Yoann FLEYTOUX
Sylvain LAUGIÉ
Alexandre SCHREIBER
Olivier SIRE DE VILAR
Groupe PC7

Projet tuteuré :
Gestionnaire de mots de passe sécurisé

Dossier de Programmation

Client : Boris VALERA
Tutrice : Nadia YASSINE-DIAB

SOMMAIRE

INTRODUCTION

Ce document constitue le dossier de programmation du projet "Gestionnaire de mots de passe". Il contient l'intégralité du code source du programme, avec les commentaires associés à chaque opération.

Le document est divisé en trois grande parties : code source serveur, code source client, code source commun. Chaque grande partie sera divisée en sous-parties, contenant chaque le code d'un fichier.

Les différents composants du logiciel sont constitués de deux fichiers : un fichier .h et un fichier .cpp. Les fichiers .h contiennent les spécifications des composants : variables et méthodes. Les fichiers .cpp contiennent le corps des méthodes (traitements).

Toutes les méthodes et fonctions (dans les fichiers .cpp) sont précédées de commentaires (en vert, commençant par //) indiquant quelle est l'opération réalisée. Les méthodes et fonctions sont de la forme :

```
// Commentaires concernant la méthode (entrées, sorties, actions)
typeSortie NomComposant::nomMethode(TypeEntree parametre, ...) {
      // Eventuels commentaires
      instruction1;
      instruction2;
      ...
}
```

I. Code source du serveur

1. main.cpp

Ce fichier est le fichier principal du serveur : il permet de charger le composant principal.

```
#include <QCoreApplication>
#include "ComposantPrincipal.h"
#include "ComposantCryptographie.h"

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

    ComposantPrincipal cp(argc, argv);

    return a.exec();
}
```

2. ComposantPrincipal

Ce composant est le composant principal du serveur : il reçoit les requêtes de ComposantReseauConnexion, les analyse et lance les traitements concernés dans ComposantBaseDeDonnees.

a. ComposantPrincipal.h

```
#ifndef COMPOSANTPRINCIPAL_H
#define COMPOSANTPRINCIPAL_H

#include <QtCore>
#include <QObject>

class ComposantBaseDeDonnees;
class ComposantCryptographie;
class ComposantReseau;

class ComposantPrincipal : public QObject
{
    Q_OBJECT

public:
    ComposantPrincipal(int argc, char *argv[]);
    ~ComposantPrincipal();

private:
    ComposantBaseDeDonnees * composantBDD;
    ComposantReseau * composantReseau;

    QString fichierConfiguration;
    QString fichierBaseDeDonnees;
    QString fichierClePriveeRSA;
    QString fichierClePubliqueRSA;
    int portEcoute;

    void lancerServeur();
    void installerServeur(int argc, char *argv[]);
    void sauvegarderServeur(int argc, char *argv[]);
    void restaurerServeur(int argc, char *argv[]);

    QByteArray sauvegardeServeur();

    QByteArray intToByteArray(unsigned int valeur);
    unsigned int byteArrayToInt(const QByteArray ba);
    QByteArray genererMotDePasse(const int longueur);

public slots:
    void traitementMessage(QByteArray message);
};

#endif // COMPOSANTPRINCIPAL_H
```

b. ComposantPrincipal.cpp

```
#include "ComposantPrincipal.h"
#include "ComposantBaseDeDonnees.h"
#include "ComposantCryptographie.h"
#include "ComposantReseau.h"
#include "ComposantReseauConnexion.h"
#include "ComposantDocumentation.h"

#include <QFile>
#include <QDir>
#include <QTextStream>
#include <cmath>
#include <iostream>


// Le constructeur analyse les paramètres de la ligne de commande
ComposantPrincipal::ComposantPrincipal(int argc, char *argv[])
{

    // Fichier de configuration par défaut
    fichierConfiguration = QDir::homePath() + QString("/spmd/spmd.conf");

    if(argc == 1) {
        lancerServeur();
        return;
    }

    // Si des paramètres sont présents, on prépare la sortie standard pour
écriture
    QTextStream out(stdout);

    // Lancement du serveur avec un fichier de configuration spécifique
    if(strcmp(argv[1], "--config") == 0) {

        if(argc != 3) {
            out << "Ligne de commande incorrecte." << endl;
            ComposantDocumentation::aide();
            exit(1);
        }
        if(!QFile::exists(argv[2])) {
            out << "Impossible de lancer le serveur : fichier de
configuration introuvable." << endl;
            exit(1);
        }
        fichierConfiguration = argv[2];
        lancerServeur();
        return;
    }

    // Installation du serveur
    if(strcmp(argv[1], "--install") == 0) {
        installerServeur(argc, argv);
        exit(0);
    }

    // Sauvegarde du serveur
    if(strcmp(argv[1], "--backup") == 0) {
        sauvegarderServeur(argc, argv);
        exit(0);
    }
```

```
    // Restauration d'une sauvegarde du serveur
    if(strcmp(argv[1], "--restore") == 0) {
        restaurerServeur(argc, argv);
        exit(0);
    }

    // Affichage de l'aide
    if(strcmp(argv[1], "--help") == 0) {
        ComposantDocumentation::aide();
        exit(0);
    }

    // Par défaut, affichage de l'aide
    out << endl << "Ligne de commande incorrecte." << endl;
    ComposantDocumentation::aide();
    exit(1);
    return;

}

ComposantPrincipal::~ComposantPrincipal() {
}
```

```
// Lancement normal du serveur
void ComposantPrincipal::lancerServeur() {

    // On prépare la sortie standard pour écriture
    QTextStream out(stdout);

    // Vérification de la présence du fichier de configuration
    QFile fichierConfig(fichierConfiguration);
    if(!fichierConfig.exists()) {
        out << "Impossible de lancer le serveur : le fichier de configuration
est introuvable." << endl;
        exit(1);
    }
    if(!fichierConfig.open(QIODevice::ReadOnly)) {
        out << "Impossible de lancer le serveur : le fichier de configuration
ne peut etre ouvert." << endl;
        exit(1);
    }

    // Chargement de la configuration
    QString fichier(fichierConfig.readAll());
    fichierConfig.close();

    QStringList lignes = fichier.split("\n");
    QStringList ligne;
    for(int l = 0; l < lignes.size(); l++) {
        ligne = lignes.at(l).split("=");
        if(ligne.at(0) == "fichierClePubliqueRSA") fichierClePubliqueRSA =
ligne.at(1).trimmed();
        else if(ligne.at(0) == "fichierClePriveeRSA") fichierClePriveeRSA =
ligne.at(1).trimmed();
        else if(ligne.at(0) == "fichierBaseDeDonnees") fichierBaseDeDonnees =
ligne.at(1).trimmed();
        else if(ligne.at(0) == "portEcoute") portEcoute =
ligne.at(1).toInt();
    }



    // Création composant base de données avec le fichier
fichierBaseDeDonnees
    composantBDD = new ComposantBaseDeDonnees(fichierBaseDeDonnees,
fichierClePriveeRSA, fichierClePubliqueRSA);
    if(!composantBDD->ouvrirBaseDeDonnees()) {
        out << "Impossible de lancer le serveur : la base de donnees ne peut
etre ouverte." << endl;
        exit(1);
    }

    // Création composant réseau avec les fichiers de cles RSA
    composantReseau = new ComposantReseau(portEcoute, fichierClePriveeRSA,
fichierClePubliqueRSA, this);
}
```

```
// Traitement des messages provenant de la connexion réseau
void ComposantPrincipal::traitementMessage(QByteArray message) {

    // On récupère la connexion qui a demandé le traitement du message
    ComposantReseauConnexion * connexion = (ComposantReseauConnexion
*)(QObject::sender());

    // Si l'utilisateur n'est pas identifié, on tente une identification
    if(connexion->getIdUtilisateur() == -1) {
        QList<QByteArray> champs = message.split(31);
        if(champs.at(0) == "connexion") {
            int id;
            QString nom;
            bool admin;
            composantBDD->connecterUtilisateur(champs.at(1), champs.at(2),
id, nom, admin);
            if(id != -1) {
                connexion->setUtilisateur(id, admin);
                QByteArray reponse = "connexionacceptee";
                reponse.append(31);
                if(admin) reponse.append("admin");
                else reponse.append("user");
                reponse.append(31);
                reponse.append(QString::number(id));
                reponse.append(31);
                reponse.append(nom);
                connexion->envoyerMessage(reponse);
            }
            else connexion->envoyerMessage("connexionrefusee");
            return;
        }
        else {
            connexion->envoyerMessage("nonconnecte");
            return;
        }
    }
    // Sinon, on analyse la requête pour exécuter le bon traitement
    else {
        QList<QByteArray> champs = message.split(30);
        if(champs.at(0) == "listemotsdepasse") {
            QByteArray liste = composantBDD->listeMotsDePasse(connexion-
>getIdUtilisateur());
            connexion->envoyerMessage(liste);
            return;
        }
        else if(connexion->getEstAdministrateur()) {
            if(champs.at(0).startsWith("admin_liste")) {
                QByteArray liste;

                // Listes générales
                if(champs.at(0) == "admin_listeutilisateurs") liste =
composantBDD->admin_listeUtilisateurs();
                else if(champs.at(0) == "admin_listegroupes") liste =
composantBDD->admin_listeGroupes();
                else if(champs.at(0) == "admin_listemotsdepasse") liste =
composantBDD->admin_listeMotsDePasse();
                else if(champs.at(0) == "admin_listeserveurs") liste =
composantBDD->admin_listeServeurs();

                // Listes particulières
                else if(champs.at(0) == "admin_listegroupesutilisateur")
liste = composantBDD->admin_listeGroupesUtilisateur(champs.at(1).toInt());
                else if(champs.at(0) == "admin_listeutilisateursgroupe")
liste = composantBDD->admin_listeUtilisateursGroupe(champs.at(1).toInt());
```

```
                else if(champs.at(0) == "admin_listemotsdepassegroupe") liste
= composantBDD->admin_listeMotsDePasseGroupe(champs.at(1).toInt());
                else if(champs.at(0) == "admin_listegroupesmotdepasse") liste
= composantBDD->admin_listeGroupesMotDePasse(champs.at(1).toInt());


                connexion->envoyerMessage(liste);
                return;
            }
            else if(champs.at(0) == "admin_gererutilisateur") {
                QByteArray reponse;
                if(champs.at(1) == "creation") reponse = composantBDD-
>admin_gererUtilisateur("creation", champs.at(2).toInt(), champs.at(3),
champs.at(4), champs.at(5), champs.at(6) == "true");
                else if(champs.at(1) == "modification") reponse =
composantBDD->admin_gererUtilisateur("modification", champs.at(2).toInt(),
champs.at(3), champs.at(4), champs.at(5), champs.at(6) == "true");
                else if(champs.at(1) == "suppression") reponse =
composantBDD->admin_gererUtilisateur("suppression", champs.at(2).toInt());
                else if(champs.at(1) == "groupes") {
                    QVector<int> groupesID;
                    QList<QByteArray> groupesListe = champs.at(3).split(31);
                    for(int g = 0; g < groupesListe.size(); g++)
                        groupesID.append(groupesListe.at(g).toInt());
                    reponse = composantBDD-
>admin_gererGroupesUtilisateur(champs.at(2).toInt(), groupesID);
                }
                connexion->envoyerMessage(reponse);
                return;
            }
            else if(champs.at(0) == "admin_gerergroupe") {
                QByteArray reponse;
                if(champs.at(1) == "creation") reponse = composantBDD-
>admin_gererGroupe("creation", champs.at(2).toInt(), champs.at(3));
                else if(champs.at(1) == "modification") reponse =
composantBDD->admin_gererGroupe("modification", champs.at(2).toInt(),
champs.at(3));
                else if(champs.at(1) == "suppression") reponse =
composantBDD->admin_gererGroupe("suppression", champs.at(2).toInt());
                else if(champs.at(1) == "utilisateurs") {
                    QVector<int> utilisateursID;
                    QList<QByteArray> utilisateursListe =
champs.at(3).split(31);
                    for(int g = 0; g < utilisateursListe.size(); g++)

utilisateursID.append(utilisateursListe.at(g).toInt());
                    reponse = composantBDD-
>admin_gererUtilisateursGroupe(champs.at(2).toInt(), utilisateursID);
                }
                else if(champs.at(1) == "motsdepasse") {
                    QVector<int> motsdepasseID;
                    QList<QByteArray> motsdepasseListe =
champs.at(3).split(31);
                    for(int g = 0; g < motsdepasseListe.size(); g++)
                        motsdepasseID.append(motsdepasseListe.at(g).toInt());
                    reponse = composantBDD-
>admin_gererMotsDePasseGroupe(champs.at(2).toInt(), motsdepasseID);
                }
                connexion->envoyerMessage(reponse);
                return;
            }
            else if(champs.at(0) == "admin_gerermotdepasse") {
                QByteArray reponse;
```

```
                if(champs.at(1) == "creation") reponse = composantBDD-
>admin_gererMotDePasse("creation", champs.at(2).toInt(), champs.at(3),
champs.at(4), champs.at(5), champs.at(6).toInt());
                else if(champs.at(1) == "modification") reponse =
composantBDD->admin_gererMotDePasse("modification", champs.at(2).toInt(),
champs.at(3), champs.at(4), champs.at(5), champs.at(6).toInt());
                else if(champs.at(1) == "suppression") reponse =
composantBDD->admin_gererMotDePasse("suppression", champs.at(2).toInt());
                else if(champs.at(1) == "groupes") {
                    QVector<int> groupesID;
                    QList<QByteArray> groupesListe = champs.at(3).split(31);
                    for(int g = 0; g < groupesListe.size(); g++)
                        groupesID.append(groupesListe.at(g).toInt());
                    reponse = composantBDD-
>admin_gererGroupesMotDePasse(champs.at(2).toInt(), groupesID);
                }
                connexion->envoyerMessage(reponse);
                return;
            }
            else if(champs.at(0) == "admin_gererserveur") {
                QByteArray reponse;
                if(champs.at(1) == "creation") reponse = composantBDD-
>admin_gererServeur("creation", champs.at(2).toInt(), champs.at(3),
champs.at(4));
                else if(champs.at(1) == "modification") reponse =
composantBDD->admin_gererServeur("modification", champs.at(2).toInt(),
champs.at(3), champs.at(4));
                else if(champs.at(1) == "suppression") reponse =
composantBDD->admin_gererServeur("suppression", champs.at(2).toInt());

                connexion->envoyerMessage(reponse);
                return;
            }
            else if(champs.at(0) == "admin_sauvegarder") {
                QByteArray reponse;
                reponse = sauvegardeServeur();
                connexion->envoyerMessage(reponse);
                return;
            }
        }
        connexion->envoyerMessage("commandeincorrecte");
    }
}
```

```
// Renvoie la sauvegarde du serveur au format "sauvegardeok" + mot de passe
généré (16 caractères) + contenu crypté sauvegarde
QByteArray ComposantPrincipal::sauvegardeServeur() {
    QByteArray sauvegarde;

    QFile fConfig(fichierConfiguration);
    QFile fClePriveeRSA(fichierClePriveeRSA);
    QFile fClePubliqueRSA(fichierClePubliqueRSA);
    QFile fBaseDeDonnees(fichierBaseDeDonnees);

    // Vérification de l'ouverture des fichiers
    if(!fConfig.open(QIODevice::ReadOnly) ||
       !fClePriveeRSA.open(QIODevice::ReadOnly) ||
       !fClePubliqueRSA.open(QIODevice::ReadOnly) ||
       !fBaseDeDonnees.open(QIODevice::ReadOnly)) return QByteArray();

    // Sauvegarde fichier de configuration
    sauvegarde.append(intToByteArray(fConfig.bytesAvailable()));
    sauvegarde.append(fConfig.readAll());

    // Sauvegarde fichier de clé privée RSA
    sauvegarde.append(intToByteArray(fClePriveeRSA.bytesAvailable()));
    sauvegarde.append(fClePriveeRSA.readAll());

    // Sauvegarde fichier de clé publique RSA
    sauvegarde.append(intToByteArray(fClePubliqueRSA.bytesAvailable()));
    sauvegarde.append(fClePubliqueRSA.readAll());

    // Sauvegarde fichier de la base de données
    sauvegarde.append(intToByteArray(fBaseDeDonnees.bytesAvailable()));
    sauvegarde.append(fBaseDeDonnees.readAll());

    // Fermeture des fichiers
    fConfig.close();
    fClePriveeRSA.close();
    fClePubliqueRSA.close();
    fBaseDeDonnees.close();

    // Génération du mot de passe utilisé en clé AES
    QByteArray motDePasse = genererMotDePasse(16);

    ComposantCryptographie c;
    if(!c.setCleAES(motDePasse + motDePasse))
        return QByteArray();

    // Création de la sauvegarde au format "sauvegardeok" + mot de passe
généré (16 caractères) + contenu crypté sauvegarde
    QByteArray sauvegardeCryptee = "sauvegardeok";
    sauvegardeCryptee.append(motDePasse);
    sauvegardeCryptee.append(c.encrypterAES(sauvegarde));

    // En cas d'erreur
    if(sauvegardeCryptee.size() <= 28) return QByteArray();

    return sauvegardeCryptee;
}
```

```
// Convertit un entier en un tableau d'octets
QByteArray ComposantPrincipal::intToByteArray(unsigned int valeur) {
    QByteArray ba;

    unsigned char aux;

    for(int c = 3; c >= 0; c--) {
        aux = (unsigned char)(valeur / (unsigned int)pow(256, c));
        ba.append(aux);
        valeur -= (unsigned int)aux * (unsigned int)pow(256, c);
    }

    return ba;
}

// Convertit un tableau d'octets en un entier
unsigned int ComposantPrincipal::byteArrayToInt(const QByteArray ba) {
    unsigned int valeur = 0;

    if(ba.size() != 4) return valeur;

    unsigned char aux;

    for(int c = 3; c >= 0; c--) {
        aux = ba.at(3 - c);
        valeur += (unsigned int)aux * (unsigned int)pow(256, c);
    }

    return valeur;
}

// Génère un mot de passe aléatoirement d'une longueur donnée
QByteArray ComposantPrincipal::genererMotDePasse(const int longueur) {
    QByteArray motDePasse;

    // Caractères utilisés dans le mot de passe
    QByteArray caracteres = "abcdefghijklmnopqrstuvwxyz";
    caracteres += "0123456789";
    caracteres += "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    caracteres += "0123456789";

    int ch;

    // Choix aléatoire d'un caractère jusqu'à obtenir la longueur voulue
    srand(time(NULL));
    for(int c = 0; c < longueur; c++) {
        ch = rand() % caracteres.size();
        motDePasse.append(caracteres.at(ch));
    }

    return motDePasse;
}
```

```
// Installation du serveur avec les paramètres donnés en ligne de commande
void ComposantPrincipal::installerServeur(int argc, char *argv[]) {

    QTextStream out(stdout);

    int p = 2;

    // Fichiers et paramètres par défaut
    fichierBaseDeDonnees = QDir::homePath() +
QString("/spmd/database.sqlite3");
    fichierClePriveeRSA = QDir::homePath() +
QString("/spmd/private_key.pem");
    fichierClePubliqueRSA = QDir::homePath() +
QString("/spmd/public_key.pem");
    portEcoute = 3665;
    bool genererCles = true;

    // Analyse de chaque argument de la ligne de commande
    while(p < argc) {
        if(strcmp(argv[p], "--config") == 0 && p + 1 < argc) {
            fichierConfiguration = argv[p+1];
            p += 2;
        }
        else if(strcmp(argv[p], "--priv-key") == 0 && p + 1 < argc) {
            fichierClePriveeRSA = argv[p+1];
            p += 2;
        }
        else if(strcmp(argv[p], "--pub-key") == 0 && p + 1 < argc) {
            fichierClePubliqueRSA = argv[p+1];
            p += 2;
        }
        else if(strcmp(argv[p], "--database") == 0 && p + 1 < argc) {
            fichierBaseDeDonnees = argv[p+1];
            p += 2;
        }
        else if(strcmp(argv[p], "--port") == 0 && p + 1 < argc) {
            portEcoute = QString(argv[p+1]).toInt();
            p += 2;
        }
        else if(strcmp(argv[p], "--use-existing-keys") == 0) {
            genererCles = false;
            p++;
        }
        else p++;
    }

    // On vérifie l'existence du dossier spmd
    QDir pathToConf(QDir::homePath() + QString("/spmd"));
    if(!pathToConf.exists()) {
        if(!pathToConf.mkpath(QDir::homePath() + QString("/spmd"))) {
            out << "Impossible de créer le dossier " << QDir::homePath() +
QString("/spmd") << " . Abandon." << endl;
            exit(1);
        }
    }

    // Ouverture fichier configuration
    if(QFile::exists(fichierConfiguration)) {
        char c = 0;
        while(c != 'o' && c != 'n') {
            out << "Le fichier " << fichierConfiguration << " existe,
remplacer ? (o/n)" << endl;
            std::cin >> c;
        }
```

```
        if(c == 'n') {
            out << "Abandon." << endl;
            exit(1);
        }
    }

    QFile fConfig(fichierConfiguration);
    if(!fConfig.open(QFile::WriteOnly | QFile::Truncate)) {
        out << "Impossible d'ecrire dans le fichier " << fichierConfiguration
<< ". Abandon." << endl;
        exit(1);
    }

    // Ouverture fichier base de donées
    if(QFile::exists(fichierBaseDeDonnees)) {
        char c = 0;
        while(c != 'o' && c != 'n') {
            out << "Le fichier " << fichierBaseDeDonnees << " existe,
remplacer ? (o/n)" << endl;
            std::cin >> c;
        }
        if(c == 'n') {
            out << "Abandon." << endl;
            exit(1);
        }
        else if(!QFile::remove(fichierBaseDeDonnees)) {
            out << "Impossible d'ecrire dans le fichier " <<
fichierBaseDeDonnees << ". Abandon." << endl;
            exit(1);
        }
    }

    // En cas de génération de clés RSA
    if(genererCles) {
        // Ouverture fichier clé privée
        if(QFile::exists(fichierClePriveeRSA)) {
            char c = 0;
            while(c != 'o' && c != 'n') {
                out << "Le fichier " << fichierClePriveeRSA << " existe,
remplacer ? (o/n)" << endl;
                std::cin >> c;
            }
            if(c == 'n') {
                out << "Abandon." << endl;
                exit(1);
            }
            else if(!QFile::remove(fichierClePriveeRSA)) {
                out << "Impossible d'ecrire dans le fichier " <<
fichierClePriveeRSA << ". Abandon." << endl;
                exit(1);
            }
        }

        // Ouverture fichier clé publique
        if(QFile::exists(fichierClePubliqueRSA)) {
            char c = 0;
            while(c != 'o' && c != 'n') {
                out << "Le fichier " << fichierClePubliqueRSA << " existe,
remplacer ? (o/n)" << endl;
                std::cin >> c;
            }
            if(c == 'n') {
                out << "Abandon." << endl;
                exit(1);
```

```
            }
            else if(!QFile::remove(fichierClePubliqueRSA)) {
                out << "Impossible d'ecrire dans le fichier " <<
fichierClePubliqueRSA << ". Abandon." << endl;
                exit(1);
            }
        }

        if(!ComposantCryptographie::genererClesRSA(fichierClePriveeRSA,
fichierClePubliqueRSA)) {
            out << "Impossible de creer les cles RSA. Abandon." << endl;
            exit(1);
        }
    }
    // Sinon on utilise les clés RSA existantes
    else {
        QFile fClePubliqueRSA(fichierClePubliqueRSA);
        QFile fClePriveeRSA(fichierClePriveeRSA);

        // Vérification de l'existence des clés
        if(!fClePubliqueRSA.exists()) {
            out << "Fichier de cle publique RSA introuvable. Abandon." <<
endl;
            exit(1);
        }
        if(!fClePriveeRSA.exists()) {
            out << "Fichier de cle privee RSA introuvable. Abandon." << endl;
            exit(1);
        }

        // Vérification de la possibilité de lecture des clés
        if(fClePriveeRSA.open(QFile::ReadOnly)) {
            if(!fClePubliqueRSA.open(QFile::ReadOnly)) {
                fClePriveeRSA.close();
                out << "Impossible d'ouvrir le fichier de cle publique RSA.
Abandon." << endl;
                exit(1);
            }
        }
        else {
            out << "Impossible d'ouvrir le fichier de cle privee RSA.
Abandon." << endl;
            exit(1);
        }
        fClePubliqueRSA.close();
        fClePriveeRSA.close();
    }

    // Création base de données à partir de la structure de base
    if(!QFile::copy("db_structure", fichierBaseDeDonnees)) {
        out << "Impossible d'ecrire dans la base de donnees " <<
fichierBaseDeDonnees << ". Abandon." << endl;
        exit(1);
    }
    composantBDD = new ComposantBaseDeDonnees(fichierBaseDeDonnees,
fichierClePriveeRSA, fichierClePubliqueRSA);
    if(!composantBDD->ouvrirBaseDeDonnees()) {
        out << "Impossible d'ouvrir la base de donnees " <<
fichierBaseDeDonnees << ". Abandon." << endl;
        delete composantBDD;
        exit(1);
    }
```

```
    // Création utilisateur root aved unt mot de passe généré aléatoirement
    QString motDePasse = genererMotDePasse(10);
    if("creationok" != composantBDD->admin_gererUtilisateur("creation", -1,
"Administrateur", "root", motDePasse, true)) {
        out << "Impossible de creer le compte root. Abandon." << endl;
        delete composantBDD;
        exit(1);
    }

    // Ecriture de la configuration
    QTextStream sortiefConfig(&fConfig);
    sortiefConfig << "fichierClePriveeRSA=" << fichierClePriveeRSA << endl;
    sortiefConfig << "fichierClePubliqueRSA=" << fichierClePubliqueRSA <<
endl;
    sortiefConfig << "fichierBaseDeDonnees=" << fichierBaseDeDonnees << endl;
    sortiefConfig << "portEcoute=" << QString::number(portEcoute) << flush;
    fConfig.close();

    // Affichage des informations du compte root
    out << "Le serveur a correctement ete installe. Le compte root a ete cree
: " << endl;
    out << "Login : root" << endl << "Mot de passe : " << motDePasse << endl;
    out << "Par mesure de securite, vous pouvez changer le mot de passe du
compte root." << endl;

    exit(0);
}
```

```
// Sauvegarde du serveur en ligne de commande
void ComposantPrincipal::sauvegarderServeur(int argc, char *argv[]) {
    QTextStream out(stdout);

    // Vérification du nombre de paramètres
    if(argc != 4 && argc != 6) {
        out << "Ligne de commande incorrecte." << endl;
        ComposantDocumentation::aide();
        exit(1);
    }

    int p = 2;

    QString fichierSauvegarde = "";

    // Analyse de chaque argument de la ligne de commande
    while(p < argc) {
        if(strcmp(argv[p], "--file") == 0 && p + 1 < argc) {
            fichierSauvegarde = argv[p+1];
            p += 2;
        }
        else if(strcmp(argv[p], "--config") == 0 && p + 1 < argc) {
            fichierConfiguration = argv[p+1];
            p += 2;
        }
        else p++;
    }

    // Vérification paramètres fournis
    if(fichierSauvegarde == "") {
        out << "Ligne de commande incorrecte." << endl;
        ComposantDocumentation::aide();
        exit(1);
    }


    // Vérification de l'existence du fichier de configuration
    QFile fichierConfig(fichierConfiguration);
    if(!fichierConfig.exists()) {
        out << "Impossible de sauvegarder le serveur : le fichier de
configuration est introuvable." << endl;
        exit(1);
    }
    if(!fichierConfig.open(QIODevice::ReadOnly)) {
        out << "Impossible de sauvegarder le serveur : le fichier de
configuration ne peut etre ouvert." << endl;
        exit(1);
    }

    // Vérification du fichier de sauvegarde
    if(QFile::exists(fichierSauvegarde)) {
        char c = 0;
        while(c != 'o' && c != 'n') {
            out << "Le fichier " << fichierSauvegarde << " existe, remplacer
? (o/n)" << endl;
            std::cin >> c;
        }
        if(c == 'n') {
            out << "Abandon." << endl;
            exit(1);
        }
        else if(!QFile::remove(fichierSauvegarde)) {
            out << "Impossible d'ecrire dans le fichier " <<
fichierSauvegarde << ". Abandon." << endl;
```

```
            exit(1);
        }
    }
    QFile fSauvegarde(fichierSauvegarde);
    if(!fSauvegarde.open(QFile::WriteOnly | QFile::Truncate)) {
        out << "Impossible d'ecrire dans le fichier " << fichierSauvegarde <<
". Abandon." << endl;
        exit(1);
    }

    // Chargement de la configuration
    QString fichier(fichierConfig.readAll());
    fichierConfig.close();

    QStringList lignes = fichier.split("\n");
    QStringList ligne;
    for(int l = 0; l < lignes.size(); l++) {
        ligne = lignes.at(l).split("=");
        if(ligne.at(0) == "fichierClePubliqueRSA") fichierClePubliqueRSA =
ligne.at(1).trimmed();
        else if(ligne.at(0) == "fichierClePriveeRSA") fichierClePriveeRSA =
ligne.at(1).trimmed();
        else if(ligne.at(0) == "fichierBaseDeDonnees") fichierBaseDeDonnees =
ligne.at(1).trimmed();
        else if(ligne.at(0) == "portEcoute") portEcoute =
ligne.at(1).toInt();
    }

    // Création de la sauvegarde
    QByteArray sauvegarde = sauvegardeServeur();

    if(!sauvegarde.startsWith("sauvegardeok")) {
        out << "Impossible de sauvegarder le serveur : erreur technique.
Veuillez réessayer." << endl;
        fSauvegarde.close();
        exit(1);
    }

    QString motDePasse = sauvegarde.mid(12, 16);

    // Ecriture de la sauvegarde
    fSauvegarde.write(sauvegarde.mid(28));
    fSauvegarde.close();

    // Affichage des informations de la sauvegarde
    out << "Le serveur a correctement ete sauvegarde dans le fichier " <<
fichierSauvegarde << endl;
    out << "Mot de passe de la sauvegarde : " << motDePasse << endl;

    exit(0);

}
```

```
// Restauration d'une sauvegarde en ligne de commande
void ComposantPrincipal::restaurerServeur(int argc, char *argv[]) {
    QTextStream out(stdout);

    // Vérification du nombre d'arguments
    if(argc != 6 && argc != 8) {
        out << "Ligne de commande incorrecte." << endl;
        ComposantDocumentation::aide();
        exit(1);
    }

    int p = 2;

    QString fichierSauvegarde = "";
    QString motDePasse = "";

    // Analyse de chaque argument de la ligne de commande
    while(p < argc) {
        if(strcmp(argv[p], "--file") == 0 && p + 1 < argc) {
            fichierSauvegarde = argv[p+1];
            p += 2;
        }
        else if(strcmp(argv[p], "--password") == 0 && p + 1 < argc) {
            motDePasse = argv[p+1];
            p += 2;
        }
        else if(strcmp(argv[p], "--config") == 0 && p + 1 < argc) {
            fichierConfiguration = argv[p+1];
            p += 2;
        }
        else p++;
    }

    // Vérification des paramètres fournis
    if(fichierSauvegarde == "" || motDePasse == "") {
        out << "Ligne de commande incorrecte." << endl;
        ComposantDocumentation::aide();
        exit(1);
    }

    // Initialisation du décryptage d'après le mot de passe
    ComposantCryptographie c;
    if(!c.setCleAES(motDePasse.toUtf8() + motDePasse.toUtf8())) {
        out << "Mot de passe non accepte. Veuillez reessayer." << endl;
        exit(1);
    }

    // On vérifie l'existence du dossier spmd
    QDir pathToConf(QDir::homePath() + QString("/spmd"));
    if(!pathToConf.exists()) {
        if(!pathToConf.mkpath(QDir::homePath() + QString("/spmd"))) {
            out << "Impossible de créer le dossier " << QDir::homePath() +
QString("/spmd") << " . Abandon." << endl;
            exit(1);
        }
    }
```

```
    // Ouverture du fichier de sauvegarde
    QFile fSauvegarde(fichierSauvegarde);
    if(!fSauvegarde.open(QFile::ReadOnly)) {
        out << "Impossible de lire le fichier " << fichierSauvegarde << ".
Abandon." << endl;
        exit(1);
    }

    // Lecture et décryptage de la sauvegarde
    QByteArray sauvegardeCryptee = fSauvegarde.readAll();
    fSauvegarde.close();
    QByteArray sauvegarde = c.decrypterAES(sauvegardeCryptee);

    if(sauvegarde.size() == 0) {
        out << "Impossible de decrypter la sauvegarde. Abandon." << endl;
        exit(1);
    }

    // Extraction de chaque fichier de la sauvegarde
    int position = 0;
    int taille = byteArrayToInt(sauvegarde.mid(position, 4));
    QByteArray contenuConfig = sauvegarde.mid(position+4, taille);

    position += taille + 4;
    taille = byteArrayToInt(sauvegarde.mid(position, 4));
    QByteArray contenuClePriveeRSA = sauvegarde.mid(position+4, taille);

    position += taille + 4;
    taille = byteArrayToInt(sauvegarde.mid(position, 4));
    QByteArray contenuClePubliqueRSA = sauvegarde.mid(position+4, taille);

    position += taille + 4;
    taille = byteArrayToInt(sauvegarde.mid(position, 4));
    QByteArray contenuBaseDeDonnees = sauvegarde.mid(position+4, taille);

    // Lecture de la configuration
    QStringList lignes = QString(contenuConfig).split("\n");
    QStringList ligne;
    for(int l = 0; l < lignes.size(); l++) {
        ligne = lignes.at(l).split("=");
        if(ligne.at(0) == "fichierClePubliqueRSA") fichierClePubliqueRSA =
ligne.at(1).trimmed();
        else if(ligne.at(0) == "fichierClePriveeRSA") fichierClePriveeRSA =
ligne.at(1).trimmed();
        else if(ligne.at(0) == "fichierBaseDeDonnees") fichierBaseDeDonnees =
ligne.at(1).trimmed();
    }

    // Ouverture fichier configuration
    if(QFile::exists(fichierConfiguration)) {
        char c = 0;
        while(c != 'o' && c != 'n') {
            out << "Le fichier " << fichierConfiguration << " existe,
remplacer ? (o/n)" << endl;
            std::cin >> c;
        }
        if(c == 'n') {
            out << "Abandon." << endl;
            exit(1);
        }
    }
```

```
    QFile fConfig(fichierConfiguration);
    if(!fConfig.open(QFile::WriteOnly | QFile::Truncate)) {
        out << "Impossible d'ecrire dans le fichier " << fichierConfiguration
<< ". Abandon." << endl;
        exit(1);
    }

    // Ouverture fichier clé privée RSA
    if(QFile::exists(fichierClePriveeRSA)) {
        char c = 0;
        while(c != 'o' && c != 'n') {
            out << "Le fichier " << fichierClePriveeRSA << " existe,
remplacer ? (o/n)" << endl;
            std::cin >> c;
        }
        if(c == 'n') {
            out << "Abandon." << endl;
            exit(1);
        }
    }
    QFile fClePriveeRSA(fichierClePriveeRSA);
    if(!fClePriveeRSA.open(QFile::WriteOnly | QFile::Truncate)) {
        out << "Impossible d'ecrire dans le fichier " << fichierClePriveeRSA
<< ". Abandon." << endl;
        exit(1);
    }

    // Ouverture fichier clé publique RSA
    if(QFile::exists(fichierClePubliqueRSA)) {
        char c = 0;
        while(c != 'o' && c != 'n') {
            out << "Le fichier " << fichierClePubliqueRSA << " existe,
remplacer ? (o/n)" << endl;
            std::cin >> c;
        }
        if(c == 'n') {
            out << "Abandon." << endl;
            exit(1);
        }
    }
    QFile fClePubliqueRSA(fichierClePubliqueRSA);
    if(!fClePubliqueRSA.open(QFile::WriteOnly | QFile::Truncate)) {
        out << "Impossible d'ecrire dans le fichier " <<
fichierClePubliqueRSA << ". Abandon." << endl;
        exit(1);
    }

    // Ouverture fichier base de données
    if(QFile::exists(fichierBaseDeDonnees)) {
        char c = 0;
        while(c != 'o' && c != 'n') {
            out << "Le fichier " << fichierBaseDeDonnees << " existe,
remplacer ? (o/n)" << endl;
            std::cin >> c;
        }
        if(c == 'n') {
            out << "Abandon." << endl;
            exit(1);
        }
    }
```

```
    QFile fBaseDeDonnees(fichierBaseDeDonnees);
    if(!fBaseDeDonnees.open(QFile::WriteOnly | QFile::Truncate)) {
        out << "Impossible d'ecrire dans le fichier " << fichierBaseDeDonnees
<< ". Abandon." << endl;
        exit(1);
    }

    // Ecriture puis fermeture des fichiers
    fConfig.write(contenuConfig);
    fClePriveeRSA.write(contenuClePriveeRSA);
    fClePubliqueRSA.write(contenuClePubliqueRSA);
    fBaseDeDonnees.write(contenuBaseDeDonnees);
    fConfig.close();
    fClePriveeRSA.close();
    fClePubliqueRSA.close();
    fBaseDeDonnees.close();

    out << "Le serveur a correctement ete restaure." << endl;

    exit(0);
}
```

3. ComposantReseau

Ce composant permet d'écouter les connexions entrantes et de créer un ComposantReseauConnexion pour chaque connexion de client.

a. ComposantReseau.h

```
#ifndef COMPOSANTRESEAU_H
#define COMPOSANTRESEAU_H

#include <QTcpServer>

class ComposantCryptographie;
class ComposantPrincipal;

class ComposantReseau : public QTcpServer
{
    Q_OBJECT

public:
    ComposantReseau(int port, QString clePriv, QString clePub,
ComposantPrincipal *parent);

private:
    ComposantPrincipal * composantPrincipal;
    QString clePriveeRSA;
    QString clePubliqueRSA;

private slots:
    void nouvelleConnexion();
};

#endif // COMPOSANTRESEAU_H
```

b. ComposantReseau.cpp

```
#include "ComposantReseau.h"
#include "ComposantReseauConnexion.h"
#include "ComposantPrincipal.h"


ComposantReseau::ComposantReseau(int port, QString clePriv, QString clePub,
ComposantPrincipal * parent) {

    clePriveeRSA = clePriv;
    clePubliqueRSA = clePub;
    composantPrincipal = parent;

    // Ouverture de la socket d'écoute
    listen(QHostAddress::Any,port);

    // Toute connexion entrante entraine l'exécution de nouvelleConnexion()
    QObject::connect(this,
SIGNAL(newConnection()),this,SLOT(nouvelleConnexion()));
}


void ComposantReseau::nouvelleConnexion() {

    // Récupération de la connexion entrante
    ComposantReseauConnexion * connexion = new
ComposantReseauConnexion(nextPendingConnection(), clePriveeRSA,
clePubliqueRSA);

    // Toute arrivée de données de la connexion sera retransmise au composant
principal
    QObject::connect(connexion, SIGNAL(traitementMessage(QByteArray)),
                     composantPrincipal,
SLOT(traitementMessage(QByteArray)));
}
```

4. ComposantReseauConnexion

Ce composant est créé pour chaque connexion de client, il reçoit les requêtes, les transmet au ComposantPrincipal, et retransmet les réponses au client.

a. ComposantReseauConnexion .h

```
#ifndef COMPOSANTRESEAUCONNEXION_H
#define COMPOSANTRESEAUCONNEXION_H

#include <QObject>

class ComposantCryptographie;
class QTcpSocket;

class ComposantReseauConnexion : public QObject
{
    Q_OBJECT

public:
    ComposantReseauConnexion(QTcpSocket *s, const QString clePriv, const
QString clePub);
    ~ComposantReseauConnexion();
    void setUtilisateur(int id, bool admin);
    int getIdUtilisateur();
    bool getEstAdministrateur();

    void envoyerMessage(QByteArray message);

private:
    bool estCryptee;
    int idUtilisateur;
    bool estAdministrateur;
    QTcpSocket * socket;
    ComposantCryptographie * composantCrypto;

    QByteArray intToByteArray(unsigned int valeur);
    unsigned int byteArrayToInt(const QByteArray ba);

private slots:
    void receptionMessage();

signals:
    void traitementMessage(QByteArray message);
};

#endif // COMPOSANTRESEAUCONNEXION_H
```

b. ComposantReseauConnexion .cpp

```
#include "ComposantReseauConnexion.h"
#include "ComposantCryptographie.h"

#include <QTcpSocket>
#include <cmath>

ComposantReseauConnexion::ComposantReseauConnexion(QTcpSocket *s, const
QString clePriv, const QString clePub)
{
    socket = s;

    // Toute arrivée de données entrainera l'exécution de receptionMessage()
    QObject::connect(socket,
SIGNAL(readyRead()),this,SLOT(receptionMessage()));

    // Si déconnexion, on supprime la connexion
    QObject::connect(socket,
SIGNAL(disconnected()),this,SLOT(deleteLater()));

    // Création du composant de cryptographie pour la connexion
    composantCrypto = new ComposantCryptographie();
    composantCrypto->loadClePriveeRSA(clePriv);
    composantCrypto->loadClePubliqueRSA(clePub);

    // Initialisation session
    estCryptee = false;
    idUtilisateur = -1;
    estAdministrateur = false;
}

ComposantReseauConnexion::~ComposantReseauConnexion()
{
    delete socket;
    delete composantCrypto;
}
```

```
// Traitement des données reçues sur la connexion
void ComposantReseauConnexion::receptionMessage() {

    // Lecture du bloc de données
    QByteArray reponse = socket->readAll();

    unsigned int taille = byteArrayToInt(reponse.mid(0, 4));

    while((unsigned int)reponse.size() < taille) {
        socket->waitForReadyRead();
        reponse.append(socket->readAll());
    }

    QByteArray messageCrypte = reponse.mid(4);

    // Si la connexion n'est pas cryptée
    if(!estCryptee) {
        QByteArray reponse;
        QByteArray message = composantCrypto->decrypterRSA(messageCrypte);

        // Si le message ne contient pas la clé AES
        if(!message.startsWith("cleaes")) {


            // Le client avait demandé la clé publique RSA du serveur (en non
crypté)
            if(messageCrypte == "clepubliquersa") {
                reponse = composantCrypto->getClePubliqueRSA();
            }
            // Erreur de message
            else reponse = "cleaeserreur";
        }

        // Sinon on récupère la clé AES
        else if(composantCrypto->setCleAES(message.mid(6))) {
            reponse = "cleaesok";
            estCryptee = true;
        }

        // Ajout de la taille du message + 4 au début
        reponse.prepend(intToByteArray(reponse.size() + 4));

        // Envoi de la réponse
        socket->write(reponse);
        socket->flush();
        return;
    }

    // Si la connexion était cryptée, on décrypte et on lance le traitement
du message
    QByteArray message = composantCrypto->decrypterAES(messageCrypte);
    emit traitementMessage(message);
}
```

```
// Envoi d'un message au client
// Le message sera crypté en AES
void ComposantReseauConnexion::envoyerMessage(QByteArray message) {

    if(!estCryptee) return;

    // Cryptage du message en AES
    QByteArray messageCrypte = composantCrypto->encrypterAES(message);

    // Ajout de la taille du message + 4 au début
    messageCrypte.prepend(intToByteArray(messageCrypte.size() + 4));

    // Envoi de la réponse
    socket->write(messageCrypte);
    socket->flush();
}

// Prise en compte des informations de l'utilisateur connecté dans la session
void ComposantReseauConnexion::setUtilisateur(int id, bool admin) {
    idUtilisateur = id;
    estAdministrateur = admin;
}

// Récupère l'id de l'utilisateur connecté
int ComposantReseauConnexion::getIdUtilisateur() {
    return idUtilisateur;
}

// Indique si l'utilisateur connecté est administrateur
bool ComposantReseauConnexion::getEstAdministrateur() {
    return estAdministrateur;
}


// Convertit un entier en un tableau d'octets
QByteArray ComposantReseauConnexion::intToByteArray(unsigned int valeur) {
    QByteArray ba;

    unsigned char aux;

    for(int c = 3; c >= 0; c--) {
        aux = (unsigned char)(valeur / (unsigned int)pow(256, c));
        ba.append(aux);
        valeur -= (unsigned int)aux * (unsigned int)pow(256, c);
    }

    return ba;
}

// Convertit un tableau d'octets en un entier
unsigned int ComposantReseauConnexion::byteArrayToInt(const QByteArray ba) {
    unsigned int valeur = 0;

    if(ba.size() != 4) return valeur;

    unsigned char aux;

    for(int c = 3; c >= 0; c--) {
        aux = ba.at(3 - c);
        valeur += (unsigned int)aux * (unsigned int)pow(256, c);
    }

    return valeur;
}
```

5. ComposantBaseDeDonnees

Ce composant permet de réaliser les traitements sur la base de données.

a. ComposantBaseDeDonnees .h

```
#ifndef COMPOSANTBASEDEDONNEES_H
#define COMPOSANTBASEDEDONNEES_H

#include <QtCore>
#include <QSql>
#include <QSqlDatabase>

class ComposantCryptographie;

class ComposantBaseDeDonnees
{
public:
    ComposantBaseDeDonnees(const QString fichier, const QString
fClePriveeRSA, const QString fClePubliqueRSA);
    ~ComposantBaseDeDonnees();
    bool ouvrirBaseDeDonnees();

    void connecterUtilisateur(const QString utilisateur, const QString
motDePasse, int & id, QString &nom, bool & admin);

    // Listes générales
    QByteArray admin_listeUtilisateurs();
    QByteArray admin_listeGroupes();
    QByteArray admin_listeMotsDePasse();
    QByteArray admin_listeServeurs();

    // Listes particulières
    QByteArray listeMotsDePasse(const int idUtilisateur);
    QByteArray admin_listeGroupesUtilisateur(const int idUtilisateur);
    QByteArray admin_listeUtilisateursGroupe(const int idGroupe);
    QByteArray admin_listeMotsDePasseGroupe(const int idGroupe);
    QByteArray admin_listeGroupesMotDePasse(const int idMotDePasse);

    // Administration des utilisateurs
    QByteArray admin_gererUtilisateur(const QString mode, const int id = -1,
const QString nom = "", const QString identifiant = "", const QString
motDePasse = "", const bool admin = false);
    QByteArray admin_gererGroupesUtilisateur(const int idUtilisateur, const
QVector<int> groupes);

    // Administration des groupes
    QByteArray admin_gererGroupe(const QString mode, const int id = -1, const
QString nom = "");
    QByteArray admin_gererUtilisateursGroupe(const int idGroupe, const
QVector<int> utilisateurs);
    QByteArray admin_gererMotsDePasseGroupe(const int idGroupe, const
QVector<int> motsDePasse);

    // Administration des mots de passe
    QByteArray admin_gererMotDePasse(const QString mode, const int id = -1,
const QString nom = "", const QString identifiant = "", const QString
motDePasse = "", const int serveur = 0);
    QByteArray admin_gererGroupesMotDePasse(const int idMotDePasse, const
QVector<int> groupes);
```

```
// Administration des serveurs
    QByteArray admin_gererServeur(const QString mode, const int id = -1,
const QString nom = "", const QString hote = "");

private:
    QSqlDatabase db;
    QString fichierBaseDeDonnees;
    QString fichierClePriveeRSA;
    QString fichierClePubliqueRSA;
    ComposantCryptographie * compCrypto;
};

#endif // COMPOSANTBASEDEDONNEES_H
```

b. ComposantBaseDeDonnees .cpp

```
#include "ComposantBaseDeDonnees.h"
#include "ComposantCryptographie.h"

#include <QSqlQuery>
#include <QSqlError>
#include <QFile>

ComposantBaseDeDonnees::ComposantBaseDeDonnees(const QString fichier, const
QString fClePriveeRSA, const QString fClePubliqueRSA) {
    fichierBaseDeDonnees = fichier;
    fichierClePriveeRSA = fClePriveeRSA;
    fichierClePubliqueRSA = fClePubliqueRSA;
    compCrypto = NULL;
}

// Ferme la base de données si elle est ouverte
// Détruit le composant de cryptographie si il existe
ComposantBaseDeDonnees::~ComposantBaseDeDonnees() {
    if(db.isOpen()) db.close();
    if(compCrypto != NULL) delete compCrypto;
}

// Ouvre la base de données à partir du chemin fichierBaseDeDonnees
// Initialise le composant de cryptographie pour les mots de passe
// Renvoie true si l'ouverture a réussie, false sinon
bool ComposantBaseDeDonnees::ouvrirBaseDeDonnees() {

    // Initialisation base de données
    db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName(fichierBaseDeDonnees);

    // Ouverture de la base de données
    if(db.open()) {
        // Initialisation du composant de cryptographie
        compCrypto = new ComposantCryptographie();
        if(!compCrypto) return false;
        if(compCrypto->loadClePriveeRSA(fichierClePriveeRSA) &&
           compCrypto->loadClePubliqueRSA(fichierClePubliqueRSA))
        {
            return true;
        }
        // Si le chargement a échoué
        delete compCrypto;
        compCrypto = NULL;
    }

    // Si l'ouverture de la base de données
    // Ou le chargement des clés RSA a échoué
    return false;
}
```

```
// Vérifie les informations de connexion utilisateur fournies
void ComposantBaseDeDonnees::connecterUtilisateur(const QString utilisateur,
const QString motDePasse, int & id, QString & nom, bool & admin) {

    // Initialisations
    id = -1;
    nom = "";
    admin = false;

    QByteArray motDePasseCrypte;
    QString motDePasseClair;

    if(db.isOpen()) {

        // Préparation de la requête - utilisateur selon identifiant
        QSqlQuery query(db);
        query.prepare("SELECT id_utilisateur, nom, motdepasse, administrateur
"
                      "FROM utilisateurs "
                      "WHERE identifiant = :identifiant");
        query.bindValue(":identifiant", utilisateur, QSql::Out);
        query.exec();

        // Récupération des valeurs en cas de résultat
        if(query.next()) {

            // Décryptage du mot de passe associé à l'identifiant
            motDePasseCrypte =
QByteArray::fromHex(query.value(2).toString().toUtf8());
            motDePasseClair = QString(compCrypto-
>decrypterRSA(motDePasseCrypte));

            // Affectation des variables de la session de connexion
            if(motDePasseClair == motDePasse) {
                id = query.value(0).toInt();
                nom = query.value(1).toString();
                admin = query.value(3).toInt() == 1;
            }
        }
    }
}
```

```
/******************************************** LISTES GENERALES
*********************************************/

// Administration : liste tous les utilisateurs du système
QByteArray ComposantBaseDeDonnees::admin_listeUtilisateurs() {

    // Préparation de la réponse
    QByteArray liste;
    liste.append("admin_listeutilisateurs");
    liste.append(30);

    QByteArray motDePasseCrypte;
    QString motDePasseClair;

    if(db.isOpen()) {

        // True lorsqu'au moins un utilisateur existe
        bool ok = false;

        // Préparation et exécution de la requête - liste utilisateurs
        QSqlQuery query(db);
        query.prepare("SELECT id_utilisateur, nom, identifiant, motdepasse,
administrateur "
                      "FROM utilisateurs ");
        query.exec();

        // Récupération des valeurs
        while(query.next()) {
            ok = true;

            // Décryptage du mot de passe de l'utilisateur
            motDePasseCrypte =
QByteArray::fromHex(query.value(3).toString().toUtf8());
            motDePasseClair = QString(compCrypto-
>decrypterRSA(motDePasseCrypte));

            liste.append(QString::number(query.value(0).toInt()));
            liste.append(31);
            liste.append(query.value(1).toString());
            liste.append(31);
            liste.append(query.value(2).toString());
            liste.append(31);
            liste.append(motDePasseClair);
            liste.append(31);
            liste.append(query.value(4).toInt() == 1 ? "admin" : "user");
            liste.append(30);
        }
        if(ok) liste.remove(liste.size() - 1, 1);
    }

    return liste;
}
```

```
// Administration : liste tous les groupes du système
QByteArray ComposantBaseDeDonnees::admin_listeGroupes() {

    // Préparation de la réponse
    QByteArray liste;
    liste.append("admin_listegroupes");
    liste.append(30);

    if(db.isOpen()) {

        // True lorsqu'au moins un groupe existe
        bool ok = false;

        // Préparation et exécution de la requête - liste groupes
        QSqlQuery query(db);
        query.prepare("SELECT id_groupe, nom "
                      "FROM groupes ");
        query.exec();

        // Récupération des valeurs en cas de résultat
        while(query.next()) {
            ok = true;
            liste.append(QString::number(query.value(0).toInt()));
            liste.append(31);
            liste.append(query.value(1).toString());
            liste.append(30);
        }
        if(ok) liste.remove(liste.size() - 1, 1);
    }

    return liste;
}
```

```
// Administration : liste tous les mots de passe du système
QByteArray ComposantBaseDeDonnees::admin_listeMotsDePasse() {

    // Préparation de la réponse
    QByteArray liste;
    liste.append("admin_listemotsdepasse");
    liste.append(30);

    QByteArray motDePasseCrypte;
    QString motDePasseClair;

    if(db.isOpen()) {

        // True lorsqu'au moins un mot de passe existe
        bool ok = false;

        // Préparation et exécution de la requête - liste mots de passe
        QSqlQuery query(db);
        query.prepare("SELECT id_motdepasse, motsdepasse.nom, identifiant,
motdepasse, serveurs.id_serveur, serveurs.nom, hote "
                      "FROM motsdepasse, serveurs "
                      "WHERE serveurs.id_serveur = motsdepasse.id_serveur");
        query.exec();

        // Récupération des valeurs en cas de résultat
        while(query.next()) {
            ok = true;

            // Décryptage du mot de passe
            motDePasseCrypte =
QByteArray::fromHex(query.value(3).toString().toUtf8());
            motDePasseClair = QString(compCrypto-
>decrypterRSA(motDePasseCrypte));

            liste.append(QString::number(query.value(0).toInt()));
            liste.append(31);
            liste.append(query.value(1).toString());
            liste.append(31);
            liste.append(query.value(2).toString());
            liste.append(31);
            liste.append(motDePasseClair);
            liste.append(31);
            liste.append(QString::number(query.value(4).toInt()));
            liste.append(31);
            liste.append(query.value(5).toString());
            liste.append(31);
            liste.append(query.value(6).toString());
            liste.append(30);
        }
        if(ok) liste.remove(liste.size() - 1, 1);
    }

    return liste;
}
```

```
// Administration : liste tous les serveurs du système
QByteArray ComposantBaseDeDonnees::admin_listeServeurs() {

    // Préparation de la réponse
    QByteArray liste;
    liste.append("admin_listeserveurs");
    liste.append(30);

    if(db.isOpen()) {

        // True lorsqu'au moins un serveur existe
        bool ok = false;

        // Préparation et exécution de la requête - liste serveurs
        QSqlQuery query(db);
        query.prepare("SELECT id_serveur, nom, hote "
                      "FROM serveurs ");
        query.exec();

        // Récupération des valeurs en cas de résultat
        while(query.next()) {
            ok = true;
            liste.append(QString::number(query.value(0).toInt()));
            liste.append(31);
            liste.append(query.value(1).toString());
            liste.append(31);
            liste.append(query.value(2).toString());
            liste.append(30);
        }
        if(ok) liste.remove(liste.size() - 1, 1);
    }

    return liste;
}
```

```
/******************************************** LISTES PARTICULIERES
*********************************************/

// Utilisateur : liste tous les mots de passe accessibles à un utilisateur
QByteArray ComposantBaseDeDonnees::listeMotsDePasse(const int idUtilisateur)
{

    // Préparation de la réponse
    QByteArray liste;
    liste.append("listemotsdepasse");
    liste.append(30);

    QByteArray motDePasseCrypte;
    QString motDePasseClair;

    if(db.isOpen()) {

        // True lorsqu'au moins un mot de passe est accessible
        bool ok = false;

        // Préparation et exécution de la requête - liste mots de passe de
l'utilisateur
        QSqlQuery query(db);
        query.prepare("SELECT DISTINCT serveurs.nom, hote, motsdepasse.nom,
identifiant, motdepasse "
                      "FROM serveurs, motsdepasse, utilisateurs_groupes,
groupes_motsdepasse "
                      "WHERE serveurs.id_serveur = motsdepasse.id_serveur "
                      "  AND motsdepasse.id_motdepasse =
groupes_motsdepasse.id_motdepasse "
                      "  AND groupes_motsdepasse.id_groupe =
utilisateurs_groupes.id_groupe "
                      "  AND utilisateurs_groupes.id_utilisateur =
:id_utilisateur");
        query.bindValue(":id_utilisateur", idUtilisateur, QSql::Out);
        query.exec();

        // Récupération des valeurs en cas de résultat
        while(query.next()) {
            ok = true;

            // Décryptage du mot de passe
            motDePasseCrypte =
QByteArray::fromHex(query.value(4).toString().toUtf8());
            motDePasseClair = QString(compCrypto-
>decrypterRSA(motDePasseCrypte));

            liste.append(query.value(0).toString());
            liste.append(31);
            liste.append(query.value(1).toString());
            liste.append(31);
            liste.append(query.value(2).toString());
            liste.append(31);
            liste.append(query.value(3).toString());
            liste.append(31);
            liste.append(motDePasseClair);
            liste.append(30);
        }
        if(ok) liste.remove(liste.size() - 1, 1);
    }

    return liste;
}
```

```
// Administration : liste les groupes dans lequel un utilisateur est affecté
QByteArray ComposantBaseDeDonnees::admin_listeGroupesUtilisateur(const int
idUtilisateur) {

    // Préparation de la réponse
    QByteArray liste;
    liste.append("admin_listegroupesutilisateur");
    liste.append(30);

    if(db.isOpen()) {

        // True lorsque l'utilisateur est affecté à au moins un groupe
        bool ok = false;

        // Préparation et exécution de la requête - liste groupes de
utilisateur
        QSqlQuery query(db);
        query.prepare("SELECT DISTINCT groupes.id_groupe, nom "
                      "FROM groupes, utilisateurs_groupes "
                      "WHERE groupes.id_groupe =
utilisateurs_groupes.id_groupe "
                      "  AND utilisateurs_groupes.id_utilisateur =
:id_utilisateur");
        query.bindValue(":id_utilisateur", idUtilisateur, QSql::Out);
        query.exec();

        // Récupération des valeurs en cas de résultat
        while(query.next()) {
            ok = true;
            liste.append(QString::number(query.value(0).toInt()));
            liste.append(31);
            liste.append(query.value(1).toString());
            liste.append(30);
        }
        if(ok) liste.remove(liste.size() - 1, 1);
    }

    return liste;
}
```

```
// Administration : liste les utilisateurs affectés à un groupe
QByteArray ComposantBaseDeDonnees::admin_listeUtilisateursGroupe(const int
idGroupe) {

    // Préparation de la réponse
    QByteArray liste;
    liste.append("admin_listeutilisateursgroupe");
    liste.append(30);

    if(db.isOpen()) {

        // True lorsqu'au moins un utilisateur est affecté au groupe
        bool ok = false;

        // Préparation et exécution de la requête - liste utilisateurs du
groupe
        QSqlQuery query(db);
        query.prepare("SELECT DISTINCT utilisateurs.id_utilisateur, nom,
identifiant "
                      "FROM utilisateurs, utilisateurs_groupes "
                      "WHERE utilisateurs.id_utilisateur =
utilisateurs_groupes.id_utilisateur "
                      "  AND utilisateurs_groupes.id_groupe = :id_groupe");
        query.bindValue(":id_groupe", idGroupe, QSql::Out);
        query.exec();

        // Récupération des valeurs en cas de résultat
        while(query.next()) {
            ok = true;
            liste.append(QString::number(query.value(0).toInt()));
            liste.append(31);
            liste.append(query.value(1).toString());
            liste.append(31);
            liste.append(query.value(2).toString());
            liste.append(30);
        }
        if(ok) liste.remove(liste.size() - 1, 1);
    }

    return liste;
}
```

```
// Administration : liste les mots de passe auxquels un groupe a accès
QByteArray ComposantBaseDeDonnees::admin_listeMotsDePasseGroupe(const int
idGroupe) {

    // Préparation de la réponse
    QByteArray liste;
    liste.append("admin_listemotsdepassegroupe");
    liste.append(30);

    if(db.isOpen()) {

        // True lorsque le groupe a accès à au moins un mot de passe
        bool ok = false;

        // Préparation et exécution de la requête - liste mots de passe du
groupe
        QSqlQuery query(db);
        query.prepare("SELECT DISTINCT motsdepasse.id_motdepasse,
motsdepasse.nom, identifiant, serveurs.nom, hote "
                      "FROM motsdepasse, serveurs, groupes_motsdepasse "
                      "WHERE serveurs.id_serveur = motsdepasse.id_serveur "
                      "  AND motsdepasse.id_motdepasse =
groupes_motsdepasse.id_motdepasse "
                      "  AND groupes_motsdepasse.id_groupe = :id_groupe");
        query.bindValue(":id_groupe", idGroupe, QSql::Out);
        query.exec();

        // Récupération des valeurs en cas de résultat
        while(query.next()) {
            ok = true;
            liste.append(QString::number(query.value(0).toInt()));
            liste.append(31);
            liste.append(query.value(1).toString());
            liste.append(31);
            liste.append(query.value(2).toString());
            liste.append(31);
            liste.append(query.value(3).toString());
            liste.append(31);
            liste.append(query.value(4).toString());
            liste.append(30);
        }
        if(ok) liste.remove(liste.size() - 1, 1);
    }

    return liste;
}
```

```
// Administration : liste les groupes qui ont accès à un mot de passe
QByteArray ComposantBaseDeDonnees::admin_listeGroupesMotDePasse(const int
idMotDePasse) {

    // Préparation de la réponse
    QByteArray liste;
    liste.append("admin_listegroupesmotdepasse");
    liste.append(30);

    if(db.isOpen()) {

        // True lorsqu'au moins un groupe a accès au mot de passe
        bool ok = false;

        // Préparation et exécution de la requête - liste groupes du mot de
passe
        QSqlQuery query(db);
        query.prepare("SELECT DISTINCT groupes.id_groupe, nom "
                      "FROM groupes, groupes_motsdepasse "
                      "WHERE groupes.id_groupe =
groupes_motsdepasse.id_groupe "
                      "  AND groupes_motsdepasse.id_motdepasse =
:id_motdepasse");
        query.bindValue(":id_motdepasse", idMotDePasse, QSql::Out);
        query.exec();

        // Récupération des valeurs en cas de résultat
        while(query.next()) {
            ok = true;
            liste.append(QString::number(query.value(0).toInt()));
            liste.append(31);
            liste.append(query.value(1).toString());
            liste.append(30);
        }
        if(ok) liste.remove(liste.size() - 1, 1);
    }

    return liste;
}
```

```
/******************************************** GESTION UTILISATEURS
*********************************************/

// Administration : gestion des utilisateurs
QByteArray ComposantBaseDeDonnees::admin_gererUtilisateur(const QString mode,
const int id, const QString nom, const QString identifiant, const QString
motDePasse, const bool admin) {

    QByteArray reponse;

    if(mode == "creation") {

        // Cryptage du mot de passe
        QString motDePasseCrypte = (compCrypto-
>encrypterRSA(motDePasse.toUtf8()).toHex());

        // Préparation de la requête - création utilisateur
        QSqlQuery query(db);
        query.prepare("INSERT INTO utilisateurs (nom, identifiant,
motdepasse, administrateur) "
                      "                VALUES (:nom, :identifiant,
:motdepasse, :administrateur)");
        query.bindValue(":nom", nom, QSql::In);
        query.bindValue(":identifiant", identifiant, QSql::In);
        query.bindValue(":motdepasse", motDePasseCrypte, QSql::In);
        query.bindValue(":administrateur", admin ? 1 : 0, QSql::In);

        // Exécution de la requête
        if(query.exec()) reponse = "creationok";
        else {
            reponse = "creationerreurtechnique";
            reponse.append(30);
            reponse.append(query.lastError().text());
        }

    }
    else if(mode == "modification") {

        // Cryptage du mot de passe
        QString motDePasseCrypte = QString(compCrypto-
>encrypterRSA(motDePasse.toUtf8()).toHex());

        // Préparation de la requête - modification utilisateur
        QSqlQuery query(db);
        query.prepare("UPDATE utilisateurs "
                      "SET nom = :nom, identifiant = :identifiant, motdepasse
= :motdepasse, administrateur = :administrateur "
                      "WHERE id_utilisateur = :id_utilisateur");
        query.bindValue(":nom", nom, QSql::In);
        query.bindValue(":identifiant", identifiant, QSql::In);
        query.bindValue(":motdepasse", motDePasseCrypte, QSql::In);
        query.bindValue(":administrateur", admin ? 1 : 0, QSql::In);
        query.bindValue(":id_utilisateur", id, QSql::In);

        // Exécution de la requête
        if(query.exec()) {
            if(query.numRowsAffected() == 1) reponse = "modificationok";
            else reponse = "modificationerreurinexistant";
        }
        else {
            reponse = "modificationerreurtechnique";
            reponse.append(30);
            reponse.append(query.lastError().text());
        }
```

```
    }
    else if(mode == "suppression") {

        // Plusieurs requêtes en modification ont lieu : on démarre une
transaction
        db.transaction();

        // Préparation et exécution de la requête - suppression utilisateur
        QSqlQuery query1(db);
        query1.prepare("DELETE FROM utilisateurs "
                       "WHERE id_utilisateur = :id_utilisateur");
        query1.bindValue(":id_utilisateur", id, QSql::In);
        query1.exec();

        // Si l'utilisateur existait bien
        if(query1.numRowsAffected() == 1) {

            // Préparation et exécution de la requête - suppression liens
avec groupes de l'utilisateur
            QSqlQuery query2(db);
            query2.prepare("DELETE FROM utilisateurs_groupes "
                           "WHERE id_utilisateur = :id_utilisateur");
            query2.bindValue(":id_utilisateur", id, QSql::In);
            query2.exec();

            // Fin de transaction
            if(db.commit())
                reponse = "suppressionok";
            else {
                db.rollback();
                reponse = "suppressionerreurtechnique";
                reponse.append(30);
                reponse.append(db.lastError().text());
            }

        }
        // Si l'utilisateur n'existait pas
        else {
            db.rollback();
            reponse = "suppressionerreurinexistant";
        }
    }

    return reponse;
}
```

```
// Administration : gestion des groupes d'un utilisateur
QByteArray ComposantBaseDeDonnees::admin_gererGroupesUtilisateur(const int
idUtilisateur, const QVector<int> groupes) {
    QByteArray reponse;

    // Vérification de l'existence de l'utilisateur
    QSqlQuery query(db);
    query.prepare("SELECT * FROM utilisateurs "
                  "WHERE id_utilisateur = :id_utilisateur");
    query.bindValue(":id_utilisateur", idUtilisateur, QSql::In);
    query.exec();

    // Si l'utilisateur n'existe pas
    if(!query.next()) reponse = "groupesutilisateurerreurinexistant";

    // Si l'utilisateur existe
    else {

        // Plusieurs requêtes en modification ont lieu : on démarre une
transaction
        db.transaction();

        // Préparation de la requête - suppression liens avec groupes de
l'utilisateurs
        QSqlQuery query1(db);
        query1.prepare("DELETE FROM utilisateurs_groupes "
                       "WHERE id_utilisateur = :id_utilisateur");
        query1.bindValue(":id_utilisateur", idUtilisateur, QSql::In);
        query1.exec();

        // Préparation de la requête - ajout liens avec nouveaux groupes de
l'utilisateur
        QSqlQuery query2(db);
        query2.prepare("INSERT INTO utilisateurs_groupes (id_utilisateur,
id_groupe) "
                       "                          VALUES (:id_utilisateur,
:id_groupe)");
        query2.bindValue(":id_utilisateur", idUtilisateur, QSql::In);

        // Exécution de la requête pour chaque id de nouveau groupe
        for(int g = 0; g < groupes.size(); g++) {
            query2.bindValue(":id_groupe", groupes.at(g), QSql::In);
            query2.exec();
        }

        // Fin de transaction
        if(db.commit())
            reponse = "groupesutilisateurok";
        else {
            db.rollback();
            reponse = "groupesutilisateurerreurtechnique";
            reponse.append(30);
            reponse.append(db.lastError().text());
        }

    }

    return reponse;
}
```

```
/******************************************** GESTION GROUPES
*********************************************/

// Administration : gestion des groupes
QByteArray ComposantBaseDeDonnees::admin_gererGroupe(const QString mode,
const int id, const QString nom) {
    QByteArray reponse;

    if(mode == "creation") {
        // Préparation de la requête - création groupe
        QSqlQuery query(db);
        query.prepare("INSERT INTO groupes (nom) "
                      "             VALUES (:nom)");
        query.bindValue(":nom", nom, QSql::In);

        // Exécution de la requête
        if(query.exec()) reponse = "creationok";
        else {
            reponse = "creationerreurtechnique";
            reponse.append(30);
            reponse.append(query.lastError().text());
        }
    }
    else if(mode == "modification") {
        // Préparation de la requête - modification groupe
        QSqlQuery query(db);
        query.prepare("UPDATE groupes "
                      "SET nom = :nom "
                      "WHERE id_groupe = :id_groupe");
        query.bindValue(":nom", nom, QSql::In);
        query.bindValue(":id_groupe", id, QSql::In);

        // Exécution de la requête
        if(query.exec()) {
            if(query.numRowsAffected() == 1) reponse = "modificationok";
            else reponse = "modificationerreurinexistant";
        }
        else {
            reponse = "modificationerreurtechnique";
            reponse.append(30);
            reponse.append(query.lastError().text());
        }
    }
    else if(mode == "suppression") {

        // Plusieurs requêtes en modification ont lieu : on démarre une
transaction
        db.transaction();

        // Préparation et exécution de la requête - suppression groupe
        QSqlQuery query1(db);
        query1.prepare("DELETE FROM groupes "
                       "WHERE id_groupe = :id_groupe");
        query1.bindValue(":id_groupe", id, QSql::In);
        query1.exec();

        // Si le groupe existait bien
        if(query1.numRowsAffected() == 1) {

            // Préparation de la requête - suppression liens avec
utilisateurs du groupe
            QSqlQuery query2(db);
            query2.prepare("DELETE FROM utilisateurs_groupes "
                           "WHERE id_groupe = :id_groupe");
```

```
            query2.bindValue(":id_groupe", id, QSql::In);
            query2.exec();

            // Préparation de la requête - suppression liens avec mots de
passe du groupe
            QSqlQuery query3(db);
            query3.prepare("DELETE FROM groupes_motsdepasse "
                           "WHERE id_groupe = :id_groupe");
            query3.bindValue(":id_groupe", id, QSql::In);
            query3.exec();

            // Fin de transaction
            if(db.commit())
                reponse = "suppressionok";
            else {
                db.rollback();
                reponse = "suppressionerreurtechnique";
                reponse.append(30);
                reponse.append(db.lastError().text());
            }

        }
        // Si le groupe n'existait pas
        else {
            db.rollback();
            reponse = "suppressionerreurinexistant";
        }
    }

    return reponse;
}
```

```
// Administration : gestion des utilisateurs d'un groupe
QByteArray ComposantBaseDeDonnees::admin_gererUtilisateursGroupe(const int
idGroupe, const QVector<int> utilisateurs) {
    QByteArray reponse;

    // Vérification de l'existence du groupe
    QSqlQuery query(db);
    query.prepare("SELECT * FROM groupes "
                  "WHERE id_groupe = :id_groupe");
    query.bindValue(":id_groupe", idGroupe, QSql::In);
    query.exec();

    // Si le groupe n'existe pas
    if(!query.next()) reponse = "utilisateursgroupeerreurinexistant";

    // Si le groupe existe bien
    else {

        // Plusieurs requêtes en modification ont lieu : on démarre une
transaction
        db.transaction();

        // Préparation de la requête - suppression liens avec utilisateurs du
groupe
        QSqlQuery query1(db);
        query1.prepare("DELETE FROM utilisateurs_groupes "
                       "WHERE id_groupe = :id_groupe");
        query1.bindValue(":id_groupe", idGroupe, QSql::In);
        query1.exec();

        // Préparation de la requête - ajout liens avec nouveaux utilisateurs
du groupe
        QSqlQuery query2(db);
        query2.prepare("INSERT INTO utilisateurs_groupes (id_utilisateur,
id_groupe) "
                       "                          VALUES (:id_utilisateur,
:id_groupe)");
        query2.bindValue(":id_groupe", idGroupe, QSql::In);

        // Exécution de la requête pour chaque id de nouvel utilisateur
        for(int u = 0; u < utilisateurs.size(); u++) {
            query2.bindValue(":id_utilisateur", utilisateurs.at(u),
QSql::In);
            query2.exec();
        }

        // Fin de transaction
        if(db.commit())
            reponse = "utilisateursgroupeok";
        else {
            db.rollback();
            reponse = "utilisateursgroupeerreurtechnique";
            reponse.append(30);
            reponse.append(db.lastError().text());
        }

    }

    return reponse;
}
```

```
// Administration : gestion des mots de passe d'un groupe
QByteArray ComposantBaseDeDonnees::admin_gererMotsDePasseGroupe(const int
idGroupe, const QVector<int> motsDePasse) {
    QByteArray reponse;

    // Vérification de l'existence du groupe
    QSqlQuery query(db);
    query.prepare("SELECT * FROM groupes "
                  "WHERE id_groupe = :id_groupe");
    query.bindValue(":id_groupe", idGroupe, QSql::In);
    query.exec();

    // Si le groupe n'existe pas
    if(!query.next()) reponse = "motsdepassegroupeerreurinexistant";

    // Si le groupe existe
    else {

        // Plusieurs requêtes en modification ont lieu : on démarre une
transaction
        db.transaction();

        // Préparation de la requête - suppression liens avec mots de passe
du groupe
        QSqlQuery query1(db);
        query1.prepare("DELETE FROM groupes_motsdepasse "
                       "WHERE id_groupe = :id_groupe");
        query1.bindValue(":id_groupe", idGroupe, QSql::In);
        query1.exec();

        // Préparation de la requête - ajout liens avec nouveaux mots de
passe
        QSqlQuery query2(db);
        query2.prepare("INSERT INTO groupes_motsdepasse (id_groupe,
id_motdepasse) "
                       "                          VALUES (:id_groupe,
:id_motdepasse)");
        query2.bindValue(":id_groupe", idGroupe, QSql::In);

        // Exécution de la requête pour chaque id de nouveau mot de passe
        for(int m = 0; m < motsDePasse.size(); m++) {
            query2.bindValue(":id_motdepasse", motsDePasse.at(m), QSql::In);
            query2.exec();
        }

        // Fin de transaction
        if(db.commit())
            reponse = "motsdepassegroupeok";
        else {
            db.rollback();
            reponse = "motsdepassegroupeerreurtechnique";
            reponse.append(30);
            reponse.append(db.lastError().text());
        }

    }

    return reponse;
}
```

```
/******************************************** GESTION MOTS DE PASSE
*********************************************/

// Administration : gestion des mots de passe
QByteArray ComposantBaseDeDonnees::admin_gererMotDePasse(const QString mode,
const int id, const QString nom, const QString identifiant, const QString
motDePasse, const int serveur) {
    QByteArray reponse;

    if(mode == "creation") {

        // Cryptage du mot de passe
        QString motDePasseCrypte = QString(compCrypto-
>encrypterRSA(motDePasse.toUtf8()).toHex());

        // Préparation de la requête - création mot de passe
        QSqlQuery query(db);
        query.prepare("INSERT INTO motsdepasse (nom, identifiant, motdepasse,
id_serveur) "
                      "               VALUES (:nom, :identifiant,
:motdepasse, :id_serveur)");
        query.bindValue(":nom", nom, QSql::In);
        query.bindValue(":identifiant", identifiant, QSql::In);
        query.bindValue(":motdepasse", motDePasseCrypte, QSql::In);
        query.bindValue(":id_serveur", serveur, QSql::In);

        // Exécution de la requête
        if(query.exec()) reponse = "creationok";
        else {
            reponse = "creationerreurtechnique";
            reponse.append(30);
            reponse.append(query.lastError().text());
        }
    }
    else if(mode == "modification") {

        // Cryptage du mot de passe
        QString motDePasseCrypte = QString(compCrypto-
>encrypterRSA(motDePasse.toUtf8()).toHex());

        // Préparation de la requête - modification mot de passe
        QSqlQuery query(db);
        query.prepare("UPDATE motsdepasse "
                      "SET nom = :nom, identifiant = :identifiant, motdepasse
= :motdepasse, id_serveur = :id_serveur "
                      "WHERE id_motdepasse = :id_motdepasse");
        query.bindValue(":nom", nom, QSql::In);
        query.bindValue(":identifiant", identifiant, QSql::In);
        query.bindValue(":motdepasse", motDePasseCrypte, QSql::In);
        query.bindValue(":id_serveur", serveur, QSql::In);
        query.bindValue(":id_motdepasse", id, QSql::In);

        // Exécution de la requête
        if(query.exec()) {
            if(query.numRowsAffected() == 1) reponse = "modificationok";
            else reponse = "modificationerreurinexistant";
        }
        else {
            reponse = "modificationerreurtechnique";
            reponse.append(30);
            reponse.append(query.lastError().text());
        }
    }
    else if(mode == "suppression") {
```

```
        // Plusieurs requêtes en modification ont lieu : on démarre une
transaction
        db.transaction();

        // Préparation et exécution de la requête - suppression mot de passe
        QSqlQuery query1(db);
        query1.prepare("DELETE FROM motsdepasse "
                       "WHERE id_motdepasse = :id_motdepasse");
        query1.bindValue(":id_motdepasse", id, QSql::In);
        query1.exec();

        // Si le mot de passe existait
        if(query1.numRowsAffected() == 1) {

            // Préparation et exécution de la requête - suppression liens
avec groupes du mot de passe
            QSqlQuery query2(db);
            query2.prepare("DELETE FROM groupes_motsdepasse "
                           "WHERE id_motdepasse = :id_motdepasse");
            query2.bindValue(":id_motdepasse", id, QSql::In);
            query2.exec();

            // Fin de transaction
            if(db.commit())
                reponse = "suppressionok";
            else {
                db.rollback();
                reponse = "suppressionerreurtechnique";
                reponse.append(30);
                reponse.append(db.lastError().text());
            }
        }
        // Si le mot de passe n'existait pas
        else {
            db.rollback();
            reponse = "suppressionerreurinexistant";
        }
    }

    return reponse;
}
```

```
// Administration : gestion des groupes d'un mot de passe
QByteArray ComposantBaseDeDonnees::admin_gererGroupesMotDePasse(const int
idMotDePasse, const QVector<int> groupes) {
    QByteArray reponse;

    // Vérification de l'existence du mot de passe
    QSqlQuery query(db);
    query.prepare("SELECT * FROM motsdepasse "
                  "WHERE id_motdepasse = :id_motdepasse");
    query.bindValue(":id_motdepasse", idMotDePasse, QSql::In);
    query.exec();

    // Si le mot de passe n'existe pas
    if(!query.next()) reponse = "groupesmotdepasseerreurinexistant";

    // Si le mot de passe existe bien
    else {

        // Plusieurs requêtes en modification ont lieu : on démarre une
transaction
        db.transaction();

        // Préparation et exécution de la requête - suppression liens avec
groupes du mot de passe
        QSqlQuery query1(db);
        query1.prepare("DELETE FROM groupes_motsdepasse "
                       "WHERE id_motdepasse = :id_motdepasse");
        query1.bindValue(":id_motdepasse", idMotDePasse, QSql::In);
        query1.exec();

        // Préparation de la requête - ajout liens avec nouveaux groupes
        QSqlQuery query2(db);
        query2.prepare("INSERT INTO groupes_motsdepasse (id_groupe,
id_motdepasse) "
                       "                          VALUES (:id_groupe,
:id_motdepasse)");
        query2.bindValue(":id_motdepasse", idMotDePasse, QSql::In);

        // Exécution de la requête pour chaque id de nouveau groupe
        for(int g = 0; g < groupes.size(); g++) {
            query2.bindValue(":id_groupe", groupes.at(g), QSql::In);
            query2.exec();
        }

        // Fin de transaction
        if(db.commit())
            reponse = "groupesmotdepasseok";
        else {
            db.rollback();
            reponse = "groupesmotdepasseerreurtechnique";
            reponse.append(30);
            reponse.append(db.lastError().text());
        }
    }

    return reponse;
}
```

```
/******************************************** GESTION SERVEURS
*********************************************/

// Administration : gestion des serveurs
QByteArray ComposantBaseDeDonnees::admin_gererServeur(const QString mode,
const int id, const QString nom, const QString hote) {
    QByteArray reponse;

    if(mode == "creation") {
        // Préparation de la requête - création serveur
        QSqlQuery query(db);
        query.prepare("INSERT INTO serveurs (nom, hote) "
                      "             VALUES (:nom, :hote)");
        query.bindValue(":nom", nom, QSql::In);
        query.bindValue(":hote", hote, QSql::In);

        // Exécution de la requête
        if(query.exec()) reponse = "creationok";
        else {
            reponse = "creationerreurtechnique";
            reponse.append(30);
            reponse.append(query.lastError().text());
        }
    }
    else if(mode == "modification") {
        // Préparation de la requête - modification serveur
        QSqlQuery query(db);
        query.prepare("UPDATE serveurs "
                      "SET nom = :nom, hote = :hote "
                      "WHERE id_serveur = :id_serveur");
        query.bindValue(":nom", nom, QSql::In);
        query.bindValue(":hote", hote, QSql::In);
        query.bindValue(":id_serveur", id, QSql::In);

        // Exécution de la requête
        if(query.exec()) {
            if(query.numRowsAffected() == 1) reponse = "modificationok";
            else reponse = "modificationerreurinexistant";
        }
        else {
            reponse = "modificationerreurtechnique";
            reponse.append(30);
            reponse.append(query.lastError().text());
        }
    }
    else if(mode == "suppression") {

        // Plusieurs requêtes en modification ont lieu : on démarre une
transaction
        db.transaction();

        // Préparation de la requête - suppression serveur
        QSqlQuery query1(db);
        query1.prepare("DELETE FROM serveurs "
                       "WHERE id_serveur = :id_serveur");
        query1.bindValue(":id_serveur", id, QSql::In);
        query1.exec();

        // Si le serveur existait bien
        if(query1.numRowsAffected() == 1) {

            // Préparation de la requête - liste mots de passe du serveur
            QSqlQuery query2(db);
            query2.prepare("SELECT id_motdepasse FROM motsdepasse "
```

```
                        "WHERE id_serveur = :id_serveur");
            query2.bindValue(":id_serveur", id, QSql::In);
            query2.exec();

            // Préparation de la requête - suppression liens avec groupes des
mots de passe
            QSqlQuery query3(db);
            query3.prepare("DELETE FROM groupes_motsdepasse "
                        "WHERE id_motdepasse = :id_motdepasse");

            // Exécution de la requête pour chaque mot de passe du serveur
supprimé
            while(query2.next()) {
                query3.bindValue(":id_motdepasse", query2.value(0).toInt());
                query3.exec();
            }

            // Préparation et exécution de la requête - suppression mots de
passe du serveur supprimé
            QSqlQuery query4(db);
            query4.prepare("DELETE FROM motsdepasse "
                        "WHERE id_serveur = :id_serveur");
            query4.bindValue(":id_serveur", id, QSql::In);
            query4.exec();

            // Fin de transaction
            if(db.commit())
                reponse = "suppressionok";
            else {
                db.rollback();
                reponse = "suppressionerreurtechnique";
                reponse.append(30);
                reponse.append(db.lastError().text());
            }

        }
        // Si le serveur n'existait pas
        else {
            db.rollback();
            reponse = "suppressionerreurinexistant";
        }
    }

    return reponse;
}
```

6. ComposantDocumentation

Ce composant permet d'afficher une aide en ligne de commande.

a. ComposantDocumentation .h

```
#ifndef COMPOSANTDOCUMENTATION_H
#define COMPOSANTDOCUMENTATION_H

class ComposantDocumentation
{
public:
    ComposantDocumentation();

    static void aide();
};

#endif // COMPOSANTDOCUMENTATION_H
```

b. ComposantDocumentation .cpp

```
#include "ComposantDocumentation.h"

#include <QTextStream>
#include <QDir>

ComposantDocumentation::ComposantDocumentation()
{
}
```

```
void ComposantDocumentation::aide() {
    QTextStream out(stdout);

    out << endl << "Usage :" << endl << endl;

    out << "  spmd" << endl;

    out << "    Lance le serveur avec comme fichier de configuration par
defaut : " << QDir::homePath() << "/spmd/spmd.conf" << endl << endl;

    out << "  spmd --config <fichier>" << endl;
    out << "    Lance le serveur avec le fichier de configuration fourni en
parametre." << endl << endl;

    out << "  spmd --install" << endl;
    out << "       [--config <fichier>]" << endl;
    out << "       [--priv-key <fichier>]" << endl;
    out << "       [--pub-key <fichier>]" << endl;
    out << "       [--database <fichier>]" << endl;
    out << "       [--port <port>]" << endl;
    out << "       [--use-existing-keys]" << endl << endl;

    out << "    Installe le serveur avec les parametres fournis." << endl;
    out << "    Le fichier de configuration est renseigne avec --config, ou
est " << QDir::homePath() << "/spmd/spmd.conf par defaut." << endl;
    out << "    Le fichier de cle privee RSA est renseigne avec --priv-key,
ou est " << QDir::homePath() << "/spmd/private_key.pem par defaut." << endl;
    out << "    Le fichier de cle publique RSA est renseigne avec --pub-key,
ou est " << QDir::homePath() << "/spmd/public_key.pem par defaut." << endl;
    out << "    Le fichier de base de donnees est renseigne avec --database,
ou est " << QDir::homePath() << "/spmd/database.sqlite3 par defaut." << endl;
    out << "    Le port d'ecoute du serveur est renseigne avec --port, ou est
3665 par defaut" << endl;
    out << "    Les cles RSA seront generees aleatoirement sauf si presence
du parametre --use-existing-keys." << endl;
    out << "    Le processus d'installation demandera confirmation en cas de
fichiers existants." << endl << endl;

    out << "  spmd --backup" << endl;
    out << "       --file <fichier>" << endl;
    out << "       [--config <fichier>]" << endl << endl;

    out << "    Effectue une sauvegarde des cles et base de donnees dans un
fichier crypte." << endl;
    out << "    Le fichier de configuration est renseigne avec --config, ou
est " << QDir::homePath() << "/spmd/spmd.conf par defaut." << endl;
    out << "    Le fichier est crypte avec un mot de passe genere
aleatoirement qui sera affiche." << endl << endl;

    out << "  spmd --restore" << endl;
    out << "       --file <fichier>" << endl;
    out << "       --password <motdepasse>" << endl;
    out << "       [--config <fichier>]" << endl << endl;

    out << "    Restaure la sauvegarde a partir du fichier crypte." << endl;
    out << "    Le fichier de configuration a ecrire est renseigne avec --
config, ou est " << QDir::homePath() << "/spmd/spmd.conf par defaut." << endl
<< endl;

    out << "  spmd --help" << endl << endl;
    out << "    Affiche cette aide." << endl << endl;
}
```

II. Code source du client

1. main.cpp

Ce fichier est le fichier principal du serveur : il permet de charger le composant principal.

```
#include "../src/ComposantPrincipal.h"
#include <QApplication>

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

    ComposantPrincipal w;
    w.show();

    return a.exec();
}
```

2. ComposantPrincipal

Ce composant permet d'effectuer les traitements relatifs à l'interface principale du logiciel : connexion, affichage des mots de passe, ouverture des fenêtres d'administration.

a. ComposantPrincipal .h

```
#ifndef COMPOSANTPRINCIPAL_H
#define COMPOSANTPRINCIPAL_H

#include <QMainWindow>
#include <QFile>

namespace Ui {
class ComposantPrincipal;
}

class ComposantReseau;
class ComposantSauvegarde;
class GestionGroupes;
class GestionMotsDePasse;
class GestionServeurs;
class GestionUtilisateurs;
class ComposantCryptographie;

class ComposantPrincipal : public QMainWindow
{
    Q_OBJECT

public:
    ComposantPrincipal(QWidget *parent = 0);
    ~ComposantPrincipal();

private slots:

    void on_enregistrerPageConfiguration_clicked();
    void on_connexionPageConnexion_clicked();
    void on_actionDeconnexion_triggered();
    void on_actionQuitter_triggered();
    void on_actionGestionUtilisateurs_triggered();
    void on_actionGestionServeurs_triggered();
    void on_actionGestionMDP_triggered();
    void on_actionGestionGroupes_triggered();
    void on_actionSauvegarder_triggered();
    void on_actualiser_clicked();
    void on_afficherMasquer_clicked();
    void on_listeMotsDePasse_clicked(const QModelIndex &index);
    void on_identifiantPageConnexion_returnPressed();
    void on_mdpPageConnexion_returnPressed();

private:

    struct MotDePasse {
        QString nomServeur;
        QString hoteServeur;
        QString nom;
        QString identifiant;
        QString motDePasse;
    };

    Ui::ComposantPrincipal *ui;
    ComposantReseau *compRes;
```

```
    ComposantSauvegarde *compSave;
    GestionGroupes *gestGroupe;
    GestionMotsDePasse *gestMDP;
    GestionServeurs *gestServ;
    GestionUtilisateurs *gestUtil;
    ComposantCryptographie *compCrypt;
    QString ipServ;
    int portServ;
    QString cleServ;
    QFile fichierConfig;
    int idUtilisateur;
    QString nomUtilisateur;
    bool estAdministrateur;

    QVector<MotDePasse> listeMotsDePasse;

    void reinitialiser();
    void rafraichirListeMotsDePasse();
};

#endif // COMPOSANTPRINCIPAL_H
```

b. ComposantPrincipal .cpp

```
#include "ComposantPrincipal.h"
#include "ComposantReseau.h"
#include "ComposantSauvegarde.h"
#include "GestionGroupes.h"
#include "GestionMotsDePasse.h"
#include "GestionServeurs.h"
#include "GestionUtilisateurs.h"
#include "../../Serveur/src/ComposantCryptographie.h"

#ifdef Q_OS_WIN
#include "ui_ComposantPrincipal_win.h"
#else
#include "ui_ComposantPrincipal_unix.h"
#endif

#include <QMessageBox>
#include <QDir>

ComposantPrincipal::ComposantPrincipal(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::ComposantPrincipal),
    compRes(NULL), compSave(NULL), gestGroupe(NULL),
    gestMDP(NULL), gestServ(NULL), gestUtil(NULL)
{

    // Initialisation des variables membre
    ui->setupUi(this);
    reinitialiser();

    // Lecture du fichier de configuration
    fichierConfig.setFileName(QDir::homePath() + QString("/spm/spm.conf"));

    if(fichierConfig.exists()) {
        if (!fichierConfig.open(QIODevice::ReadOnly | QIODevice::Text)) {
            QMessageBox::critical(this, "Erreur fichier", QString("Impossible
d'ouvrir le fichier de configuration :\n\n")
                                                                +
fichierConfig.fileName());
            exit(1);
        }
        if(fichierConfig.size() != 0) {
            QTextStream config(&fichierConfig);
            QString line;
            while (!config.atEnd()) {
                line = config.readLine();
                QStringList list = line.split("=");
                if(list[0] == "serveur") ipServ = list.at(1).trimmed();
                else if(list[0] == "port") portServ = list.at(1).toInt();
                else if(list[0] == "cleServeur") cleServ =
list.at(1).trimmed();
            }

            // Affichage de la page de connexion
            ui->stackedWidget->setCurrentIndex(0);
            return;
        }
        else fichierConfig.close();
    }
    // Aucun fichier de configuration : installation du client
    ui->stackedWidget->setCurrentIndex(2);
}
```

```
// Réinitialisation des variables membre et interfaces
void ComposantPrincipal::reinitialiser() {
    idUtilisateur = -1;
    nomUtilisateur = "";
    estAdministrateur = false;
    listeMotsDePasse.clear();
    ui->listeMotsDePasse->clear();
    ui->stackedWidget->setCurrentIndex(0);
    ui->menuAdministration->menuAction()->setVisible(false);
    ui->actionDeconnexion->setEnabled(false);
    ui->connexionPageConnexion->setEnabled(true);
    ui->connexionPageConnexion->setText("Connecter");
    ui->identifiantPageConnexion->clear();
    ui->mdpPageConnexion->clear();
}

// Destruction des composants
ComposantPrincipal::~ComposantPrincipal()
{
    delete ui;
    if(compRes != NULL) delete compRes;
    if(compSave != NULL) delete compSave;
    if(gestGroupe != NULL) delete gestGroupe;
    if(gestMDP != NULL) delete gestMDP;
    if(gestServ != NULL) delete gestServ;
    if(gestUtil != NULL) delete gestUtil;
}
```

```
// Ecriture de la configuration client
void ComposantPrincipal::on_enregistrerPageConfiguration_clicked()
{
    // L'IP et le port sont récuperés
    ipServ = ui->serveurPageConfiguration->text();
    portServ = ui->portPageConfiguration->value();

    // On vérifie l'existence du dossier spm
    QDir pathToConf(QDir::homePath() + QString("/spm"));
    if(!pathToConf.exists()) {
        if(!pathToConf.mkpath(QDir::homePath() + QString("/spm"))) {
            QMessageBox::critical(this, "Erreur système de fichiers",
QString("Impossible de créer le dossier spm :\n\n")
                                                                    +
QDir::homePath() + QString("/spm"));
            return;
        }
    }

    // On ouvre le fichier
    if (!fichierConfig.open(QFile::WriteOnly | QFile::Truncate)) {
        QMessageBox::critical(this, "Erreur fichier", QString("Impossible
d'ouvrir le fichier de configuration :\n\n")
                                                        +
fichierConfig.fileName());
        return;
    }
    QTextStream sortie(&fichierConfig);

    // Téléchargement de la clé publique du serveur
    compRes = new ComposantReseau(ipServ,portServ,NULL);
    QByteArray cle = compRes->getClePubliqueRSA();
    if(cle.size() == 0) {
        fichierConfig.close();
        delete compRes;
        compRes = NULL;
        QMessageBox::critical(this, "Erreur réseau", "Impossible de
télécharger la clé publique RSA.");
        return;
    }

    // Enregistrement de la clé publique du serveur
    cleServ = QDir::homePath() + QString("/spm/server-pubkey.pem");
    QFile fichierCleRSA(cleServ);
    if(!fichierCleRSA.open(QFile::WriteOnly | QFile::Truncate)) {
        fichierConfig.close();
        delete compRes;
        compRes = NULL;
        QMessageBox::critical(this, "Erreur réseau", QString("Impossible
d'écrire dans le fichier ") + cleServ);
        return;
    }
    fichierCleRSA.write(cle);
    fichierCleRSA.close();

    // On écrit la configuration
    sortie << "serveur=" << ipServ << endl;
    sortie << "port=" << portServ << endl;
    sortie << "cleServeur=" << cleServ << flush;
    fichierConfig.close();

    // On affiche la page de connexion
    ui->stackedWidget->setCurrentIndex(0);
}
```

```
// Connexion de l'utilisateur
void ComposantPrincipal::on_connexionPageConnexion_clicked()
{

    // Création des composants de cryptographie et réseau
    compCrypt = new ComposantCryptographie();
    compCrypt->loadClePubliqueRSA(cleServ);
    compRes = new ComposantReseau(ipServ,portServ,compCrypt);

    // Connexion au serveur
    ui->connexionPageConnexion->setEnabled(false);
    ui->connexionPageConnexion->setText("Connexion en cours");
    if(!compRes->connecter()) {
        QMessageBox::critical(this, "Erreur réseau", "Le serveur ne répond
pas.\nVérifiez votre connexion réseau.", QMessageBox::Ok);
        ui->connexionPageConnexion->setEnabled(true);
        ui->connexionPageConnexion->setText("Connecter");
        delete compRes;
        compRes = NULL;
        delete compCrypt;
        compCrypt = NULL;
        return;
    }

    // Création message de connexion
    QString user = ui->identifiantPageConnexion->text();
    QString pass = ui->mdpPageConnexion->text();
    QByteArray message = "connexion";
    message.append(31);
    message.append(user);
    message.append(31);
    message.append(pass);
    QByteArray reponse = compRes->envoyerMessage(message);

    QList<QByteArray> champs = reponse.split(31);

    // Vérification de la réponse du serveur
    if(champs.at(0) == "connexionacceptee") {
        if(champs.at(1) == "admin") {
            ui->menuAdministration->menuAction()->setVisible(true);
            gestUtil = new GestionUtilisateurs(compRes, this);
            gestServ = new GestionServeurs(compRes, this);
            gestMDP = new GestionMotsDePasse(compRes, this);
            gestGroupe = new GestionGroupes(compRes, this);
            compSave = new ComposantSauvegarde(compRes, this);
            estAdministrateur = true;
        }
        idUtilisateur = champs.at(2).toInt();
        nomUtilisateur = champs.at(3);
        ui->identifiantPageConnexion->setFocus();
        ui->actionDeconnexion->setEnabled(true);
        ui->stackedWidget->setCurrentIndex(1);
        ui->mdpPageConnexion->clear();
        rafraichirListeMotsDePasse();
    }
    else if(champs.at(0) == "connexionrefusee") {
        QMessageBox::critical(this, "Connexion refusée", "Connexion
refusée.\nVérifiez vos informations de connexion.", QMessageBox::Ok);
        delete compRes;
        delete compCrypt;
    }
```

```
    else {
        QMessageBox::critical(this, "Erreur inconnue", reponse,
QMessageBox::Ok);
        delete compRes;
        delete compCrypt;
    }
    ui->connexionPageConnexion->setEnabled(true);
    ui->connexionPageConnexion->setText("Connecter");
}


// Récupération de la liste des mots de passe accessibles à l'utilisateur
void ComposantPrincipal::rafraichirListeMotsDePasse() {

    // Envoi de la requête et réception de la réponse
    QByteArray message = "listemotsdepasse";
    QByteArray reponse = compRes->envoyerMessage(message);

    QList<QByteArray> lignes = reponse.split(30);

    listeMotsDePasse.clear();
    ui->listeMotsDePasse->clear();

    // Affichage des résultats
    if(lignes.at(0) == "listemotsdepasse" && lignes.at(1).size() > 0) {
        QList<QByteArray> champs;
        MotDePasse motDePasse;

        for(int m = 1; m < lignes.size(); m++) {
            champs = lignes.at(m).split(31);
            motDePasse.nomServeur = champs.at(0).trimmed();
            motDePasse.hoteServeur = champs.at(1).trimmed();
            motDePasse.nom = champs.at(2).trimmed();
            motDePasse.identifiant = champs.at(3).trimmed();
            motDePasse.motDePasse = champs.at(4).trimmed();
            listeMotsDePasse.append(motDePasse);
            ui->listeMotsDePasse->addItem(motDePasse.nomServeur + " : " +
                                          motDePasse.nom + " @ " +
motDePasse.hoteServeur);
        }
    }

    // Initialisations des zones d'affichage
    ui->labelNomServeur->clear();
    ui->labelHoteServeur->clear();
    ui->labelNom->clear();
    ui->labelIdentifiant->clear();
    ui->motDePasse->clear();
    ui->motDePasse->setEchoMode(QLineEdit::Password);
    ui->afficherMasquer->setText("Afficher");
    ui->afficherMasquer->setEnabled(false);

}
```

```
// Sélection d'un mot de passe : affichage des informations
void ComposantPrincipal::on_listeMotsDePasse_clicked(const QModelIndex
&index)
{
    int i = index.row();
    if(i == -1 || i >= listeMotsDePasse.size()) return;

    MotDePasse motDePasse = listeMotsDePasse.at(i);

    // Affichage des informations du mot de passe
    ui->labelNomServeur->setText(motDePasse.nomServeur);
    ui->labelHoteServeur->setText(motDePasse.hoteServeur);
    ui->labelNom->setText(motDePasse.nom);
    ui->labelIdentifiant->setText(motDePasse.identifiant);
    ui->motDePasse->setText(motDePasse.motDePasse);

    // Le mot de passe est masqué
    ui->motDePasse->setEchoMode(QLineEdit::Password);
    ui->afficherMasquer->setText("Afficher");
    ui->afficherMasquer->setEnabled(true);
}


// Action du menu Fichier - Déconnexion
void ComposantPrincipal::on_actionDeconnexion_triggered()
{
    if(idUtilisateur != -1) {
        // Le composant réseau envoie le message de déconnexion lors de sa
destruction
        if(estAdministrateur) {
            delete gestUtil;
            gestUtil = NULL;
            delete gestGroupe;
            gestGroupe = NULL;
            delete gestMDP;
            gestMDP = NULL;
            delete gestServ;
            gestServ = NULL;
            delete compSave;
            compSave = NULL;
        }
        if(compRes != NULL) {
            delete compRes;
            compRes = NULL;
        }
        if(compCrypt != NULL) {
            delete compCrypt;
            compCrypt = NULL;
        }
        reinitialiser();
    }

}

// Action du menu Fichier - Quitter
void ComposantPrincipal::on_actionQuitter_triggered()
{
    on_actionDeconnexion_triggered();
    this->close();
}
```

```
// Action du menu Administration - Gestion des utilisateurs
void ComposantPrincipal::on_actionGestionUtilisateurs_triggered()
{
    // Actualisation de la liste si nécessaire
    if(!gestUtil->isVisible()) gestUtil->rafraichirListeUtilisateurs();
    gestUtil->show();
}

// Action du menu Administration - Gestion des serveurs
void ComposantPrincipal::on_actionGestionServeurs_triggered()
{
    // Actualisation de la liste si nécessaire
    if(!gestServ->isVisible()) gestServ->rafraichirListeServeurs();
    gestServ->show();
}

// Action du menu Administration - Gestion des mots de passe
void ComposantPrincipal::on_actionGestionMDP_triggered()
{
    // Actualisation de la liste si nécessaire
    if(!gestMDP->isVisible()) gestMDP->rafraichirListeMotsDePasse();
    gestMDP->show();
}

// Action du menu Administration - Gestion des groupes
void ComposantPrincipal::on_actionGestionGroupes_triggered()
{
    // Actualisation de la liste si nécessaire
    if(!gestGroupe->isVisible()) gestGroupe->rafraichirListeGroupes();
    gestGroupe->show();
}

// Action du menu Administration - Sauvegarde du serveur
void ComposantPrincipal::on_actionSauvegarder_triggered()
{
    compSave->show();
}

// Action du bouton Actualiser pour la liste des mots de passe
void ComposantPrincipal::on_actualiser_clicked()
{
    rafraichirListeMotsDePasse();
}

// Action du bouton Afficher/Masquer pour le mot de passe sélectionné
void ComposantPrincipal::on_afficherMasquer_clicked()
{
    if(ui->listeMotsDePasse->currentRow() == -1) return;

    // Si le mot de passe est masqué on l'affiche, et inversement
    if(ui->motDePasse->echoMode() == QLineEdit::Password) {
        ui->motDePasse->setEchoMode(QLineEdit::Normal);
        ui->afficherMasquer->setText("Masquer");
    }
    else {
        ui->motDePasse->setEchoMode(QLineEdit::Password);
        ui->afficherMasquer->setText("Afficher");
    }
}
```

```
// La touche Entrée fait passer au champ mot de passe si on est dans le champ
identifiant
void ComposantPrincipal::on_identifiantPageConnexion_returnPressed()
{
    ui->mdpPageConnexion->setFocus();
}

// La touche Entrée lance la connexion si on est dans le champ mot de passe
void ComposantPrincipal::on_mdpPageConnexion_returnPressed()
{
    on_connexionPageConnexion_clicked();
}
```

3. ComposantReseau

Ce composant permet d'établir la connexion au serveur, de transmettre les requêtes et récupérer les réponses aux requêtes.

a. ComposantReseau.h

```
#ifndef COMPOSANTRESEAU_H
#define COMPOSANTRESEAU_H

#include <QObject>
#include <QTcpSocket>

class ComposantCryptographie;

class ComposantReseau : public QObject
{
    Q_OBJECT

public:
    ComposantReseau(const QString ip, const int port, ComposantCryptographie
* c);
    ~ComposantReseau();

    bool connecter();

    QByteArray envoyerMessage(const QByteArray message);
    QByteArray getClePubliqueRSA();

private:
    QTcpSocket * socket;
    ComposantCryptographie * compCrypt;
    QString ipServ;
    int portServ;

    QByteArray envoyerMessage_private(QByteArray message);

    QByteArray intToByteArray(unsigned int valeur);
    unsigned int byteArrayToInt(const QByteArray ba);
};

#endif // COMPOSANTRESEAU_H
```

b. ComposantReseau.cpp

```
#include "ComposantReseau.h"
#include "../../Serveur/src/ComposantCryptographie.h"

#include <QThread>
#include <QMessageBox>
#include <cmath>

// Constructeur de la connexion au serveur avec ip et port
ComposantReseau::ComposantReseau(const QString ip, const int port,
ComposantCryptographie * c) :
    socket(new QTcpSocket()),
    compCrypt(c),
    ipServ(ip),
    portServ(port)
{
}

// Destructeur
ComposantReseau::~ComposantReseau() {
    // On ferme la socket si elle est ouverte
    if(compCrypt && socket->isOpen()) {
        envoyerMessage("disconnect");
        socket->disconnectFromHost();
        if(socket->state() != QAbstractSocket::UnconnectedState)
            socket->waitForDisconnected(5000);
    }
    delete socket;
}

// Télécharge et renvoie la clé publique RSA
QByteArray ComposantReseau::getClePubliqueRSA() {
    socket->connectToHost(ipServ, portServ);
    if(socket->waitForConnected()) {
        QByteArray reponse = envoyerMessage_private("clepubliquersa");
        return reponse;
    }
    return QByteArray();
}
```

```
// Ouvre la connexion et initie le cryptage AES
bool ComposantReseau::connecter() {

    // Ouverture connexion
    socket->connectToHost(ipServ, portServ);
    if(socket->waitForConnected()) {

        // Génération clé AES
        compCrypt->genererCleAES();

        // Préparation message de clé AES
        QByteArray messageCleAES = "cleaes";
        messageCleAES.append(compCrypt->getCleAES());
        if(messageCleAES.size() == 6) return false;

        // Cryptage du message en RSA et envoi
        QByteArray messageCleAESCrypte = compCrypt-
>encrypterRSA(messageCleAES);
        QByteArray reponse = envoyerMessage_private(messageCleAESCrypte);

        return reponse == "cleaesok";
    }
    return false;
}


// Envoie un message au serveur après l'avoir crypté en AES
QByteArray ComposantReseau::envoyerMessage(const QByteArray message) {

    // Cryptage du message en AES
    QByteArray messageCrypte = compCrypt->encrypterAES(message);

    // Envoi et décryptage de la réponse
    QByteArray reponseCryptee = envoyerMessage_private(messageCrypte);
    QByteArray reponse = compCrypt->decrypterAES(reponseCryptee);

    return reponse;
}


// Envoie un message au serveur
QByteArray ComposantReseau::envoyerMessage_private(QByteArray message) {
    QByteArray reponse;

    // Ajout de la taille du message + 4 au début
    message.prepend(intToByteArray(message.size() + 4));

    // Envoi du message
    socket->write(message);
    socket->flush();

    // Réception de tous les blocs de données de la réponse
    socket->waitForReadyRead();
    reponse.append(socket->readAll());

    int taille = byteArrayToInt(reponse.mid(0, 4));

    while(reponse.size() != taille) {
        socket->waitForReadyRead();
        reponse.append(socket->readAll());
    }

    return reponse.mid(4);
}
```

```
// Convertit un entier en un tableau d'octets
QByteArray ComposantReseau::intToByteArray(unsigned int valeur) {
    QByteArray ba;

    unsigned char aux;

    for(int c = 3; c >= 0; c--) {
        aux = (unsigned char)(valeur / (unsigned int)pow(256, c));
        ba.append(aux);
        valeur -= (unsigned int)aux * (unsigned int)pow(256, c);
    }

    return ba;
}


// Convertit un tableau d'octets en un entier
unsigned int ComposantReseau::byteArrayToInt(const QByteArray ba) {
    unsigned int valeur = 0;

    if(ba.size() != 4) return valeur;

    unsigned char aux;

    for(int c = 3; c >= 0; c--) {
        aux = ba.at(3 - c);
        valeur += (unsigned int)aux * (unsigned int)pow(256, c);
    }

    return valeur;
}
```

4. GestionUtilisateurs

Ce composant gère les actions et traitements liés à la gestion des utilisateurs.

a. GestionUtilisateurs.h

```
#ifndef GESTIONUTILISATEURS_H
#define GESTIONUTILISATEURS_H

#include <QDialog>

class ComposantReseau;

namespace Ui {
class GestionUtilisateurs;
}

class GestionUtilisateurs : public QDialog
{
    Q_OBJECT

public:
    explicit GestionUtilisateurs(ComposantReseau *p_compRes, QWidget *parent
= 0);
    ~GestionUtilisateurs();

    void rafraichirListeUtilisateurs();

private slots:
    void on_creer_clicked();
    void on_modifier_clicked();
    void on_supprimer_clicked();
    void on_listeUtilisateurs_currentRowChanged(int currentRow);
    void on_listeUtilisateurs_doubleClicked(const QModelIndex &index);
    void on_gererGroupes_clicked();
    void on_validerUtilisateur_clicked();
    void on_annulerUtilisateur_clicked();
    void on_ajouterGroupe_clicked();
    void on_enleverGroupe_clicked();
    void on_validerGroupes_clicked();
    void on_annulerGroupes_clicked();
    void on_actualiser_clicked();

private:

    Ui::GestionUtilisateurs *ui;
    ComposantReseau *compRes;

    struct Utilisateur {
        int id;
        QString nom;
        QString identifiant;
        QString motDePasse;
        bool administrateur;
    };

    struct Groupe {
        int id;
        QString nom;
    };

    Utilisateur utilisateur;
```

```
    QVector<Utilisateur> listeUtilisateurs;
    QVector<Groupe> listeGroupes;
    QVector<Groupe> listeGroupesUtilisateur;

    void closeEvent(QCloseEvent *);
    void keyPressEvent(QKeyEvent *);
};

#endif // GESTIONUTILISATEURS_H
```

b. GestionUtilisateurs.cpp

```
#include "GestionUtilisateurs.h"
#include "ComposantReseau.h"

#ifdef Q_OS_WIN
#include "ui_GestionUtilisateurs_win.h"
#else
#include "ui_GestionUtilisateurs_unix.h"
#endif

#include <QMessageBox>
#include <QCloseEvent>
#include <QKeyEvent>

GestionUtilisateurs::GestionUtilisateurs(ComposantReseau *p_compRes, QWidget
*parent) :
    QDialog(parent,
Qt::WindowSystemMenuHint|Qt::WindowCloseButtonHint|Qt::WindowMinimizeButtonHi
nt),
    ui(new Ui::GestionUtilisateurs),
    compRes(p_compRes)
{
    ui->setupUi(this);
}

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

// Récupère la liste des utilisateurs du système
void GestionUtilisateurs::rafraichirListeUtilisateurs() {

    // Envoi de la requête et réception de la réponse
    QByteArray message = "admin_listeutilisateurs";
    QByteArray reponse = compRes->envoyerMessage(message);

    QList<QByteArray> lignes = reponse.split(30);

    listeUtilisateurs.clear();
    ui->listeUtilisateurs->clear();

    // Affichage des résultats
    if(lignes.at(0) == "admin_listeutilisateurs" && lignes.at(1).size() > 0){
        QList<QByteArray> champs;

        for(int m = 1; m < lignes.size(); m++) {
            champs = lignes.at(m).split(31);
            utilisateur.id = champs.at(0).trimmed().toInt();
            utilisateur.nom = champs.at(1).trimmed();
            utilisateur.identifiant = champs.at(2).trimmed();
            utilisateur.motDePasse = champs.at(3).trimmed();
            utilisateur.administrateur = champs.at(4).trimmed() == "admin";
            listeUtilisateurs.append(utilisateur);
            ui->listeUtilisateurs->addItem(utilisateur.nom + " (" +
utilisateur.identifiant + ")" + (utilisateur.administrateur ? ",
administrateur" : ""));
        }
    }

    utilisateur.motDePasse.clear();
    ui->listeUtilisateurs->setCurrentRow(-1);
}
```

```
// Entre en mode création d'un utilisateur
void GestionUtilisateurs::on_creer_clicked()
{
    ui->stackedWidget->setCurrentIndex(1);
    ui->titreCreerModifier->setText("Créer un utilisateur");

    utilisateur.id = -1;
    ui->nom->clear();
    ui->identifiant->clear();
    ui->motdepasse->clear();
    ui->administrateur->setChecked(false);
}


// Entre en mode modification d'un utilisateur
void GestionUtilisateurs::on_modifier_clicked()
{
    int i = ui->listeUtilisateurs->currentRow();
    if(i < 0 || i >= listeUtilisateurs.size()) return;

    ui->stackedWidget->setCurrentIndex(1);
    ui->titreCreerModifier->setText("Modifier un utilisateur");

    utilisateur = listeUtilisateurs.at(i);
    ui->nom->setText(utilisateur.nom);
    ui->identifiant->setText(utilisateur.identifiant);
    ui->motdepasse->setText(utilisateur.motDePasse);
    ui->administrateur->setChecked(utilisateur.administrateur);
}
```

```
// Suppression d'un utilisateur
void GestionUtilisateurs::on_supprimer_clicked()
{
    int i = ui->listeUtilisateurs->currentRow();
    if(i < 0 || i >= listeUtilisateurs.size()) return;

    // Message de confirmation
    QMessageBox::StandardButton retour;
    retour = QMessageBox::question(this, "Confirmation", "La suppression de
l'utilisateur sera définitive. Confirmer ?");

    if(retour != QMessageBox::Yes) return;

    // Envoi de la requête et réception de la réponse
    QByteArray message = "admin_gererutilisateur";
    message.append(30);
    message.append("suppression");
    message.append(30);
    message.append(QString::number(listeUtilisateurs.at(i).id));

    QByteArray reponse = compRes->envoyerMessage(message);

    // Affichage de l'état de la requête
    if(reponse == "suppressionok") {
        QMessageBox::information(this, "Suppression effectuée",
"L'utilisateur a été supprimé");
        rafraichirListeUtilisateurs();
    }

    else if(reponse == "suppressionerreurinexistant") {
        QMessageBox::critical(this, "Suppression échouée", "L'utilisateur
n'existe pas (id_utilisateur introuvable).");
        rafraichirListeUtilisateurs();
    }

    else if(reponse.startsWith("suppressionerreurtechnique")) {
        QMessageBox::critical(this, "Suppression échouée", "Une erreur
technique a eu lieu :\n\n" + reponse.split(30).at(1));
    }
    else {
        QMessageBox::critical(this, "Erreur serveur", "Pas de réponse ou
réponse malformée du serveur. Veuillez réessayer.");
    }
}

// Entre en mode gestion des groupe de l'utilisateur
void GestionUtilisateurs::on_gererGroupes_clicked()
{
    int i = ui->listeUtilisateurs->currentRow();
    if(i < 0 || i >= listeUtilisateurs.size()) return;

    listeGroupesUtilisateur.clear();
    ui->groupesUtilisateur->clear();
    listeGroupes.clear();
    ui->groupes->clear();

    utilisateur = listeUtilisateurs.at(i);

    ui->labelUtilisateur->setText(utilisateur.nom + " (" +
utilisateur.identifiant + ")");

    QByteArray message;
    QList<QByteArray> lignes;
    QList<QByteArray> champs;
```

```
    Groupe groupe;

    // Récupération des groupes actuels de l'utilisateur
    QByteArray groupesUtilisateur;
    message = "admin_listegroupesutilisateur";
    message.append(30);
    message.append(QString::number(utilisateur.id));
    groupesUtilisateur = compRes->envoyerMessage(message);

    lignes = groupesUtilisateur.split(30);

    // Affichage des résultats
    if(lignes.at(0) == "admin_listegroupesutilisateur" && lignes.at(1).size()
> 0) {
        listeGroupesUtilisateur.clear();
        ui->groupesUtilisateur->clear();
        for(int m = 1; m < lignes.size(); m++) {
            champs = lignes.at(m).split(31);
            groupe.id = champs.at(0).trimmed().toInt();
            groupe.nom = champs.at(1).trimmed();
            listeGroupesUtilisateur.append(groupe);
            ui->groupesUtilisateur->addItem(groupe.nom);
        }
    }

    lignes.clear();
    champs.clear();

    // Récupération des groupes du système
    QByteArray groupes;
    groupes = compRes->envoyerMessage("admin_listegroupes");
    bool trouve;

    lignes = groupes.split(30);

    // Affichage des résultats
    if(lignes.at(0) == "admin_listegroupes" && lignes.at(1).size() > 0) {
        listeGroupes.clear();
        ui->groupes->clear();
        for(int m = 1; m < lignes.size(); m++) {
            champs = lignes.at(m).split(31);
            groupe.id = champs.at(0).trimmed().toInt();
            groupe.nom = champs.at(1).trimmed();

            // On exclue les groupes qui sont dans les groupes de
l'utilisateur
            for(int g = 0; g < listeGroupesUtilisateur.size(); g++) {
                if(listeGroupesUtilisateur.at(g).id == groupe.id) {
                    trouve = true;
                    break;
                }
            }
            if(!trouve) {
                listeGroupes.append(groupe);
                ui->groupes->addItem(groupe.nom);
            }
            trouve = false;
        }
    }

    ui->stackedWidget->setCurrentIndex(2);
}
```

```
// Validation de la création / modification de l'utilisateur
void GestionUtilisateurs::on_validerUtilisateur_clicked() {
    QString mode = utilisateur.id == -1 ? "creation" : "modification";

    // Envoi de la requête et réception de la réponse
    QByteArray message = "admin_gererutilisateur";
    message.append(30);
    message.append(mode);
    message.append(30);
    message.append(QString::number(utilisateur.id));
    message.append(30);
    message.append(ui->nom->text());
    message.append(30);
    message.append(ui->identifiant->text());
    message.append(30);
    message.append(ui->motdepasse->text());
    message.append(30);
    message.append(ui->administrateur->isChecked() ? "true" : "false");

    QByteArray reponse = compRes->envoyerMessage(message);

    // Affichage de l'état de la requête
    if(mode == "creation") {
        if(reponse == "creationok") {
            ui->stackedWidget->setCurrentIndex(0);
            ui->motdepasse->clear();
            QMessageBox::information(this, "Création réussie", "L'utilisateur
a bien été créé");
            rafraichirListeUtilisateurs();
        }
        else if(reponse.startsWith("creationerreurtechnique")) {
            QMessageBox::critical(this, "Création échouée", "Une erreur
technique a eu lieu :\n\n" + reponse.split(30).at(1));
        }
        else {
            QMessageBox::critical(this, "Erreur serveur", "Pas de réponse ou
réponse malformée du serveur. Veuillez réessayer.");
        }
    }
    else if(mode == "modification") {
        if(reponse == "modificationok") {
            ui->stackedWidget->setCurrentIndex(0);
            ui->motdepasse->clear();
            QMessageBox::information(this, "Modification réussie",
"L'utilisateur a bien été modifié");
            rafraichirListeUtilisateurs();
        }
        else if(reponse == "modificationerreurinexistant") {
            ui->stackedWidget->setCurrentIndex(0);
            ui->motdepasse->clear();
            QMessageBox::critical(this, "Modification échouée",
"L'utilisateur n'existe pas (id_utilisateur introuvable).");
            rafraichirListeUtilisateurs();
        }
        else if(reponse.startsWith("modificationerreurtechnique")) {
            QMessageBox::critical(this, "Modification échouée", "Une erreur
technique a eu lieu :\n\n" + reponse.split(30).at(1));
        }
        else {
            QMessageBox::critical(this, "Erreur serveur", "Pas de réponse ou
réponse malformée du serveur. Veuillez réessayer.");
        }
    }
}
```

```
// Annulation de la création / modification de l'utilisateur
void GestionUtilisateurs::on_annulerUtilisateur_clicked()
{
    QMessageBox::StandardButton retour;
    retour = QMessageBox::question(this, "Confirmation", "Les modifications
seront abandonnées. Confirmer ?");

    if(retour != QMessageBox::Yes) return;

    ui->stackedWidget->setCurrentIndex(0);
    ui->motdepasse->clear();
}


// Ajout d'un groupe du système aux groupes de l'utilisateur
void GestionUtilisateurs::on_ajouterGroupe_clicked()
{
    int i = ui->groupes->currentRow();
    if(i < 0 || i >= listeGroupes.size()) return;

    Groupe groupe = listeGroupes.at(i);
    listeGroupes.remove(i);
    delete ui->groupes->takeItem(i);
    listeGroupesUtilisateur.append(groupe);
    ui->groupesUtilisateur->addItem(groupe.nom);
}


// Retrait d'un groupes des groupes de l'utilisateur
void GestionUtilisateurs::on_enleverGroupe_clicked()
{
    int i = ui->groupesUtilisateur->currentRow();
    if(i < 0 || i >= listeGroupesUtilisateur.size()) return;

    Groupe groupe = listeGroupesUtilisateur.at(i);
    listeGroupesUtilisateur.remove(i);
    delete ui->groupesUtilisateur->takeItem(i);
    listeGroupes.append(groupe);
    ui->groupes->addItem(groupe.nom);
}
```

```
// Validation de la liste des groupes de l'utilisateur
void GestionUtilisateurs::on_validerGroupes_clicked()
{
    // Envoi de la requête et réception de la réponse
    QByteArray message = "admin_gererutilisateur";
    message.append(30);
    message.append("groupes");
    message.append(30);
    message.append(QString::number(utilisateur.id));
    message.append(30);

    for(int g = 0; g < listeGroupesUtilisateur.size(); g++) {
        message.append(QString::number(listeGroupesUtilisateur.at(g).id));
        if(g != listeGroupesUtilisateur.size() - 1) message.append(31);
    }

    QByteArray reponse = compRes->envoyerMessage(message);

    // Affichage de l'état de la requête
    if(reponse == "groupesutilisateurok") {
        ui->stackedWidget->setCurrentIndex(0);
        QMessageBox::information(this, "Modification réussie", "Les groupes
de l'utilisateur ont été mis à jour.");
        rafraichirListeUtilisateurs();
    }
    else if(reponse == "groupesutilisateurerreurinexistant") {
        ui->stackedWidget->setCurrentIndex(0);
        QMessageBox::critical(this, "Modification échouée", "L'utilisateur
n'existe pas (id_utilisateur introuvable).");
        rafraichirListeUtilisateurs();
    }
    else if(reponse.startsWith("groupesutilisateurerreurtechnique")) {
        QMessageBox::critical(this, "Modification échouée", "Une erreur
technique a eu lieu :\n\n" + reponse.split(30).at(1));
    }
    else {
        QMessageBox::critical(this, "Erreur serveur", "Pas de réponse ou
réponse malformée du serveur. Veuillez réessayer.");
    }
}
```

```
// Annulation de la modification de la liste des groupes de l'utilisateur
void GestionUtilisateurs::on_annulerGroupes_clicked()
{
    QMessageBox::StandardButton retour;
    retour = QMessageBox::question(this, "Confirmation", "Les modifications
seront abandonnées. Confirmer ?");

    if(retour != QMessageBox::Yes) return;

    ui->stackedWidget->setCurrentIndex(0);
}


// Activation des boutons de modification lors de la sélection d'un
utilisateur
void GestionUtilisateurs::on_listeUtilisateurs_currentRowChanged(int
currentRow)
{
    bool actif = (currentRow != -1);
    ui->modifier->setEnabled(actif);
    ui->supprimer->setEnabled(actif);
    ui->gererGroupes->setEnabled(actif);
}


// Simulation du bouton modifier lors du double-clic sur un utilisateurs
void GestionUtilisateurs::on_listeUtilisateurs_doubleClicked(const
QModelIndex &index)
{
    on_modifier_clicked();
}


// Actualisation de la liste des utilisateurs
void GestionUtilisateurs::on_actualiser_clicked()
{
    rafraichirListeUtilisateurs();
}


// Réimplémenté : exécuté lors de la fermeture de la fenêtre (croix)
void GestionUtilisateurs::closeEvent(QCloseEvent * e) {
    // Message de confirmation si on est sur une fenêtre d'édition
    if(ui->stackedWidget->currentIndex() != 0) {
        QMessageBox::StandardButton retour;
        retour = QMessageBox::question(this, "Confirmation", "Les
modifications seront abandonnées. Confirmer ?");

        if(retour != QMessageBox::Yes) e->ignore();
        else {
            ui->stackedWidget->setCurrentIndex(0);
            utilisateur.id = -1;
            ui->motdepasse->clear();
            e->accept();
        }
    }
    else e->accept();
}
```

```
// Réimplémenté : exécuté lorsqu'une touche est appuyée
void GestionUtilisateurs::keyPressEvent(QKeyEvent * e) {
    // On demande la fermeture de la fenêtre si la touche est Echap
    if(e->key() == Qt::Key_Escape) {
        this->close();
        e->accept();
    }
    else e->ignore();
}
```

5. GestionGroupes

Ce composant gère les actions et traitements liés à la gestion des groupes.

a. GestionGroupes.h

```
#ifndef GESTIONGROUPES_H
#define GESTIONGROUPES_H

#include <QDialog>

class ComposantReseau;

namespace Ui {
class GestionGroupes;
}

class GestionGroupes : public QDialog
{
    Q_OBJECT

public:
    explicit GestionGroupes(ComposantReseau *p_compRes, QWidget *parent = 0);
    ~GestionGroupes();

    void rafraichirListeGroupes();

private slots:
    void on_creer_clicked();
    void on_gererUtilisateurs_clicked();
    void on_gererMotsDePasse_clicked();
    void on_modifier_clicked();
    void on_supprimer_clicked();
    void on_validerGroupe_clicked();
    void on_annulerGroupe_clicked();
    void on_ajouterUtilisateur_clicked();
    void on_enleverUtilisateur_clicked();
    void on_validerUtilisateurs_clicked();
    void on_annulerUtilisateurs_clicked();
    void on_ajouterMotDePasse_clicked();
    void on_enleverMotDePasse_clicked();
    void on_validerMotsDePasse_clicked();
    void on_annulerMotsDePasse_clicked();
    void on_listeGroupes_currentRowChanged(int currentRow);
    void on_listeGroupes_doubleClicked(const QModelIndex &index);
    void on_actualiser_clicked();

private:
    Ui::GestionGroupes *ui;
    ComposantReseau *compRes;

    struct Groupe {
        int id;
        QString nom;
    };

    struct Utilisateur {
        int id;
        QString nom;
        QString identifiant;
    };
```

```
    struct MotDePasse {
        int id;
        QString nom;
        QString identifiant;
        QString nomServeur;
        QString hoteServeur;
    };

    Groupe groupe;
    QVector<Groupe> listeGroupes;
    QVector<Utilisateur> listeUtilisateurs;
    QVector<Utilisateur> listeUtilisateursGroupe;
    QVector<MotDePasse> listeMotsDePasse;
    QVector<MotDePasse> listeMotsDePasseGroupe;

    void closeEvent(QCloseEvent *);
    void keyPressEvent(QKeyEvent *);
};

#endif // GESTIONGROUPES_H
```

b. GestionGroupes.cpp

```
#include "GestionGroupes.h"
#include "ComposantReseau.h"

#ifdef Q_OS_WIN
#include "ui_GestionGroupes_win.h"
#else
#include "ui_GestionGroupes_unix.h"
#endif

#include <QMessageBox>
#include <QCloseEvent>
#include <QKeyEvent>

GestionGroupes::GestionGroupes(ComposantReseau *p_compRes, QWidget *parent) :
    QDialog(parent,
Qt::WindowSystemMenuHint|Qt::WindowCloseButtonHint|Qt::WindowMinimizeButtonHi
nt),
    ui(new Ui::GestionGroupes),
    compRes(p_compRes)
{
    ui->setupUi(this);
}

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

// Récupère la liste des groupes du système
void GestionGroupes::rafraichirListeGroupes() {

    // Envoi de la requête et réception de la réponse
    QByteArray message = "admin_listegroupes";
    QByteArray reponse = compRes->envoyerMessage(message);

    QList<QByteArray> lignes = reponse.split(30);

    listeGroupes.clear();
    ui->listeGroupes->clear();

    // Affichage des résultats
    if(lignes.at(0) == "admin_listegroupes" && lignes.at(1).size() > 0) {
        QList<QByteArray> champs;

        for(int m = 1; m < lignes.size(); m++) {
            champs = lignes.at(m).split(31);
            groupe.id = champs.at(0).trimmed().toInt();
            groupe.nom = champs.at(1).trimmed();
            listeGroupes.append(groupe);
            ui->listeGroupes->addItem(groupe.nom);
        }
    }

    ui->listeGroupes->setCurrentRow(-1);

}
```

```
// Entre en mode création d'un groupe
void GestionGroupes::on_creer_clicked()
{
    ui->stackedWidget->setCurrentIndex(1);
    ui->titreCreerModifier->setText("Créer un groupe");

    groupe.id = -1;
    ui->nom->clear();
}


// Entre en mode modification d'un groupe
void GestionGroupes::on_modifier_clicked()
{
    int i = ui->listeGroupes->currentRow();
    if(i < 0 || i >= listeGroupes.size()) return;

    ui->stackedWidget->setCurrentIndex(1);
    ui->titreCreerModifier->setText("Modifier un groupe");

    groupe = listeGroupes.at(i);
    ui->nom->setText(groupe.nom);
}
```

```
// Suppression d'un groupe
void GestionGroupes::on_supprimer_clicked()
{
    int i = ui->listeGroupes->currentRow();
    if(i < 0 || i >= listeGroupes.size()) return;

    // Message de confirmation
    QMessageBox::StandardButton retour;
    retour = QMessageBox::question(this, "Confirmation", "La suppression du
groupe sera définitive. Confirmer ?");

    if(retour != QMessageBox::Yes) return;

    // Envoi de la requête et réception de la réponse
    QByteArray message = "admin_gerergroupe";
    message.append(30);
    message.append("suppression");
    message.append(30);
    message.append(QString::number(listeGroupes.at(i).id));

    QByteArray reponse = compRes->envoyerMessage(message);

    // Affichage de l'état de la requête
    if(reponse == "suppressionok") {
        QMessageBox::information(this, "Suppression effectuée", "Le groupe a
été supprimé");
        rafraichirListeGroupes();
    }

    else if(reponse == "suppressionerreurinexistant") {
        QMessageBox::critical(this, "Suppression échouée", "Le groupe
n'existe pas (id_groupe introuvable).");
        rafraichirListeGroupes();
    }

    else if(reponse.startsWith("suppressionerreurtechnique")) {
        QMessageBox::critical(this, "Suppression échouée", "Une erreur
technique a eu lieu :\n\n" + reponse.split(30).at(1));
    }


    else {
        QMessageBox::critical(this, "Erreur serveur", "Pas de réponse ou
réponse malformée du serveur. Veuillez réessayer.");
    }
}
```

```
// Entre en mode gestion des utilisateurs du groupe
void GestionGroupes::on_gererUtilisateurs_clicked()
{
    int i = ui->listeGroupes->currentRow();
    if(i < 0 || i >= listeGroupes.size()) return;

    listeUtilisateursGroupe.clear();
    ui->utilisateursGroupe->clear();
    listeUtilisateurs.clear();
    ui->utilisateurs->clear();

    groupe = listeGroupes.at(i);

    ui->labelGroupe1->setText(groupe.nom);

    QByteArray message;
    QList<QByteArray> lignes;
    QList<QByteArray> champs;
    Utilisateur utilisateur;

    // Récupération des utilisateurs actuels du groupe
    QByteArray utilisateursGroupe;
    message = "admin_listeutilisateursgroupe";
    message.append(30);
    message.append(QString::number(groupe.id));
    utilisateursGroupe = compRes->envoyerMessage(message);

    lignes = utilisateursGroupe.split(30);

    // Affichage des résultats
    if(lignes.at(0) == "admin_listeutilisateursgroupe" && lignes.at(1).size()
> 0) {
        for(int m = 1; m < lignes.size(); m++) {
            champs = lignes.at(m).split(31);
            utilisateur.id = champs.at(0).trimmed().toInt();
            utilisateur.nom = champs.at(1).trimmed();
            utilisateur.identifiant = champs.at(2).trimmed();
            listeUtilisateursGroupe.append(utilisateur);
            ui->utilisateursGroupe->addItem(utilisateur.nom + " (" +
utilisateur.identifiant + ")");
        }
    }

    lignes.clear();
    champs.clear();

    // Récupération des utilisateurs du système
    QByteArray utilisateurs;
    utilisateurs = compRes->envoyerMessage("admin_listeutilisateurs");
    bool trouve;

    lignes = utilisateurs.split(30);

    // Affichage des résultats
    if(lignes.at(0) == "admin_listeutilisateurs" && lignes.at(1).size() > 0)
{
        listeUtilisateurs.clear();
        ui->utilisateurs->clear();
        for(int m = 1; m < lignes.size(); m++) {
            champs = lignes.at(m).split(31);
            utilisateur.id = champs.at(0).trimmed().toInt();
            utilisateur.nom = champs.at(1).trimmed();
            utilisateur.identifiant = champs.at(2).trimmed();
```

```
            // On exclue les utilisateurs qui sont dans les utilisateurs du
groupe
            for(int g = 0; g < listeUtilisateursGroupe.size(); g++) {
                if(listeUtilisateursGroupe.at(g).id == utilisateur.id) {
                    trouve = true;
                    break;
                }
            }
            if(!trouve) {
                listeUtilisateurs.append(utilisateur);
                ui->utilisateurs->addItem(utilisateur.nom + " (" +
utilisateur.identifiant + ")");
            }
            trouve = false;
        }
    }

    ui->stackedWidget->setCurrentIndex(2);
}
```

```
// Entre en mode gestion des mots de passe du groupe
void GestionGroupes::on_gererMotsDePasse_clicked()
{
    int i = ui->listeGroupes->currentRow();
    if(i < 0 || i >= listeGroupes.size()) return;

    listeMotsDePasseGroupe.clear();
    ui->motsDePasseGroupe->clear();
    listeMotsDePasse.clear();
    ui->motsDePasse->clear();

    groupe = listeGroupes.at(i);

    ui->labelGroupe2->setText(groupe.nom);

    QByteArray message;
    QList<QByteArray> lignes;
    QList<QByteArray> champs;
    MotDePasse motDePasse;

    // Récupération des mots de passe actuels du groupe
    QByteArray motsDePasseGroupe;

    message = "admin_listemotsdepassegroupe";
    message.append(30);
    message.append(QString::number(groupe.id));
    motsDePasseGroupe = compRes->envoyerMessage(message);

    lignes = motsDePasseGroupe.split(30);

    // Affichage des résultats
    if(lignes.at(0) == "admin_listemotsdepassegroupe" && lignes.at(1).size()
> 0) {
        for(int m = 1; m < lignes.size(); m++) {
            champs = lignes.at(m).split(31);
            motDePasse.id = champs.at(0).trimmed().toInt();
            motDePasse.nom = champs.at(1).trimmed();
            motDePasse.identifiant = champs.at(2).trimmed();
            motDePasse.nomServeur = champs.at(3).trimmed();
            motDePasse.hoteServeur = champs.at(4).trimmed();
            listeMotsDePasseGroupe.append(motDePasse);
            ui->motsDePasseGroupe->addItem(motDePasse.nomServeur + " : " +
                                           motDePasse.nom + " @ " +
motDePasse.hoteServeur);
        }
    }

    lignes.clear();
    champs.clear();

    // Récupération des mots de passe du système
    QByteArray motsDePasse;
    motsDePasse = compRes->envoyerMessage("admin_listemotsdepasse");
    bool trouve;

    lignes = motsDePasse.split(30);

    // Affichage des résultats
    if(lignes.at(0) == "admin_listemotsdepasse" && lignes.at(1).size() > 0) {
        for(int m = 1; m < lignes.size(); m++) {
            champs = lignes.at(m).split(31);
            motDePasse.id = champs.at(0).trimmed().toInt();
            motDePasse.nom = champs.at(1).trimmed();
            motDePasse.identifiant = champs.at(2).trimmed();
```

```
            motDePasse.nomServeur = champs.at(5).trimmed();
            motDePasse.hoteServeur = champs.at(6).trimmed();

            // On exclue les mots de passe qui sont dans les mots de passe du
groupe
            for(int g = 0; g < listeMotsDePasseGroupe.size(); g++) {
                if(listeMotsDePasseGroupe.at(g).id == motDePasse.id) {
                    trouve = true;
                    break;
                }
            }
            if(!trouve) {
                listeMotsDePasse.append(motDePasse);
                ui->motsDePasse->addItem(motDePasse.nomServeur + " : " +
                                         motDePasse.nom + " @ " +
motDePasse.hoteServeur);
            }
            trouve = false;
        }
    }

    ui->stackedWidget->setCurrentIndex(3);
}
```

```
// Validation de la création / modification du groupe
void GestionGroupes::on_validerGroupe_clicked()
{
    QString mode = groupe.id == -1 ? "creation" : "modification";

    // Envoi de la requête et réception de la réponse
    QByteArray message = "admin_gerergroupe";
    message.append(30);
    message.append(mode);
    message.append(30);
    message.append(QString::number(groupe.id));
    message.append(30);
    message.append(ui->nom->text());

    QByteArray reponse = compRes->envoyerMessage(message);

    // Affichage de l'état de la requête
    if(mode == "creation") {
        if(reponse == "creationok") {
            ui->stackedWidget->setCurrentIndex(0);
            QMessageBox::information(this, "Création réussie", "Le groupe a
bien été créé");
            rafraichirListeGroupes();
        }
        else if(reponse.startsWith("creationerreurtechnique")) {
            QMessageBox::critical(this, "Création échouée", "Une erreur
technique a eu lieu :\n\n" + reponse.split(30).at(1));
        }
        else {
            QMessageBox::critical(this, "Erreur serveur", "Pas de réponse ou
réponse malformée du serveur. Veuillez réessayer.");
        }
    }
    else if(mode == "modification") {
        if(reponse == "modificationok") {
            ui->stackedWidget->setCurrentIndex(0);
            QMessageBox::information(this, "Modification réussie", "Le groupe
a bien été modifié");
            rafraichirListeGroupes();
        }
        else if(reponse == "modificationerreurinexistant") {
            ui->stackedWidget->setCurrentIndex(0);
            QMessageBox::critical(this, "Modification échouée", "Le groupe
n'existe pas (id_groupe introuvable).");
            rafraichirListeGroupes();
        }
        else if(reponse.startsWith("modificationerreurtechnique")) {
            QMessageBox::critical(this, "Modification échouée", "Une erreur
technique a eu lieu :\n\n" + reponse.split(30).at(1));
        }
        else {
            QMessageBox::critical(this, "Erreur serveur", "Pas de réponse ou
réponse malformée du serveur. Veuillez réessayer.");
        }
    }
}
```

```
// Annulation de la création / modification du groupe
void GestionGroupes::on_annulerGroupe_clicked()
{
    QMessageBox::StandardButton retour;
    retour = QMessageBox::question(this, "Confirmation", "Les modifications
seront abandonnées. Confirmer ?");

    if(retour != QMessageBox::Yes) return;

    ui->stackedWidget->setCurrentIndex(0);
}


// Ajout d'un utilisateur du système aux utilisateurs du groupe
void GestionGroupes::on_ajouterUtilisateur_clicked()
{
    int i = ui->utilisateurs->currentRow();
    if(i < 0 || i >= listeUtilisateurs.size()) return;

    Utilisateur utilisateur = listeUtilisateurs.at(i);
    listeUtilisateurs.remove(i);
    delete ui->utilisateurs->takeItem(i);
    listeUtilisateursGroupe.append(utilisateur);
    ui->utilisateursGroupe->addItem(utilisateur.nom + " (" +
utilisateur.identifiant + ")");
}


// Retrait d'un utilisateur des utilisateurs du groupe
void GestionGroupes::on_enleverUtilisateur_clicked()
{
    int i = ui->utilisateursGroupe->currentRow();
    if(i < 0 || i >= listeUtilisateursGroupe.size()) return;

    Utilisateur utilisateur = listeUtilisateursGroupe.at(i);
    listeUtilisateursGroupe.remove(i);
    delete ui->utilisateursGroupe->takeItem(i);
    listeUtilisateurs.append(utilisateur);
    ui->utilisateurs->addItem(utilisateur.nom + " (" +
utilisateur.identifiant + ")");
}
```

```
// Validation de la liste des utilisateurs du groupe
void GestionGroupes::on_validerUtilisateurs_clicked()
{

    // Envoi de la requête et réception de la réponse
    QByteArray message = "admin_gerergroupe";
    message.append(30);
    message.append("utilisateurs");
    message.append(30);
    message.append(QString::number(groupe.id));
    message.append(30);

    for(int g = 0; g < listeUtilisateursGroupe.size(); g++) {
        message.append(QString::number(listeUtilisateursGroupe.at(g).id));
        if(g != listeUtilisateursGroupe.size() - 1) message.append(31);
    }

    QByteArray reponse = compRes->envoyerMessage(message);

    // Affichage de l'état de la requête
    if(reponse == "utilisateursgroupeok") {
        ui->stackedWidget->setCurrentIndex(0);
        QMessageBox::information(this, "Modification réussie", "Les
utilisateurs du groupe ont été mis à jour.");
        rafraichirListeGroupes();
    }
    else if(reponse == "utilisateursgroupeerreurinexistant") {
        ui->stackedWidget->setCurrentIndex(0);
        QMessageBox::critical(this, "Modification échouée", "Le groupe
n'existe pas (id_groupe introuvable).");
        rafraichirListeGroupes();
    }
    else if(reponse.startsWith("utilisateursgroupeerreurtechnique")) {
        QMessageBox::critical(this, "Modification échouée", "Une erreur
technique a eu lieu :\n\n" + reponse.split(30).at(1));
    }
    else {
        QMessageBox::critical(this, "Erreur serveur", "Pas de réponse ou
réponse malformée du serveur. Veuillez réessayer.");
    }
}


// Annulation de la modification de la liste des utilisateurs du groupe
void GestionGroupes::on_annulerUtilisateurs_clicked()
{
    QMessageBox::StandardButton retour;
    retour = QMessageBox::question(this, "Confirmation", "Les modifications
seront abandonnées. Confirmer ?");

    if(retour != QMessageBox::Yes) return;

    ui->stackedWidget->setCurrentIndex(0);
}
```

```
// Ajout d'un mot de passe du système aux mots de passe du groupe
void GestionGroupes::on_ajouterMotDePasse_clicked()
{
    int i = ui->motsDePasse->currentRow();
    if(i < 0 || i >= listeMotsDePasse.size()) return;

    MotDePasse motDePasse = listeMotsDePasse.at(i);
    listeMotsDePasse.remove(i);
    delete ui->motsDePasse->takeItem(i);
    listeMotsDePasseGroupe.append(motDePasse);
    ui->motsDePasseGroupe->addItem(motDePasse.nomServeur + " : " +
                                   motDePasse.nom + " @ " +
motDePasse.hoteServeur);
}


// Ajout d'un mot de passe des mots de passe du groupe
void GestionGroupes::on_enleverMotDePasse_clicked()
{
    int i = ui->motsDePasseGroupe->currentRow();
    if(i < 0 || i >= listeMotsDePasseGroupe.size()) return;

    MotDePasse motDePasse = listeMotsDePasseGroupe.at(i);
    listeMotsDePasseGroupe.remove(i);
    delete ui->motsDePasseGroupe->takeItem(i);
    listeMotsDePasse.append(motDePasse);
    ui->motsDePasse->addItem(motDePasse.nomServeur + " : " +
                             motDePasse.nom + " @ " +
motDePasse.hoteServeur);
}
```

```
// Validation de la liste des mots de passe du groupe
void GestionGroupes::on_validerMotsDePasse_clicked()
{

    // Envoi de la requête et réception de la réponse
    QByteArray message = "admin_gerergroupe";
    message.append(30);
    message.append("motsdepasse");
    message.append(30);
    message.append(QString::number(groupe.id));
    message.append(30);

    for(int g = 0; g < listeMotsDePasseGroupe.size(); g++) {
        message.append(QString::number(listeMotsDePasseGroupe.at(g).id));
        if(g != listeMotsDePasseGroupe.size() - 1) message.append(31);
    }

    QByteArray reponse = compRes->envoyerMessage(message);

    // Affichage de l'état de la requête
    if(reponse == "motsdepassegroupeok") {
        ui->stackedWidget->setCurrentIndex(0);
        QMessageBox::information(this, "Modification réussie", "Les mots de
passe du groupe ont été mis à jour.");
        rafraichirListeGroupes();
    }
    else if(reponse == "motsdepassegroupeerreurinexistant") {
        ui->stackedWidget->setCurrentIndex(0);
        QMessageBox::critical(this, "Modification échouée", "Le groupe
n'existe pas (id_groupe introuvable).");
        rafraichirListeGroupes();
    }
    else if(reponse.startsWith("motsdepassegroupeerreurtechnique")) {
        QMessageBox::critical(this, "Modification échouée", "Une erreur
technique a eu lieu :\n\n" + reponse.split(30).at(1));
    }
    else {
        QMessageBox::critical(this, "Erreur serveur", "Pas de réponse ou
réponse malformée du serveur. Veuillez réessayer.");
    }
}
```

```
// Annulation de la modification de la liste des mots de passe du groupe
void GestionGroupes::on_annulerMotsDePasse_clicked()
{
    QMessageBox::StandardButton retour;
    retour = QMessageBox::question(this, "Confirmation", "Les modifications
seront abandonnées. Confirmer ?");

    if(retour != QMessageBox::Yes) return;

    ui->stackedWidget->setCurrentIndex(0);
}

// Activation des boutons de modification lors de la sélection d'un groupe
void GestionGroupes::on_listeGroupes_currentRowChanged(int currentRow)
{
    bool actif = (currentRow != -1);
    ui->modifier->setEnabled(actif);
    ui->supprimer->setEnabled(actif);
    ui->gererUtilisateurs->setEnabled(actif);
    ui->gererMotsDePasse->setEnabled(actif);
}

// Simulation du bouton modifier lors du double-clic sur un groupe
void GestionGroupes::on_listeGroupes_doubleClicked(const QModelIndex &index)
{
    on_modifier_clicked();
}

// Actualisation de la liste des groupes
void GestionGroupes::on_actualiser_clicked()
{
    rafraichirListeGroupes();
}

// Réimplémenté : exécuté lors de la fermeture de la fenêtre (croix)
void GestionGroupes::closeEvent(QCloseEvent * e) {
    // Message de confirmation si on est sur une fenêtre d'édition
    if(ui->stackedWidget->currentIndex() != 0) {
        QMessageBox::StandardButton retour;
        retour = QMessageBox::question(this, "Confirmation", "Les
modifications seront abandonnées. Confirmer ?");

        if(retour != QMessageBox::Yes) e->ignore();
        else {
            ui->stackedWidget->setCurrentIndex(0);
            groupe.id = -1;
            e->accept();
        }
    }
    else e->accept();
}

// Réimplémenté : exécuté lorsqu'une touche est appuyée
void GestionGroupes::keyPressEvent(QKeyEvent * e) {
    // On demande la fermeture de la fenêtre si la touche est Echap
    if(e->key() == Qt::Key_Escape) {
        this->close();
        e->accept();
    }
    else e->ignore();
}
```

6. GestionServeurs

Ce composant gère les actions et traitements liés à la gestion des serveurs.

a. GestionServeurs.h

```
#ifndef GESTIONSERVEURS_H
#define GESTIONSERVEURS_H

#include <QDialog>

class ComposantReseau;

namespace Ui {
class GestionServeurs;
}

class GestionServeurs : public QDialog
{
    Q_OBJECT

public:
    explicit GestionServeurs(ComposantReseau *p_compRes, QWidget *parent =
0);
    ~GestionServeurs();

    void rafraichirListeServeurs();

private slots:
    void on_creer_clicked();
    void on_modifier_clicked();
    void on_supprimer_clicked();
    void on_validerServeur_clicked();
    void on_annulerServeur_clicked();
    void on_listeServeurs_currentRowChanged(int currentRow);
    void on_listeServeurs_doubleClicked(const QModelIndex &index);
    void on_actualiser_clicked();

private:
    Ui::GestionServeurs *ui;
    ComposantReseau *compRes;

    struct Serveur {
        int id;
        QString nom;
        QString hote;
    };

    Serveur serveur;
    QVector<Serveur> listeServeurs;

    void closeEvent(QCloseEvent *);
    void keyPressEvent(QKeyEvent *);
};

#endif // GESTIONSERVEURS_H
```

b. GestionServeurs.cpp

```
#include "GestionServeurs.h"
#include "ComposantReseau.h"

#ifdef Q_OS_WIN
#include "ui_GestionServeurs_win.h"
#else
#include "ui_GestionServeurs_unix.h"
#endif

#include <QMessageBox>
#include <QCloseEvent>
#include <QKeyEvent>

GestionServeurs::GestionServeurs(ComposantReseau *p_compRes, QWidget *parent)
:
    QDialog(parent,
Qt::WindowSystemMenuHint|Qt::WindowCloseButtonHint|Qt::WindowMinimizeButtonHi
nt),
    ui(new Ui::GestionServeurs),
    compRes(p_compRes)
{
    ui->setupUi(this);
}

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

// Récupère la liste des serveurs du système
void GestionServeurs::rafraichirListeServeurs() {

    // Envoi de la requête et réception de la réponse
    QByteArray message = "admin_listeserveurs";
    QByteArray reponse = compRes->envoyerMessage(message);

    QList<QByteArray> lignes = reponse.split(30);

    listeServeurs.clear();
    ui->listeServeurs->clear();

    // Affichage des résultats
    if(lignes.at(0) == "admin_listeserveurs" && lignes.at(1).size() > 0) {
        QList<QByteArray> champs;

        for(int m = 1; m < lignes.size(); m++) {
            champs = lignes.at(m).split(31);
            serveur.id = champs.at(0).trimmed().toInt();
            serveur.nom = champs.at(1).trimmed();
            serveur.hote = champs.at(2).trimmed();
            listeServeurs.append(serveur);
            ui->listeServeurs->addItem(serveur.nom + " (" + serveur.hote +
")");
        }
    }

    ui->listeServeurs->setCurrentRow(-1);

}
```

```
// Entre en mode création d'un serveur
void GestionServeurs::on_creer_clicked()
{
    ui->stackedWidget->setCurrentIndex(1);
    ui->titreCreerModifier->setText("Créer un serveur");

    serveur.id = -1;
    ui->nom->clear();
    ui->hote->clear();
}


// Entre en mode modification d'un serveur
void GestionServeurs::on_modifier_clicked()
{
    int i = ui->listeServeurs->currentRow();
    if(i < 0 || i >= listeServeurs.size()) return;

    ui->stackedWidget->setCurrentIndex(1);
    ui->titreCreerModifier->setText("Modifier un serveur");

    serveur = listeServeurs.at(i);
    ui->nom->setText(serveur.nom);
    ui->hote->setText(serveur.hote);
}
```

```
// Suppression d'un serveur
void GestionServeurs::on_supprimer_clicked()
{
    int i = ui->listeServeurs->currentRow();
    if(i < 0 || i >= listeServeurs.size()) return;

    // Messages de confirmation
    QMessageBox::StandardButton retour;

    retour = QMessageBox::question(this, "Confirmation", "La suppression du
serveur sera définitive. Confirmer ?");
    if(retour != QMessageBox::Yes) return;

    retour = QMessageBox::question(this, "Confirmation", "Tous les mots de
passe associés au serveur seront également supprimés. Confirmer ?");
    if(retour != QMessageBox::Yes) return;

    // Envoi de la requête et réception de la réponse
    QByteArray message = "admin_gererserveur";
    message.append(30);
    message.append("suppression");
    message.append(30);
    message.append(QString::number(listeServeurs.at(i).id));

    QByteArray reponse = compRes->envoyerMessage(message);

    // Affichage de l'état de la requête
    if(reponse == "suppressionok") {
        QMessageBox::information(this, "Suppression effectuée", "Le serveur a
été supprimé");
        rafraichirListeServeurs();
    }

    else if(reponse == "suppressionerreurinexistant") {
        QMessageBox::critical(this, "Suppression échouée", "Le serveur
n'existe pas (id_serveur introuvable).");
        rafraichirListeServeurs();
    }

    else if(reponse.startsWith("suppressionerreurtechnique")) {
        QMessageBox::critical(this, "Suppression échouée", "Une erreur
technique a eu lieu :\n\n" + reponse.split(30).at(1));
    }
    else {
        QMessageBox::critical(this, "Erreur serveur", "Pas de réponse ou
réponse malformée du serveur. Veuillez réessayer.");
    }
}
```

```
// Validation de la création / modification du serveur
void GestionServeurs::on_validerServeur_clicked()
{
    QString mode = serveur.id == -1 ? "creation" : "modification";

    // Envoi de la requête et réception de la réponse
    QByteArray message = "admin_gererserveur";
    message.append(30);
    message.append(mode);
    message.append(30);
    message.append(QString::number(serveur.id));
    message.append(30);
    message.append(ui->nom->text());
    message.append(30);
    message.append(ui->hote->text());

    QByteArray reponse = compRes->envoyerMessage(message);

    // Affichage de l'état de la requête
    if(mode == "creation") {
        if(reponse == "creationok") {
            ui->stackedWidget->setCurrentIndex(0);
            QMessageBox::information(this, "Création réussie", "Le serveur a
bien été créé");
            rafraichirListeServeurs();
        }
        else if(reponse.startsWith("creationerreurtechnique")) {
            QMessageBox::critical(this, "Création échouée", "Une erreur
technique a eu lieu :\n\n" + reponse.split(30).at(1));
        }
        else {
            QMessageBox::critical(this, "Erreur serveur", "Pas de réponse ou
réponse malformée du serveur. Veuillez réessayer.");
        }
    }
    else if(mode == "modification") {
        if(reponse == "modificationok") {
            ui->stackedWidget->setCurrentIndex(0);
            QMessageBox::information(this, "Modification réussie", "Le
serveur a bien été modifié");
            rafraichirListeServeurs();
        }
        else if(reponse == "modificationerreurinexistant") {
            ui->stackedWidget->setCurrentIndex(0);
            QMessageBox::critical(this, "Modification échouée", "Le serveur
n'existe pas (id_serveur introuvable).");
            rafraichirListeServeurs();
        }
        else if(reponse.startsWith("modificationerreurtechnique")) {
            QMessageBox::critical(this, "Modification échouée", "Une erreur
technique a eu lieu :\n\n" + reponse.split(30).at(1));
        }
        else {
            QMessageBox::critical(this, "Erreur serveur", "Pas de réponse ou
réponse malformée du serveur. Veuillez réessayer.");
        }
    }
}
```

```
// Annulation de la création / modification du serveur
void GestionServeurs::on_annulerServeur_clicked()
{
    QMessageBox::StandardButton retour;
    retour = QMessageBox::question(this, "Confirmation", "Les modifications
seront abandonnées. Confirmer ?");

    if(retour != QMessageBox::Yes) return;

        ui->stackedWidget->setCurrentIndex(0);
}


// Activation des boutons de modification lors de la sélection d'un serveur
void GestionServeurs::on_listeServeurs_currentRowChanged(int currentRow)
{
    bool actif = (currentRow != -1);
    ui->modifier->setEnabled(actif);
    ui->supprimer->setEnabled(actif);
}


// Simulation du bouton modifier lors du double-clic sur un serveur
void GestionServeurs::on_listeServeurs_doubleClicked(const QModelIndex
&index)
{
    on_modifier_clicked();
}


// Actualisation de la liste des serveurs
void GestionServeurs::on_actualiser_clicked()
{
    rafraichirListeServeurs();
}


// Réimplémenté : exécuté lors de la fermeture de la fenêtre (croix)
void GestionServeurs::closeEvent(QCloseEvent * e) {
    // Message de confirmation si on est sur une fenêtre d'édition
    if(ui->stackedWidget->currentIndex() != 0) {
        QMessageBox::StandardButton retour;
        retour = QMessageBox::question(this, "Confirmation", "Les
modifications seront abandonnées. Confirmer ?");

        if(retour != QMessageBox::Yes) e->ignore();
        else {
            ui->stackedWidget->setCurrentIndex(0);
            serveur.id = -1;
            e->accept();
        }
    }
    else e->accept();
}

// Réimplémenté : exécuté lorsqu'une touche est appuyée
void GestionServeurs::keyPressEvent(QKeyEvent * e) {
    // On demande la fermeture de la fenêtre si la touche est Echap
    if(e->key() == Qt::Key_Escape) {
        this->close();
        e->accept();
    }
    else e->ignore();
}
```

6. GestionMotsDePasse

Ce composant gère les actions et traitements liés à la gestion des mots de passe.

a. GestionMotsDePasse.h

```
#ifndef GESTIONMOTSDEPASSE_H
#define GESTIONMOTSDEPASSE_H

#include <QDialog>

class ComposantReseau;

namespace Ui {
class GestionMotsDePasse;
}

class GestionMotsDePasse : public QDialog
{
    Q_OBJECT

public:
    explicit GestionMotsDePasse(ComposantReseau *p_compRes, QWidget *parent =
0);
    ~GestionMotsDePasse();

    void rafraichirListeMotsDePasse();

private slots:
    void on_creer_clicked();
    void on_modifier_clicked();
    void on_supprimer_clicked();
    void on_validerMotDePasse_clicked();
    void on_annulerMotDePasse_clicked();
    void on_ajouterGroupe_clicked();
    void on_enleverGroupe_clicked();
    void on_validerGroupes_clicked();
    void on_annulerGroupes_clicked();
    void on_listeMotsDePasse_currentRowChanged(int currentRow);
    void on_listeMotsDePasse_doubleClicked(const QModelIndex &index);
    void on_actualiser_clicked();
    void on_gererGroupes_clicked();

private:
    Ui::GestionMotsDePasse *ui;
    ComposantReseau *compRes;

    struct MotDePasse {
        int id;
        QString nom;
        QString identifiant;
        QString motDePasse;
        int idServeur;
        QString nomServeur;
        QString hoteServeur;
    };

    struct Serveur {
        int id;
        QString nom;
        QString hote;
    };
```

```
    struct Groupe {
        int id;
        QString nom;
    };

    MotDePasse motDePasse;
    QVector<MotDePasse> listeMotsDePasse;
    QVector<Serveur> listeServeurs;
    QVector<Groupe> listeGroupes;
    QVector<Groupe> listeGroupesMotDePasse;

    void rafraichirListeServeurs(const int idServeur = -1);

    void closeEvent(QCloseEvent *);
    void keyPressEvent(QKeyEvent *);
};

#endif // GESTIONMOTSDEPASSE_H
```

b. GestionMotsDePasse.cpp

```
#include "GestionMotsDePasse.h"
#include "ComposantReseau.h"

#ifdef Q_OS_WIN
#include "ui_GestionMotsDePasse_win.h"
#else
#include "ui_GestionMotsDePasse_unix.h"
#endif

#include <QMessageBox>
#include <QCloseEvent>
#include <QKeyEvent>

GestionMotsDePasse::GestionMotsDePasse(ComposantReseau *p_compRes, QWidget
*parent) :
    QDialog(parent,
Qt::WindowSystemMenuHint|Qt::WindowCloseButtonHint|Qt::WindowMinimizeButtonHi
nt),
    ui(new Ui::GestionMotsDePasse),
    compRes(p_compRes)
{
    ui->setupUi(this);
}

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

// Récupère la liste des mots de passe du système
void GestionMotsDePasse::rafraichirListeMotsDePasse() {

    // Envoi de la requête et réception de la réponse
    QByteArray message = "admin_listemotsdepasse";
    QByteArray reponse = compRes->envoyerMessage(message);

    QList<QByteArray> lignes = reponse.split(30);

    listeMotsDePasse.clear();
    ui->listeMotsDePasse->clear();

    // Affichage des résultats
    if(lignes.at(0) == "admin_listemotsdepasse" && lignes.at(1).size() > 0) {
        QList<QByteArray> champs;

        for(int m = 1; m < lignes.size(); m++) {
            champs = lignes.at(m).split(31);
            motDePasse.id = champs.at(0).trimmed().toInt();
            motDePasse.nom = champs.at(1).trimmed();
            motDePasse.identifiant = champs.at(2).trimmed();
            motDePasse.motDePasse = champs.at(3).trimmed();
            motDePasse.idServeur = champs.at(4).trimmed().toInt();
            motDePasse.nomServeur = champs.at(5).trimmed();
            motDePasse.hoteServeur = champs.at(6).trimmed();
            listeMotsDePasse.append(motDePasse);
            ui->listeMotsDePasse->addItem(motDePasse.nomServeur + " : " +
                                          motDePasse.nom + " @ " +
motDePasse.hoteServeur);
        }
    }
    ui->listeMotsDePasse->setCurrentRow(-1);
}
```

```
// Récupère la liste des serveurs du système
void GestionMotsDePasse::rafraichirListeServeurs(const int idServeur) {

    // Envoi de la requête et réception de la réponse
    QByteArray message = "admin_listeserveurs";
    QByteArray reponse = compRes->envoyerMessage(message);

    QList<QByteArray> lignes = reponse.split(30);

    int ligneServeur = 0;

    listeServeurs.clear();
    ui->serveur->clear();

    // Affichage des résultats
    if(lignes.at(0) == "admin_listeserveurs" && lignes.at(1).size() > 0) {
        QList<QByteArray> champs;

        Serveur serveur;
        for(int m = 1; m < lignes.size(); m++) {
            champs = lignes.at(m).split(31);
            serveur.id = champs.at(0).trimmed().toInt();
            serveur.nom = champs.at(1).trimmed();
            serveur.hote = champs.at(2).trimmed();
            listeServeurs.append(serveur);
            ui->serveur->addItem(serveur.nom + " (" + serveur.hote + ")");
            if(serveur.id == idServeur) ligneServeur = m - 1;
        }
    }

    ui->serveur->setCurrentIndex(ligneServeur);
}


// Entre en mode création d'un mot de passe
void GestionMotsDePasse::on_creer_clicked()
{
    ui->stackedWidget->setCurrentIndex(1);
    ui->titreCreerModifier->setText("Créer un mot de passe");

    motDePasse.id = -1;
    ui->nom->clear();
    ui->identifiant->clear();
    ui->motdepasse->clear();

    rafraichirListeServeurs();
}

// Entre en mode modification d'un mot de passe
void GestionMotsDePasse::on_modifier_clicked()
{
    int i = ui->listeMotsDePasse->currentRow();
    if(i < 0 || i >= listeMotsDePasse.size()) return;

    ui->stackedWidget->setCurrentIndex(1);
    ui->titreCreerModifier->setText("Modifier un mot de passe");

    motDePasse = listeMotsDePasse.at(i);
    ui->nom->setText(motDePasse.nom);
    ui->identifiant->setText(motDePasse.identifiant);
    ui->motdepasse->setText(motDePasse.motDePasse);

    rafraichirListeServeurs(motDePasse.idServeur);
}
```

```
// Suppression d'un mot de passe
void GestionMotsDePasse::on_supprimer_clicked()
{
    int i = ui->listeMotsDePasse->currentRow();
    if(i < 0 || i >= listeMotsDePasse.size()) return;

    // Message de confirmation
    QMessageBox::StandardButton retour;
    retour = QMessageBox::question(this, "Confirmation", "La suppression du
mot de passe sera définitive. Confirmer ?");

    if(retour != QMessageBox::Yes) return;

    // Envoi de la requête et réception de la réponse
    QByteArray message = "admin_gerermotdepasse";
    message.append(30);
    message.append("suppression");
    message.append(30);
    message.append(QString::number(listeMotsDePasse.at(i).id));

    QByteArray reponse = compRes->envoyerMessage(message);

    // Affichage de l'état de la requête
    if(reponse == "suppressionok") {
        QMessageBox::information(this, "Suppression effectuée", "Le mot de
passe a été supprimé");
        rafraichirListeMotsDePasse();
    }

    else if(reponse == "suppressionerreurinexistant") {
        QMessageBox::critical(this, "Suppression échouée", "Le mot de passe
n'existe pas (id_motdepasse introuvable).");
        rafraichirListeMotsDePasse();
    }

    else if(reponse.startsWith("suppressionerreurtechnique")) {
        QMessageBox::critical(this, "Suppression échouée", "Une erreur
technique a eu lieu :\n\n" + reponse.split(30).at(1));
    }
    else {
        QMessageBox::critical(this, "Erreur serveur", "Pas de réponse ou
réponse malformée du serveur. Veuillez réessayer.");
    }
}
```

```
// Validation de la création / modification du mot de passe
void GestionMotsDePasse::on_validerMotDePasse_clicked()
{
    QString mode = motDePasse.id == -1 ? "creation" : "modification";

    // Envoi de la requête et réception de la réponse
    QByteArray message = "admin_gerermotdepasse";
    message.append(30);
    message.append(mode);
    message.append(30);
    message.append(QString::number(motDePasse.id));
    message.append(30);
    message.append(ui->nom->text());
    message.append(30);
    message.append(ui->identifiant->text());
    message.append(30);
    message.append(ui->motdepasse->text());
    message.append(30);
    message.append(QString::number(listeServeurs.at(ui->serveur-
>currentIndex()).id));

    QByteArray reponse = compRes->envoyerMessage(message);

    // Affichage de l'état de la requête
    if(mode == "creation") {
        if(reponse == "creationok") {
            ui->stackedWidget->setCurrentIndex(0);
            QMessageBox::information(this, "Création réussie", "Le mot de
passe a bien été créé");
            rafraichirListeMotsDePasse();
        }
        else if(reponse.startsWith("creationerreurtechnique")) {
            QMessageBox::critical(this, "Création échouée", "Une erreur
technique a eu lieu :\n\n" + reponse.split(30).at(1));
        }
        else {
            QMessageBox::critical(this, "Erreur serveur", "Pas de réponse ou
réponse malformée du serveur. Veuillez réessayer.");
        }
    }
    else if(mode == "modification") {
        if(reponse == "modificationok") {
            ui->stackedWidget->setCurrentIndex(0);
            QMessageBox::information(this, "Modification réussie", "Le mot de
passe a bien été modifié");
            rafraichirListeMotsDePasse();
        }
        else if(reponse == "modificationerreurinexistant") {
            ui->stackedWidget->setCurrentIndex(0);
            QMessageBox::critical(this, "Modification échouée", "Le mot de
passe n'existe pas (id_motdepasse introuvable).");
            rafraichirListeMotsDePasse();
        }
        else if(reponse.startsWith("modificationerreurtechnique")) {
            QMessageBox::critical(this, "Modification échouée", "Une erreur
technique a eu lieu :\n\n" + reponse.split(30).at(1));
        }
        else {
            QMessageBox::critical(this, "Erreur serveur", "Pas de réponse ou
réponse malformée du serveur. Veuillez réessayer.");
        }
    }
}
```

```
// Annulation de la création / modification du mot de passe
void GestionMotsDePasse::on_annulerMotDePasse_clicked()
{
    QMessageBox::StandardButton retour;
    retour = QMessageBox::question(this, "Confirmation", "Les modifications
seront abandonnées. Confirmer ?");

    if(retour != QMessageBox::Yes) return;

    ui->stackedWidget->setCurrentIndex(0);
    ui->motdepasse->clear();
}


// Ajout d'un groupe du système aux groupes du mot de passe
void GestionMotsDePasse::on_ajouterGroupe_clicked()
{
    int i = ui->groupes->currentRow();
    if(i < 0 || i >= listeGroupes.size()) return;

    Groupe groupe = listeGroupes.at(i);
    listeGroupes.remove(i);
    delete ui->groupes->takeItem(i);
    listeGroupesMotDePasse.append(groupe);
    ui->groupesMotDePasse->addItem(groupe.nom);
}


// Retrait d'un groupe des groupes du mot de passe
void GestionMotsDePasse::on_enleverGroupe_clicked()
{
    int i = ui->groupesMotDePasse->currentRow();
    if(i < 0 || i >= listeGroupesMotDePasse.size()) return;

    Groupe groupe = listeGroupesMotDePasse.at(i);
    listeGroupesMotDePasse.remove(i);
    delete ui->groupesMotDePasse->takeItem(i);
    listeGroupes.append(groupe);
    ui->groupes->addItem(groupe.nom);
}
```

```
// Validation de la liste des groupes du mot de passe
void GestionMotsDePasse::on_validerGroupes_clicked()
{

    // Envoi de la requête et réception de la réponse
    QByteArray message = "admin_gerermotdepasse";
    message.append(30);
    message.append("groupes");
    message.append(30);
    message.append(QString::number(motDePasse.id));
    message.append(30);

    for(int g = 0; g < listeGroupesMotDePasse.size(); g++) {
        message.append(QString::number(listeGroupesMotDePasse.at(g).id));
        if(g != listeGroupesMotDePasse.size() - 1) message.append(31);
    }

    QByteArray reponse = compRes->envoyerMessage(message);

    // Affichage de l'état de la requête
    if(reponse == "groupesmotdepasseok") {
        ui->stackedWidget->setCurrentIndex(0);
        QMessageBox::information(this, "Modification réussie", "Les groupes
du mot de passe ont été mis à jour.");
        rafraichirListeMotsDePasse();
    }
    else if(reponse == "groupesmotdepasseerreurinexistant") {
        ui->stackedWidget->setCurrentIndex(0);
        QMessageBox::critical(this, "Modification échouée", "Le mot de passe
n'existe pas (id_motdepasse introuvable).");
        rafraichirListeMotsDePasse();
    }
    else if(reponse.startsWith("groupesmotdepasseerreurtechnique")) {
        QMessageBox::critical(this, "Modification échouée", "Une erreur
technique a eu lieu :\n\n" + reponse.split(30).at(1));
    }
    else {
        QMessageBox::critical(this, "Erreur serveur", "Pas de réponse ou
réponse malformée du serveur. Veuillez réessayer.");
    }
}

// Annulation de la modification de la liste des groupes du mot de passe
void GestionMotsDePasse::on_annulerGroupes_clicked()
{
    QMessageBox::StandardButton retour;
    retour = QMessageBox::question(this, "Confirmation", "Les modifications
seront abandonnées. Confirmer ?");

    if(retour != QMessageBox::Yes) return;

    ui->stackedWidget->setCurrentIndex(0);
}
```

```
// Activation des boutons de modification lors de la sélection d'un mot de
passe
void GestionMotsDePasse::on_listeMotsDePasse_currentRowChanged(int
currentRow)
{
    bool actif = (currentRow != -1);
    ui->modifier->setEnabled(actif);
    ui->supprimer->setEnabled(actif);
    ui->gererGroupes->setEnabled(actif);
}


// Simulation du bouton modifier lors du double-clic sur un mot de passe
void GestionMotsDePasse::on_listeMotsDePasse_doubleClicked(const QModelIndex
&index)
{
    on_modifier_clicked();
}


// Actualisation de la liste des mots de passe
void GestionMotsDePasse::on_actualiser_clicked()
{
    rafraichirListeMotsDePasse();
}


// Réimplémenté : exécuté lors de la fermeture de la fenêtre (croix)
void GestionMotsDePasse::closeEvent(QCloseEvent * e) {
    // Message de confirmation si on est sur une fenêtre d'édition
    if(ui->stackedWidget->currentIndex() != 0) {
        QMessageBox::StandardButton retour;
        retour = QMessageBox::question(this, "Confirmation", "Les
modifications seront abandonnées. Confirmer ?");

        if(retour != QMessageBox::Yes) e->ignore();
        else {
            ui->stackedWidget->setCurrentIndex(0);
            motDePasse.id = -1;
            ui->motdepasse->clear();
            e->accept();
        }
    }
    else e->accept();
}


// Réimplémenté : exécuté lorsqu'une touche est appuyée
void GestionMotsDePasse::keyPressEvent(QKeyEvent * e) {
    // On demande la fermeture de la fenêtre si la touche est Echap
    if(e->key() == Qt::Key_Escape) {
        this->close();
        e->accept();
    }
    else e->ignore();
}
```

```
// Entre en mode gestion des groupes d'un mot de passe
void GestionMotsDePasse::on_gererGroupes_clicked()
{
    int i = ui->listeMotsDePasse->currentRow();
    if(i < 0 || i >= listeMotsDePasse.size()) return;

    listeGroupesMotDePasse.clear();
    ui->groupesMotDePasse->clear();
    listeGroupes.clear();
    ui->groupes->clear();

    motDePasse = listeMotsDePasse.at(i);

    ui->labelMotDePasse->setText(motDePasse.nomServeur + " : " +
                                 motDePasse.nom + " @ " +
motDePasse.hoteServeur);

    QByteArray message;
    QList<QByteArray> lignes;
    QList<QByteArray> champs;
    Groupe groupe;

    QByteArray groupesMotDePasse;

    message = "admin_listegroupesmotdepasse";
    message.append(30);
    message.append(QString::number(motDePasse.id));
    groupesMotDePasse = compRes->envoyerMessage(message);

    lignes = groupesMotDePasse.split(30);

    if(lignes.at(0) == "admin_listegroupesmotdepasse" && lignes.at(1).size()
> 0) {
        listeGroupesMotDePasse.clear();
        ui->groupesMotDePasse->clear();
        for(int m = 1; m < lignes.size(); m++) {
            champs = lignes.at(m).split(31);
            groupe.id = champs.at(0).trimmed().toInt();
            groupe.nom = champs.at(1).trimmed();
            listeGroupesMotDePasse.append(groupe);
            ui->groupesMotDePasse->addItem(groupe.nom);
        }
    }

    lignes.clear();
    champs.clear();

    QByteArray groupes;
    groupes = compRes->envoyerMessage("admin_listegroupes");
    bool trouve;

    lignes = groupes.split(30);

    if(lignes.at(0) == "admin_listegroupes" && lignes.at(1).size() > 0) {
        listeGroupes.clear();
        ui->groupes->clear();
        for(int m = 1; m < lignes.size(); m++) {
            champs = lignes.at(m).split(31);
            groupe.id = champs.at(0).trimmed().toInt();
            groupe.nom = champs.at(1).trimmed();
```

```
            for(int g = 0; g < listeGroupesMotDePasse.size(); g++) {
                if(listeGroupesMotDePasse.at(g).id == groupe.id) {
                    trouve = true;
                    break;
                }
            }
            if(!trouve) {
                listeGroupes.append(groupe);
                ui->groupes->addItem(groupe.nom);
            }
            trouve = false;
        }
    }

    ui->stackedWidget->setCurrentIndex(2);
}
```

6. ComposantSauvegarde

Ce composant permet de télécharger une sauvegarde du serveur.

a. ComposantSauvegarde.h

```
#ifndef COMPOSANTSAUVEGARDE_H
#define COMPOSANTSAUVEGARDE_H

#include <QDialog>

class ComposantReseau;

namespace Ui {
class ComposantSauvegarde;
}

class ComposantSauvegarde : public QDialog
{
    Q_OBJECT

public:
    explicit ComposantSauvegarde(ComposantReseau *p_compRes, QWidget *parent
= 0);
    ~ComposantSauvegarde();

private slots:
    void on_demarrerSauvegarde_clicked();

private:
    Ui::ComposantSauvegarde *ui;
    ComposantReseau *compRes;

    void closeEvent(QCloseEvent *);
    void keyPressEvent(QKeyEvent *);
};

#endif // COMPOSANTSAUVEGARDE_H
```

b. ComposantSauvegarde.cpp

```
#include "ComposantSauvegarde.h"
#include "ComposantReseau.h"

#ifdef Q_OS_WIN
#include "ui_ComposantSauvegarde_win.h"
#else
#include "ui_ComposantSauvegarde_unix.h"
#endif

#include <QFileDialog>
#include <QMessageBox>
#include <QCloseEvent>
#include <QKeyEvent>
```

```
ComposantSauvegarde::ComposantSauvegarde(ComposantReseau *p_compRes, QWidget
*parent) :
    QDialog(parent),
    ui(new Ui::ComposantSauvegarde),
    compRes(p_compRes)
{
    ui->setupUi(this);
}

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


// Lance le processus de sauvegarde
void ComposantSauvegarde::on_demarrerSauvegarde_clicked()
{
    //Envoi du message et réception de la réponse
    QByteArray reponse = compRes->envoyerMessage("admin_sauvegarder");

    // Vérification de la réponse
    if(reponse.startsWith("sauvegardeok")) {

        // Demande de l'emplacement de la sauvegarde à écrire
        QString nomFichier = QFileDialog::getSaveFileName(this, "Emplacement
de la sauvegarde", "",
                                                          "Sauvegarde SPM
(*.spm)");

        QFile fichier(nomFichier);
        if(fichier.open(QFile::WriteOnly | QFile::Truncate)) {
            // On doit lire la sauvegarde à partir de la position 28
            // 12 caractères pour "sauvegardeok", 16 caractètes pour le mot
de passe
            fichier.write(reponse.mid(28));
            fichier.close();

            QMessageBox::information(this, "Sauvegarde réussie",
                                     "La sauvegarde a été effectuée avec
succès.\n"
                                     "Veuillez noter le mot de passe puis
fermer la fenêtre.");

            ui->motDePasse->setText(reponse.mid(12, 16));
        }
        else {
            QMessageBox::warning(this, "Sauvegarde échouée", "Impossible
d'écrire dans le fichier.");
        }
    }
    else QMessageBox::warning(this, "Sauvegarde échouée", "La sauvegarde a
échoué. Veuillez réessayer.");
}
```

```
// Réimplémenté : exécuté lors de la fermeture de la fenêtre (croix)
void ComposantSauvegarde::closeEvent(QCloseEvent * e) {
    ui->motDePasse->clear();
    e->accept();
}


// Réimplémenté : exécuté lorsqu'une touche est appuyée
void ComposantSauvegarde::keyPressEvent(QKeyEvent * e) {
    // On demande la fermeture de la fenêtre si la touche est Echap
    if(e->key() == Qt::Key_Escape) {
        this->close();
        e->accept();
    }
    else e->ignore();
}
```

III. ComposantCryptographie

Le code source relatif au ComposantCryptographie est identique pour le serveur et le client. Il permet de réaliser les différentes opérations de cryptage/décryptage des données avec les méthodes RSA et AES.

1. ComposantCryptographie.h

```
#ifndef COMPOSANTCRYPTOGRAPHIE_H
#define COMPOSANTCRYPTOGRAPHIE_H

// Librairies OpenSSL
#include <openssl/pem.h>
#include <openssl/aes.h>
#include <openssl/rand.h>

// Librairies Qt
#include <QObject>

class ComposantCryptographie : public QObject
{
    Q_OBJECT

public:
    ComposantCryptographie();
    ~ComposantCryptographie();

    // RSA
    bool loadClePubliqueRSA(const QString fichier);
    bool loadClePriveeRSA(const QString fichier);
    QByteArray getClePubliqueRSA();
    QByteArray decrypterRSA(const QByteArray messageCrypte);
    QByteArray encrypterRSA(const QByteArray messageClair);
    static bool genererClesRSA(const QString fichierClePriveeRSA, const
QString fichierClePubliqueRSA);

    // AES
    bool genererCleAES();
    QByteArray getCleAES();
    bool setCleAES(const QByteArray cle);
    QByteArray decrypterAES(const QByteArray messageCrypte);
    QByteArray encrypterAES(const QByteArray messageClair);

private:
    // RSA
    QString fichierCleRSA;
    RSA * clePriveeRSA;
    RSA * clePubliqueRSA;

    // AES
    EVP_CIPHER_CTX * encAES;
    EVP_CIPHER_CTX * decAES;
    unsigned char * cleAES;
    bool setCleAES();

};

#endif
```

2. ComposantCryptographie.cpp

```
#include "ComposantCryptographie.h"

#include <QFile>

#ifdef Q_OS_WIN
#include <openssl/applink.c>
#endif

ComposantCryptographie::ComposantCryptographie()
{
    clePubliqueRSA = NULL;
    clePriveeRSA = NULL;
    encAES = NULL;
    decAES = NULL;
    cleAES = NULL;
    fichierCleRSA = "";
}


// Destructeur : libère toutes les clés RSA et AES
ComposantCryptographie::~ComposantCryptographie()
{
    RSA_free(clePubliqueRSA);

    RSA_free(clePriveeRSA);

    if(encAES) {
        EVP_CIPHER_CTX_cleanup(encAES);
        free(encAES);
        encAES = NULL;
    }

    if(decAES) {
        EVP_CIPHER_CTX_cleanup(decAES);
        free(decAES);
        decAES = NULL;
    }

    free(cleAES);
    cleAES = NULL;
}
```

```
/****************************** RSA ******************************/

// Charge une clé publique RSA à partir d'un fichier
// Renvoie true si le chargement a réussi, false sinon
bool ComposantCryptographie::loadClePubliqueRSA(const QString fichier) {

    // Vérification de l'existence du fichier
    if(!QFile::exists(fichier)) return false;

    // Ouverture du fichier
    FILE * fp = fopen(fichier.toUtf8().data(), "r");
    if(fp == NULL) return false;

    // Chargement de la clé
    PEM_read_RSA_PUBKEY(fp, &clePubliqueRSA, NULL, NULL);
    fclose(fp);
    if(clePubliqueRSA == NULL) return false;

    fichierCleRSA = fichier;

    return true;

}


// Charge une clé privée RSA à partir d'un fichier
// Renvoie true si le chargement a réussi, false sinon
bool ComposantCryptographie::loadClePriveeRSA(const QString fichier) {

    // Vérification de l'existence du fichier
    if(!QFile::exists(fichier)) return false;

    // Ouverture du fichier
    FILE * fp = fopen(fichier.toUtf8().data(), "r");
    if(fp == NULL) return false;

    // Chargement de la clé
    PEM_read_RSAPrivateKey(fp, &clePriveeRSA, NULL, NULL);
    fclose(fp);
    if(clePriveeRSA == NULL) return false;

    return true;
}


// Retourne la clé publique RSA au format PEM
QByteArray ComposantCryptographie::getClePubliqueRSA() {
    if(fichierCleRSA == "") return QByteArray();
    QFile fichier(fichierCleRSA);
    if(!fichier.open(QFile::ReadOnly)) return QByteArray();
    QByteArray cleRSA = fichier.readAll();
    fichier.close();
    return cleRSA;
}
```

```
// Crypte un message avec RSA avec la clé publique actuellement chargée
QByteArray ComposantCryptographie::encrypterRSA(const QByteArray
messageClair) {

    // Initialisations
    QByteArray messageCrypte, aux;
    int num;
    int size = RSA_size(clePubliqueRSA);
    unsigned char * from;
    unsigned char * to = (unsigned char *)malloc(size);
    size -= 11;

    // Le message doit être découpé en plusieurs morceaux pour être crypté
    for(int c = 0; c < messageClair.size(); c += size) {
        aux = messageClair.mid(c, size);
        from = (unsigned char *)aux.data();
        num = RSA_public_encrypt(aux.size(), from, to, clePubliqueRSA,
RSA_PKCS1_PADDING);
        aux = QByteArray((char *)to, num);
        messageCrypte.append(aux);
    }

    free(to);

    return messageCrypte;
}


// Décrypte un message crypté avec RSA avec la clé privée actuellement
chargée
QByteArray ComposantCryptographie::decrypterRSA(const QByteArray
messageCrypte) {

    // Initialisations
    QByteArray messageClair, aux;
    int num;
    int size = RSA_size(clePriveeRSA);
    unsigned char * from;
    unsigned char * to = (unsigned char *)malloc(size);

    // Le message doit être découpé en plusieurs morceaux pour être décrypté
    for(int c = 0; c < messageCrypte.size(); c += size) {
        aux = messageCrypte.mid(c, size);
        from = (unsigned char *)aux.data();
        num = RSA_private_decrypt(aux.size(), from, to, clePriveeRSA,
RSA_PKCS1_PADDING);
        messageClair.append(QByteArray((char *)to, num));
    }

    free(to);

    return messageClair;
}
```

```
// Génère un couple de clés RSA privée/publique et les stocke dans des
fichiers
bool ComposantCryptographie::genererClesRSA(const QString
fichierClePriveeRSA, const QString fichierClePubliqueRSA) {

    // Initialisations
    RSA * clesRSA = RSA_new();
    BIGNUM * e = BN_new();

    // Génération de l'exposant et de la paire de clés RSA
    BN_set_word(e, 65537);
    if(!RSA_generate_key_ex(clesRSA, 2048, e, NULL)) return false;

    // Ouverture et écriture du fichier de la clé privée
    FILE * fp = fopen(fichierClePriveeRSA.toUtf8().data(), "w");
    if(fp == NULL) return false;

    if(!PEM_write_RSAPrivateKey(fp, clesRSA, NULL, NULL, 0, NULL, NULL))
return false;
    fclose(fp);

    // Ouverture et écriture du fichier de la clé publique
    FILE * fp2 = fopen(fichierClePubliqueRSA.toUtf8().data(), "w");
    if(fp2 == NULL) return false;

    if(!PEM_write_RSA_PUBKEY(fp2, clesRSA)) {
        QFile::remove(fichierClePriveeRSA);
        return false;
    }
    fclose(fp2);

    RSA_free(clesRSA);

    return true;
}
```

```
/****************************** AES ******************************/

// Génère une clé AES
// Renvoie true si la génération a réussi, false sinon
bool ComposantCryptographie::genererCleAES() {

    // Allocation de mémoire pour la clé AES
    if(cleAES) free(cleAES);
    cleAES = (unsigned char *)malloc(32);
    if(cleAES == NULL) return false;

    // Génération aléatoire de la clé
    int i = RAND_bytes(cleAES, 32);
    if(i == 0) {
        free(cleAES);
        cleAES = NULL;
        return false;
    }

    // Chargement de la clé
    if(!setCleAES()) {
        free(cleAES);
        cleAES = NULL;
        return false;
    }

    return true;
}


// Renvoie la clé AES actuelle
QByteArray ComposantCryptographie::getCleAES() {

    // Vérification de l'existence de la clé
    if(cleAES == NULL) return QByteArray();

    return QByteArray((char *)cleAES, 32);
}


// Charge une clé AES
// Renvoie true si le chargement a réussi, false sinon
bool ComposantCryptographie::setCleAES(const QByteArray cle) {

    // Vérification de la taille de clé
    if(cle.size() != 32) return false;

    // Allocation de mémoire pour la clé AES
    cleAES = (unsigned char *)malloc(32);
    if(cleAES == NULL) return false;

    // Copie de la clé
    memcpy((void*)cleAES, (void *)cle.data(), 32);

    // Chargement de la clé
    if(!setCleAES()) {
        free(cleAES);
        cleAES = NULL;
        return false;
    }

    return true;
}
```

```
// Génère les contextes encAES et decAES à partir de la cleAES actuelle
// Renvoie true si la génération a réussi, false sinon
bool ComposantCryptographie::setCleAES() {

    // Vérification de l'existence de la clé
    if(cleAES == NULL) return false;

    // Initialisations key et iv
    unsigned char key[32], iv[32];
    memset(key, 0, 32);
    memset(iv, 0, 32);

    // Génération des key et iv
    int i = EVP_BytesToKey(EVP_aes_256_cbc(), EVP_sha1(), NULL, cleAES, 32,
5, key, iv);
    if (i == 0) return false;

    // Génération du contexte encAES
    encAES = (EVP_CIPHER_CTX*)malloc(sizeof(EVP_CIPHER_CTX));
    EVP_CIPHER_CTX_init(encAES);
    i = EVP_EncryptInit_ex(encAES, EVP_aes_256_cbc(), NULL, key, iv);
    if(i == 0) {
        EVP_CIPHER_CTX_cleanup(encAES);
        free(encAES);
        encAES = NULL;
        return false;
    }

    // Génération du contexte decAES
    decAES = (EVP_CIPHER_CTX*)malloc(sizeof(EVP_CIPHER_CTX));
    EVP_CIPHER_CTX_init(decAES);
    EVP_DecryptInit_ex(decAES, EVP_aes_256_cbc(), NULL, key, iv);
    if(i == 0) {
        EVP_CIPHER_CTX_cleanup(encAES);
        free(encAES);
        encAES = NULL;
        EVP_CIPHER_CTX_cleanup(decAES);
        free(decAES);
        decAES = NULL;
        return false;
    }

    return true;
}
```

```
// Décrypte un message crypté avec AES avec la clé actuellement chargée
QByteArray ComposantCryptographie::decrypterAES(const QByteArray
messageCrypte) {

    // Vérifications du contexte de décryptage et du message crypté
    if(decAES == NULL || messageCrypte.size() == 0) return QByteArray();

    // Initialisations
    unsigned char * ciphertext = (unsigned char *)messageCrypte.data();
    int p_len = messageCrypte.size(), f_len = 0;
    unsigned char *plaintext = (unsigned char *)malloc(p_len);

    // Décryptage
    if(!EVP_DecryptInit_ex(decAES, NULL, NULL, NULL, NULL))
        return QByteArray();

    if(!EVP_DecryptUpdate(decAES, plaintext, &p_len, ciphertext,
messageCrypte.size()))
        return QByteArray();

    if(!EVP_DecryptFinal_ex(decAES, plaintext+p_len, &f_len))
        return QByteArray();

    QByteArray messageClair((char *)plaintext, p_len + f_len);

    free(plaintext);

    return messageClair;
}


// Crypte un message avec AES avec la clé actuellement chargée
QByteArray ComposantCryptographie::encrypterAES(const QByteArray
messageClair) {

    // Vérification du contexte d'encryptage
    if(encAES == NULL) return QByteArray();

    // Initialisations
    unsigned char * plaintext = (unsigned char *)messageClair.data();
    int c_len = messageClair.size() + AES_BLOCK_SIZE - 1, f_len = 0;
    unsigned char *ciphertext = (unsigned char *)malloc(c_len);

    // Encryptage
    if(!EVP_EncryptInit_ex(encAES, NULL, NULL, NULL, NULL))
        return QByteArray();

    if(!EVP_EncryptUpdate(encAES, ciphertext, &c_len, plaintext,
messageClair.size()))
        return QByteArray();

    if(!EVP_EncryptFinal_ex(encAES, ciphertext+c_len, &f_len))
        return QByteArray();

    QByteArray messageCrypte((char *)ciphertext, c_len + f_len);

    free(ciphertext);

    return messageCrypte;
}
```

CONCLUSION

Ce dossier contient le code source final du logiciel prêt à être déployé pour utilisation par le client.