# Module Chat

## Acteurs du module

Toutes les méthodes possèdent un entête précis et clair dans le code, détaillant leur fonctionnement. Pour cette raisons, nous ne nous étendrons pas (ou peu) sur celles-ci dans cette partie.

### ControllerChat

Il s’agit du contrôleur du module. Il gère notamment les interactions entre le modèle (**ModeleChator**), la vue (**ViewChat**) et le serveur, et agit donc comme « noyau central » du module.  
Il est créé dans le *main.cpp* principal de l’application, en même temps que tous les autres contrôleurs, et devient actif directement après qu’une demande de connexion ait été validée par le serveur.

Il connaît, par attributs privés :

* Sa vue (**ViewChat**).
* Le modèle (**ModeleChator**).
* Un objet de type « pointeur sur ModelUser » représentant l’utilisateur courant (connecté) qui a été paramétré lors de la connexion de ce dernier.
* Les différents objets en rapport avec la communication avec le serveur.
* Le contrôleur du module Salle (**ControllerRoom**), qui lui sera utile pour ouvrir les différentes fenêtres de gestion de salles (ajout, édition, …).
* L’objet **Cryptor** lui permettant d’implémenter la sécurité.

Tout ce qui doit transiter de la vue à une autre entité, ou d’une entité à la vue, passe par ce contrôleur ; il effectue les contrôles nécessaires et traite les données avant de les envoyer plus loin. C’est aussi lui qui lancera la vue au moment opportun.

### ViewChat

Il s’agit de la vue principale du module. Elle contient notamment l’interface graphique (*viewChat.ui* – voir plus loin pour plus de détails) et est lancée par le contrôleur (**ControllerChat**) après la connexion d’un utilisateur.

Son but principal est de mettre en style et d’afficher les différentes données reçues par le modèle et le serveur, qui transitent via le contrôleur.

### ViewMembershipRequests

Cette vue contient une interface graphique permettant de gérer les demandes d’adhésions aux salles privées auxquelles l’utilisateur connecté est administrateur. Elle est lancée par le contrôleur (**ControllerChat**) lorsque l’utilisateur appuie sur le menu « Notifications > Demandes d’adhésion… » (notifié par la vue **ViewChat**).

Cette vue est composée d’un arbre (QTreeWidget – plus de détails plus bas) contenant les demandes d’adhésion, ainsi que de deux boutons permettant à l’administrateur d’accepter ou de refuser une demande (pour plus de détails, se référer au chapitre « Liste des événements pouvant survenir » -> « Un utilisateur a fait une demande d’adhésion sur une salle privée dont je suis administrateur »).

### ViewAbout

Cette vue affiche les détails d’à-propos de l’application. Elle est lancée par le contrôleur (**ControllerChat**) lorsque l’utilisateur appuie sur le menu « ? > À propos… » (notifié pr la vue **ViewChat**).

## Détails sur l’interface graphique de la vue principale

L’interface graphique de ce module a été réalisée à l’aide du générateur intégré à l’IDE Qt Creator.



Figure 1 : Création de l'interface graphique avec Qt Creator.

Elle est lancée et stockée dans la vue du module, **ViewChat**.

### Présentation

Elle se présente de la manière suivante :

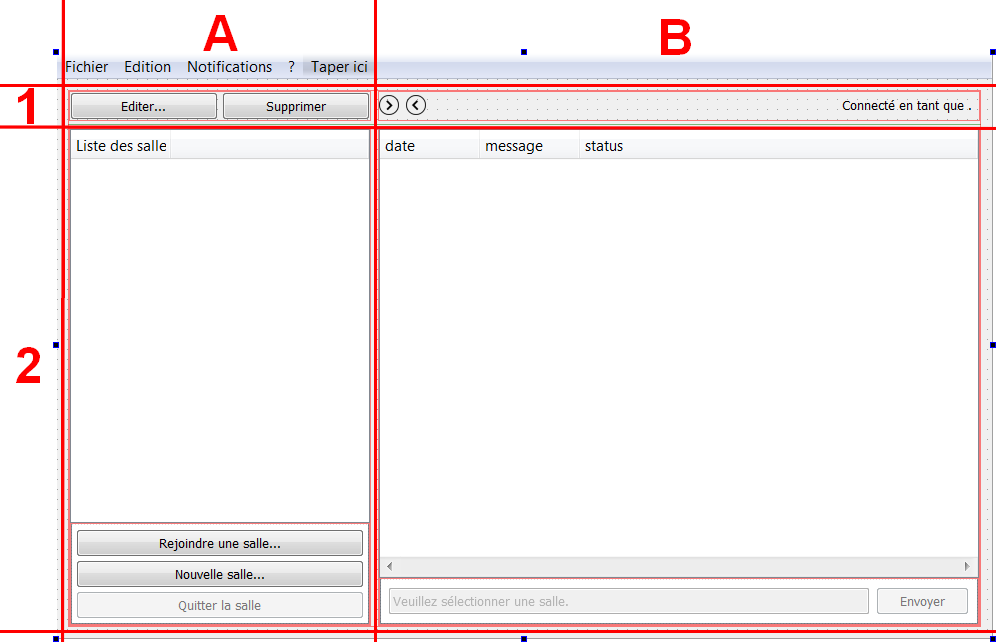


Figure 2 : Interface graphique du module Chat.

Les blocs sont organisés en « layouts » (mises en page), de la manière suivante (en essayant de rester bref) : la fenêtre contient un QGridLayout (mise en page dans une grille contenant des cellules) de deux lignes (1 et 2) et deux colonnes (A et B). La première ligne (1) est fine, et ne s’agrandit qu’en largeur (jamais en hauteur) ; elle contient des layouts de type QHBoxLayout (alignement horizontal des éléments). La deuxième ligne (2) s’agrandit par contre en largeur et en hauteur. Elle contient elle-même des layouts de type QVBoxLayout (alignement vertical des éléments) contenant deux éléments : celui du haut (les arbres de salles (A2) et de messages (B2)) qui peut s’agrandir verticalement, alors que celui du bas qui ne le peut pas.

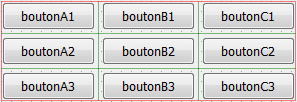
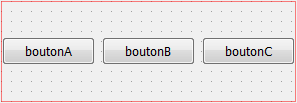
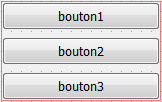
6

Figure 3 : Exemple de QGridLayout (3x3).

Figure 4 : Exemple de QHBoxLayout.

Figure 5 : Exemple de QVBoxLayout.

Trois blocs principaux sont présents dans cette interface :

1. La barre de menu, permettant d’accéder à diverses fonctionnalités :
   1. Quitter l’application : Fichier > Quitter
   2. Afficher les informations du compte de l’utilisateur courant (anciennement édition de compte) : Edition > Informations utilisateur…
   3. Gérer les demandes d’adhésion aux salles privées auxquelles nous sommes administrateur : Notifications > Demandes d’adhésion…
   4. Afficher une fenêtre d’à-propos : « ? » > A propos.
2. Le bloc de gestion des salles (A, sur le schéma), il est composé (de haut en bas) de :
   1. (1) Deux boutons d’administration de salle, permettant de l’éditer, respectivement de la supprimer ; ils sont visibles et actifs uniquement si l’utilisateur courant est administrateur de la salle sélectionnée.
   2. (2) La liste des salles, représentée par un arbre (fonctionnement détaillé plus bas) donc les nœuds de premier niveau correspondent au nom des salles, et ceux de deuxième niveau à ceux des utilisateurs présents dans la salle. Les utilisateurs connectés apparaissent en gras.

**SCREENSHOT**

* 1. (2) Trois boutons permettant respectivement de rejoindre une salle (ouvre une nouvelle fenêtre), d’en créer une nouvelle (pareil) ou de quitter celle qui est sélectionnée dans l’arbre (après confirmation).

1. Le bloc de messagerie à proprement parler (B, sur le schéma), qui est composé de (de haut en bas, gauche à droite) :
   1. (1) Deux boutons permettant respectivement de dérouler tous les messages contenus dans les nœuds (a), ainsi que de les enrouler (b).

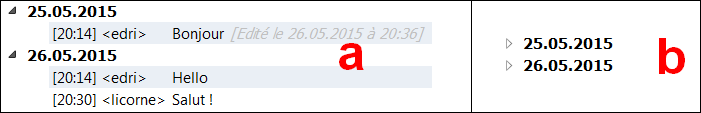


Figure 6 : Dérouler les messages (a), ou les enrouler (b).

* 1. (1) Un libellé indiquant le nom de l’utilisateur courant.
  2. (2) Un arbre contenant la liste des messages de la salle, organisés par date ; les nœuds de premier niveau de cet arbre représentent les différentes dates, et ceux de deuxième niveau les messages de la salle. Un message est ordré en trois colonnes :
     1. La date et l’utilisateur.
     2. Le contenu du message.
     3. La date de dernière édition du message (facultative).

Les messages de l’utilisateur courant sont mis en valeur. Lorsque l’on double-clique sur l’un de ceux-ci (peu importe la colonne), il devient éditable ; il est alors possible de le modifier, ce qui enverra une requête au serveur.



Figure 7 : Edition d'un message.

Il est aussi possible de réaliser un clic-droit sur l’un de nos messages, afin de faire apparaître un menu contextuel qui permet de l’éditer (comme ci-dessus) ou de le supprimer après confirmation.

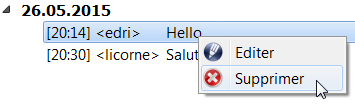


Figure 8 : Menu contextuel qui apparaît lors d'un clic-droit sur l'un de nos messages.

* 1. (2) Un champ de texte permettant d’écrire un message, ainsi que le bouton permettant de l’envoyer au serveur.

### Détails techniques

Les différentes actions réalisables sur les composants (clic sur un bouton, appui sur une touche, etc.) sont réalisées à l’aide de signaux et de slots, afin de respecter les conventions de Qt. En deux mots, l’action envoie un signal qui est récupéré dans une méthode (que l’on appellera un slot, dans ce contexte) qui agit en conséquence.

De nombreux composants C++ possèdent leur « homonyme » dans Qt, qui commencent tous par la lettre ‘Q’ par convention : string <=> QString; unsigned int <=> quint32, etc. Ils permettent notamment d’assurer une portabilité robuste entre les systèmes (un quint32 fera toujours 32 bits, quel que soit le système utilisé, par exemple), et de simplifier les opérations (un QString agira plus comme un String en Java que comme un string en C++, par exemple).  
A noter que l’objet de type QVariant accepte n’importe quelle valeur, un peu à la manière d’un Object en Java.

#### Arbres

Les arbres contenus dans l’interface (liste des salles et liste des messages) sont des composants du type QTreeWidget, qui contiennent des nœuds de type QTreeWidgetItem (qui contient du texte et éventuellement des données « cachées » sur lesquelles nous pourrons travailler, le tout ordré par colonnes). Ces composants possèdent de nombreux paramètres et de nombreuses fonctionnalités qui sont détaillés précisément dans la documentation officielle de Qt, raison pour laquelle ils ne seront pas tous expliqués ici.

Lorsque nous souhaitons ajouter un nœud enfant à l’arbre, il faut :

1. Créer un nouvel objet de type (pointeur sur) QTreeWidgetItem.
2. Lui ajouter du texte dans les colonnes, à l’aide de la méthode setText(int colonne, const QString& texte).

Par exemple : « item->setText(0, "Coucou "); » ajoutera le texte « Coucou » dans la première colonne du nœud.

1. Lui ajouter éventuellement des données « cachées », à l’aide de la méthode setData(int colonne, in role, const QVariant& value). Le rôle à mettre par défaut est Qt::UserRole ; il devrai être indiqué aussi lors de la récupération de la donnée.

Par exemple : « item->setData(0, Qt::UserRole, 50); » ajoutera la donnée « 50 » dans la première colonne du nœud.

Pour pouvoir récupérer ces données par la suite, il suffira d’appeler la méthode data(int colonne, in role), avec éventuellement une fonction de conversion.  
Par exemple : « int nb = item->data(0, Qt::UserRole).toInt(); » récupèrera la donnée ajoutée précédemment.

1. Eventuellement lui ajouter d’autres paramètres (police d’écriture, taille, couleur, etc.).

A noter que Qt implémente un système de fanions (flags) permettant de paramétrer le comportement de certains objets, donc les QTreeWidgetItem : il est possible d’en ajouter à l’aide de la méthode setFlags(Qt::ItemFlags flags) ; nous pouvons en placer plusieurs en série à l’aide de l’opérateur OU binaire (« | »). La documentation possède de nombreux détails quant à ce système de fanions ; voici cependant quelques exemples utilisés dans le code :

1. Qt::NoItemFlags => aucun fanion (le nœud est désactivé).
2. Qt::ItemIsEnabled => le nœud est activé (n’apparaît plus en grisé).
3. Qt::ItemIsEditable => le nœud est éditable (à nous de gérer le signal).
4. Qt::ItemIsSelectable => le nœud est sélectionnable.
5. Insérer le nœud dans l’arbre à la position souhaitée, à l’aide de la méthode insertTopLevelItem(int index, QTreeWidgetItem\* item), si nous souhaitons l’ajouter en tant qu’enfant de premier niveau. Si nous souhaitons l’ajouter en tant qu’enfant de niveaux inférieurs, il faut y aller récursivement, en récupérant le nœud de premier niveau concerné, et en lui ajoutant l’objet en enfant : topLevelItem(int index)->insertChild(int index, messageItem). Nous pouvons aller encore plus loin avec la fonction child(int index)…

#### Menus contextuels

Le menu contextuel est représenté par un objet de type QMenu, stocké dans notre cas comme attribut privé de la vue, afin d’éviter de le recréer à chaque fois (vu qu’il pourra être appelé souvent). Il faut indiquer à cet objet quel est son parent, à l’aide de son constructeur (dans notre cas, la vue, donc *this*), afin qu’il sache où et comment il doit se lancer.

Nous pouvons ensuite ajouter des éléments à ce menu, qui sont représentés par des objets de type QAction, pouvant posséder une icône (facultative) et un texte, à l’aide de la méthode QAction\* addAction(*[icône, texte, …]*).  
Par exemple : « QAction\* editAct = \_menu->addAction(QIcon(":/icons/img/edit.png"), tr("Editer")); » ajoute l’élément « Editer » au menu.

Finalement, il faut exécuter le menu à l’aide de la méthode QAction\* exec(), puis récupérer les actions avec des conditions, par exemple :

// Nous plaçons le menu sur le curseur de la souris, dans l'arbre des messages.

QAction\* act = \_menu->exec(\_ui->tre\_messages->viewport()->mapToGlobal(pos));

if (act == editAct)

{

// FAIRE QUELQUE CHOSE...

}

#### Messages de confirmation

Les différents messages de confirmation qui peuvent apparaître sont des objets de type QMessageBox (qui héritent de la classe QDialog), à partir desquels nous pouvons appeler des méthodes statiques, selon le type de message désiré :

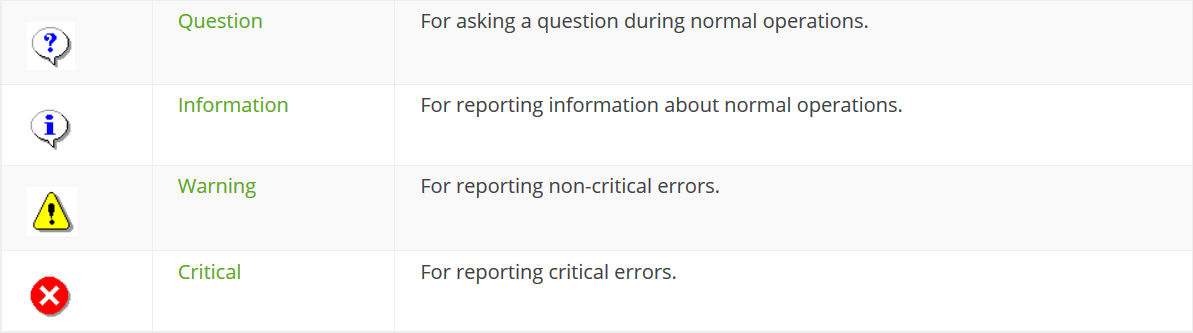


Figure 9 : Les différents types de messages de QMessageBox disponibles, tiré de la documentation de Qt (http://doc.qt.io/qt-4.8/qmessagebox.html).

Par exemple, pour construire un message de type « Warning » :

int ret = QMessageBox::warning(this, tr("Attention"),

tr("Êtes-vous sûr de vouloir supprimer cette

salle ?"),

tr("Oui"), tr("Non"));

Cela fera apparaître une fenêtre jaillissante (pop-up) avec le titre « Attention », et deux boutons « Oui » et « Non ». La valeur entière de retour permet de connaître le bouton qui a été cliqué (0 => premier bouton – Oui ; 1 => deuxième bouton – Non). Résultat :

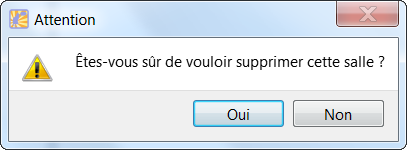


Figure 10 : Pop-up de suppression d'une salle.

## Procédure post-connexion

Lorsque la connexion a été effectuée avec succès, le « module Chat » démarre : le **ControllerUser** passe l’objet représentant l’utilisateur connecté au **ControllerChat**, lui indique qu’il peut afficher sa vue (**ViewChat**), puis se ferme.

Dans un même temps, le serveur envoie tout d’abord la liste de toutes les salles auxquelles l’utilisateur connecté appartient (contenant aussi leurs messages respectifs), ainsi que la liste des utilisateurs qui appartiennent au moins à l’une de ces salles.

Côté client, la méthode « join » du **ClientControllerInput** insert en premier lieu chaque salle, chaque message et chaque utilisateur reçus dans le modèle – via la **ControllerChat** – puis indique à ce dernier qu’il peut charger la liste des salles dans la vue (à l’aide de la méthode « loadUserRooms »). A noter que l’ajout d’une salle privée au modèle s’occupe aussi de récupérer et de stocker sa clé secrète.

Cette méthode ajoute les salles une par une à la vue, ainsi que la liste des utilisateurs, pour chacune d’entre-elles.

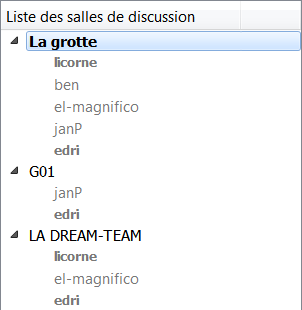


Figure 11 : Liste des salles de discussion et de leurs utilisateurs, dans la vue.

Chacun de ces objets contient des données cachées, utiles pour les traitements auxiliaires :

* Salles : l’ID de la salle dans la première colonne, ainsi que le nombre de nouveaux messages non-lus dans la deuxième (qui vaut 0, par défaut).
* Utilisateurs : l’ID de l’utilisateur dans la première colonne.

Une fois les salles chargées, la vue sélectionne la première de la liste, puis charge ses messages dans l’arbre de droite. Ceci se fait en envoyant le signal « requestLoadRoomMessages » au contrôleur, avec l’ID de la salle en paramètre ; celui-ci le rattrape dans le slot « loadRoomMessages » qui récupère la liste des messages depuis le modèle et les passe à la vue. A noter qu’il s’occupe aussi d’indiquer à cette dernière si les boutons d’administration (« Editer » et « Supprimer ») s’affichent ou non, selon le rôle de l’utilisateur courant.



Figure 12 : Boutons d'administration d'une salle, affichés uniquement si l’utilisateur est administrateur de la salle sélectionnée.

L’ajout des messages à l’arbre se déroule de la manière suivante dans la vue :

1. Pour chaque message de la liste de messages reçue :
   1. Créer un nouvel objet du type QTreeWidgetItem (qui représente un enfant d’un arbre Qt) contenant la date du message dans la première colonne, son contenu dans la deuxième, et éventuellement sa dernière date de modification dans la troisième.  
      Chacun de ces objets contient des données cachées, utiles pour les traitements auxiliaires : la date du message, son ID, ainsi qu’une variable booléenne indiquant si le message appartient à l’utilisateur courant (*true*), ou non (*false*).
   2. Si le message appartient à l’utilisateur courant, il sera mis en évidence.



Figure 13 : Mise en évidence des messages de l'utilisateur courant (ici, licorne).

* 1. La vue regarde ensuite si l’arbre possède déjà un nœud correspondant à la date du message ; deux actions sont possibles à partir de là :
     1. Le nœud existe déjà => ajout du message à la fin de ce dernier.
     2. Le nœud n’existe pas encore => il faut tout d’abord le créer, l’ajouter après le dernier nœud existant, puis finalement y ajouter le message.

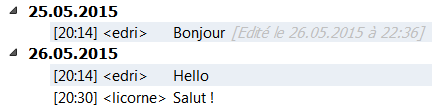


Figure 14 : Des messages dans des nœuds représentant des dates.

A noter que le serveur envoie la liste des messages dans l’ordre croissant de leur date de création ; ceci nous permet d’ajouter le message courant ou le nouveau nœud simplement à la fin de la liste pour assurer l’ordre, sans avoir à y réfléchir plus.

1. Redimensionner les colonnes de l’arbre, et s’y déplacer à la fin.

Une fois les messages chargés, le module est considéré comme lancé et chargé, et attend désormais des événements.

## Liste des événements pouvant survenir

*Dans les cas qui suivent, nous identifierons l’utilisateur actuellement connecté par « moi », ou « je », afin de simplifier les descriptions.*

#### J’ai sélectionné une autre salle que celle qui était sélectionnée auparavant dans l’arbre

Lorsque l’utilisateur change la salle sélectionnée dans l’arbre des salles, il faut recharger les messages contenus dans l’arbre des messages. Pour cela, la vue notifie le contrôleur qui lui renvoie la liste des messages de la nouvelle salle sélectionnée, récupérée dans le modèle du client (pas de requête au serveur, car nous connaissons déjà tous les messages de toutes les salles). Cela assure un accès rapide aux données, empêchant ainsi que des problèmes de latence surviennent.

#### Envoi d’un message

Lorsque j’appuie sur le bouton « Envoyer » et que j’ai entré un message, son texte est converti en format binaire (après avoir été chiffré avec la clé de la salle, si je me trouve dans une salle privée) et est encapsulé dans un objet de type ModelMessage. Cet objet est envoyé au serveur qui va le stocker dans la base de données.

Côté vue, rien ne se passe lors de cet événement, si ce n’est l’effacement du contenu du champ de texte du message. On ne peut en effet pas ajouter le message à l’arbre sans être sûr qu’il ait été reçu par le serveur, pour des questions de cohérence.

#### Réception d’un nouveau message ou d’un message édité

Lorsqu’un utilisateur a envoyé/édité un message et que le serveur l’a reçu, ce dernier va le renvoyer en retour (un objet ModelMessage – avec un contenu binaire) à tous les utilisateurs présents dans la salle concernée (y compris l’utilisateur qui a envoyé le message, et qui a besoin d’une confirmation).

Le message est reçu par le contrôleur qui va tout d’abord le déchiffrer, si la salle du message est privée. Il va ensuite agir différemment selon le type du message :

* s’il s’agit d’un nouveau message, il va le stocker dans le modèle, dans la bonne salle.
* s’il s’agit d’un message qui a été édité, il va modifier le message déjà existant dans le modèle, afin qu’il soit à jour, puis va mettre à jour la dernière date de modification de ce dernier.

Le contrôleur va ensuite notifier la vue ; à partir de là, deux cas sont possibles :

* si la salle sélectionnée actuellement dans la vue correspond à la salle du message, la vue va afficher le contenu du message sous format texte dans l’arbre des messages s’il s’agit d’un nouveau message, ou qui va ajouter/mettre à jour la date d’édition de celui-ci dans la colonne la plus à droite de l’arbre.



Figure 15 : Affichage d'un message édité.

* si la salle sélectionnée actuellement dans la vue ne correspond pas à la salle du message et qu’il s’agit d’un nouveau message, la vue va ajouter une notification à côté de la salle concernée, indiquant qu’un nouveau message non-lu est disponible. Le nombre de nouveaux messages non-lus est stocké à l’aide d’une donnée cachée liée à la salle, dans l’arbre.

A noter que s’il s’agit d’une modification de message, rien ne va se passer dans ce cas.



Figure 16 : Notification indiquant de nouveaux messages non-lus, dans une salle.

#### Suppression d’un message

Lorsque l’utilisateur supprime l’un de ces messages, une notification est envoyée au serveur (en indiquant l’ID de la salle, ainsi que l’ID du message) qui le supprimera dans la base de données, et renverra une confirmation à tous les utilisateurs de la salle (contenant elle-aussi les deux IDs).

Lorsque le contrôleur reçoit la confirmation du serveur, il va tout d’abord supprimer le message du modèle, puis indiquer à la vue qu’elle peut retirer le message de l’arbre, uniquement si la salle sélectionnée correspond à la salle du message supprimé (dans le cas contraire, elle n’a rien besoin de faire).

#### Un utilisateur s’est connecté/déconnecté

Lorsqu’un utilisateur se connecte/déconnecte, le serveur notifie tous les clients susceptibles d’être intéressés par cette action, à savoir tous les utilisateurs qui ont au moins une salle en commun avec lui.

Lorsque le contrôleur reçoit cette information (l’ID de l’utilisateur, avec une variable booléenne indiquant si l’utilisateur s’est connecté (*true*) ou s’il s’est déconnecté (*false*)), il la transmet à la vue qui s’occupera simplement d’afficher l’utilisateur en gras à chaque fois qu’il apparaît dans la liste des utilisateurs d’une salle s’il s’est connecté, ou de retirer l’affichage en gras s’il s’est déconnecté.  
L’utilisateur est recherché dans l’arbre à l’aide de son ID, qui est un champ caché (data) des éléments de cet arbre.

#### Un utilisateur a rejoint/quitté une salle dans laquelle je me situe

Lorsque qu’un utilisateur rejoint/quitte une salle, le serveur notifie tous les clients susceptibles d’être intéressés, à savoir les utilisateurs de cette salle.

Lorsque le contrôleur reçoit cette information (l’ID de la salle, ainsi que l’ID de l’utilisateur), il met à jour son modèle, puis appelle la vue qui retirera/ajoutera cet utilisateur à la liste des utilisateurs de la salle.

#### Je souhaite supprimer une salle dont je suis administrateur

La suppression se déroule en deux temps : on notifie premièrement le serveur, puis ce dernier envoie une confirmation à tous les utilisateurs de la salle.

Lorsque l’administrateur clique sur le bouton « Supprimer », puis sur celui de confirmation de suppression de la salle, la vue notifie le contrôleur qui envoie une requête (contenant l’ID de la salle) au serveur. Celui-ci supprime la salle, ses messages, ainsi que les liaisons des utilisateurs à la salle, puis leur envoie une confirmation.

Lorsque le contrôleur reçoit la confirmation, il supprime la salle du modèle, puis notifie la vue qui enlèvera de la liste des salles le nœud représentant la salle.

#### Je souhaite éditer les informations d’une salle dont je suis administrateur

Lorsque l’administrateur clique sur le bouton « Editer… », le module Chat ouvre la fenêtre d’édition de salle, et le module Salle prend le relai.

Lorsque le serveur envoie la notification d’édition à tous les utilisateurs de la salle (contenant un objet de type ModelRoom), le contrôleur notifie la vue qui va mettre à jour les informations de la salle, dans l’arbre des salles.

#### J’ai été ajouté à / ai rejoint une salle

Lorsqu’un utilisateur rejoint une salle, le serveur lui envoie l’objet ModelRoom correspondant, ainsi que les différents objets ModelUser correspondant à ses utilisateurs. Le contrôleur met à jour le modèle, en y ajoutant la salle et les utilisateurs encore non-connus (c’est-à-dire, les utilisateurs qui n’ont actuellement aucune salle en commune avec moi), puis indique à la vue qu’elle peut ajouter la salle et des utilisateurs à l’arbre des salles (de la même manière que pour la procédure post-connexion, plus haut).

#### Je souhaite quitter une salle

Lorsqu’un utilisateur appuie sur le bouton « Quitter la salle », la vue notifie le contrôleur qui envoie une requête au serveur, contenant l’ID de l’utilisateur ainsi que celui de la salle.

Une fois que le serveur a traité la requête, il en renvoie une à tous les utilisateurs de la salle concernée. Le contrôleur retire l’utilisateur de la salle dans le modèle, puis notifie la vue qui va retirer l’utilisateur de la salle, dans l’arbre (s’il s’agit d’un utilisateur autre que *moi*), ou alors va retirer la salle de l’arbre (s’il s’agit de *moi*).

A noter que si je suis le dernier utilisateur de la salle, celle-ci sera supprimée définitivement.

#### Je souhaite créer une nouvelle salle

Lorsque l’utilisateur clique sur le bouton « Nouvelle salle… », le module Chat ouvre la fenêtre de création de salle, et le module Salle reprend la main.

Une fois la salle créée, le serveur envoie une notification (contenant un objet de type ModelRoom) à ses utilisateurs. Le contrôleur notifie la vue qui va ajouter la salle ainsi que ses utilisateurs à l’arbre des salles.

#### Un utilisateur a fait une demande d’adhésion sur une salle privée dont je suis administrateur

Le serveur envoie au client une requête contenant l’ID de l’utilisateur qui fait la demande, sa clé publique, ainsi que l’ID de la salle privée. Lorsqu’il reçoit cette requête, le contrôleur l’ajoute à la liste des requêtes dans le modèle (ModelRequest), puis indique à la vue principale (**ViewChat**) qu’une nouvelle notification est disponible, et rafraîchit l’arbre de la vue de gestion des demandes (**ViewMembershipRequests**).

A noter qu’à la connexion d’un administrateur, le serveur lui envoie la liste de toutes les demandes d’adhésion.

La vue principale (**ViewChat**) possède un attribut interne *\_nbNotifications* qui comptabilise le nombre de notifications disponibles et les affiche dans la barre de menu. A noter que le nombre de notifications ne peut pas être négatif, et que s’il vaut 0, la vue n’affiche simplement pas de nombre.

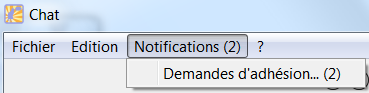


Figure 17 : Affichage des notifications (ici, 2) dans la vue principale du module Chat.

Lorsque l’administrateur appuie sur le menu « Notifications > Demandes d’adhésions… », la vue notifie le contrôleur qui va afficher la vue de gestion des demandes d’adhésion (**ViewMembershipRequests**), après l’avoir rafraîchie au préalable. Cette vue possède en effet une méthode lui permettant de mettre à jour son arbre de requêtes, à partir d’une QMap de ModelRequest. Si des demandes sont disponibles, les deux boutons « Accepter » et « Refuser » sont activés ; s’il n’y en a aucune, ils sont désactivés.

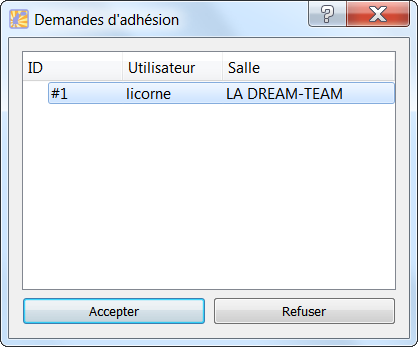


Figure 18 : fenêtre de gestion des demandes d'adhésion.

Les requêtes sont identifiées par des IDs, qui permettent au contrôleur de gérer plus facilement les requêtes. Pour chaque requête, l’administrateur possède deux choix : l’accepter ou la refuser.

Dans les deux cas, la vue se fige, puis envoie au contrôleur un signal qui contient une variable booléenne indiquant si la demande a été acceptée (*true*) ou non (*false*). Le contrôleur va ensuite envoyer au serveur le résultat, en lui indiquant l’ID de la salle privée, l’ID de l’utilisateur concerné, le statut de la requête (acceptée/refusée), ainsi qu’un tableau d’octets (QByteArray) qui possèdera une valeur différente en fonction du statut :

* Requête acceptée : le tableau possèdera la clé privée de la salle, chiffrée avec la clé publique de l’utilisateur qui a fait la demande (reçue lorsque le serveur avait envoyé la requête au client), puis convertie en tableau d’octets. Cette clé permet au serveur de gérer plus facilement l’adhésion du client, en la lui renvoyant.
* Requête refusée : le tableau est vide, car l’utilisateur n’a pas besoin de connaître la clé de la salle, dans ce cas.

Lorsque la requête a été traitée, le contrôleur indique à la vue (**ViewMembershipRequests**) qu’elle peut enlever l’objet sélectionné de son arbre, puis la réactive.

Si la requête a été acceptée, le serveur notifie l’utilisateur concerné qui met à jour sa vue.

#### Je souhaite accéder aux détails de mon compte (anciennement édition du compte)

Lorsque l’utilisateur appuie sur « Edition > Compte », ou sur la combinaison de touches Ctrl+Shift+C, le module Chat ouvre la fenêtre détails du compte, et le module User reprend la main.

#### Je souhaite afficher la fenêtre d’à-propos

En cliquant sur « ? > A propos », la fenêtre d’à-propos s’affiche.

#### Je ferme la fenêtre

L’application se termine si la fenêtre est fermée (« Fichier > Quitter », Alt+F4, Ctrl+Q, croix rouge).