RESTful API requests using Qt/C++ for Linux, Mac OSX, MS Windows

In a recent article we showed how HTTP requests are formed in low level. We demonstrated how to create a simple HTTP GET request, a URL encoded POST request, and an upload request (multipart POST HTTP request).

This article has working examples on how to create those HTTP requests programmatically using Qt for Linux, Mac OSX or MS Windows.

The next sections of this article include the code that does the job, and then a couple of examples that demonstrate ways to create different types of HTTP requests.

The code

Create a new C++ Class with the name `HttpRequestWorker` with a base class of `QObject`.

Copy the following code in the `httprequestworker.h` file.

```
#ifndef HTTPREQUESTWORKER_H
#define HTTPREQUESTWORKER_H

#include <QObject>
#include <QString>
#include <QMap>
#include <QNetworkAccessManager>
#include <QNetworkReply>


enum HttpRequestVarLayout {NOT_SET, ADDRESS, URL_ENCODED, MULTIPART};


class HttpRequestInputFileElement {

public:
    QString variable_name;
    QString local_filename;
    QString request_filename;
    QString mime_type;

};


class HttpRequestInput {

public:
    QString url_str;
    QString http_method;
    HttpRequestVarLayout var_layout;
    QMap<QString, QString> vars;
    QList<HttpRequestInputFileElement> files;

    HttpRequestInput();
    HttpRequestInput(QString v_url_str, QString v_http_method);
    void initialize();
    void add_var(QString key, QString value);
    void add_file(QString variable_name, QString local_filename, QString request_filename, QString mime_type);
```

```cpp

};


class HttpRequestWorker : public QObject {
    Q_OBJECT

public:
    QByteArray response;
    QNetworkReply::NetworkError error_type;
    QString error_str;

    explicit HttpRequestWorker(QObject *parent = 0);

    QString http_attribute_encode(QString attribute_name, QString input);
    void execute(HttpRequestInput *input);

signals:
    void on_execution_finished(HttpRequestWorker *worker);

private:
    QNetworkAccessManager *manager;

private slots:
    void on_manager_finished(QNetworkReply *reply);

};

#endif // HTTPREQUESTWORKER_H
```

Copy the following code in the `httprequestworker.cpp` file.

```cpp
#include "httprequestworker.h"
#include <QDateTime>
#include <QUrl>
#include <QFileInfo>
#include <QBuffer>


HttpRequestInput::HttpRequestInput() {
    initialize();
}

HttpRequestInput::HttpRequestInput(QString v_url_str, QString v_http_method) {
    initialize();
    url_str = v_url_str;
    http_method = v_http_method;
}

void HttpRequestInput::initialize() {
    var_layout = NOT_SET;
    url_str = "";
    http_method = "GET";
}

void HttpRequestInput::add_var(QString key, QString value) {
    vars[key] = value;
}

void HttpRequestInput::add_file(QString variable_name, QString local_filename, QString request_filename, QStri
    HttpRequestInputFileElement file;
    file.variable_name = variable_name;
    file.local_filename = local_filename;
    file.request_filename = request_filename;
    file.mime_type = mime_type;
    files.append(file);
}


HttpRequestWorker::HttpRequestWorker(QObject *parent)
    : QObject(parent), manager(NULL)
{
    qsrand(QDateTime::currentDateTime().toTime_t());

    manager = new QNetworkAccessManager(this);
    connect(manager, SIGNAL(finished(QNetworkReply*)), this, SLOT(on_manager_finished(QNetworkReply*)));
}

QString HttpRequestWorker::http_attribute_encode(QString attribute_name, QString input) {
    // result structure follows RFC 5987
    bool need_utf_encoding = false;
    QString result = "";
    QByteArray input_c = input.toLocal8Bit();
    char c;
    for (int i = 0; i < input_c.length(); i++) {
        c = input_c.at(i);
        if (c == '\\' || c == '/' || c == '\0' || c < ' ' || c > '~') {
            // ignore and request utf-8 version
            need_utf_encoding = true;
        }
        else if (c == '"') {
            result += "\\\"";
        }
        else {
            result += c;
        }
    }

    if (result.length() == 0) {
        need_utf_encoding = true;
    }

    if (!need_utf_encoding) {
        // return simple version
        return QString("%1=\"%2\"").arg(attribute_name, result);
    }

    QString result_utf8 = "";
    for (int i = 0; i < input_c.length(); i++) {
        c = input_c.at(i);
        if (
            (c >= '0' && c <= '9')
            || (c >= 'A' && c <= 'Z')
            || (c >= 'a' && c <= 'z')
        ) {
            result_utf8 += c;
        }
        else {
            result_utf8 += "%" + QString::number(static_cast<unsigned char>(input_c.at(i)), 16).toUpper();
        }
    }

    // return enhanced version with UTF-8 support
    return QString("%1=\"%2\"; %1*=utf-8''%3").arg(attribute_name, result, result_utf8);
}

void HttpRequestWorker::execute(HttpRequestInput *input) {

    // reset variables

    QByteArray request_content = "";
    response = "";
    error_type = QNetworkReply::NoError;
    error_str = "";


    // decide on the variable layout

    if (input->files.length() > 0) {
        input->var_layout = MULTIPART;
    }
    if (input->var_layout == NOT_SET) {
        input->var_layout = input->http_method == "GET" || input->http_method == "HEAD" ? ADDRESS : URL_ENCODE
    }


    // prepare request content

    QString boundary = "";

    if (input->var_layout == ADDRESS || input->var_layout == URL_ENCODED) {
        // variable layout is ADDRESS or URL_ENCODED

        if (input->vars.count() > 0) {
            bool first = true;
            foreach (QString key, input->vars.keys()) {
                if (!first) {
                    request_content.append("&");
                }
                first = false;

                request_content.append(QUrl::toPercentEncoding(key));
                request_content.append("=");
                request_content.append(QUrl::toPercentEncoding(input->vars.value(key)));
            }

            if (input->var_layout == ADDRESS) {
                input->url_str += "?" + request_content;
                request_content = "";
            }
        }
    }
    else {
        // variable layout is MULTIPART

        boundary = "__------------------------"
            + QString::number(QDateTime::currentDateTime().toTime_t())
            + QString::number(qrand());
        QString boundary_delimiter = "--";
        QString new_line = "\r\n";

        // add variables
        foreach (QString key, input->vars.keys()) {
            // add boundary
            request_content.append(boundary_delimiter);
            request_content.append(boundary);
            request_content.append(new_line);

            // add header
            request_content.append("Content-Disposition: form-data; ");
            request_content.append(http_attribute_encode("name", key));
            request_content.append(new_line);
            request_content.append("Content-Type: text/plain");
            request_content.append(new_line);

            // add header to body splitter
            request_content.append(new_line);

            // add variable content
            request_content.append(input->vars.value(key));
            request_content.append(new_line);
        }

        // add files
        for (QList<HttpRequestInputFileElement>::iterator file_info = input->files.begin(); file_info != input
            QFileInfo fi(file_info->local_filename);

            // ensure necessary variables are available
            if (
                file_info->local_filename == NULL || file_info->local_filename.isEmpty()
                || file_info->variable_name == NULL || file_info->variable_name.isEmpty()
                || !fi.exists() || !fi.isFile() || !fi.isReadable()
            ) {
                // silent abort for the current file
                continue;
            }

            QFile file(file_info->local_filename);
            if (!file.open(QIODevice::ReadOnly)) {
                // silent abort for the current file
                continue;
            }

            // ensure filename for the request
            if (file_info->request_filename == NULL || file_info->request_filename.isEmpty()) {
                file_info->request_filename = fi.fileName();
                if (file_info->request_filename.isEmpty()) {
                    file_info->request_filename = "file";
                }
            }

            // add boundary
            request_content.append(boundary_delimiter);
            request_content.append(boundary);
            request_content.append(new_line);

            // add header
            request_content.append(QString("Content-Disposition: form-data; %1; %2").arg(
                http_attribute_encode("name", file_info->variable_name),
                http_attribute_encode("filename", file_info->request_filename)
            ));
            request_content.append(new_line);

            if (file_info->mime_type != NULL && !file_info->mime_type.isEmpty()) {
                request_content.append("Content-Type: ");
                request_content.append(file_info->mime_type);
                request_content.append(new_line);
            }

            request_content.append("Content-Transfer-Encoding: binary");
            request_content.append(new_line);

            // add header to body splitter
            request_content.append(new_line);

            // add file content
            request_content.append(file.readAll());
            request_content.append(new_line);

            file.close();
        }

        // add end of body
        request_content.append(boundary_delimiter);
        request_content.append(boundary);
        request_content.append(boundary_delimiter);
    }


    // prepare connection

    QNetworkRequest request = QNetworkRequest(QUrl(input->url_str));
    request.setRawHeader("User-Agent", "Agent name goes here");

    if (input->var_layout == URL_ENCODED) {
        request.setHeader(QNetworkRequest::ContentTypeHeader, "application/x-www-form-urlencoded");
    }
    else if (input->var_layout == MULTIPART) {
        request.setHeader(QNetworkRequest::ContentTypeHeader, "multipart/form-data; boundary=" + boundary);
    }

    if (input->http_method == "GET") {
        manager->get(request);
    }
    else if (input->http_method == "POST") {
        manager->post(request, request_content);
    }
    else if (input->http_method == "PUT") {
        manager->put(request, request_content);
    }
    else if (input->http_method == "HEAD") {
        manager->head(request);
```

```cpp
    }
    else if (input->http_method == "DELETE") {
        manager->deleteResource(request);
    }
    else {
        QBuffer buff(&request_content);
        manager->sendCustomRequest(request, input->http_method.toLatin1(), &buff);
    }

}

void HttpRequestWorker::on_manager_finished(QNetworkReply *reply) {
    error_type = reply->error();
    if (error_type == QNetworkReply::NoError) {
        response = reply->readAll();
    }
    else {
        error_str = reply->errorString();
    }

    reply->deleteLater();

    emit on_execution_finished(this);
}
```

The controller class should contain one or more places that call the worker unit and one event handler function. The event handler is required because this example's code makes asynchronous HTTP calls in order for the main thread to keep being responsive and not freeze.

In our example the main class of the application is called `MainWindow` and it contains a button which triggers the examples that follow.

Here is the code for the header file of the MainWindow class (mainwindow.h).

```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "httprequestworker.h"

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;

private slots:
    void on_pushButton_clicked();
    void handle_result(HttpRequestWorker *worker);

};

#endif // MAINWINDOW_H
```

Here is the code for the main file of the MainWindow class (mainwindow.cpp).

```cpp
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QNetworkReply>
#include <QMessageBox>


MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow() {
    delete ui;
}

void MainWindow::on_pushButton_clicked() {
    // trigger the request - see the examples in the following sections
}

void MainWindow::handle_result(HttpRequestWorker *worker) {
    QString msg;

    if (worker->error_type == QNetworkReply::NoError) {
        // communication was successful
        msg = "Success - Response: " + worker->response;
    }
    else {
        // an error occurred
        msg = "Error: " + worker->error_str;
    }

    QMessageBox::information(this, "", msg);
}
```

There are a couple of notable things in this code:

1. The use of asynchronous methods for the communication process.

   There are a couple of ways you can implement an HTTP communication in Qt. We chose to use `QNetworkAccessManager`. In order to use our code without problems you need to have Qt v4.7 or newer installed.

   Known problems:
   The HTTP methods (aka HTTP verbs) GET, POST, PUT, HEAD and DELETE work smoothly. Other HTTP methods will make use of `QNetworkAccessManager::sendCustomRequest()` which did not work properly in our tests. We were unable to find the source of this issue but we do not intend to give it any more time because other HTTP methods are rarely used. It feels as a Qt bug. For future reference this issue was found using Qt 5.2.1, Qt Creator 3.0.1, Qmake on a Mac OSX 10.9.4 (Maverics) .

2. The significance of specialized classes for the HTTP input.

   This code uses the class `HttpRequestInput` as an encapsulation for input variables of the HTTP request.

3. The role of `http_attribute_encode()`

   The original HTTP protocols are not very good at handling non-latin characters in attribute values. RFC 5987 makes the necessary provisions for a well thought out way to handle UTF-8 (unicode) characters. This function implements the rules set to handle non-latin characters in HTTP attribute values.

4. Trigger code

   The code we just showed includes the function `on_pushButton_clicked()`, an event handler of a button click from the UI. The next sections of this article will differentiate this function to implement different types of HTTP requests.

5. Result handling

   This code uses the `handle_result()` function to handle the asynchronous communication result.

6. Little things **you need to adjust** in your implementation:

   This code sets the HTTP header for the name of the HTTP client/agent. You need to change "Agent name goes here" with the name of your client.

   You obviously need to set the right code for triggering the code in `on_pushButton_clicked()` . Read the next sections for more examples on this function.

   Naturally, you should customize `handle_result()` . Our code is just an example of the different things you can do when the communication is completed.

Simple HTTP GET requests

Lets see how to run a simple GET request.

```
void MainWindow::on_pushButton_clicked() {
    QString url_str = "http://www.example.com/path/to/page.php";

    HttpRequestInput input(url_str, "GET");

    HttpRequestWorker *worker = new HttpRequestWorker(this);
    connect(worker, SIGNAL(on_execution_finished(HttpRequestWorker*)), this, SLOT(handle_result(HttpRequestWorke
    worker->execute(&input);
}
```

This will produce the following HTTP request:

```
GET /path/to/page.php HTTP/1.1
User-Agent: Agent name goes here
Connection: Keep-Alive
Accept-Encoding: gzip, deflate
Accept-Language: en-US,*
Host: www.example.com
```

The previous example calls a plain URL. Lets add a few variables.

```
void MainWindow::on_pushButton_clicked() {
    QString url_str = "http://www.example.com/path/to/page.php";

    HttpRequestInput input(url_str, "GET");

    input.add_var("key1", "value1");
    input.add_var("key2", "value2");

    HttpRequestWorker *worker = new HttpRequestWorker(this);
    connect(worker, SIGNAL(on_execution_finished(HttpRequestWorker*)), this, SLOT(handle_result(HttpRequestWork
    worker->execute(&input);
}
```

This will produce the following HTTP request:

```
GET /path/to/page.php?key1=value1&key2=value2 HTTP/1.1
User-Agent: Agent name goes here
Connection: Keep-Alive
Accept-Encoding: gzip, deflate
Accept-Language: en-US,*
Host: www.example.com
```

URL encoded HTTP POST requests

We can make a slight adjustment and turn the GET request to a URL encoded POST request.

```
void MainWindow::on_pushButton_clicked() {
    QString url_str = "http://www.example.com/path/to/page.php";

    HttpRequestInput input(url_str, "POST");

    input.add_var("key1", "value1");
    input.add_var("key2", "value2");

    HttpRequestWorker *worker = new HttpRequestWorker(this);
    connect(worker, SIGNAL(on_execution_finished(HttpRequestWorker*)), this, SLOT(handle_result(HttpRequestWork
    worker->execute(&input);
}
```

This will produce the following HTTP request:

```
POST /path/to/page.php HTTP/1.1
User-Agent: Agent name goes here
```

```
Content-Type: application/x-www-form-urlencoded
Content-Length: 23
Connection: Keep-Alive
Accept-Encoding: gzip, deflate
Accept-Language: en-US,*
Host: www.example.com

key1=value1&key2=value2
```

Multipart HTTP POST requests

Finally, lets push it to the limits. Lets upload some files using a multipart POST request.

```
void MainWindow::on_pushButton_clicked() {
    QString url_str = "http://www.example.com/path/to/page.php";

    HttpRequestInput input(url_str, "POST");

    input.add_var("key1", "value1");
    input.add_var("key2", "value2");

    input.add_file("file1", "/path/to/file1.png", NULL, "image/png");
    input.add_file("file2", "/path/to/file2.png", NULL, "image/png");

    HttpRequestWorker *worker = new HttpRequestWorker(this);
    connect(worker, SIGNAL(on_execution_finished(HttpRequestWorker*)), this, SLOT(handle_result(HttpRequestWork
    worker->execute(&input);
}
```

This will produce the following HTTP request:

```
POST /path/to/page.php HTTP/1.1
User-Agent: Agent name goes here
Content-Type: multipart/form-data; boundary=__-----------------------9446182961397930864818
Content-Length: 686
Connection: Keep-Alive
Accept-Encoding: gzip, deflate
Accept-Language: en-US,*
Host: www.example.com

--__-----------------------9446182961397930864818
Content-Disposition: form-data; name="key1"
Content-Type: text/plain

value1
--__-----------------------9446182961397930864818
Content-Disposition: form-data; name="key2"
Content-Type: text/plain

value2
--__-----------------------9446182961397930864818
Content-Disposition: form-data; name="file1"; filename="file1.png"
Content-Type: image/png
Content-Transfer-Encoding: binary

[... contents of /path/to/file1.png ...]
--__-----------------------9446182961397930864818
Content-Disposition: form-data; name="file2"; filename="file2.png"
Content-Type: image/png
Content-Transfer-Encoding: binary

[... contents of /path/to/file2.png ...]
--__-----------------------9446182961397930864818--
```

Written by Alex Stylianos on 24 July, 2014

18 Comments Creative Pulse Login

18 Comments   Creative Pulse   1 Login

Recommend   Share   Sort by Best

Join the discussion…

yuliyang · a month ago
awesome work! thanks so much
1 · Reply · Share ›

Shailesh Kumar · 2 months ago
Hi Alex, I implemented code for http Rest API for qt client as you mentioned above successfully and it is working fine for GET and POST requests (for simple string to multiple files). Now my task is to convert this unsecured connection with server in to SECURE connection using SSL in qt. I enabled ssl in my system and tested few qt example code and all working fine, but not able to figure out how I can reuse this (yours) code for ssl connection? My interest is to use multipart code... I can't find any example code for sending multiple files for ssl.
1 · Reply · Share ›

> **alt_x** Mod → Shailesh Kumar · a month ago
> Hello Shailesh. If you look into the documentation of QNetworkAccessManager, you will find connectToHostEncrypted() that does want you ask for. This is probably getting somewhere before sendCustomRequest() in the code above, but cannot be sure until I find some time to do it myself.
> Reply · Share ›

Yash pal · 4 months ago
What is License of above code.
1 · Reply · Share ›

> **alt_x** Mod → Yash pal · 4 months ago
> No license. If you want to use it, it would be appreciated if you put credits but it's not required :) I guess that would make it an MIT License.
> Reply · Share ›
>
> > **Yash pal** → alt_x · 4 months ago
> > https://github.com/speedovatio...
> >
> > Thanks :)
> > 2 · Reply · Share ›
> >
> > **Yash pal** → alt_x · 4 months ago
> > ok cool. I will give credit for sure :)
> >
> > Thanks
> > Yash
> > 2 · Reply · Share ›

ixtl · 4 months ago
Many thanks for this article ! Clearly documented, very usefull.

By adding the following code you will add basic authentication capabilities :

1. Update httprequestworker.h with :

```
#include <qauthenticator>
#include <qdebug>
...

private slots:
...
// New slot triggered when authentication is needed
void provideAuthentication(QNetworkReply *reply, QAuthenticator *ator);
```

2. Update httprequestnetworker.cpp

```
// Add this new methode
void HttpRequestWorker::provideAuthentication(QNetworkReply *reply, QAuthenticator *ator)
```

see more

1 · Reply · Share ›

alt_x Mod ixtl · 4 months ago

Thank you Patrick

Reply · Share ›

Med Bm · 5 months ago

Thanks a lot for this Article :) , but i have question, could you please guide me how to change this code a little bit to make it working on Mobile Service of Microsoft Windows Azure , thanks ^^

1 · Reply · Share ›

alt_x Mod Med Bm · 5 months ago

I'm afraid I can't because I haven't tried that myself yet

Reply · Share ›

jurado91 · 5 months ago

Congratulations!!! And thanks for this article, it's very useful

1 · Reply · Share ›

Liberathor · 5 months ago

Hi, nice post!!!!

Do you have a github repository from this example?

Thanks!

1 · Reply · Share ›

alt_x Mod Liberathor · 5 months ago

Thank you Liberathor. No, I am afraid this particular article is not published in our github account. It will be sometime though.

Reply · Share ›

Xeyos · a month ago

Hi,
how can I make request to https?
Currently returns an error: "Protocol "https" is unknown

Currently returns an error: "Protocol "https" is unknown.

Can you help me?

Reply · Share ›

alt_x Mod ➦ Xeyos · a month ago

Hello Xeyos. As I explained to Shailesh Kumar earlier I need to test a new version that supports SSL before I can answer that. It has to do with a code modification that starts the connection with QNetworkAccessManager::connectToHostEncrypted()

Reply · Share ›

Xeyos ➦ alt_x · a month ago

Hello,

I managed to solve my problem using OSX on windows I still have some problems.

They took the additions of 3 lines of code.

Now I wanted to create libraries, but I wanted to know if there was a way to read the response of the worker without using a signal function and a purpose, but directly under the call.

Reply · Share ›

Xeyos ➦ alt_x · a month ago

Thanks a lot,

waiting I'm trying to solve alone, but i have this error.

connectToHostEncrypted is not a member of QNEtworkAccessManager ...

Your code is very useful, I look forward to the version with SSL

Reply · Share ›

ALSO ON **CREATIVE PULSE** WHAT'S THIS?

GMT vs. UTC, a daylight saving story
1 comment • 2 years ago

Why sending email using PHP from localhost fails
2 comments • a year ago

How to reset the admin password in Joomla! 2.5
1 comment • 2 years ago

How to add Google Maps in an Android app in 34592 simple steps
9 comments • a year ago

Subscribe Add Disqus to your site Privacy DISQUS