



Documentation technique

Projet de semestre

**Agenda étudiant**

**Étudiants :** Jérôme Moret, Paul Ntawuruhunga,

Thibaud Duchoud, David Kunzmann et Mario Ferreira

**Professeur :** Dr. René Rentsch

**Date :** 1er juin 2015 **Classe :** PRO-1-B

# Introduction

# Vue « Tâche »

## Structure

L’affichage dans le Framework Qt est basé sur des widgets. Ces widget sont simplement des objets affichables. Une fenêtre de notre application est une collection de widget. Ceci est le cas pour notre fenêtre tâche qui compte quatre widgets principaux encapsulant eux-mêmes d’autre widgets.

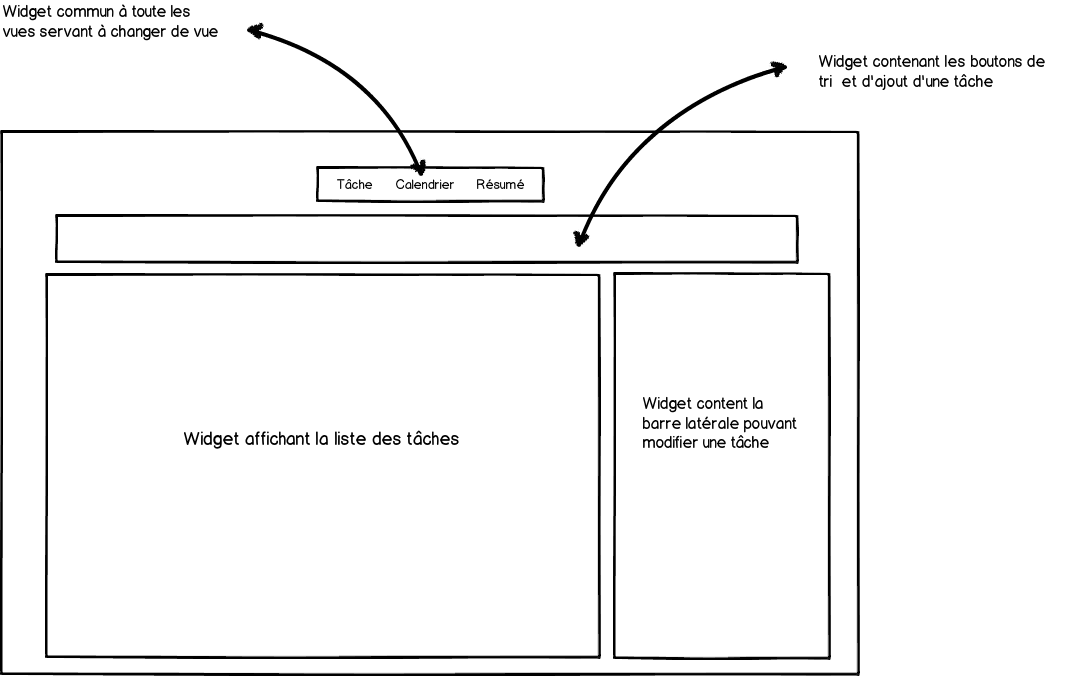


Figure 1 : Structure de la fenêtre pour la vue Tâche

Chacun de ces widgets contient plusieurs autre widgets tels que les boutons ou les champs de texte. Cette structure nous permet de modifier un widget sans devoir modifier la totalité de l’application. Par exemple en passant de la vue Tâche à la vue Résumé nous avons simplement à cacher le widget visible et à le remplacer par un autre widget. Cette opération est réalisée avec une pile de widget, structure propre au Framework Qt **QStackedWidget**.

## Récupération et affichage des données

La vue tâche consiste à afficher les tâches créer par l’utilisateur. Afin de se faire nous avions pour but d’aller récupérer les différentes tâche stockées dans notre base de données à l’aide de requêtes SQL. Pour cela nous avons créé une classe **sqlConnection** qui permet de se connecter à la base de données mais également à réaliser certaine action telles que récupérer toute les tâches. Cette façon de procéder est simple d’implémentation mais demande beaucoup de lignes de code surtout lorsqu’il faut ensuite afficher ces données. C’est pour cette raison que nous avons décidé, dans le mesure du possible, non pas de créer nous-même nos requêtes SQL mais d’utiliser les fonctionnalités offertes par le Framework Qt. En effet ce Framework nous met à disposition un outil très puissant qu’est les classes **QSqlTableModel** ainsi que **QSqlRelationTableModel**.

## QSqlTableModel

Cette classe nous permet de lier un modèle à une base de données. En d’autre mot nous lions chaque champ de notre modèle Task avec un champ dans notre base de données. Ceci a pour but que lorsque nous chargeons une tâche depuis la base de donnée, cette dernière est instanciée dans notre modèle. Il est ensuite extrêmement pratique de réaliser des actions sur notre modèle, actions qui seront répercutées directement sur notre base de données.

L’étape suivante est de pouvoir afficher ces données et une fois encore Qt nous met à disposition des méthodes permettant de lier une vue à un modèle (**mapper**). C’est de cette manière que nous pouvons réaliser des changements depuis l’interface de l’utilisateur (par exemple créer une nouvelle tâche) qui se répercuteront directement dans notre base de données. Cela sans avoir à créer chacune des commandes SQL requises pour ces opérations.

Sur le schéma ci-dessous nous voyons une représentation du lien entre la vue (view) et les données (data). Ces dernières étant stockées dans notre base de données. Le mapper cité plus haut est représenté par le cercle vert (delegate).

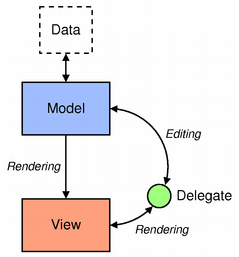


Figure 2 : Schéma du Model-View-Programming[[1]](#footnote-1)

## QSqlRelationTableModel

Ce deuxième outil nous a permis d’améliorer l’affichage de nos données en utilisant le moins possible de requête SQL et donc en gardant un code plus simple et plus lisible. **QSqlRelationTableModel** nous est utile lorsque nous avons une « Foreign Key » dans notre table SQL. En effet si nous utilisons seulement **QSqlTableModel** nous stockerions dans le modèle les informations qui sont dans une table de notre base de données et donc pour le modèle de tâche au lieu de nous d’obtenir le nom du cours auquel notre tâche est lié nous obtiendrons son identifiant. Il faudrait pouvoir ensuite rechercher cette « id » et récupérer le nom auquel elle correspond. C’est exactement à cela que sert **QSqlRelationTableModel**. Nous lions une colonne dans une table SQL à une autre colonne dans une autre table. Ceci a pour effet d’afficher le nom du cours plutôt que son identifiant.

## Les slots et les signaux

Un point important dans le Framework Qt est les slots ainsi que les signaux. Nous pouvons grâce aux signaux passer un message à un slot, ceci permet à différentes classes et/ou objets de communiquer. Nous avons donc utilisé ces signaux lors de modification des mappers et/ou des modèles afin de pouvoir appeler des fonctions pour modifier l’affichage par exemple. Dans la vue tâche, les boutons de tris sont par exemple tous connectés à un slot. Ces boutons émettent un signal lorsque l’on clique dessus, le slot récupère ce signal et le tri sélectionné est appliqué. Nous procédons ainsi car l’affichage des tâches triées et les boutons de tri se situent dans deux widgets différents.

## Tri des tâches

Dans le but de pouvoir trier nos tâches par ordre d’échéance ou par ordre de priorité, nous avons utilisé des fonctionnalités qu’offrent Qt, la classe **QSqlTableModel** et la méthode **sortBy** qui trie notre modèle, lié à une table SQL, en fonction de la colonne dans cette table (par exemple la priorité).

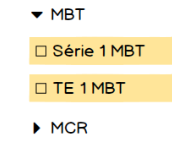
# Problèmes rencontrés

## Tri et sélection

Lors de notre implémentation de la vue tâche nous avons remarqué, que lors d’un tri des tâches, la tâche sélectionnée était différente avant et après le tri. Ceci est dû au fait que la sélection concerne la ligne sur laquelle se trouve la tâche. En raison d’un retard sur le développement de notre application nous n’avons pas eu le temps de régler ce problème de la meilleure manière qui soit. Cependant nous avons cherché un moyen de résoudre cela et avons trouvé que Qt offrait la possibilité de trier nos modèles en gardant la sélection. Ceci se ferai à l’aide de la classe **QSortFilterProxyModel** cette couche se situe entre le modèle et la vue et nous permettrai de garder la sélection entre les différents tris et filtres. Nous n’avons pu implémenter cette classe dans notre application mais nous avons implémenté une solution non définitive mais entièrement fonctionnelle.

## Affichage

Nous n’avons également pas pu obtenir la structure que nous avions prévue dans le mockup présenté dans le cahier des charges. Nous avions pour but d’afficher la liste des tâches triées par cours de la manière suivante :



La représentation sous forme d’arbre n’a pas été implémentée car afin d’obtenir une bonne lisibilité avec la grande quantité d’informations que représente une tâche, il a été plus pratique d’orienté notre design vers une simple liste triable par cours.

# Conclusion

# Annexes

1. http://doc.qt.io/qt-4.8/model-view-programming.html [↑](#footnote-ref-1)