрили злостный антипаттерн "Factory" (интересно, я писал уже про мою ненависть к этому антипаттерну), и там все кошерно делать через него. Типа, есть фабрика клиента. Из нее генерируешь соединения, из него генерируешь стримы на HTTP, на HTTPHeaders, передаешь туда текст и манипуляторы, и когда надоедает — закрываешь соединение.

Но конкретики а) не помню б) давно туда не лазил в принципе.

Надеюсь, чем-то хоть помог….

Ответить

*Владимир*:
Февраль 13, 2019 в 13:47
Спасибо. Уже реализовал. Сейчас клиентом отправляю запрос POST. В данный на сервере думаю как извлечь информацию из такого запроса.

Ответить

*cadovvl*:
Февраль 13, 2019 в 14:09
Не совсем понял: извлечь информацию из POST запроса. Для получения информации GET запросы есть…

Ответить

*Владимир*:
Февраль 28, 2019 в 11:02
Получилось, что нужно было реализовать POST запрос. У него в добавок к URI(как у GET) есть тело(body). В теле нужно было определенный JSON передать. На сервере принять и распарсить.

Ответить