

Un updater avec Qt : le téléchargement de fichiers

par Thibaut Cuvelier ([Site web](#)) ([Blog](#))

Date de publication : 19/06/2009

Dernière mise à jour : 25/10/2009

Nous allons maintenant rentrer dans le vif du sujet : le téléchargement de fichiers, et leur écriture sur le disque dur. D'abord en théorie : quelles classes peut-on utiliser, comment les utiliser ; puis en pratique, par la création de notre première fenêtre, qui proposera un bouton pour lancer la mise à jour, sans plus.

N'hésitez pas à commenter cet article !

I - QNetworkAccessManager et compagnie.....	3
I-A - Gestion des proxies.....	3
I-B - Requêtes POST et PUT.....	4
I-C - Requêtes HEAD et GET.....	4
I-D - En cas d'erreur.....	4
I-D-1 - QNetworkAccessManager.....	4
I-D-2 - QNetworkRequest.....	5
I-D-3 - QNetworkReply.....	5
II - QFile.....	6
III - Notre premier updater.....	8
III-A - La philosophie.....	8
III-B - L'implémentation.....	8
IV - Divers.....	10

I - QNetworkAccessManager et compagnie

 Les classes qui seront présentées ici ont été introduites avec Qt 4.4.

QNetworkAccessManager est devenue, avec Qt 4.4, la classe de référence pour gérer le réseau : elle contient la configuration générale, les propriétés pour chaque requête, elle gère le proxy et le cache, et peut être utilisée pour l'entièreté des protocoles supportés par Qt (HTTP, HTTPS, FTP...).

Dès qu'un objet **QNetworkAccessManager** a été créé, il peut être utilisé par l'application pour envoyer des requêtes sur le réseau. Une série de fonctions est disponible pour s'occuper de la requête et des données optionnelles.

Chaque requête crée un nouvel objet **QNetworkRequest**. Cet objet contient l'URL de la ressource vers laquelle il pointe. Cette requête ne contient strictement aucune donnée : elle représente les entêtes de chaque requête.

Les données, quant à elles, sont transmises par l'objet **QNetworkReply**.

Voici un petit exemple pour synthétiser toutes ces informations.

```
QNetworkAccessManager * manager = new QNetworkAccessManager();

QUrl url ("http://qt.developpez.com");

QNetworkReply * reply = manager->get(url);
```

manager enverra le signal finished (**QNetworkReply ***) dès que le téléchargement sera terminé.

I-A - Gestion des proxies

Notre manager peut passer à travers un proxy, quand cela lui est demandé par la fonction **setProxy()**. Cette fonction prend en paramètre un objet **QNetworkProxy**.

Voici deux exemples de création de proxy, assez explicites.

```
QNetworkProxy proxy;
proxy.setType(QNetworkProxy::Socks5Proxy);
proxy.setHostName("proxy.exemple.com");
proxy.setPort(1080);
proxy.setUser("username");
proxy.setPassword("password");
```

```
QNetworkProxy proxy (
    QNetworkProxy::Socks5Proxy,
    "proxy.exemple.com",
    1080,
    "username",
    "password"
);
```

L'ensemble des types de proxy possibles est défini par l'énumération **QNetworkProxy::ProxyType** que voici. Les valeurs possibles sont, là encore, très explicites.

```
enum ProxyType { NoProxy, DefaultProxy, Socks5Proxy, HttpProxy,
                HttpCachingProxy, FtpCachingProxy }
```

Pour plus d'informations, reportez-vous à la documentation.

I-B - Requêtes POST et PUT

Notre **QNetworkAccessManager** peut aussi envoyer des données au serveur, par les méthodes POST et PUT du protocole HTTP. La plupart des serveurs n'acceptent pas la méthode PUT, mais elle est quand même proposée, car elle fait partie du standard HTTP/1.1.

Ces requêtes fonctionnent très simplement : avec une requête et les données à envoyer, elles sont prêtes ! Quand les données sont entièrement envoyées, le signal **finished()** est envoyé.

Voici leurs prototypes.

```
QNetworkReply * QNetworkAccessManager::post ( const QNetworkRequest & request,          QIODevice *
data );
QNetworkReply * QNetworkAccessManager::post ( const QNetworkRequest & request, const QByteArray &
data );

QNetworkReply * QNetworkAccessManager::put ( const QNetworkRequest & request,          QIODevice *
data );
QNetworkReply * QNetworkAccessManager::put ( const QNetworkRequest & request, const QByteArray &
data );
```

Un **QByteArray** est un tableau de bits, un équivalent de **QString**, mais en plus généraliste ; un **QIODevice** est un périphérique de lecture et/ou d'écriture (comme, par exemple, un fichier, ou bien un **QNetworkReply**). Nous les étudierons dans la prochaine section

I-C - Requêtes HEAD et GET

Ces requêtes servent à rapatrier des fichiers : HEAD demande simplement l'existence du fichier, GET permet d'en récupérer le contenu.

Voici les prototypes de ces fonctions, qui ne devraient vous poser strictement aucun problème.

```
QNetworkReply * get ( const QNetworkRequest & request )
QNetworkReply * head ( const QNetworkRequest & request )
```

I-D - En cas d'erreur

Parce que tout n'est pas fiable à 100%, il arrive qu'il y ait des erreurs. Ces erreurs sont signalées au moyen de signaux. Les voici rassemblés, classe par classe. J'en profite pour vous montrer les autres signaux qui n'ont pas été précisés plus haut.

I-D-1 - QNetworkAccessManager

Signal	Signification
authenticationRequired (QNetworkReply * reply, QAuthenticator * authenticator)	Une authentification est requise par le serveur. Le slot devrait utiliser reply pour compléter authenticator avec les informations requises.
proxyAuthenticationRequired (const QNetworkProxy & proxy, QAuthenticator * authenticator)	Une authentification est requise par le proxy. Le slot devrait utiliser reply pour compléter authenticator avec les informations requises. Ces informations seront maintenues dans un cache, et ne seront plus demandées.
sslErrors (QNetworkReply * reply, const QList < QSslError > & errors)	Il y a eu une erreur au niveau SSL (certificat non reconnu, par exemple).

I-D-2 - QNetworkRequest

Cette classe n'a pas besoin d'avoir de signaux (il s'agit juste d'un conteneur).

I-D-3 - QNetworkReply

Signal	Signification
downloadProgress (qint64 bytesReceived, qint64 bytesTotal)	Ce signal est envoyé dès qu'il y a des changements. Les paramètres sont évidents.
error (QNetworkReply::NetworkError code)	En cas d'erreur, renvoie le code.
metaDataChanged ()	Signifie un changement dans les métadonnées (entêtes...). Il peut y en avoir même pendant un téléchargement, même si cela est rare.
sslErrors (const QList < QSslError > & errors)	Il y a eu une erreur au niveau SSL (certificat non reconnu, par exemple).
uploadProgress (qint64 bytesSent, qint64 bytesTotal)	Ce signal est envoyé dès qu'il y a des changements. Les paramètres sont évidents.

II - QFile

QFile fait partie de la grande famille des dérivés de **QIODevice**.

Cette classe est une couche d'abstraction pour toutes les classes d'E/S de Qt. Ceci permet d'utiliser cette classe pour désigner toutes les méthodes d'E/S qui coexistent dans Qt : fichiers, buffers, sockets, processus, ou bien **QNetworkReply**.

Un **QFile** représente un fichier en local. Pour en construire un, la marche à suivre est simple. Elle est détaillée dans cet exemple.

```
QFile file("in.txt");

if (! file.open(QIODevice::ReadOnly))
    return;

int i = 1;

while (!file.atEnd())
{
    QByteArray line = file.readLine();
    qDebug() << "Ligne " << i << " : " << line;
    ++i;
}
```

On initialise un **QFile** en passant au constructeur le nom du fichier à ouvrir. Ensuite, on donne le mode d'ouverture. Puis, on traite le fichier.

Voici les différents modes de lecture autorisés. Ils peuvent être facilement associés avec le pipe (|).

- QIODevice::NotOpen
- QIODevice::ReadOnly
- QIODevice::WriteOnly
- QIODevice::ReadWrite
- QIODevice::Append
- QIODevice::Truncate
- QIODevice::Text
- QIODevice::Unbuffered

Voici un nouvel exemple qui montre la combinaison de plusieurs modes.

```
QFile file("in.txt");

if (! file.open(QIODevice::ReadOnly | QIODevice::Text))
    return;

while (!file.atEnd())
{
    QByteArray line = file.readLine();
    qDebug() << "Ligne " << i << " : " << line;
    ++i;
}
```

Pour écrire dans un fichier, rien de plus simple. On peut utiliser un **QByteArray** ou bien un `const char *`.

```
file.write ("Du texte\n");

QByteArray text = new QByteArray ("Du texte");
file.write (text);
```

Les **QByteArray** sont indispensables pour pouvoir utiliser les **QIODevice**. En effet, pour lire des données d'un **QIODevice**, les fonctions fournies retournent des **QByteArray** ou des types de la STL. Même chose pour l'écriture.

Voici les fonctions employées pour la lecture et l'écriture dans des **QIODevice**.

- **qint64 read** (char * data, **qint64** maxSize)
- **QByteArray read** (qint64 maxSize)
- **QByteArray readAll** ()
- **qint64 readLine** (char * data, **qint64** maxSize)
- **QByteArray readLine** (**qint64** maxSize = 0)

- **qint64 write** (const char * data, **qint64** maxSize)
- **qint64 write** (const char * data)
- **qint64 write** (const **QByteArray** & byteArray)

QFile dérive de **QIODevice**, tout comme **QNetworkReply** : en tant que tels, ils ont tous deux une série de fonctions en commun. Par exemple, toutes les méthodes de lecture et d'écriture présentées ci-dessus.


Un petit exemple : comment écrire le contenu d'une réponse dans un fichier ?

```
QFile * file = new QFile ( QString("fichier") );
file->open(QIODevice::ReadOnly);

QNetworkAccessManager manager (this);

QNetworkRequest request ( "http://qt.developpez.com/index.php" );
QNetworkReply * reply = manager.get(request);

file->write( reply->readAll() );
```

 *Dans cet exemple, il est peu probable que des données soient écrites sur le disque dur. En effet, on commence à les écrire avant de savoir si elles sont disponibles. Pour cela, le signal **finished()** est proposé par **QNetworkReply**. La dernière ligne de code ira dans le slot connecté à ce signal.*

L'inverse est aussi possible.

```
QFile * file = new QFile ( QString("fichier") );
file->open(QIODevice::WriteOnly);

QNetworkAccessManager manager (this);

QNetworkRequest request ( "http://qt.developpez.com/index.php" );

QNetworkReply * reply = manager.put(request, file->readAll() );
```

III - Notre premier updater

III-A - La philosophie

Nous allons créer une classe updaterHandler, qui s'occupera de la partie mise à jour.

Elle possèdera le gestionnaire d'accès réseau, les requêtes et les réponses. Elle aura deux slots : un pour commencer le téléchargement, et un pour mettre à jour le fichier en local.

Le premier slot sera appelé par le bouton qui lancera la mise à jour. Le second, par le signal finished de la réponse (pour que notre updater n'essaye pas d'écrire un fichier qui n'est pas encore entièrement téléchargé).

III-B - L'implémentation

Par habitude, nous allons laisser le main assez léger.

```
#include <QtGui/QApplication>
#include <QtGui/QPushButton>

#include <updaterHandler.hpp>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QPushButton bouton("Mettre à jour");

    updaterHandler handler = new updaterHandler ();

    QObject::connect (& bouton , SIGNAL( clicked() ),
                     & handler, SLOT ( launchUpdate() ) );

    bouton.show();

    return a.exec();
}
```

Concernant notre handler : on en crée un, puis on connecte son slot launchUpdate() au signal clicked() du bouton.

Voici la définition de notre classe de handling.

```
class updaterHandler : public QObject
{
    Q_OBJECT

public:
    updaterHandler(QObject * parent = 0);
    ~updaterHandler();

public slots:
    void launchUpdate();
    void writeUpdate();

private:
    QNetworkAccessManager manager;
    QNetworkRequest request;
    QUrl url;
    QFile file;
    QNetworkReply * reply;
};
```


Le constructeur permet de désigner toute instance comme étant enfant d'une autre classe : dès que l'on supprime le parent, l'enfant est détruit. Ensuite viennent les deux slots. Finalement, tous les objets dont nous aurons besoin pour la mise à jour : un gestionnaire d'accès réseau, une requête avec son URL, une réponse et un fichier dans lequel la réponse sera écrite.

```
updaterHandler::updaterHandler(QObject *parent)
    : QObject(parent)
{
    QNetworkAccessManager manager (this);
    QUrl url ("http://qt.developpez.com/");

    QFile * file = new QFile ( QString("index.php"), this );
    file->open(QIODevice::WriteOnly);

    QObject::connect ( reply, SIGNAL ( finished() ),
                      this , SLOT ( writeUpdate() ) );
}
```

Le constructeur instancie notre gestionnaire, l'URL à partir de laquelle nous allons faire la mise à jour du fichier, le fichier cible (que nous ouvrons en écriture seule), et nous connectons la fin du téléchargement de la mise à jour à son application. Remarquez que, à chaque fois, ces objets sont instanciés en tant qu'enfants. Ceci nous permet d'avoir un destructeur très léger.

```
updaterHandler::~updaterHandler() {}
```

```
void updaterHandler::launchUpdate()
{
    QNetworkRequest request (this->url);
    QNetworkReply * reply = manager.get(request);
}
```

Dès que l'utilisateur a cliqué sur le bouton, le téléchargement commence : on crée la requête associée à l'URL, puis on récupère le contenu du fichier dans la réponse.

```
void updaterHandler::writeUpdate()
{
    this->file->write(reply->readAll());
}
```

Dès que le fichier sera entièrement lu, on l'écrira sur le disque.

Comme vous pouvez le voir, créer un updater avec Qt est tout simplement facile.

IV - Divers

Le **Source code source** est disponible.

Un tout grand merci à **yan**, **Alp** et à **superjaja** pour leurs encouragements et conseils lors de la rédaction de cet article !