Nous avons bien avancé sur les différents concepts associés à la mise en œuvre de projets à base de **QML**, appelée projets **Quick**. Nous allons approfondir nos connaissances afin de permettre une communication en réseau au travers des **Websockets** (rien de bien nouveau puisque nous ferons toute la partie communication par les classes C++).

Mise en œuvre d'un système client-serveur au travers de « sockets »

Je vous propose maintenant de voir comment communiquer dans le réseau local avec un service développé en mode **console** et un client avec un projet de type « **quick** ». Le service, très simple ici, consiste à renvoyer un texte en majuscule soumis par le client.

> ***L'objectif est simplement de voir comment mettre en œuvre la communication réseau sans se préoccuper d'un traitement trop sophistiqué. Nous le ferons ultérieurement.***

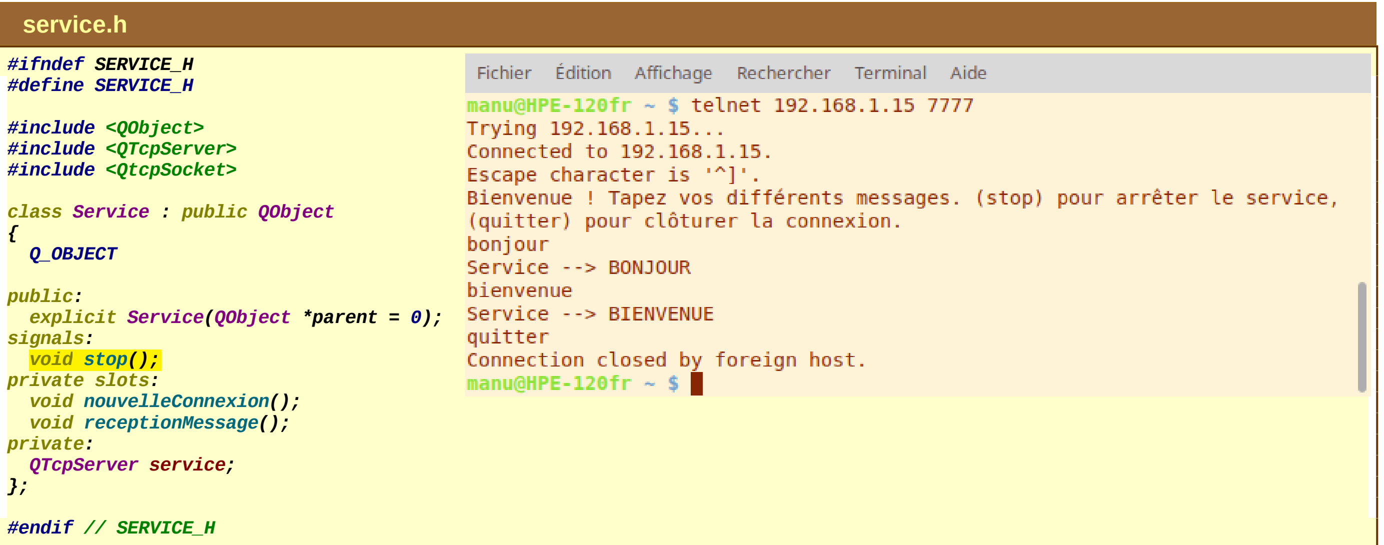
service.pro

```
QT += core network
QT -= gui

TARGET = service
CONFIG += console c++11
CONFIG -= app_bundle
TEMPLATE = app

SOURCES += main.cpp service.cpp
HEADERS += service.h
```

> ***Nous devons prendre une application QT en mode console afin de pouvoir gérer les événements associés au réseau.***

service.h

```
#ifndef SERVICE_H
#define SERVICE_H

#include <QObject>
#include <QTcpServer>
#include <QtcpSocket>

class Service : public QObject
{
  Q_OBJECT

public:
  explicit Service(QObject *parent = 0);
signals:
  void stop();
private slots:
  void nouvelleConnexion();
  void receptionMessage();
private:
  QTcpServer service;
};

#endif // SERVICE_H
```



> ***Nous prévoyons un signal stop() afin de pouvoir arrêter le service définitivement à distance.***

main.cpp

```
#include <QCoreApplication>
#include "service.h"

int main(int argc, char *argv[])
{
  QCoreApplication a(argc, argv);
  Service service;
  Service::connect(&service, SIGNAL(stop()), &a, SLOT(quit()));
  return a.exec();
}
```

> ***Pour quitter l'application qui contient le service, nous avons mis en place une gestion événementielle adaptée.***

service.cpp

```
#include "service.h"

Service::Service(QObject *parent) : QObject(parent)
{
  connect(&service, SIGNAL(newConnection()), this, SLOT(nouvelleConnexion()));
  service.listen(QHostAddress::Any, 7777);
}

void Service::nouvelleConnexion()
{
  QTcpSocket* client = service.nextPendingConnection();
  connect(client, SIGNAL(readyRead()), this, SLOT(receptionMessage()));
  QTextStream soumettre(client);
  soumettre << "Bienvenue !" << endl;
  soumettre << QString("'stop' pour arrêter définitivement le service à distance") << endl;
```

```
   soumettre << QString("'quitter' pour clôturer la connexion") << endl;
   soumettre << QString("Tapez vos différents messages :") << endl;
}

void Service::receptionMessage()
{
   QTcpSocket* client = (QTcpSocket*) sender();
   QTextStream reponse(client);
   QByteArray message = client→readAll().simplified();

   if (message=="stop") stop();
   else if (message=="quitter")
   {
     client->close();
     disconnect(client, SIGNAL(readyRead()), this, SLOT(receptionMessage()));
   }
   else reponse << "Service --> " << message.toUpper() << endl;
}
```

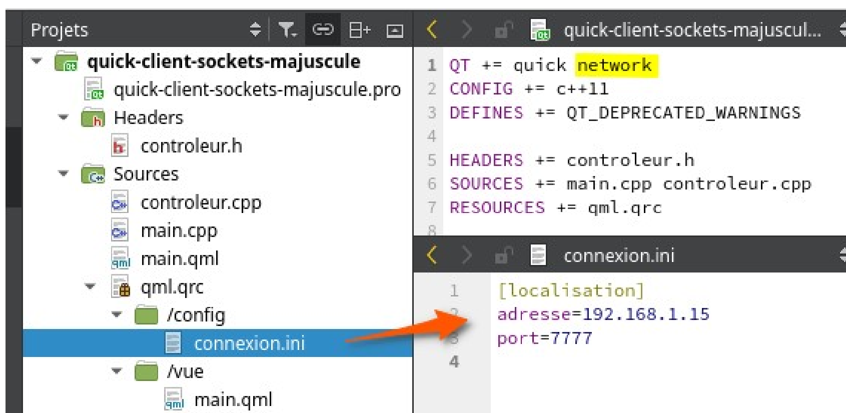
À chaque fois qu'un nouveau client se connecte, la méthode nouvelleConnexion() est automatiquement sollicitée. Le service envoie alors une série de message qui permet au client de savoir quoi faire.

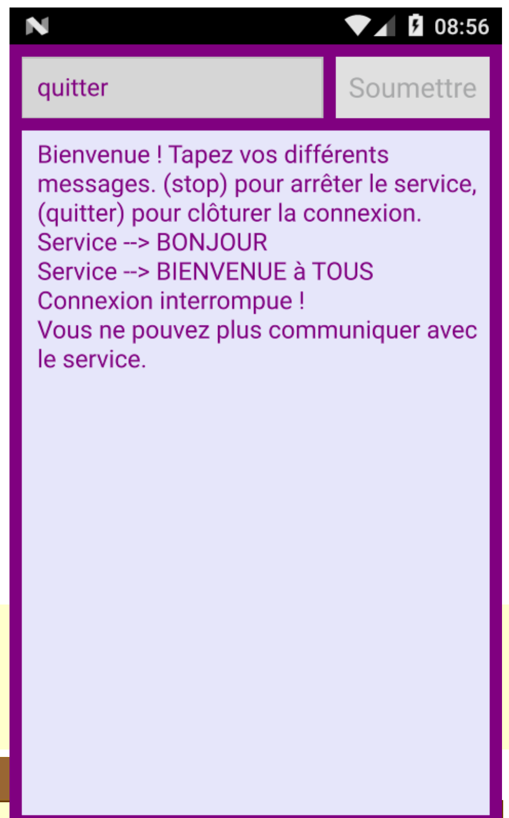
De plus, nous prévoyons une gestion événementielle de telle sorte que lorsque le client envoie un message au service, la méthode receptionMessage() est automatiquement appelée. C'est à l'intérieur de celle-ci que nous prévoyons les différents cas de figure.

Lors de la demande expresse du client de la clôture de la connexion, nous désactivons la gestion événementielle précédemment activée. Cela permet de ne pas prendre trop de ressources lorsque beaucoup de clients se connectent.

*Lorsque nous travaillons avec les sockets, deux types de communications peuvent se faire : soit sous forme de texte, soit avec un flot d'octets non interprétés. Pour le besoin de l'application , nous avons besoin ici de mettre en place un flux de texte, ce qui se fait au travers de la classe **QTextStream**.*

Intéressons-nous maintenant à l'application cliente développée au travers d'un projet de type **Quick**. Nous proposons une vue relativement modeste avec juste trois composants basiques, une zone de saisie, une zone de résultat et un bouton de soumission.





*Notre projet est structuré avec un **contrôleur** et une **vue** séparée, avec également un fichier de configuration qui nous permettra de spécifier l'adresse du serveur et son numéro de service (il faut profiter du fichier de ressources pour cela).*

Lorsque vous faites un développement réseau, pensez à rajouter le module network dans la description du projet.

main.cpp

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <controleur.h>
#include <QtQml>

int main(int argc, char *argv[])
{
   QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
   QGuiApplication app(argc, argv);

   Controleur controleur;
   QQmlApplicationEngine engine;

   engine.rootContext()->setContextProperty("controleur", &controleur);

   engine.load(QUrl(QStringLiteral("qrc:/vue/main.qml")));
   if (engine.rootObjects().isEmpty()) return -1;
   return app.exec();
}
```

Votre programme principal doit prendre en compte le contrôleur. Comme par hasard, nous nommons notre contrôleur « Controleur ». C'est cette classe qui s'occupe de la communication avec le service avec une gestion événementielle adaptée.

controleur.h

```
#ifndef CONTROLEUR_H
#define CONTROLEUR_H
#include <QObject>
#include <QtcpSocket>

class Controleur : public QObject
{
  Q_OBJECT
  Q_PROPERTY(QString message READ getMessage NOTIFY messageChanged)

public:
  explicit Controleur(QObject *parent = nullptr);
  QString getMessage() const { return message; }
private:
  void configuration();

signals:
  void messageChanged();
  void plusDeConnexion();
public slots:
  void soumettre(const QString& texte);
  void reception();
  void cloture();
  void deconnexion();
private:
  QTcpSocket service;
  QString adresse, message;
  QTextStream requete;
  int port;
};

#endif // CONTROLEUR_H
```

La propriété message sera utilisée par la vue pour remplir successivement sa zone de résultat. À chaque nouveau message reçu par le service, le signal messageChanged() sera automatiquement sollicité. Si vous tapez « stop » ou « quitter » dans la zone d'édition, la connexion sera alors interrompue, et le signal plusDeConnexion() sera également activé.

*La socket de communication avec le service s'appelle ici **service** de type **QTcpSocket** et tous les messages que nous soumettrons se feront au travers du flux de texte « **requete** ».*

controleur.cpp

```
#include "controleur.h"
#include <Qsettings>

Controleur::Controleur(QObject *parent) : QObject(parent), requete(&service)
{
  connect(&service, SIGNAL(readyRead()), this, SLOT(reception()));
  connect(&service, SIGNAL(disconnected()), this, SLOT(deconnexion()));
  configuration();
  service.connectToHost(adresse, port);
}

void Controleur::configuration()
{
  QSettings config(":/config/connexion.ini", QSettings::IniFormat);
  adresse = config.value("localisation/adresse").toString();
  port = config.value("localisation/port").toInt();
}

void Controleur::soumettre(const QString &texte)
{
  requete << texte << endl;
}

void Controleur::reception()
{
  message = service.readAll().simplified();
  messageChanged();
}

void Controleur::cloture()
{
  requete << "quitter" << endl;
}

void Controleur::deconnexion()
{
  message = "Connexion interrompue !\n"
            "Vous ne pouvez plus communiquer avec le service.";
  plusDeConnexion();
}
```

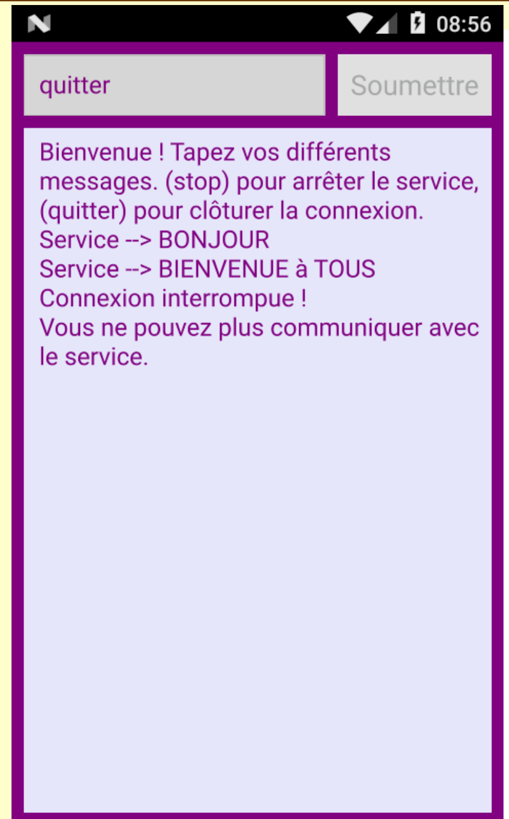
À part le constructeur, l'ensemble des méthodes sont très courtes. Nous aurions pu proposer leurs définitions directement dans le fichier en-tête. La méthode configuration() nous permet de récupérer les informations de connexion avec le service. Pensez bien que le

fichier de configuration est intégré à l'exécutable. Du coup, cela évite de l'oublier dans le déploiement. Par contre, si vous devez changer les réglages, n'oubliez pas de recompiler complètement votre projet.

*Dans le constructeur, il est important de relier le flux de texte « **requete** » au service.*

main.qml

```
import QtQuick 2.9
import QtQuick.Controls 2.2

ApplicationWindow {
  visible: true
  width: 320
  height: 480
  color: "purple"
  title: qsTr("Client majuscule")
  onClosing: controleur.cloture()


  TextField {
    id: envoi
    anchors {
      top: parent.top
      topMargin: 8
      left: parent.left
      leftMargin: 8
      right: bouton.left
      rightMargin: 8
    }
    placeholderText: qsTr("Message à envoyer")
    color: "purple"
    onAccepted: {
      controleur.soumettre(text)
      selectAll()
    }
  }

  Button {
    id: bouton
    anchors {
      top: parent.top
      topMargin: 8
      right: parent.right
      rightMargin: 8
    }
    text: qsTr("Soumettre")
    onClicked: controleur.soumettre(envoi.text)
  }

  ScrollView {
    anchors {
      top: envoi.bottom
      topMargin: 8
      left: parent.left
      leftMargin: 8
      right: parent.right
      rightMargin: 8
      bottom: parent.bottom
      bottomMargin: 8
    }

    TextArea {
      id: réponse
      background: Rectangle { color: "lavender" }
      color: "purple"
      readOnly: true
      wrapMode: "Wrap"
    }
  }

  Connections {
    target: controleur
    onMessageChanged: réponse.append(controleur.message)
    onPlusDeConnexion: {
      réponse.append(controleur.message)
      bouton.enabled = false
      envoi.enabled = false
    }
  }
}
```



Client-serveur avec le protocole « websocket »

Nous allons reprendre le même projet, mais cette fois-ci avec le protocole « **websocket** ». Ce que nous avons réalisé précédemment fonctionne parfaitement bien. Le seul problème, c'est qu'il ne fonctionne qu'en réseau local. Si vous souhaitez communiquer par Internet, vous devez passer par un service de type **WebSocket**.

Le service websocket utilise au moment de la connexion le protocole HTTP sur le port 80 au 8080, ce qui fait que le pare-feu autorise les connexions venant de l'extérieur. Une fois que la connexion est établie, le système demande une migration du protocole HTTP vers le protocole WebSocket.

*Dans le cas du protocole **HTTP**, le client propose une requête, le service donne sa réponse et la connexion est alors interrompue. Dans le cas du protocole **WebSocket**, la connexion reste active constamment, le client ou le service peuvent alors communiquer quand ils le souhaitent, il n'y a pas d'ordre établi (comme pour les sockets classiques). Le fait que le service puissent communiquer quand il le désire permet une communication entre clients, ce que ne peut faire le protocole **HTTP**.*

Nous allons reprendre le service précédant pour utiliser ce protocole WebSocket. Attention au fichier de configuration du projet.

service.pro

```
QT += core websockets
QT -= gui

TARGET = service
CONFIG += console c++11
CONFIG -= app_bundle
TEMPLATE = app

SOURCES += main.cpp service.cpp
HEADERS += service.h
```

service.h

```
#ifndef SERVICE_H
#define SERVICE_H
#include <QObject>
#include <QWebSocketServer>

class Service : public QObject
{
  Q_OBJECT
public:
  explicit Service(QObject *parent = 0);
signals:
  void stop();
private slots:
  void nouvelleConnexion();
  void receptionMessage(const QString& message);
private:
  QWebSocketServer service;
};

#endif // SERVICE_H
```

L'ossature de notre classe demeure identique si ce n'est l'utilisation de la classe QWebSocketServer en lieu et place de QTcpServer.

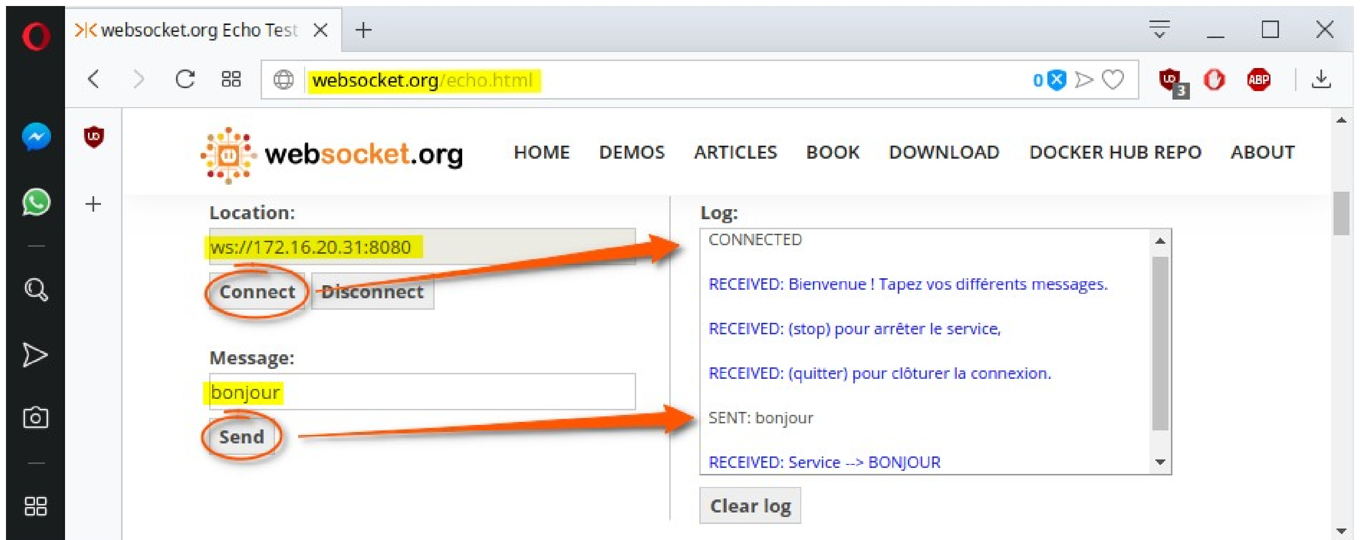
service.cpp

```
#include <QWebSocket>
#include "service.h"

Service::Service(QObject *parent) : QObject(parent),
                                    service("majuscule", QWebSocketServer::NonSecureMode, this)
{
  if (service.listen(QHostAddress::Any, 8080))
    connect(&service, SIGNAL(newConnection()), this, SLOT(nouvelleConnexion()));
  else stop();
}

void Service::nouvelleConnexion()
{
  QWebSocket* client = service.nextPendingConnection();
  connect(client, SIGNAL(textMessageReceived(QString)), this, SLOT(receptionMessage(QString)));
  client->sendTextMessage("Bienvenue ! Tapez vos différents messages. ");
  client->sendTextMessage("(stop) pour arrêter le service, ");
  client->sendTextMessage("(quitter) pour clôturer la connexion.");
}

void Service::receptionMessage(const QString& message)
{
  QWebSocket* client = (QWebSocket*) sender();
  if (message=="stop") stop();
  else if (message=="quitter")
  {
    client->close();
    disconnect(client, SIGNAL(textMessageReceived(QString)), this, SLOT(receptionMessage(QString)));
  }
  else client->sendTextMessage("Service --> "+message.toUpper());
}
```

Deux différences fondamentales apparaissent par rapport aux sockets classiques. Tout d'abord, vous devez initialiser l'objet service de type QWebSocketServer, dans la liste d'initialisation du constructeur. Vous précisez alors un nom logique sous forme de chaîne de caractères, et vous spécifiez ensuite si vous désirez avoir un échange crypté ou pas. Vous lancer alors le service à l'aide de la méthode listen() de cet objet service.

La deuxième particularité, c'est que vous n'avez pas besoin d'implémenter un flux de texte, vous recevez automatiquement votre chaîne de caractères grâce au signal textMessageReceived(QString) (ce qui est plus simple par rapport aux sockets classiques).

Par contre, pour tester votre service, vous devez utiliser un client qui prend en compte ce protocole, comme ci-dessous :



Côté client, nous conservons l'aspect visuel, seul le **contrôleur** est à modifier puisque c'est à ce niveau là que nous avons une modification à apporter, d'où l'utilité de la séparation entre la **vue** et le **contrôleur**.

Connexion.ini

```
[localisation]
adresse=172.16.20.31
port=8080
```

08:56
quitter
Soumettre
Bienvenue ! Tapez vos différents messages. (stop) pour arrêter le service, (quitter) pour clôturer la connexion.
Service --> BONJOUR
Service --> BIENVENUE à TOUS
Connexion interrompue !
Vous ne pouvez plus communiquer avec le service.

controleur.h

```
#ifndef CONTROLEUR_H
#define CONTROLEUR_H

#include <QObject>
#include <QwebSocket>

class Controleur : public QObject
{
  Q_OBJECT
  Q_PROPERTY(QString message READ getMessage NOTIFY messageChanged)
public:
  explicit Controleur(QObject *parent = nullptr);
  QString getMessage() const { return message; }
private:
  void configuration();
signals:
  void messageChanged();
  void plusDeConnexion();
public slots:
  void soumettre(const QString& texte);
  void reception(const QString& texte);
  void cloture();
  void deconnexion();
private:
  QWebSocket service;
  QString url, message;
};

#endif // CONTROLEUR_H
```

La déclaration de la classe du contrôleur est très similaire à la version des sockets classiques, si ce n'est la classe QWebSocket et la méthode reception() qui prend maintenant un paramètre de type QString.

controleur.cpp

```
#include "controleur.h"
#include <Qsettings>

Controleur::Controleur(QObject *parent) : QObject(parent)
{
  connect(&service, SIGNAL(textMessageReceived(QString)), this, SLOT(reception(QString)));
  connect(&service, SIGNAL(disconnected()), this, SLOT(deconnexion()));
  configuration();
  service.open(QUrl(url));
}

void Controleur::configuration()
{
  QSettings config(":/config/connexion.ini", QSettings::IniFormat);
  QString adresse = config.value("localisation/adresse").toString();
```

```
  int port = config.value("localisation/port").toInt();
  url = QString("ws://%1:%2").arg(adresse).arg(port);
}

void Controleur::soumettre(const QString &texte)
{
  service.sendTextMessage(texte);
}

void Controleur::reception(const QString& texte)
{
  message = texte;
  messageChanged();
}

void Controleur::cloture()
{
  service.sendTextMessage("quitter");
}

void Controleur::deconnexion()
{
  message = "Connexion interrompue !\n"
            "Vous ne pouvez plus communiquer avec le service.";
  plusDeConnexion();
}
```

Globalement, l'utilisation de ce protocole WebSocket permet d'éviter d'utiliser les flux de texte, ce qui rend le code encore plus concis. Pour se connecter au service, vous devez soumettre une url avec le schéma suivant : « ws://localisation:port ».

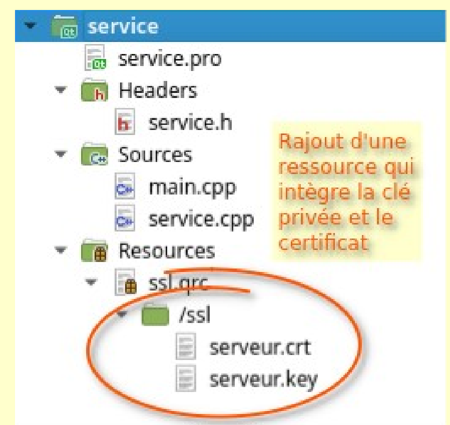
Cryptage de la communication

Lorsque vous prévoyez de communiquer par Internet grâce à ces **WebSockets**, il est préférable de crypter les échanges afin de garder une certaine confidentialité. Vous devez fabriquer les clés privées et les certificats publics tels que nous l'avons déjà expérimenté lors d'une étude précédente.

Nous montrerons ici comment modifier nos différents codes (côté serveur et côté client) et quels sont les configurations nécessaires pour que l'implémentation se déroule correctement. Nous supposons que les clés sont déjà générées.

service.h

```
#ifndef SERVICE_H
#define SERVICE_H

#include <QObject>
#include <QWebSocketServer>

class Service : public QObject
{
  Q_OBJECT
public:
  explicit Service(QObject *parent = 0);
private:
  void configurationSSL();
signals:
  void stop();
private slots:
  void nouvelleConnexion();
  void receptionMessage(const QString& message);
private:
  QWebSocketServer service;
};

#endif // SERVICE_H
```



Une méthode supplémentaire configurationSSL() est rajoutée par rapport au projet précédent.

service.cpp

```
#include <QWebSocket>
#include <QSslKey>
#include <QFile>
#include "service.h"

Service::Service(QObject *parent) : QObject(parent), service("majuscule", QWebSocketServer::SecureMode, this)
{
  configurationSSL();
  if (service.listen(QHostAddress::Any, 8080))
    connect(&service, SIGNAL(newConnection()), this, SLOT(nouvelleConnexion()));
  else stop();
}

void Service::configurationSSL()
{
   QFile certification(":/ssl/serveur.crt");
   QFile clePrivee(":/ssl/serveur.key");
   certification.open(QIODevice::ReadOnly);
   clePrivee.open(QIODevice::ReadOnly);

   QSslConfiguration ssl;
```

```
   ssl.setPeerVerifyMode(QSslSocket::VerifyNone);
   ssl.setLocalCertificate(QSslCertificate(&certification));
   ssl.setPrivateKey(QSslKey(&clePrivee, QSsl::Rsa));
   service.setSslConfiguration(ssl);
}

void Service::nouvelleConnexion()
{
  QWebSocket* client = service.nextPendingConnection();
  connect(client, SIGNAL(textMessageReceived(QString)), this, SLOT(receptionMessage(QString)));
  client->sendTextMessage("Bienvenue ! Tapez vos différents messages. ");
  client->sendTextMessage("(stop) pour arrêter le service, ");
  client->sendTextMessage("(quitter) pour clôturer la connexion.");
}

void Service::receptionMessage(const QString& message)
{
  QWebSocket* client = (QWebSocket*) sender();
  if (message=="stop") stop();
  else if (message=="quitter")
  {
    client->close();
    disconnect(client, SIGNAL(textMessageReceived(QString)), this, SLOT(receptionMessage(QString)));
  }
  else client->sendTextMessage("Service --> "+message.toUpper());
}
```

Nous retrouvons la méthode configurationSSL() qui prend un peu de place. Elle s'occupe de la gestion des clés. Vous remarquez d'ailleurs que la moitié de la méthode consiste à récupérer et ouvrir les fichiers concernés. L'autre moitié de cette méthode utilise des classes spécialisées, une pour la configuration générale QSslConfiguration, une pour la clé privée QSslKey et une pour le certificat QSslCertificat.

***Attention**, pour prendre en compte ces deux clés, vous devez spécifier impérativement le **mode sécurisé** lors de la création de l'objet **service**.*

Côté client, nous allons retrouver le même principe de structure que celui proposé par le service avec ses propres clés (du coup le cryptage de la requête sera différent du cryptage de la réponse se qui rajoute une sécurité supplémentaire).

controleur.h

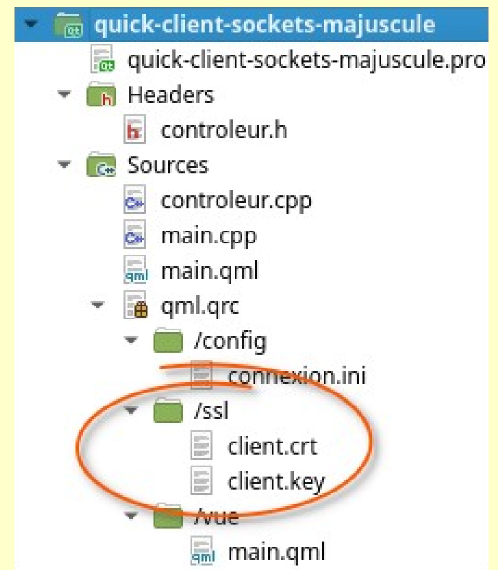
```
#ifndef CONTROLEUR_H
#define CONTROLEUR_H

#include <QObject>
#include <QwebSocket>

class Controleur : public QObject
{
  Q_OBJECT
  Q_PROPERTY(QString message READ getMessage NOTIFY messageChanged)
public:
  explicit Controleur(QObject *parent = nullptr);
  QString getMessage() const { return message; }
private:
  void configuration();
  void configurationSSL();
signals:
  void messageChanged();
  void plusDeConnexion();
public slots:
  void soumettre(const QString& texte);
  void reception(const QString& texte);
  void cloture();
  void deconnexion();
private:
  QWebSocket service;
  QString url, message;
};

#endif // CONTROLEUR_H
```

Comme pour le service, la méthode configurationSSL() est rajoutée au code précédent.

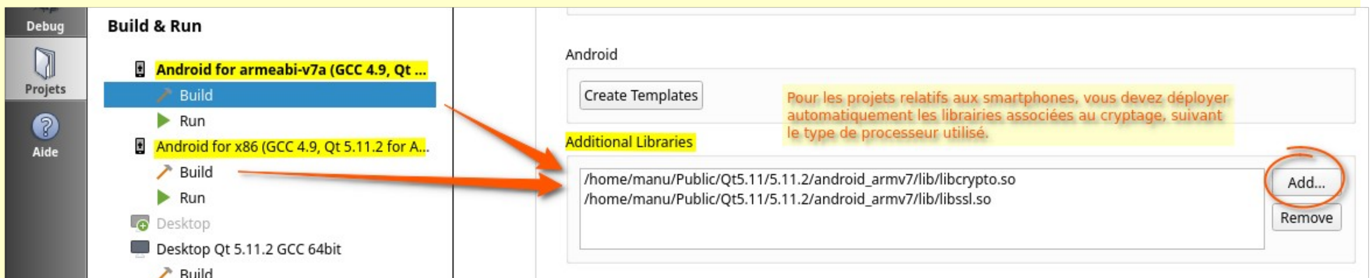
controleur.cpp

```
#include "controleur.h"
#include <QFile>
#include <QSslKey>
#include <Qsettings>

Controleur::Controleur(QObject *parent) : QObject(parent)
{
  connect(&service, SIGNAL(textMessageReceived(QString)), this, SLOT(reception(QString)));
  connect(&service, SIGNAL(disconnected()), this, SLOT(deconnexion()));
  configuration();
  configurationSSL();
  service.open(QUrl(url));
}

void Controleur::configuration()
{
```

```
  QSettings config(":/config/connexion.ini", QSettings::IniFormat);
  QString adresse = config.value("localisation/adresse").toString();
  int port = config.value("localisation/port").toInt();
  url = QString("wss://%1:%2").arg(adresse).arg(port);
}

void Controleur::configurationSSL()
{
   QFile certificat(":/ssl/client.crt");
   QFile clePrivee(":/ssl/client.key");
   certificat.open(QIODevice::ReadOnly);
   clePrivee.open(QIODevice::ReadOnly);

   QSslConfiguration ssl;
   ssl.setPeerVerifyMode(QSslSocket::VerifyNone);
   ssl.setLocalCertificate(QSslCertificate(&certificat));
   ssl.setPrivateKey(QSslKey(&clePrivee, QSsl::Rsa));
   service.setSslConfiguration(ssl);
}

void Controleur::soumettre(const QString &texte)
{
  service.sendTextMessage(texte);
}

void Controleur::reception(const QString& texte)
{
  message = texte;
  messageChanged();
}

void Controleur::cloture()
{
  service.sendTextMessage("quitter");
}

void Controleur::deconnexion()
{
  message = "Connexion interrompue !\n"
            "Vous ne pouvez plus communiquer avec le service.";
  plusDeConnexion();
}
```



Si ce n'est le nom des clés qui changent, nous retrouvons exactement le même code que nous avons écrit au niveau du serveur. Attention, le protocole doit suivre le mode sécurisé et du coup l'url de connexion doit commencer par « wss:// » (websocket sécurisée) et non plus « ws:// » (websocket).



Pour les projets clients relatifs aux smartphones Android, il faut que la librairie correspondant à OpenSSL puissent être déployée automatiquement avec le projet. Pour cela, vous devez récupérer les fichiers binaires correspondant au smartphone réel et au smartphone émulé. Ces fichiers sont disponibles sur Internet. Deux sont nécessaires « libcrypto.so » et « libssl.so ». Vous allez ensuite dans la rubrique « Projets » de QtCreator et plus précisément dans la zone « Build Android APK ».

Service de communication instantané – chat

Je vous propose de réaliser un service « **chat** » non crypté qui permet d'avoir une communication instantanée entre différents clients connectés. Nous prenons le protocole « **websocket** » comme protocole de communication (communication par Internet et facile à mettre en œuvre). Le serveur doit connaître en temps réels les différents clients déjà connectés et avertir tout le monde dès qu'un nouveau client se joint à la discussion.

service.h

```
#ifndef SERVICE_H
#define SERVICE_H

#include <QObject>
#include <QWebSocketServer>
#include <map>
using namespace std;

class Service : public QObject
{
  Q_OBJECT
public:
  explicit Service(QObject *parent = nullptr);
```

```
signals:
  void stop();
private:
  void envoyerATous(const QString &message);
  void listeDesConnectes();
private slots:
  void nouvelleConnexion();
  void receptionMessage(const QString& texte);
  void deconnexion();
private:
  QWebSocketServer service;
  map<QWebSocket*, QString> connectes;
};
#endif // SERVICE_H
```

Nous remarquons la présence de deux méthodes envoyerATous() et listeDesConnectes() qui permettent de dialoguer avec l'ensemble des clients déjà présents. C'est l'objet connectes qui enregistre tous les clients présents sur ce service et avec lesquels nous pouvons communiquer.

service.cpp

```
#include <QWebSocket>
#include "service.h"

Service::Service(QObject *parent) : QObject(parent), service("chat", QWebSocketServer::NonSecureMode, this)
{
  if (service.listen(QHostAddress::Any, 8080))
    connect(&service, SIGNAL(newConnection()), this, SLOT(nouvelleConnexion()));
  else stop();
}

void Service::nouvelleConnexion()
{
  QWebSocket *client = service.nextPendingConnection();
  QString nom = client->requestUrl().path();
  nom.remove('/');
  client->sendTextMessage("service@Bonjour "+nom);
  envoyerATous("service@"+ nom +" vient de se connecter");
  connectes[client] = nom;
  listeDesConnectes();
  connect(client, SIGNAL(textMessageReceived(QString)), this, SLOT(receptionMessage(QString)));
  connect(client, SIGNAL(disconnected()), this, SLOT(deconnexion()));
}

void Service::deconnexion()
{
  QWebSocket *client = (QWebSocket *) sender();
  client->deleteLater();
  QString nom = connectes[client];
  connectes.erase(client);
  envoyerATous("service@"+ nom + " vient de se déconnecter");
  listeDesConnectes();
  disconnect(client, SIGNAL(textMessageReceived(QString)), this, SLOT(receptionMessage(QString)));
  disconnect(client, SIGNAL(disconnected()), this, SLOT(deconnexion()));
}

void Service::receptionMessage(const QString& texte)
{
  if (texte=="stop") { stop(); return; }
  QWebSocket *client = (QWebSocket *) sender();
  QString expediteur = connectes[client];
  QStringList protocole = texte.split('@');
  QString destinataire = protocole[0];
  QString message = protocole[1];
  bool trouve = false;
  for (auto &cible : connectes)
    if (cible.second == destinataire) {
      cible.first->sendTextMessage(expediteur+"@"+message);
      trouve = true; break;
    }
  if (!trouve) client->sendTextMessage("service@Cible inconnue");
}

void Service::envoyerATous(const QString &message)
{
  for (auto &client : connectes) client.first->sendTextMessage(message);
}

void Service::listeDesConnectes()
{
  QString liste = "[";
  for (auto &client : connectes) liste.append(" "+client.second);
  liste.append("]");
  envoyerATous("Connectés@"+liste);
}
```

Côté client, nous conservons l'aspect visuel général du projet précédent. La seule différence concerne le bouton de soumission qui est remplacé pour l'instant par un champ représentant le destinataire du message. Nous devons d'ailleurs, dans le fichier de configuration prévoir un champ supplémentaire qui représente le login, c'est-à-dire l'expéditeur du

message (le titulaire du smartphone). Pour chaque application cliente, vous devrez régler ce login pour qu'il corresponde bien au bon interlocuteur.

connexion.ini

```
[localisation]
adresse=192.168.1.15
port=8080
login=alice
```

controleur.h

```
#ifndef CONTROLEUR_H
#define CONTROLEUR_H

#include <QObject>
#include <QwebSocket>

class Controleur : public QObject
{
  Q_OBJECT
  Q_PROPERTY(QString message READ getMessage NOTIFY messageChanged)
public:
  explicit Controleur(QObject *parent = nullptr);
  QString getMessage() const { return message; }
private:
  void configuration();
signals:
  void messageChanged();
  void plusDeConnexion();
public slots:
  void soumettre(const QString& texte);
  void reception(const QString& texte);
private:
  QWebSocket service;
  QString url, message, login;
};

#endif // CONTROLEUR_H
```

controleur.cpp

```
#include "controleur.h"
#include <QSettings>

Controleur::Controleur(QObject *parent) : QObject(parent)
{
  connect(&service, SIGNAL(textMessageReceived(QString)), this, SLOT(reception(QString)));
  configuration();
  service.open(QUrl(url));
}

void Controleur::configuration()
{
  QSettings config(":/config/connexion.ini", QSettings::IniFormat);
  QString adresse = config.value("localisation/adresse").toString();
  int port = config.value("localisation/port").toInt();
  QString login = config.value("localisation/login").toString();
  url = QString("ws://%1:%2/%3").arg(adresse).arg(port).arg(login);
}

void Controleur::soumettre(const QString &texte)
{
  service.sendTextMessage(texte);
}

void Controleur::reception(const QString& texte)
{
  message = texte;
  messageChanged();
}
```

main.qml

```
import QtQuick 2.9
import QtQuick.Controls 2.2

ApplicationWindow {
  visible: true
  width: 320
  height: 480
  color: "purple"
  title: qsTr("Client chat")

  TextField {
    id: message
    anchors {
      top: parent.top
      topMargin: 8
      left: parent.left
      leftMargin: 8
```

```
        right: destinataire.left
        rightMargin: 8
      }
      placeholderText: qsTr("Message à envoyer")
      color: "purple"
      onAccepted: {
        controleur.soumettre(destinataire.text+'@'+text)
        selectAll()
      }
    }

    TextField {
      id: destinataire
      anchors {
        top: parent.top
        topMargin: 8
        right: parent.right
        rightMargin: 8
      }
      placeholderText: qsTr("Destinataire")
      width: 120
    }

    ScrollView {
      anchors {
        top: message.bottom
        topMargin: 8
        left: parent.left
        leftMargin: 8
        right: parent.right
        rightMargin: 8
        bottom: parent.bottom
        bottomMargin: 8
      }

      TextArea {
        id: réponse
        background: Rectangle { color: "lavender" }
        color: "purple"
        readOnly: true
        wrapMode: "Wrap"
      }
    }

    Connections {
      target: controleur
      onMessageChanged: réponse.append(controleur.message)
    }
}
```

Messagerie instantanée – chat – nouvelle présentation

Je vous propose de modifier le client précédent pour qu'il soit plus agréable à utiliser et plus en conformité avec ce qu'il se fait habituellement, notamment pour choisir le destinataire au travers de la liste de tous les connectés sous forme de « **ComboBox** ». Par ailleurs, chaque message sera séparé sous forme de bulle avec le nom de l'interlocuteur.

Dans un premier temps, nous devons modifier le protocole d'échange pour la liste des connectés, afin que cela soit plus facile à discriminer côté client. Il suffit alors de prévoir un séparateur (@) entre chaque connecté.

service.cpp

```
#include <QWebSocket>
#include "service.h"

Service::Service(QObject *parent) : QObject(parent), service("chat", QWebSocketServer::NonSecureMode, this)
{
  if (service.listen(QHostAddress::Any, 8080))
    connect(&service, SIGNAL(newConnection()), this, SLOT(nouvelleConnexion()));
  else stop();
}

void Service::nouvelleConnexion()
{
  QWebSocket *client = service.nextPendingConnection();
  QString nom = client->requestUrl().path();
  nom.remove('/');
  client->sendTextMessage("service@Bonjour "+nom);
  envoyerATous("service@"+ nom +" vient de se connecter");
  connectes[client] = nom;
  listeDesConnectes();
  connect(client, SIGNAL(textMessageReceived(QString)), this, SLOT(receptionMessage(QString)));
  connect(client, SIGNAL(disconnected()), this, SLOT(deconnexion()));
}

void Service::deconnexion()
{
  QWebSocket *client = (QWebSocket *) sender();
  client->deleteLater();
  QString nom = connectes[client];
  connectes.erase(client);
```

```
   envoyerATous("service@"+ nom + " vient de se déconnecter");
   listeDesConnectes();
   disconnect(client, SIGNAL(textMessageReceived(QString)), this, SLOT(receptionMessage(QString)));
   disconnect(client, SIGNAL(disconnected()), this, SLOT(deconnexion()));
}

void Service::receptionMessage(const QString& texte)
{
   if (texte=="stop") { stop(); return; }
   QWebSocket *client = (QWebSocket *) sender();
   QString expediteur = connectes[client];
   QStringList protocole = texte.split('@');
   QString destinataire = protocole[0];
   QString message = protocole[1];
   bool trouve = false;
   for (auto &cible : connectes)
     if (cible.second == destinataire) {
       cible.first->sendTextMessage(expediteur+"@"+message);
       trouve = true; break;
     }
   if (!trouve) client->sendTextMessage("service@Cible inconnue");
}

void Service::envoyerATous(const QString &message)
{
   for (auto &client : connectes) client.first->sendTextMessage(message);
}

void Service::listeDesConnectes()
{
   QString liste;
   for (auto &client : connectes)
     liste.append("@"+client.second);
   envoyerATous("Connectés"+liste);
}
```

Bien entendu, côté client, le contrôleur doit également être modifié pour s'ajuster aux nouvelles contraintes et au petit changement du protocole. De nouvelles propriétés apparaissent pour soumettre à la vue la liste des connectés et aussi quel est l'expéditeur du dernier message reçu.

controleur.h

```
#ifndef CONTROLEUR_H
#define CONTROLEUR_H

#include <QObject>
#include <QwebSocket>

class Controleur : public QObject
{
   Q_OBJECT
   Q_PROPERTY(QString message READ getMessage NOTIFY messageChanged)
   Q_PROPERTY(QStringList connectes READ getconnectes NOTIFY connectesChanged)
   Q_PROPERTY(QString qui READ getExpediteur)
public:
   explicit Controleur(QObject *parent = nullptr);
   QString getMessage() const { return message; }
   QStringList getconnectes() { return connectes; }
   QString getExpediteur()    { return expediteur; }
private:
   void configuration();
signals:
   void messageChanged();
   void plusDeConnexion();
   void connectesChanged();
public slots:
   void soumettre(const QString& texte);
   void reception(const QString& texte);
private:
   QWebSocket service;
   QString url, message, login, expediteur;
   QStringList connectes;
};

#endif // CONTROLEUR_H
```

controleur.cpp

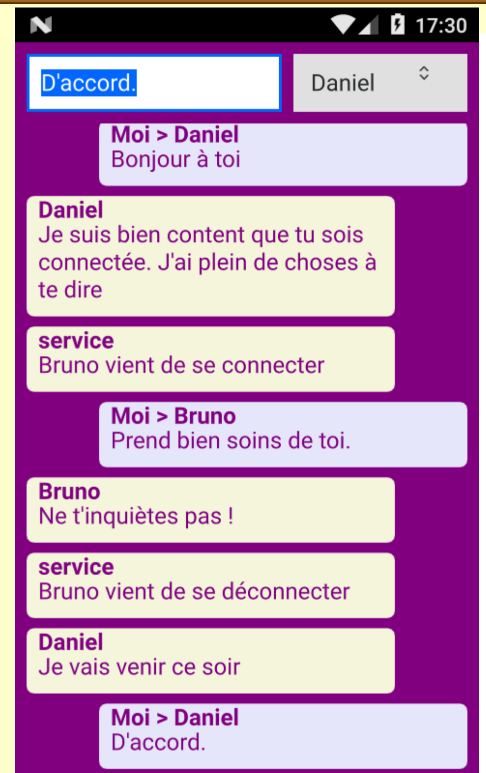
```
#include "controleur.h"
#include <Qsettings>

Controleur::Controleur(QObject *parent) : QObject(parent)
{
   connect(&service, SIGNAL(textMessageReceived(QString)), this, SLOT(reception(QString)));
   configuration();
   service.open(QUrl(url));
}

void Controleur::configuration()
{
   QSettings config(":/config/connexion.ini", QSettings::IniFormat);
```

```
  QString adresse = config.value("localisation/adresse").toString();
  int port = config.value("localisation/port").toInt();
  QString login = config.value("localisation/login").toString();
  url = QString("ws://%1:%2/%3").arg(adresse).arg(port).arg(login);
}

void Controleur::soumettre(const QString &texte)
{
  service.sendTextMessage(texte);
}

void Controleur::reception(const QString& texte)
{
  QStringList protocole = texte.split('@');
  if (protocole[0]=="Connectés")
  {
    protocole.removeAt(0);
    connectes = protocole;
    connectesChanged();
  }
  else {
    expediteur = protocole[0];
    message = protocole[1];
    messageChanged();
  }
}
```

Le dernier point concerne la vue et c'est là que nous avons le plus de changement. Il est possible de créer des listes personnalisés qui qui de développent dynamiquement, ici notamment pour garder les derniers messages envoyés ou reçus. Cela se fait tout simplement au travers du composant « ListModel ». Il suffit d'utiliser la méthode « append() » avec la liste des paramètres à inclure.

main.qml

```
import QtQuick 2.9
import QtQuick.Controls 2.2

ApplicationWindow {
  property int largeurMessage: 50
  visible: true
  width: 320
  height: 480
  color: "purple"
  title: qsTr("Messagerie instantanée")

  ListModel { id: messages }

  ScrollView {
    anchors {
      top: message.bottom
      topMargin: 8
      left: parent.left
      leftMargin: 8
      right: parent.right
      rightMargin: 8
      bottom: parent.bottom
      bottomMargin: 8
    }

    Flickable {
      contentWidth: parent.width
      contentHeight: parent.height

      ListView {
        id: listeMessages
        anchors.fill: parent
        spacing: 7
        model: messages
        delegate: Rectangle {
          width: parent.width - largeurMessage
          height: qui.height + contenu.height + 7
          radius: 5
          color: couleur
          x: position

          Text {
            id: qui
            x:8
            text: expéditeur
            font.bold: true
            color: "purple"
          }

          Text {
            id: contenu
            x: 8
            y: 17
            text: texte
            color: "purple"
            wrapMode: Text.Wrap
            width: parent.width-16
          }
```



```
          }
        }
      }
    }

    Rectangle {
      color: "purple"
      width: parent.width
      height: message.height+16
    }

    TextField {
      id: message
      anchors {
        top: parent.top
        topMargin: 8
        left: parent.left
        leftMargin: 8
        right: destinataire.left
        rightMargin: 8
      }
      placeholderText: qsTr("Message à envoyer")
      color: "purple"
      onAccepted: {
        controleur.soumettre(destinataire.displayText+'@'+text)
        selectAll()
        messages.append({texte: text, expéditeur: "Moi > "+destinataire.displayText,
                         couleur: "lavender", position: largeurMessage})
        listeMessages.positionViewAtEnd()
      }
    }

    ComboBox {
      id: destinataire
      anchors {
        top: parent.top
        topMargin: 8
        right: parent.right
        rightMargin: 8
      }
      model: controleur.connectes
      width: 120
    }

    Connections {
      target: controleur
      onMessageChanged: {
        messages.append({texte: controleur.message, expéditeur: controleur.qui, couleur: "beige", position: 0})
        listeMessages.positionViewAtEnd()
      }
    }
}
```

Le composant « ListView » permet de prendre en compte le modèle personnalisé (avec la propriété « model ») et d'afficher ainsi, les différents messages qui transitent les uns au dessus des autres avec un petit espace entre eux.

Dans la propriété « delegate », vous devez prévoir comment visualiser chaque élément de la liste. Une fois que tous ces réglages sont réalisés, les messages s'affichent au fur et à mesure automatiquement.

Insertion d'une StackView afin de pouvoir configurer la connexion

Toujours sur ce projet passionnant, je vous propose de le compléter avec une vue partielle supplémentaire qui va servir à régler les paramètres de configuration de la connexion, notamment pour choisir le nom de login depuis votre smartphone. Il sera ainsi possible de couper la communication quand nous le désirons. Une autre fonctionnalité est d'effacer complètement l'ensemble des messages qui risquent d'encombrer la vue principale.

Une fois que les paramètres de configuration sont établis, il est possible de faire en sorte que la communication avec le service se fasse automatiquement dès le lancement de l'application. C'est au choix de l'utilisateur.

Le fichier de configuration correspondant « connexion.ini » ne doit plus faire partie de la ressource, mais doit être un fichier séparé qui sera automatiquement constitué au premier démarrage de l'application.

Nous profitons de l'occasion pour que la vue soit découpée an plusieurs fichiers qui vont répondre aux différents critères spécifiques et permettre ainsi de s'y retrouver plus facilement lors de l'évolution de l'application.

connexion.ini

```
[localisation]
adresse=192.168.1.15
port=8080
login=Martine
automatique=true
```

main.cpp

```cpp
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <controleur.h>
#include <QtQml>

int main(int argc, char *argv[])
{
  QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
  QGuiApplication app(argc, argv);

  Controleur controleur;
  QQmlApplicationEngine engine;

  engine.rootContext()->setContextProperty("controleur", &controleur);

  engine.load(QUrl(QStringLiteral("qrc:/vues/main.qml")));
  if (engine.rootObjects().isEmpty()) return -1;
  return app.exec();
}
```

Projets
- quick-client-websockets-chat-stack
  - quick-client-websockets-chat-stack.pro
  - Headers
    - controleur.h
  - Sources
    - Configuration.qml
    - controleur.cpp
    - EnTete.qml
    - main.cpp
    - main.qml
    - PagePrincipale.qml
    - qml.qrc
      - /icônes
        - checklist.png
        - contacts.png
        - liste-contact.png
        - tâches.png
        - Vista.png
      - /vues
        - Configuration.qml
        - EnTete.qml
        - main.qml
        - PagePrincipale.qml

Le contrôleur doit être revu afin de rendre accessible les paramètres de configuration avec des propriétés adaptées. Voici le code correspondant.

controleur.h

```cpp
#ifndef CONTROLEUR_H
#define CONTROLEUR_H

#include <QObject>
#include <QwebSocket>

class Controleur : public QObject
{
  Q_OBJECT
  Q_PROPERTY(QString message READ getMessage NOTIFY messageChanged)
  Q_PROPERTY(QStringList connectes READ getconnectes NOTIFY connectesChanged)
  Q_PROPERTY(QString qui READ getExpediteur)
  Q_PROPERTY(QString adresse MEMBER adresse NOTIFY adresseChanged)
  Q_PROPERTY(int port MEMBER port NOTIFY portChanged)
  Q_PROPERTY(QString login MEMBER login NOTIFY loginChanged)
  Q_PROPERTY(bool automatique MEMBER automatique NOTIFY autoChanged)
  Q_PROPERTY(bool enService MEMBER enService)
public:
  explicit Controleur(QObject *parent = nullptr);
  QString getMessage() const        { return message; }
  QStringList getconnectes() const  { return connectes; }
  QString getExpediteur() const     { return expediteur; }
private:
  void configuration();
signals:
  void messageChanged();
  void plusDeConnexion();
  void connectesChanged();
  void adresseChanged();
  void portChanged();
  void loginChanged();
  void autoChanged();
public slots:
  void soumettre(const QString& texte);
  void reception(const QString& texte);
  void seConnecter();
  void seDeconnecter();
private:
  QWebSocket service;
  QString url, adresse, message, login, expediteur;
  int port=8080;
  QStringList connectes;
  bool automatique, enService=false;
};

#endif // CONTROLEUR_H
```

controleur.cpp

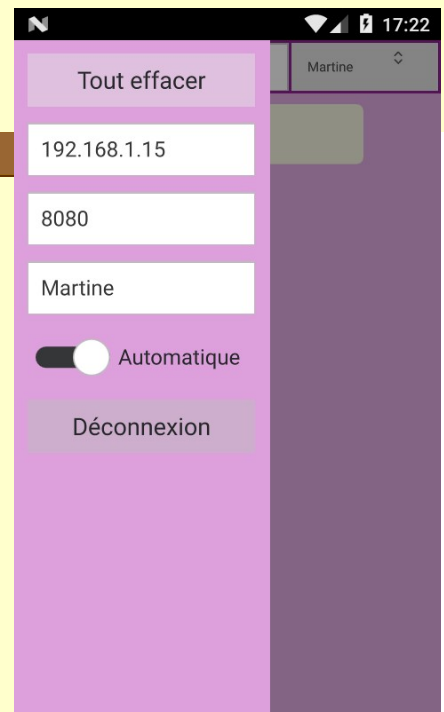
```cpp
#include "controleur.h"
#include <QFile>
#include <QSslKey>
#include <Qsettings>

Controleur::Controleur(QObject *parent) : QObject(parent)
{
  connect(&service, SIGNAL(textMessageReceived(QString)), this, SLOT(reception(QString)));
  configuration();
  if (automatique) {
    service.open(QUrl(url));
    enService=true;
  }
}
```

```
void Controleur::configuration()
{
  QSettings config("connexion.ini", QSettings::IniFormat);
  adresse = config.value("localisation/adresse").toString();
  adresseChanged();
  port = config.value("localisation/port", 8080).toInt();
  portChanged();
  login = config.value("localisation/login").toString();
  loginChanged();
  automatique = config.value("localisation/automatique").toBool();
  autoChanged();
  url = QString("ws://%1:%2/%3").arg(adresse).arg(port).arg(login);
}

void Controleur::seConnecter()
{
  QSettings config("connexion.ini", QSettings::IniFormat);
  config.setValue("localisation/adresse", adresse);
  config.setValue("localisation/port", port);
  config.setValue("localisation/login", login);
  config.setValue("localisation/automatique", automatique);
  url = QString("ws://%1:%2/%3").arg(adresse).arg(port).arg(login);
  service.open(QUrl(url));
}

void Controleur::seDeconnecter()
{
  service.close();
  connectes.clear();
  connectesChanged();
}

void Controleur::soumettre(const QString &texte)
{
  service.sendTextMessage(texte);
}

void Controleur::reception(const QString& texte)
{
  QStringList protocole = texte.split('@');
  if (protocole[0]=="Connectés")
  {
    protocole.removeAt(0);
    connectes = protocole;
    connectesChanged();
  }
  else {
    expediteur = protocole[0];
    message = protocole[1];
    messageChanged();
  }
}
```

Le dernier point concerne les vues constituant l'aspect visuel de notre application. Comme pour le projet de type « Swipe » que nous avons élaboré dans une étude précédente, un projet de type « StackView » utilise la classe principale « ApplicationWindow » en lieu et place de « Window ».

Cette nouvelle classe propose une ossature qui permet d'intégrer une en-tête ou un pied de page, respectivement « header » et « footer ». Ensuite, cette application principale permet de gérer un ensemble de pages, soit les unes à côté des autres comme avec « Swipe », soit les unes au-dessus des autres, comme avec « StackView ».

Dans ce dernier cas, il est possible d'avoir une page plus réduite qui apparaît sur le côté afin de contrôler différentes options, ceci au travers de classe spécifique « Drawer ».



main.qml

```
import QtQuick 2.9
import QtQuick.Controls 2.2

ApplicationWindow {
  id: fenêtre
  visible: true
  width: 320
  height: 480
  title: qsTr("Messagerie instantanée")
  property int largeurMessage: 50

  signal deconnexion()

  onDeconnexion:
    messages.append({texte: "Vous êtes déconnecté du service",
                     expéditeur: "Système", couleur: "pink",
                     position: largeurMessage})

  header: EnTete {}

  Configuration {
    id: configuration
  }
```

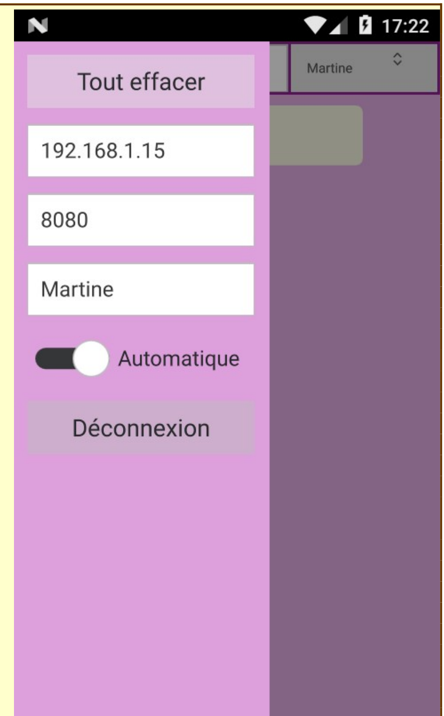
```
  ListModel { id: messages }

  Connections {
    target: controleur
    onMessageChanged: {
      messages.append({texte: controleur.message,
                       expéditeur: controleur.qui,
                       couleur: "beige",
                       position: 0})
      page.pageEnBas()
    }
  }

  StackView {
    id: pile
    anchors.fill: parent
    initialItem: PagePrincipale { id: page }
  }
}
```

EnTete.qml

```
import QtQuick 2.0
import QtQuick.Controls 2.4

ToolBar {
  contentHeight: menu.implicitHeight
  background: Rectangle { color: "purple" }
  ToolButton {
    id: menu
    icon {
      color: "transparent"
      source: "qrc:/icônes/checklist.png"
    }
    font.pixelSize: Qt.application.font.pixelSize * 1.6
    onClicked: configuration.open()
  }

  TextField {
    id: message
    anchors {
      top: parent.top
      topMargin: 2
      left: menu.right
      leftMargin: 2
      right: destinataire.left
      rightMargin: 2
      bottom: menu.bottom
      bottomMargin: 2
    }
    placeholderText: qsTr("Message à envoyer")
    color: "purple"
    onAccepted: {
      controleur.soumettre(destinataire.displayText+'@'+text)
      messages.append({texte: text,
                       expéditeur: "Moi > "+destinataire.displayText,
                       couleur: "lavender",
                       position: largeurMessage})
      text=""
      page.pageEnBas()
    }
  }

  ComboBox {
    id: destinataire
    font.pixelSize: 10
    anchors {
      top: parent.top
      topMargin: 2
      right: parent.right
      rightMargin: 2
      bottom: menu.bottom
      bottomMargin: 2
    }
    model: controleur.connectes
    width: 110
  }
}
```

Configuration.qml

```
import QtQuick 2.0
import QtQuick.Controls 2.4

Drawer {
  width: fenêtre.width * 0.6
  height: fenêtre.height
  background: Rectangle { color: "plum" }

  Column {
    anchors.fill: parent
    anchors.margins: 10
    spacing: 12
```




```
    ToolButton {
      width: parent.width
      text: "Tout effacer"
      onClicked: {
        messages.clear()
        configuration.close()
      }
    }

    TextField {
      id: adresse
      width: parent.width
      placeholderText: "@IP"
      text: controleur.adresse
      inputMethodHints: Qt.ImhDigitsOnly
    }

    TextField {
      id: port
      width: parent.width
      placeholderText: "n° Port"
      text: controleur.port
      inputMethodHints: Qt.ImhDigitsOnly
    }

    TextField {
      id: login
      width: parent.width
      placeholderText: "Login"
      text: controleur.login
    }

    Switch {
      id: automatique
      checked: controleur.automatique
      text: "Automatique"
    }

    ToolButton {
      id: connexion
      width: parent.width
      text: "Connexion"
      checkable: true
      Component.onCompleted: checked = controleur.enService
      onCheckedChanged:
        if (checked) {
          controleur.adresse = adresse.text
          controleur.port = port.text
          controleur.login = login.text
          controleur.automatique = automatique.checked
          controleur.seConnecter()
          text = "Déconnexion"
          configuration.close()
        }
        else {
          controleur.seDeconnecter()
          text = "Connexion"
          fenêtre.deconnexion()
          configuration.close()
        }
      }
  }
}
```

PagePrincipale.qml

```
import QtQuick 2.9
import QtQuick.Controls 2.2

Page {
  anchors.fill: parent
  background: Rectangle { color: "plum" }

  signal pageEnBas()
  onPageEnBas: listeMessages.positionViewAtEnd()

  ScrollView {
    anchors {
      top: parent.top
      topMargin: 8
      left: parent.left
      leftMargin: 8
      right: parent.right
      rightMargin: 8
      bottom: parent.bottom
      bottomMargin: 8
    }

    Flickable {
      contentWidth: parent.width
      contentHeight: parent.height
```

```
      ListView {
        id: listeMessages
        anchors.fill: parent
        spacing: 7
        model: messages
        delegate: Rectangle {
          width: parent.width - largeurMessage
          height: qui.height + contenu.height + 7
          radius: 5
          color: couleur
          x: position

          Text {
            id: qui
            x:8
            text: expéditeur
            font.bold: true
            color: "purple"
          }

          Text {
            id: contenu
            x: 8
            y: 17
            text: texte
            color: "purple"
            wrapMode: Text.Wrap
            width: parent.width-16
          }
        }
      }
    }
  }
}
```