Projet Developpement Logiciel

> Membres : François BONNIN et Rémi FALCATI

Sommaire

Présentation du projet

Dans le cadre de notre deuxième année à Ynov, nous avons découvert plusieurs technologies orienté sur la programmation logiciel (Architecture Logiciel, les diagrammes UML, Java, Arduino pour faire de l'iOT et C++).

Il nous est donc demandé, en cette fin d'année de présenter un projet fonctionnel développer à partir de certaines de ces technologies par groupe de 2.

Nous étions libres sur le choix du projet (à condition d'être validé par le responsable de notre suivi de projet).
Nous avons donc choisi de répondre à une problématique que beaucoup de personnes se pose à présent que cette crise sanitaire s'atténue :

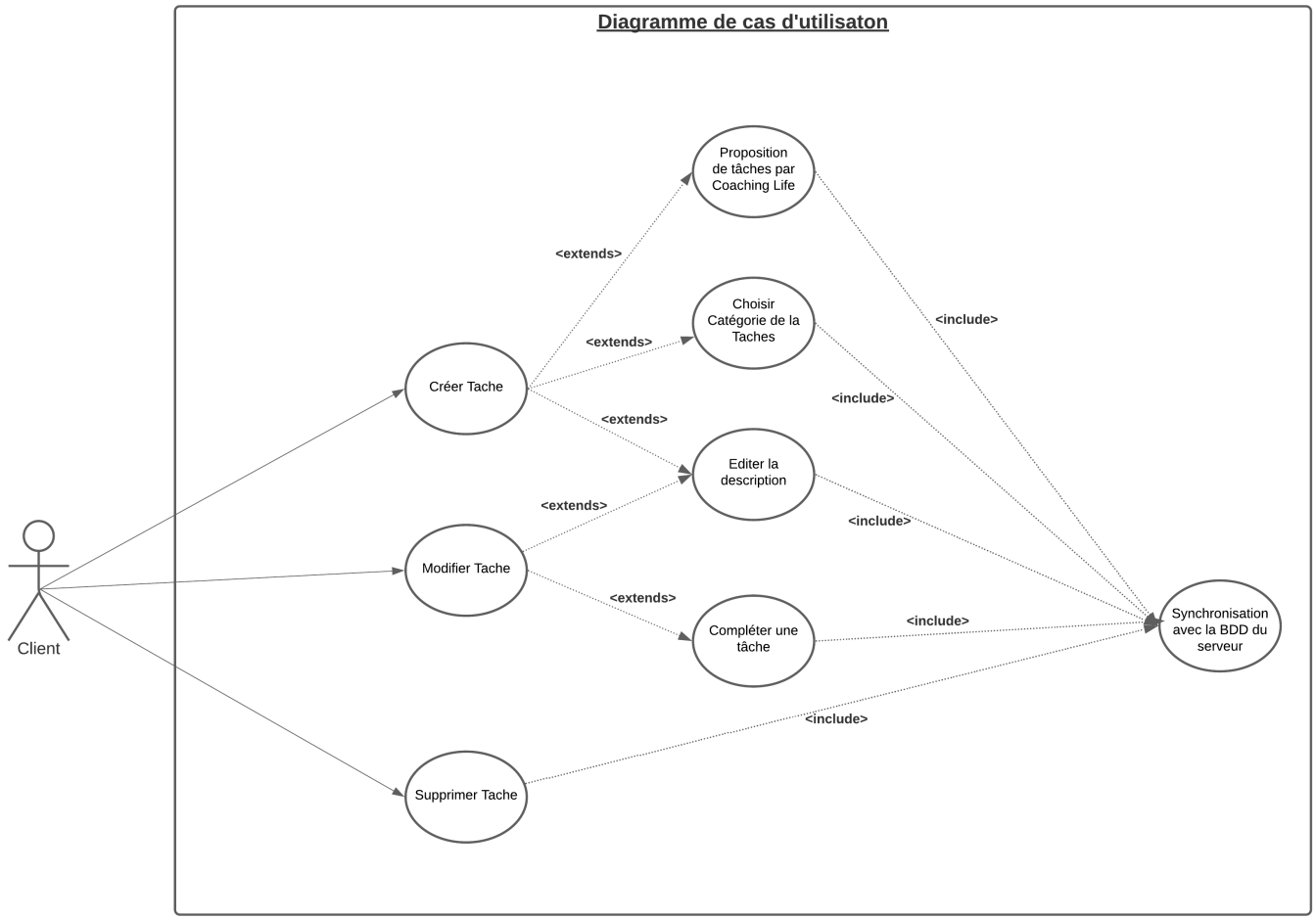
> **Problématique :** Comment accompagner les personnes, dans l'organisation de ses tâches journalière et dans l'amélioration de son bien-être personnel ?

Cette problématique est principallement destinés à des personnes qui veulent mieux organiser leurs journées, et se reprendre en main après cette période de crises sanitaires où ils se sont un peu laissés aller.

Nous avions un temps limité non négligeable pour mettre en place un projet pouvant répondre à cette problématique.

Ainsi nous mettons en oeuvre une Todo-List permettant également de nous proposer des tâches quotidienne, hebdomadaire, mensuel ou annuel. Pour que l'application permette à l'utilisateur d'accéder à ses tâches depuis différents postes, nous connecterons une API reliés à une base de données SQLite.

Voici un diagramme de cas d'utilisation dans lequel nous avons définis les principales fonctionnalités du projet :



L'utilisateur pourra créer une tâche en lui donnant un nom et si besoin, une description. Il pourra affecter à cette tâche une catégorie.
Il pourra également générer une tâches Coaching Life, qui s'ajoutera automatiquement dans sa liste de tâche à faire.

Si des tâches sont déjà présentes l'utilisateur peut modifier son nom ou sa description. Il peut aussi modifier l'état de la tâche en `completed` si celle-ci à été réalisé, ou est à refaire.

Afin de faire le ménage dans ces tâches, il pourra les supprimer.

Après chaque opérations, les tâches sont automatiquement synchronisés à la base de données, pour en être sur, un bouton synchroniser à été mis en place, pour "forcer" cette synchronisation.

Choix technologique :

Une fois avoir fais notre diagramme de cas d'utilisation il faut choisir une technologie. Une technologie de développement logiciel uniquement.

Nous avions retenue 2 langages :

- Soit Python pour sa simplicité syntaxique et notamment pour une futur utilisation personnel avec Django, un framework utilisé dans le développement web (spécialité choisi l'année prochaine).
- Soit le C++ découvert cette année, plus particulièrement Qt Creator qui possède une interface graphique plus avancés. Le problème issue de cette technologie c'est que ce n'est pas réellement du C++. Qt possède presque son propre langage basé sur le C++.

Nous nous sommes donc dirigé dans un premier temps vers Python, mais après réflexion et au vu du temps qu'il restait, nous pouvons pas nous permettre de recommencer l'apprentissage d'un nouveau langage en si peu de temps.

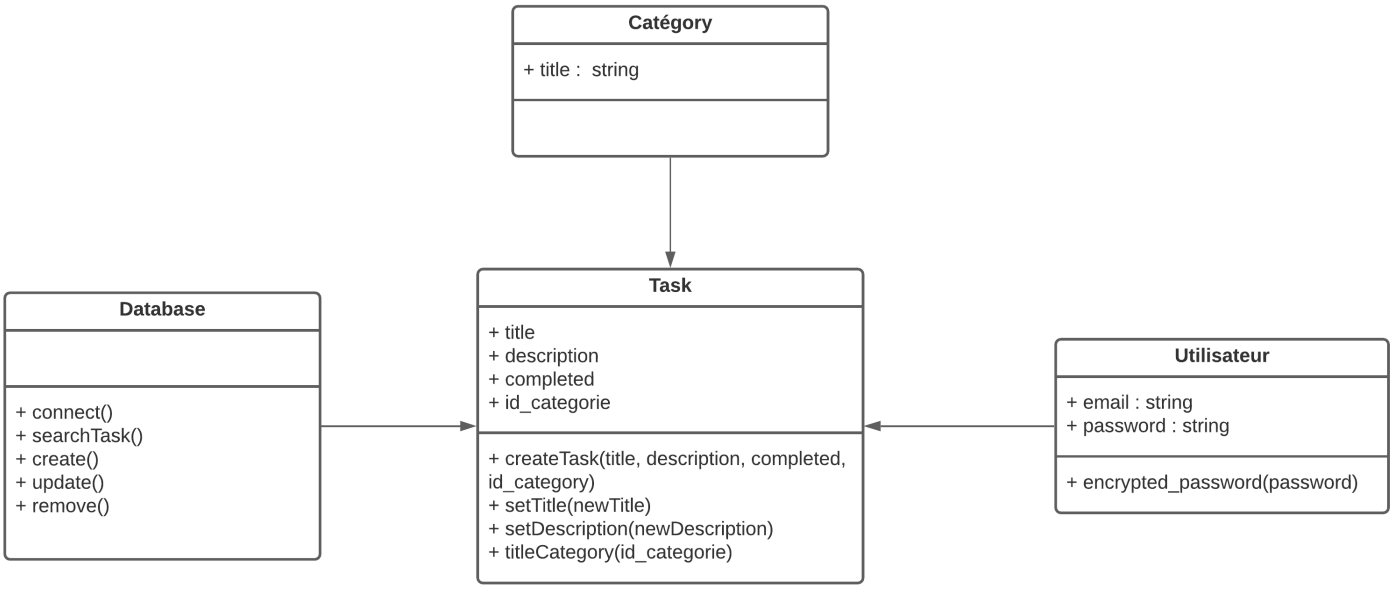
Ainsi nous avons décidé de nous rediriger vers la technologie vu en cours : **C++ / Qt Creator**.

Cette année, deux technologies nous on permit de mettre en place une **API** : Java Sping ou Laravel. En tant que futur développeur web, nous avons sélectionné **Laravel**, plus précisément un micro-framework de Laravel : **Lumen**.

Programmation Orienté Objet :

Juste après avoir mis en place notre diagramme de cas d'utilisation nous nous sommes penchés sur le diagramme de classe. Un diagramme de classe qui nous permet de mieux comprendre le schéma de notre application :

Diagramme de classe



Afin d'avoir un code productif et plus modulable notre projet contient plusieurs objets (`class`) qui intéragissent entre eux :

- **Tâches** (alias `Task`) : Elle permettra d'instancier un objet "task" qui sera ajouter à `QList <Task> listTask` et nous permettra d'afficher chacune des "task" dans notre interface.

```
class Task
{
    private:
        QString m_titre;
        QString m_description;
        int m_id_category;
        int m_completed;

    public:
        Task(QString titre, QString description, int completed, int id_category);

        QString titre() const;
        void setTitre(const QString &titre);

        QString description() const;
        void setDescription(const QString &description);

        int id_categorie() const;
        void setId_categorie(const int id_cat);
        QString titleCat(int id_cat) const;
```

- **Catégories** (alias `Categories`) : De la même façon que la classe Task, on instancie un objet "category" qui sera ajouter à `QList <Category> listCat` et nous permettra d'afficher chacune des "category" dans notre interface.

```
class Category
{


public:
    int m_id;
    QString m_title;
    void fillListCat();

    Category(int id, QString title);
    static QString idToTitle(int id);

    QString title();
};
```

- **Base de données** (alias `Database`) : Cette classe à été créer afin d'effectuer les requêtes sql sur la base de donnée.

```
class Database
{
public:
    Database();
    bool connect(QString bdd_name);

    static Task search(QString search);
    static bool findCoach();
    static bool insertTask(QString title, QString description, int completed, int id_ca
    static bool update(QString title, QString description, int completed, int id_catego
    static bool updateTodo(QString title);
    static bool updateFinish(QString title);
    void show(QString title);
    static bool remove(QString title);
    static int countToDo();
    static int countFinish();

};
```

> Remarque : Cette classe se nommait API, et nous permettait de la même façon, d'intéragir avec notre API.

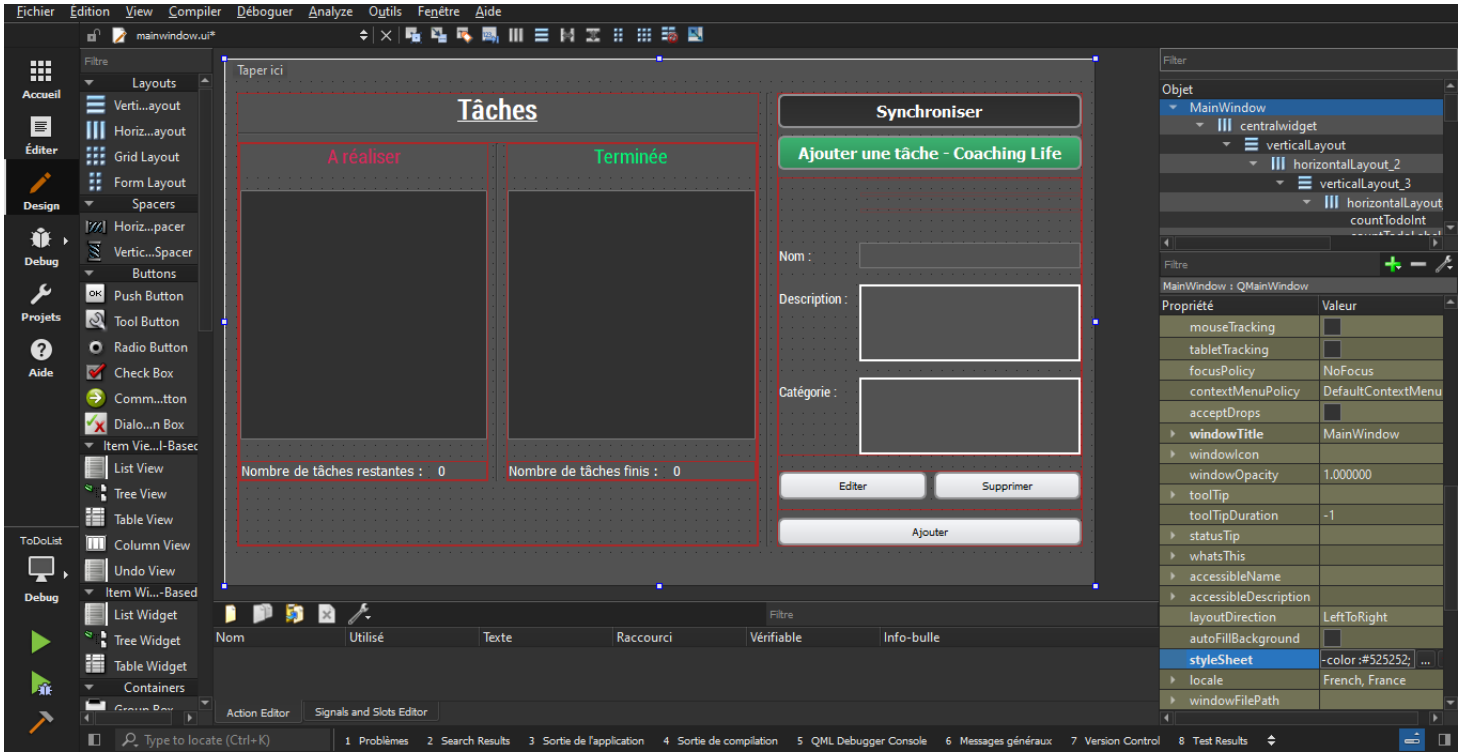
Par exemple la fonction `connect()` nous permet d'établir une connection avec la base de données SQLite.

```
bool Database::connect(QString bdd_name)
{
    const QString DRIVER("QSQLITE");

    if (QSqlDatabase::isDriverAvailable(DRIVER)) {
        QSqlDatabase db = QSqlDatabase::addDatabase(DRIVER);

        db.setDatabaseName(bdd_name);

        if (!db.open()) {
            qWarning() << __FUNCTION__ << "  ERROR: " << db.lastError().text();
            return false;
        }
    }
    else {
        qWarning() << __FUNCTION__ << " ERROR: no driver " << DRIVER << " available";
        return false;
    }
    return true;
}
```

Interaction avec utilisateur

L'avantage de **Qt Creator** et de posséder un outil complet d'interface. Une fois les layouts (horizontal et vertical) ajouté, notre fenêtre est entièrement redimensionnable et responsive.

Le principe d'interface dans Qt nécessite cependant une classe (créer par défaut) qui se nomme `MainWindow` . Notre interface est stocké dans un fichier mainwindow.ui au format XML au fur et à mesure qu'on l'a créer avec le mode *Design*.

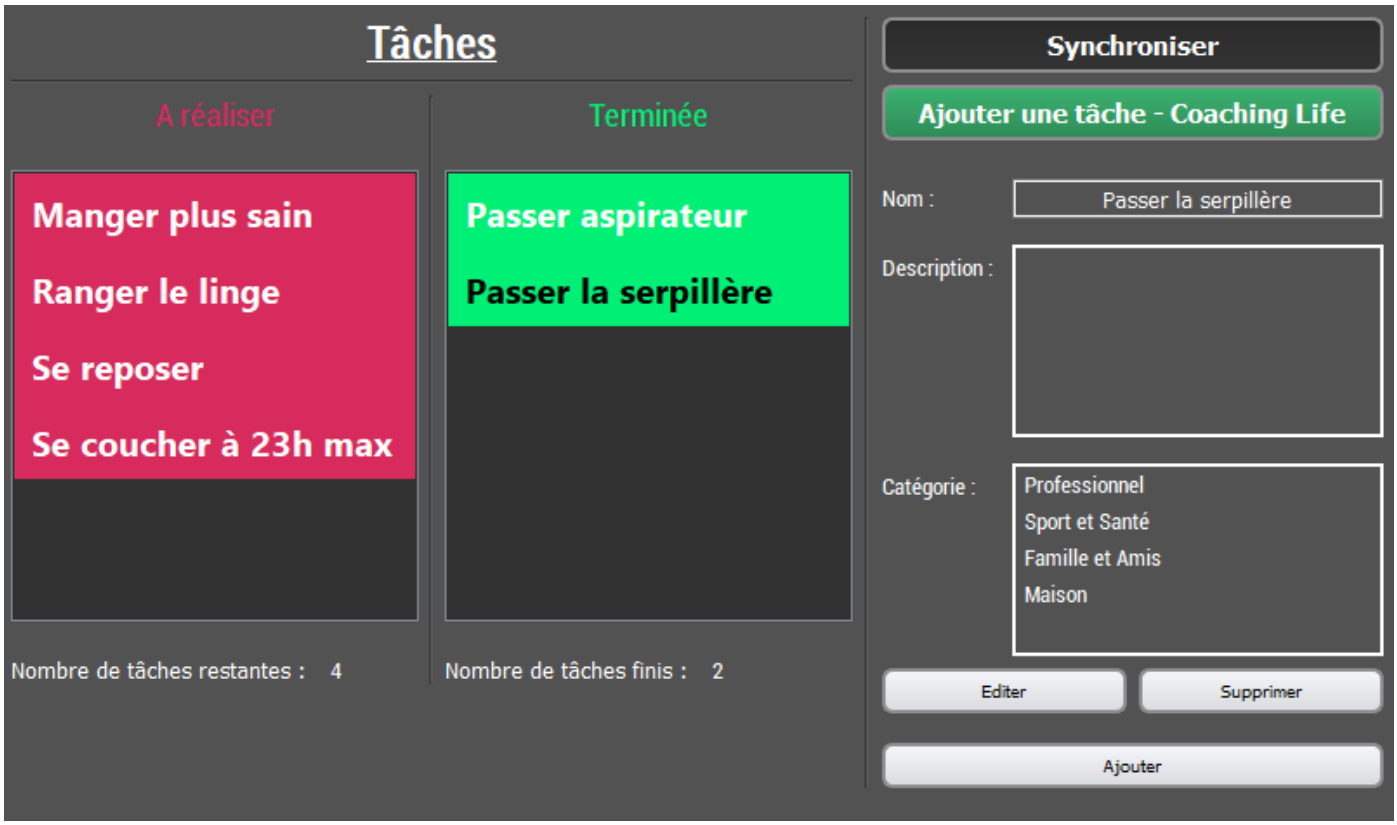


Fonctionnalités du projet :

CRUD :

Le CRUD (Create Read Update Delete) est un éléments majeur d'une Todo-list.

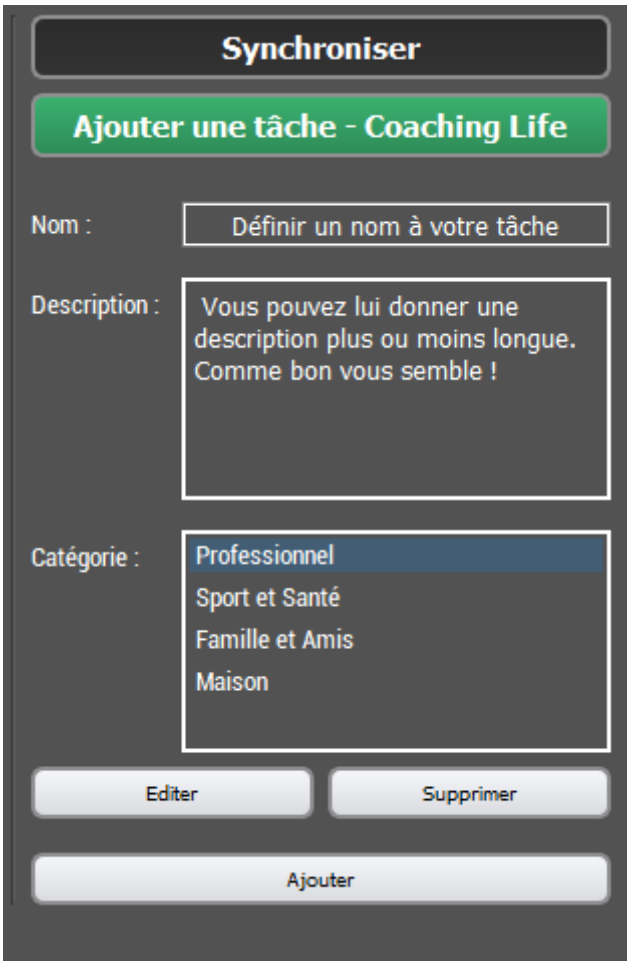
- **Voir** l'ensemble de ses tâches :
  Interface :



Voici la procédure nous permettant d'afficher les données dans la colonne **"A réaliser"** :

```
// On récupère toutes les tasks qui sont complétés
QSqlQuery query ("SELECT * FROM tasks WHERE completed=0");
    //On récupère le numéro de la colonne de chaque attributs
    int idtitle = query.record().indexOf("title");
    int iddescription = query.record().indexOf("description");
    int idcompleted = query.record().indexOf("completed");
    int idcategory = query.record().indexOf("id_category");

    // On vide la liste d'objet Task à faire
    listTasksTodo.clear();

    // On parcourt les résultats de la requête
    while (query.next()) {
        QString title = query.value(idtitle).toString();
        QString description = query.value(iddescription).toString();
        int completed = query.value(idcompleted).toInt();
        int id_category = query.value(idcategory).toInt();
        /* Une fois qu'on a récupéré les valeurs dans les variables
        correspondantes on créé un nouvel objet Task */
        Task* newTask = new Task(
            title,
            description,
            completed,
            id_category
        );

        if(newTask != nullptr)
        {
            /* Si cet objet n'est pas null alors on l'ajoute à la
            liste d'objet Task à faire*/
            listTasksTodo.append(newTask);
        }
    }
    /* A la fin, on parcourt la liste et on remplit la colonne
    "A réaliser" de l'interface avec chacun des éléments*/
    for(const auto task : listTasksTodo){
        ui->listTodo->addItem(task->titre());
    }
```

- **Créer** (bouton *Ajouter*):
  - Définir un nom
  - Ajouter une description détaillée pour ses tâches.
  - Choisir une catégorie par tâches (*cette fonctionnalité est en cours de maintenance*)
    Interface :



Code de la procédure :

```
//On récupère la valeur de chaque champ du formulaire
QString title = ui->nomEdit->text();
QString description = ui->descriptionEdit->toPlainText();
int completed = 0;
int id = 1;
//catégorie sélectionné = ui->catListWidget->currentItem()->text()

/*On créé la tache en utilisant la méthode Database::insertTask()
INSERT INTO tasks...*/
if(Database::insertTask(title, description, completed, id)){
    // On affiche un message de succès
    statusBar()->showMessage("Tâche créée", 4000);

    // On vide les champs du formulaire
    ui->nomEdit->clear();
    ui->descriptionEdit->clear();
}
// On synchronise avec la Base de donnée
populateList();
```

- **Modifier** (bouton *Editer*):
  Code de la fonction Database::update() :

```
bool Database::update(QString title, QString description, int completed, int id_category)
{
    QSqlQuery query;

    QString sql_query = QString("UPDATE tasks SET title = '%1', description = '%2', completed =
                        .arg(title).arg(description).arg(completed).arg(id_category);

    if (!query.exec(sql_query)){
        qWarning() <<  __FUNCTION__ << "   ERROR: " << query.lastError().text();
        return false;
    }

    return true;
}
```

- **Supprimer** (bouton *Supprimer*)
  Code de la fonction `Database::remove()` :

```
bool Database::remove(QString title)
{
    QSqlQuery query;

    QString sql_query = QString("DELETE FROM tasks WHERE title = '%1' ").arg(title);

    if (!query.exec(sql_query)){
        qWarning() <<  __FUNCTION__ << "   ERROR: " << query.lastError().text();
        return false;
    }

    return true;
}
```

Coaching Life:

Grâce à ce button, l'utilisateur se voit attribué une tâche déjà créer dans la base de données. Une tâche qui l'aidera à se sentir mieux une fois qu'il l'aura accomplie.
Elle peut être de différentes catégorie : Professionel, Santé & Sport, Famille et Amis, ou Maison (tâches ménagère...)

Drag and Drop

Grâce à Qt et à ses fonctionnalités, on a pu mettre en place le système de drag & drop (glisser - déposer) pour permettre à l'utilisateur de changer de liste sa tâche une fois qu'elle est terminée.
Une fois le drag & drop effectué, le changement d'état de la tâche se fera automatiquement dans la Base de donnée.

API

Afin de pouvoir avoir accès à ces tâches à partir de différents postes (à condition d'avoir accès à internet), nous avons créé une API avec Laravel (plus précisément Lumen).
Cette API (REST) à pour but de connecter l'application à une base de donnée distante.
Pour cela nous avons utiliser Laragon et une base de donnée Sqlite.

Afin d'activer l'API, il faut taper la commande (dans le terminal Laragon par exemple) :

```
php -S localhost:8000 -t ./public
```

Liste des différentes requête (précédé de `localhost:8000`):

- GET : `'/tasks'` = affiche la liste des tâches
- GET : `'/category'` = affiche la liste des catégories
- GET : `'/coach'` = affiche la liste des tâches Coaching Life
- GET : `'/search'` = rechercher une tâche via son titre, sa description si elle est réalisé ou non
- POST : `'/task/create'` = créer une tâche
- POST : `'/task/edit/{id}'` = modifier une tâche
- POST : `'/task/completed/{id}'` = passe la tâche en "terminée"
- POST : `'/task/discompleted/{id}'` = passe la tâche en "à faire"
- POST : `'/coach/completed/{id}'` = passe la tâche coaching-life en "terminée"
- POST : `'/coach/discompleted/{id}'` = passe la tâche coaching-life en "à faire"
- DELETE : `'/task/delete/{id}'` = supprime la tâche

Difficultés rencontrées :

Lors du développement d'un projet, nous rencontrons pratiquement toujours certaines difficultés face à des choses nouvelles.

- Choix du langage (Python vs C++) : Notre première difficulté fut Python, évaluer la durée restante du projet et l'utilisation d'un nouveau langage, d'un nouvel IDE et d'une nouvelle bibliothèque graphique Tkinter. Après 10 jours d'évaluation, il a été plus judicieux de se tourner vers Qt Creator.
- Connexion à l'API : Du côté de Qt Creator, nous n'avons pas réussi à faire fonctionner *QNetworkAccessManager* et *QJsonDocument* afin d'envoyer les requêtes et de lire les réponses

au format JSON. Voyant la date de rendu approcher à grands pas, nous avons décider de basculer avec une base de donnée en local.
- Driver MySQL : Après avoir échoué avec le fonctionnement de l'API avec Qt, nous avons donc décidé de basculer sur une base de donnée en local et nous nous sommes diriger vers MySQL. Il nous auras fallu presque jours et de multiples tutoriel pour essayer de faire fonctionner Qt Creator avec MySQL. En effet il n'y a pas de partenariat établie entre les deux, il faut donc créer le driver soit-même. Nous sommes donc passé sur une base de donnée SQLite...
- Apostrophe dans les `title` ou `description` pour les requêtes SQL. Une des difficultés que j'ai rencontrés est l'intégration des apostrophes dans les champs, pour les requêtes SQL. En effet, cela fausse toute la requete, même en passant par une variable.

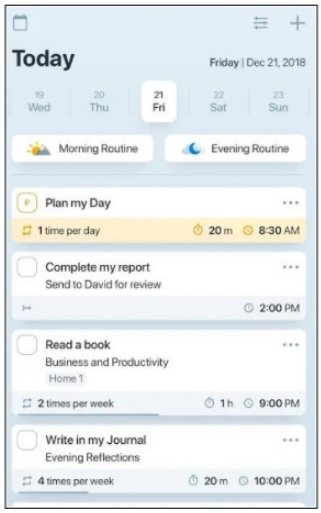
Fonctionnalités supplémentaire :

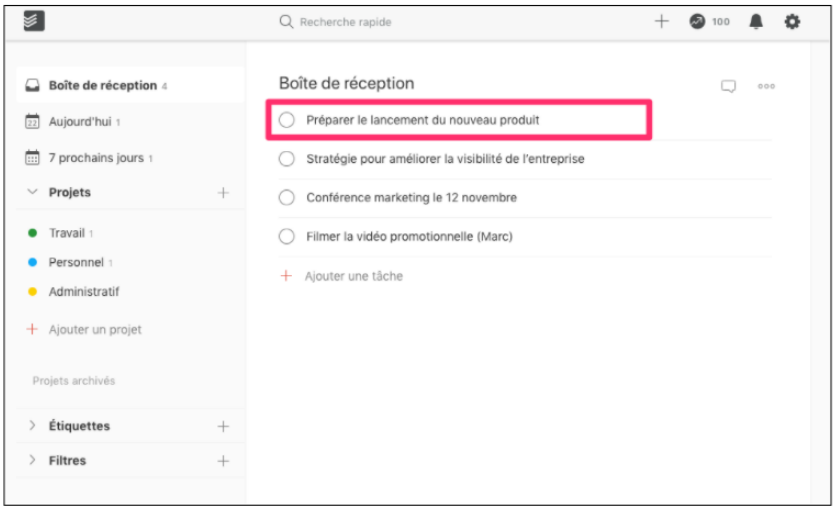
Nous pouvons envisager des fonctionnalités supplémentaires pour la suite du développement de l'application comme par exemple :

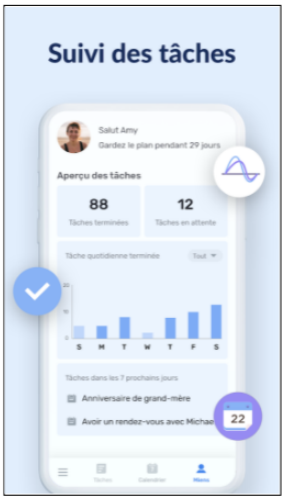
- Ajouter un système d'authentification avec un compte et un mot de passe par personne.
- Implémenter des statistiques sur la réalisations des tâches par catégories afin de proposer des tâches plus ciblés.
- A cause d'un léger problème technique, la fonctionnalité "associer une catégorie à une tâche est en maintenance, elle sera disponible dès la prochaine mise à jour"

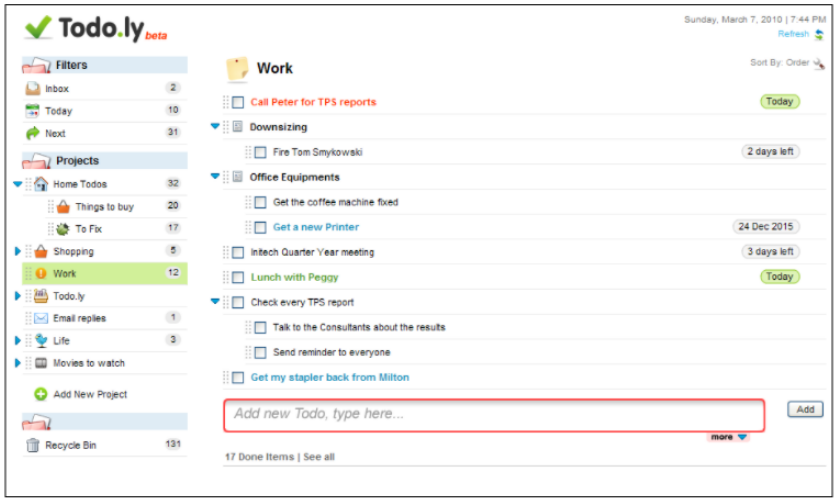
Moodboard :

Au début de notre projet, afin de mieux visualiser ce dont on vouler développer. Nous avons mis en place un moodboard avec divers applications déjà sur le marché.









Contact

François BONNIN - francois.bonnin@ynov.com
Rémi FALCATI - remi.falcati@ynov.com

Lien du repository Github - https://github.com/Francois-BONNIN/Coaching-Life-CPP