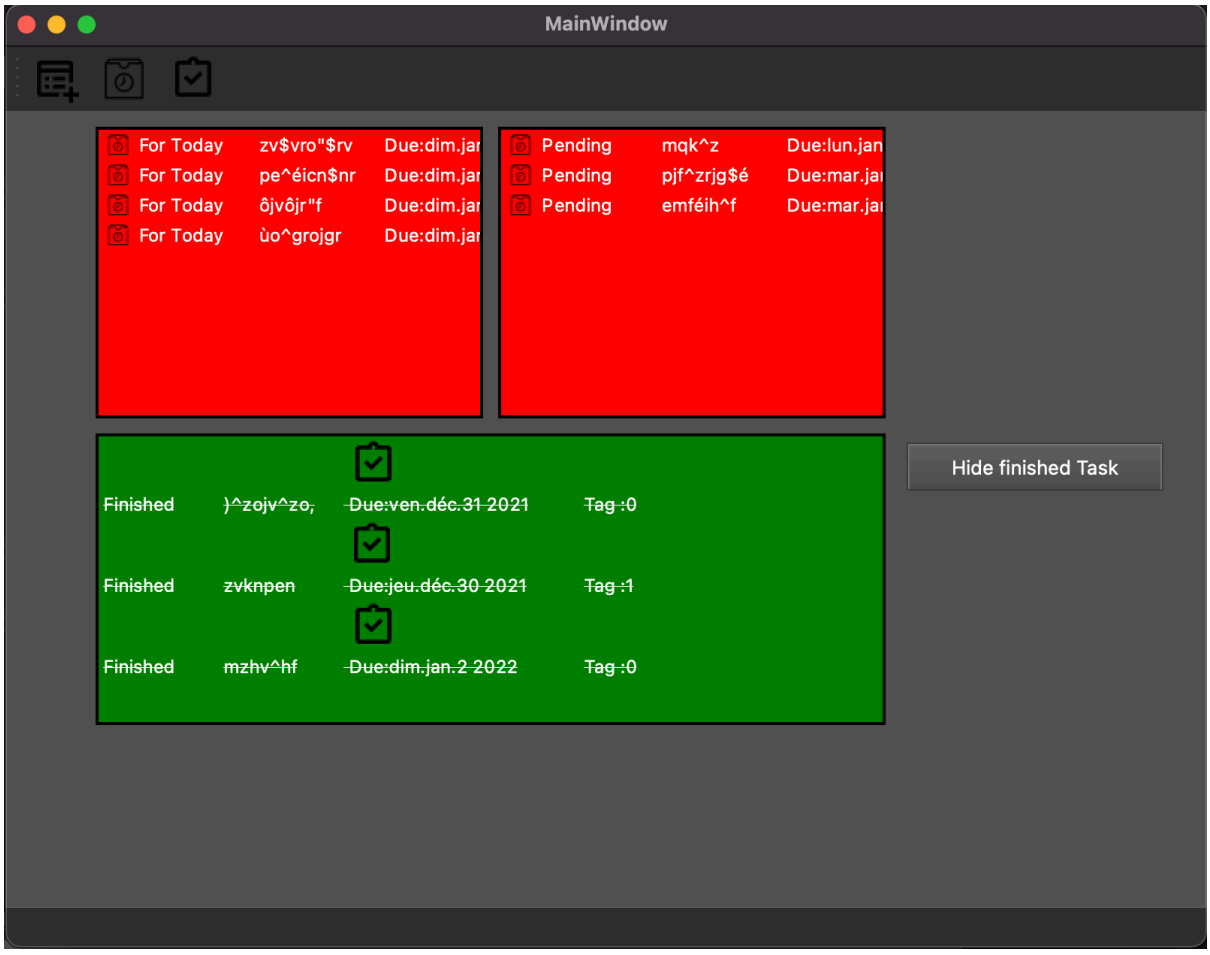
Tarek Kamel
Mohamed Abdallaoui
Encadré par :
Pr Anass Belcaid

# To Do Application with Qt designer




The present report maps the entire process our team have been through to implement a QMainWindow performing a basic To Do Application to manage tasks. It should have all the features of **main application** such as menues, actions and toolbar. The application must store an **archive** of all the **pending** and **finished** tasks.

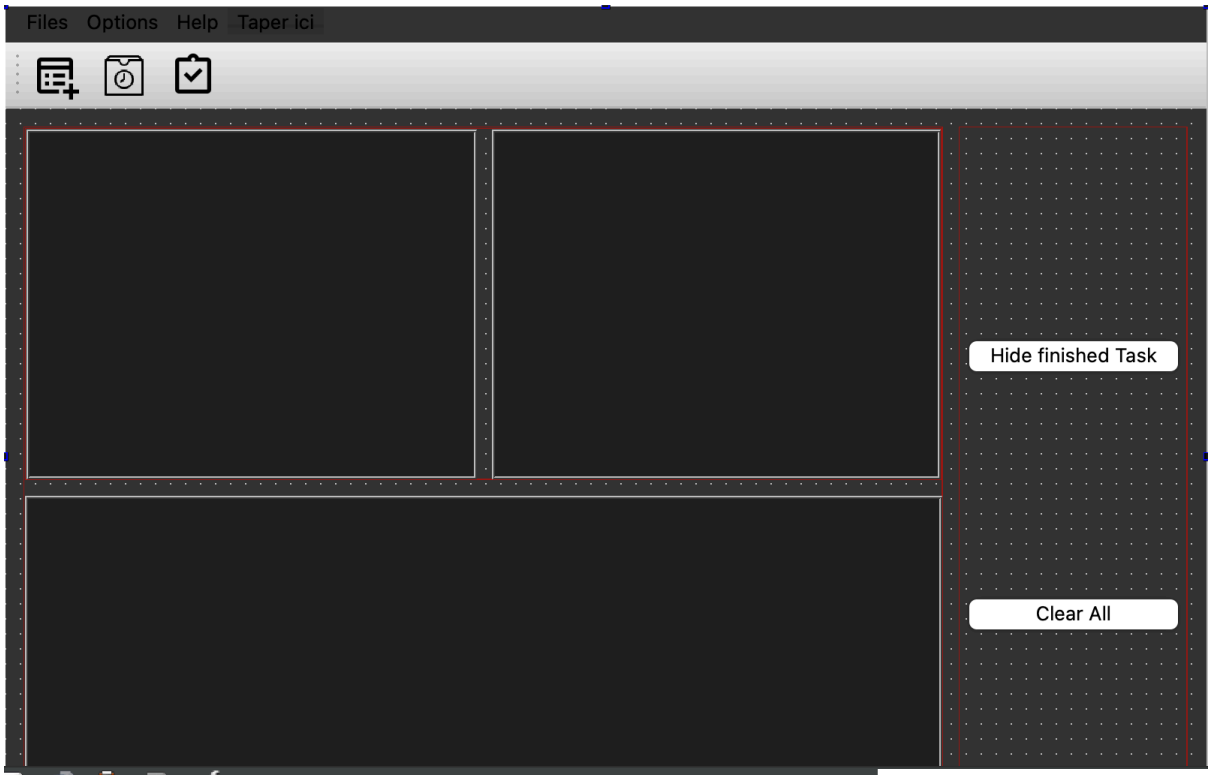
# TABLE OF CONTENT

# I - ToDo with QWidget

## 1 - Introduction :

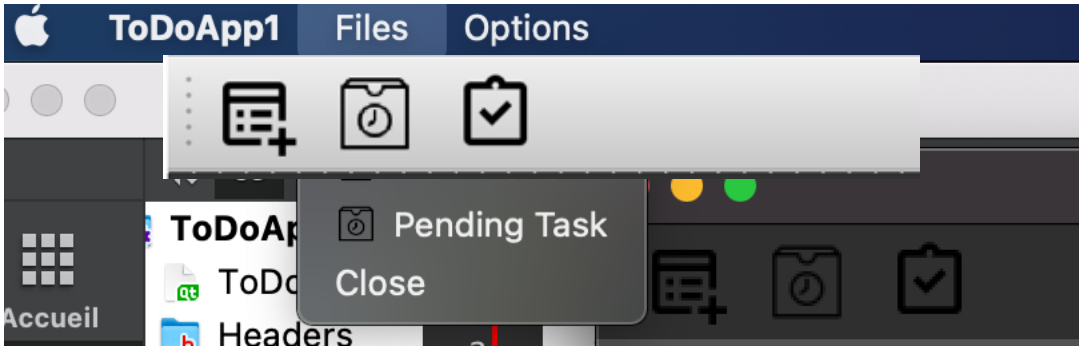
In order to implement this application, we have first utilized qt Ui to create the main window that displays the tasks and performs all the methods, as well as the Dialog used for prompting the user for the new task details (description , tag …) ,
In the second part of the process, we implemented the slots to each QAction of the window, and connected them with the corresponding slots; most of the those slots weren't predefined so we had to implement them,
In addition to that, and in order to keep track on the previous tasks the user entered, we created a Qfile to store and read all the data.

## 2 - Window creation using UI:




Our Main window is split into 3 QLIstWidgets (we could have used QTableWiget as well) the top right one for todays Tasks, top left one for the pending tasks and the last for the finished tasks.
The window also has a menu bar that allows the user to add tasks, to make a pending task done,
Or the opposite.

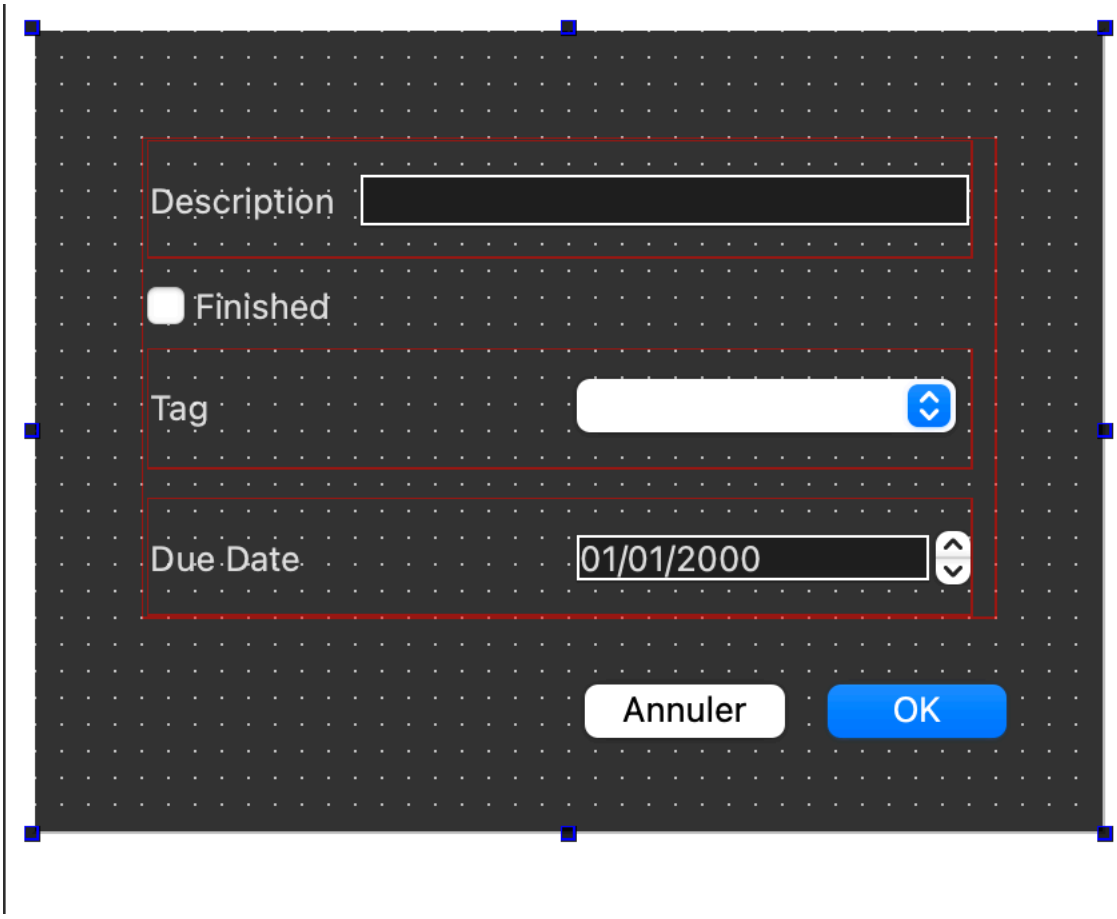
And to make this functions more accessible to the user, we added them to a tool bar that displays them with their respective icons stored in the ressource file we added to the project.




Finally we set up the layouts of the window.

| Objet | Classe |
|---|---|
| MainWindow | QMainWindow |
| centralwidget | QWidget |
| verticalLayout | QVBoxLayout |
| horizontalLayout | QHBoxLayout |
| pendingTask | QListWidget |
| taskForToday | QListWidget |
| Finished | QListWidget |

Once the main window set up, we have to create a dialog that appears every time the user wants to enter a task




containing an edit line for task description, a checkbox to notify if the task if finished and returns a boolean, a comboBox comprising the 3 tags « work », « life » , « other » and finally a DateEdit to enter the task due date.

Here's a how layouts are setUp

| Objet | Classe |
|---|---|
| addDialog | QDialog |
| verticalLayout | QVBoxLayout |
| horizontalLayout | QHBoxLayout |
| description | QLineEdit |
| label | QLabel |
| horizontalLayout_2 | QHBoxLayout |
| comboBox | QComboBox |
| label_2 | QLabel |
| horizontalLayout_3 | QHBoxLayout |
| dateEdit | QDateEdit |
| label_3 | QLabel |
| checkBox | QCheckBox |
| okButton | QDialogButtonBox |

## 3 - Main application features in detail

Here is a list of **cases** that the user could **perform** with our app:

1. A user should be able to **close** the application of course.

```
connect(ui->actionClose,&QAction::triggered,this,&MainWindow::close);
```

2. A Todo application cannot be useful, unless it offers the possibility of **creating new tasks**.

This slot is triggered by clicking the Add Task QAction, it shows a add Dialog that we created previously,

```
void MainWindow::on_actionAdd_Task_triggered()
{

    addDialog D ;
    D.setModal(false);
    D.exec();
```

« addDialog.h »

```
class addDialog : public QDialog
{

    Q_OBJECT

public:
    explicit addDialog(QWidget *parent = nullptr);
    ~addDialog();

    QString getDescription();
    QString getTag();
    QString getFinished();
    QDate getDueDate();
    void showEvent(QShowEvent * event);
private slots:
```

```
    QString newTask ;
    // Get the line edit text
    QString description = D.getDescription();
    if (description!=NULL){
        //Get Finished bool
        QString finished = D.getFinished();
    // Get current date
         QDate curDate = D.getDueDate();
        // QString date = D.getDueDate().toString();
        newTask = description +"\t Due:"+ curDate.toString() ;
        QString Tag= D.getTag();

         if (finished=="finished" || curDate < QDate::currentDate())
         {
             newTask = "Finished\t"+newTask+"\tTag :"+Tag+"\n";
              ui->Finished->addItem(new QListWidgetItem(QIcon(":/new/prefix1/done.png"),newTask));
}
         else if (curDate == QDate::currentDate()){
              newTask = "For Today\t"+newTask+"\tTag :"+Tag+"\n";
                ui->taskForToday->addItem(new QListWidgetItem(QIcon(":/new/prefix1/pending.png"),newTask));
         }
         else{
                 newTask = "Pending\t"+newTask+"\tTag :"+Tag+"\n";

                ui->pendingTask->addItem(new QListWidgetItem(QIcon(":/new/prefix1/pending.png"),newTask));

         }
    }
```

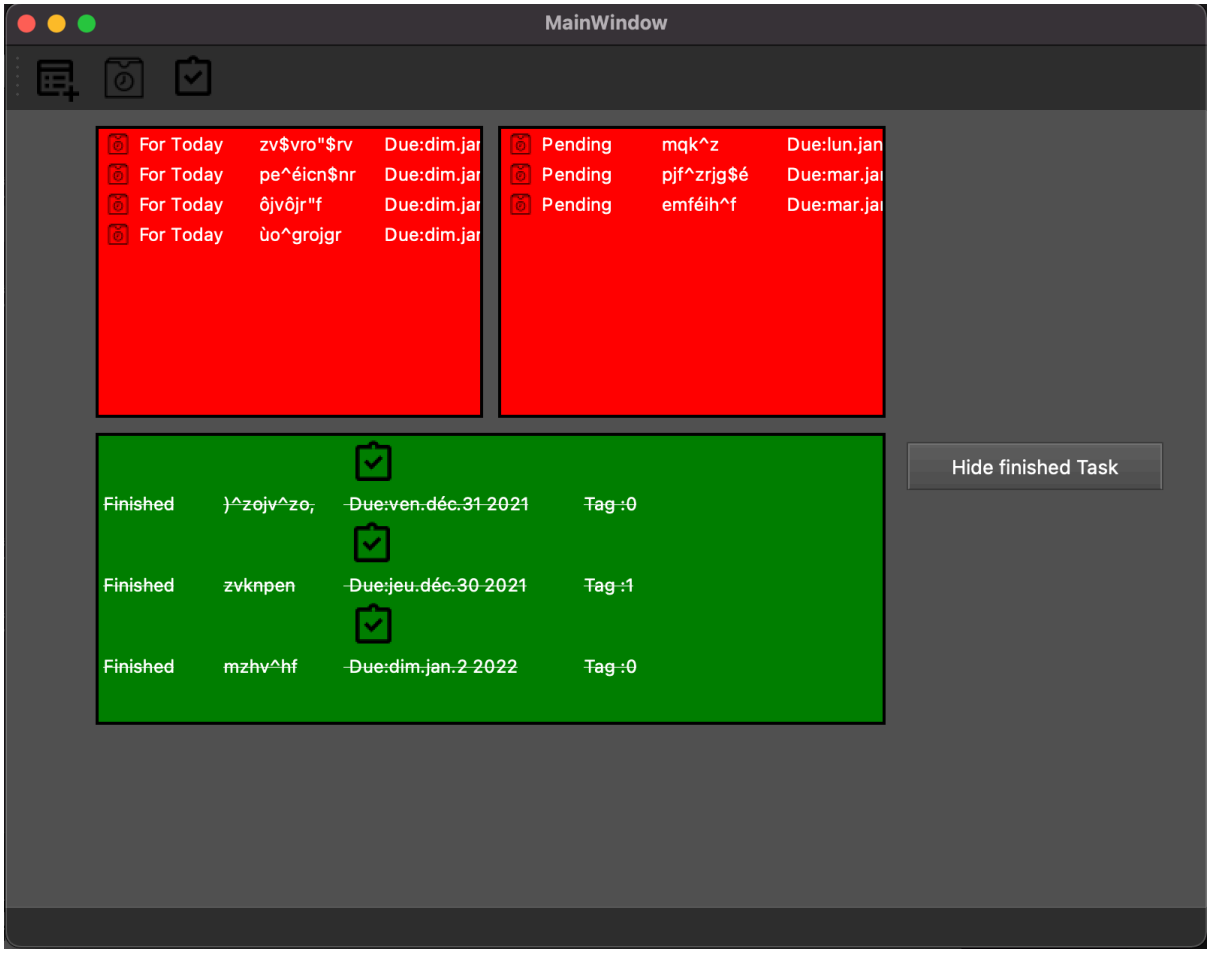
Then we created a new String that stores the values of the addDialog getters values,

Then some conditions to segment the tasks depending on the due date, if it is identical to the users operating system date the tasks moves to Today's tasks, if it is trespassed the task moves to finished tasks…

And in order to keep a record on the user's tasks, the addAction creates a file to store the data, this topics will be explained in-depth further in this chapter.

3. The View of the main widget is **split** into **three** areas as shown in the screenshot:
    - The first (en persistent) area shows the list of **today** tasks.
    - The second one is reserved for **pending** task (tasks for the future).
    - Finally, the third one shows the set of **finished** tasks.

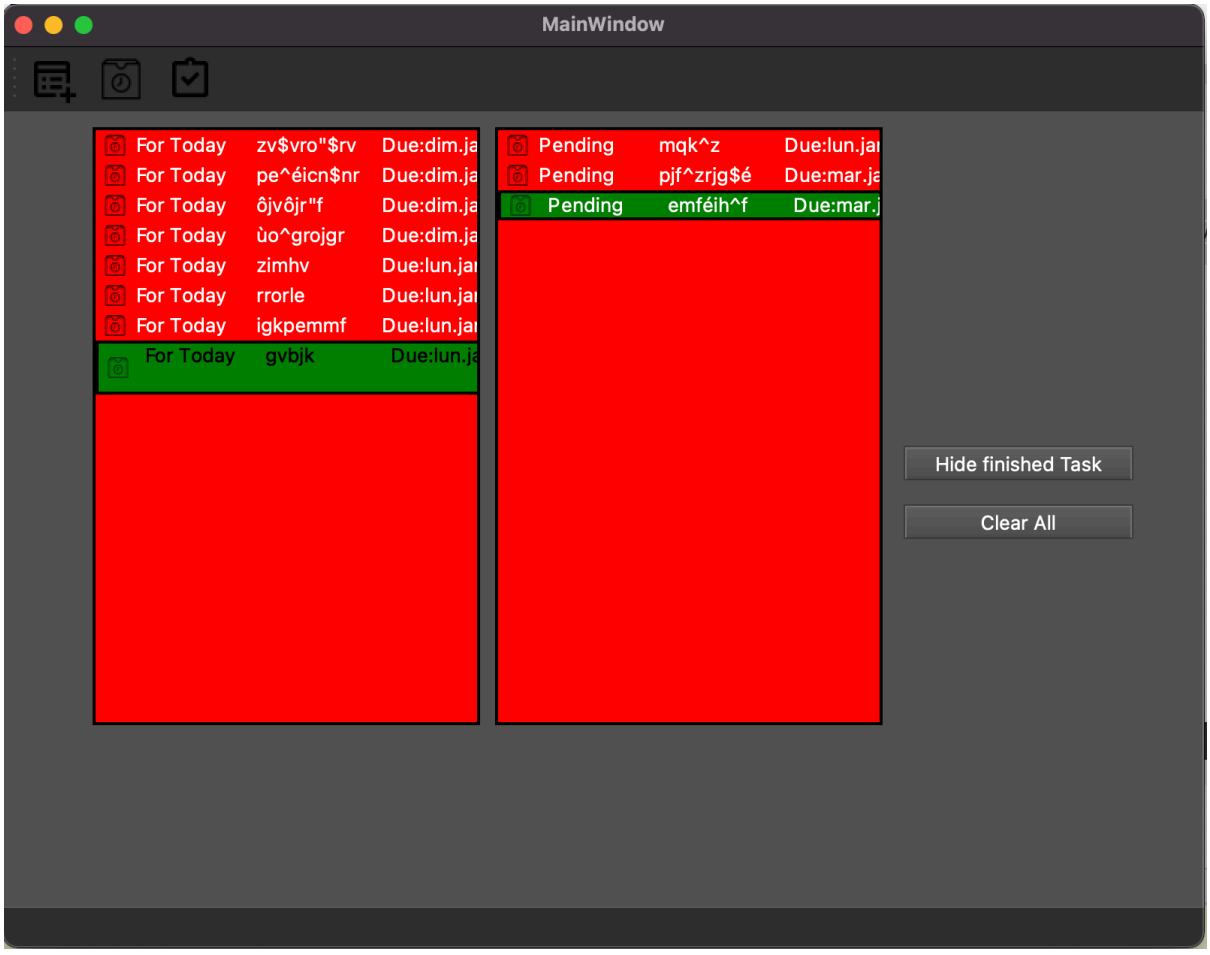
4. Each category must be shown with a **custom** icon.



5. The user could either `hide/show` the pending and finished views.

For this purpose, we added a Push button, that allows the user to hide the Finished tasks QListWidget , once triggered it calls on_cacher_clicked() method

```
void MainWindow::on_cacher_clicked()
{
    ui->Finished->setVisible(false);
}
```



6. Finally, the tasks entered to your application must remains in the app in future use.

Inside the addTask slot, that gets the user's task informations we have created a QFile named « myFile.txt ».

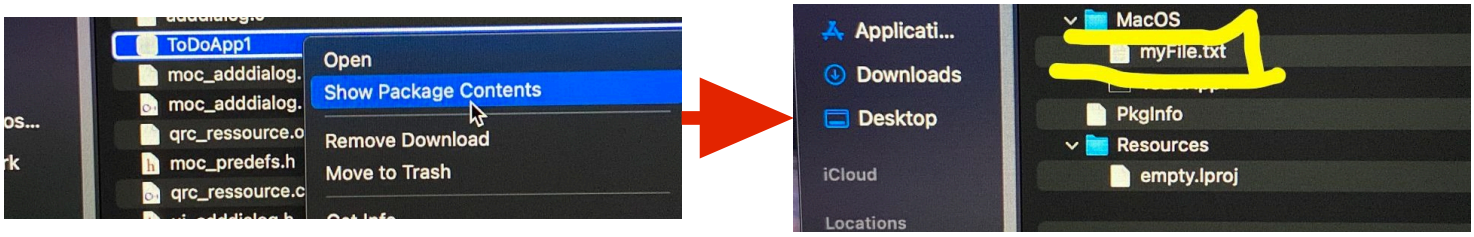
```
QString fichier = "myFile.txt";

QFile file(fichier); // Appel du constructeur de la classe QFile

if (file.open(QIODevice::Append | QIODevice::Text)) {
// Si l'ouverture du fichier en écriture à réussie

// écrire dans le fichier en utilisant un flux :
QTextStream out(&file);
out << newTask;

// Fermer le fichier
file.close();
}
```




And populate it with our values using a QTextStream.

In order to recover the data when re-opening the window we also implemented a method in the main.cpp file, that gets back the tasks line by line from the file and re-positions each line in the corresponding ListWidget.

```
void MainWindow::chargerTasks(QString myFile){
    QFile fichier(myFile);

    if(fichier.open(QIODevice::ReadOnly | QIODevice::Text)) // ReadOnly on lecture // ::Text si le fichier est deja ouvert
    {
        QTextStream flux(&fichier);
        while(!flux.atEnd())
        {
            QString temp = flux.readLine();
            if(  temp.startsWith("Finished"))
            ui->Finished->addItem(new QListWidgetItem(QIcon(":/new/prefix1/done.png"),temp));
            else if( temp.startsWith("Pending"))
                    ui->pendingTask->addItem(new QListWidgetItem(QIcon(":/new/prefix1/pending.png"),temp));
            else
                ui->taskForToday->addItem(new QListWidgetItem(QIcon(":/new/prefix1/pending.png"),temp));
        }
        fichier.close();
    }
```

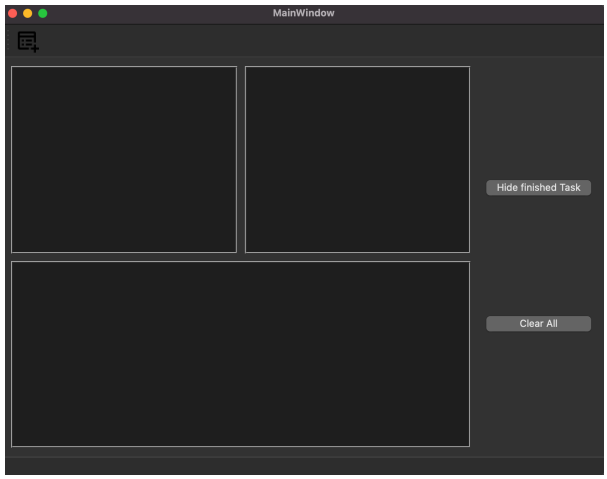
# 4 - Style enhancements :

How an application looks, is an essential criteria for a user friendly window, at the beginning our to do app looked like this :

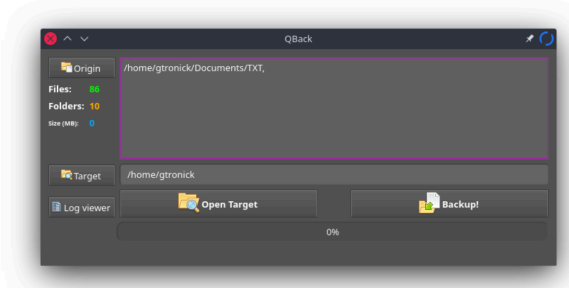
We needed to integrate a styleSheet, after some research we found a very useful templates available on openSource in the form of a .Qss file similar to .Css but made for Qt hence the « Q »

After adding this qss file to the ressources here is how we set up the style sheet to our MainWindow

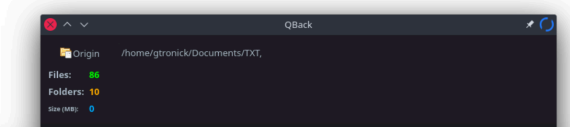
```
    //set style sheet
    QFile styleSheetFile (":/new/prefix1/ElegantDark.qss");
    styleSheetFile.open(QIODevice::ReadOnly);
    QString styleSheet = QLatin1String(styleSheetFile.readAll());
    this->setStyleSheet(styleSheet);
    ui->setupUi(this);
}
```




2. ElegantDark

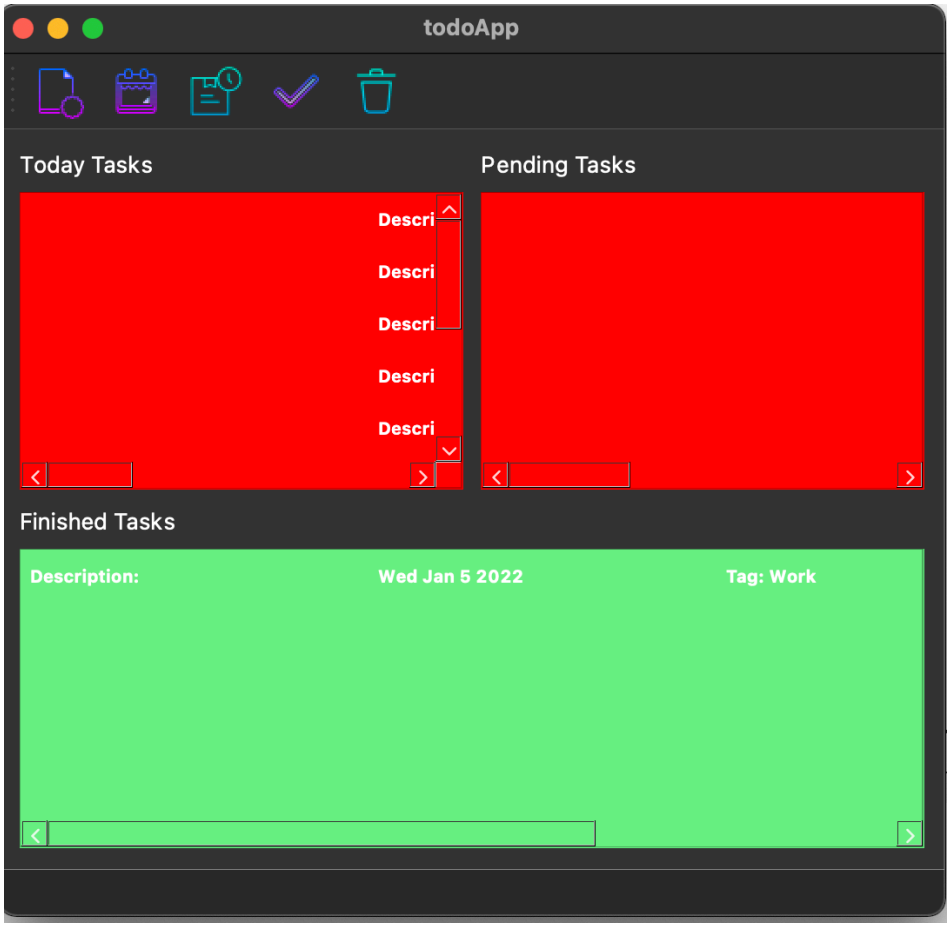



3. MaterialDark



# II - ToDo with MVC

Now, we will try a different approche of qt gui programming based on the MVC model ,Qt contains a set of item view classes that use a model/view architecture to manage the relationship between data and the way it is presented to the user. The separation of functionality introduced by this architecture gives developers greater flexibility to customize the presentation of items, and provides a standard model interface to allow a wide range of data sources to be used with existing item views. In this document, we give a brief introduction to the model/view paradigm, outline the concepts involved, and describe the architecture of the item view system. Each of the components in the architecture is explained, and examples are given that show how to use the classes provided.
Our application will store the data provided by the user into a SQL DataBase



In this version of the app we will use QTableView instead of QListWidget used previously these will represent a view for our model

Starting off with the constructor where we've defined the model, All item models are based on the QAbstractItemModel class. This class defines an interface that is used by views and delegates to access data. The data itself does not have to be stored in the model; it can be held in a data structure or repository provided by a separate class, a file, a database, or some other application component.
In our case we will use a data base that stores all data
then we have enabled the task edition, to change the due date or the description  for instance when the task is double clicked

```
todoApp::todoApp(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::todoApp)
{
    ui->setupUi(this);


    //Définir le model
    todayTaskModel = new QSqlQueryModel;
    pendingTaskModel = new QSqlQueryModel;
    finishedTaskModel = new QSqlQueryModel;

    //se connecter à la base de données
    connectDB();

    //Afficher le contenu de la base de données
    showDB();

    //Make connections
    connect(ui->todayTask, &QTableView::doubleClicked, this, &todoApp::editTaskSlot);

}
```

Moving on to the connectDB method which allows us to add the database with the SQLITE driver, set its path, and check if it was properly opened. Then we've assigned our database to the query

```
void todoApp::connectDB()
{
    //Add the database with the SQLITE driver
    db = QSqlDatabase::addDatabase("QSQLITE");

    //Set the database path
    db.setDatabaseName(this->filename);

    //check if the database is opened
    if(!db.open())
        QMessageBox::critical(this, "FAILED", "DB is not opened");

    //définir la requete sur la base de données
    auto todayTaskQuery = QSqlQuery(db);
```

After defining the query  were going to execute it, and Create the different tables

```
    //Exécuter la requete create
    if(!todayTaskQuery.exec(createTodayTask))
        QMessageBox::critical(this,"Info","Cannot create the table");

    //create the pendingtasks table
    auto  pendingTaskQuery = QSqlQuery(db);
    QString createPendingTask{"CREATE TABLE IF NOT EXISTS pendingtasks"
                              "(description VARCHAR(80),date DATE,"
                              "tag VARCHAR(10),finished BOOLEAN)"};
    if(!pendingTaskQuery.exec(createPendingTask))
        QMessageBox::critical(this,"Info","Cannot create the table");

    //create the finishedtasks table
    auto  finishedTaskQuery = QSqlQuery(db);
    QString createfinishedTask{"CREATE TABLE IF NOT EXISTS finishedtasks"
                               "(description VARCHAR(80),date DATE,"
                               "tag VARCHAR(10),finished BOOLEAN)"};
    if(!finishedTaskQuery.exec(createfinishedTask))
        QMessageBox::critical(this,"Info","Cannot create the table");
```

showDB method basically gets the data from our database and displays in our views

```
void todoApp::showDB()
{
    //Show today tasks
        auto todayQuery = QSqlQuery(db);
        QString today{"SELECT * FROM todaytasks"};
        todayQuery.exec(today);
        todayTaskModel->setQuery(todayQuery);
        ui->todayTask->setModel(todayTaskModel);

    //Show pending tasks
         auto pendingQuery = QSqlQuery(db);
         QString pending{"SELECT * FROM pendingtasks"};
         pendingQuery.exec(pending);
         pendingTaskModel->setQuery(pendingQuery);
         ui->pendingTask->setModel(pendingTaskModel);



    //Show finished tasks
        auto finishedQuery = QSqlQuery(db);
        QString finished{"SELECT * FROM finishedtasks"};
        finishedQuery.exec(finished);
        finishedTaskModel->setQuery(finishedQuery);
        ui->finishedTask->setModel(finishedTaskModel);

}
```

## Dialog creating

When the user clicks “Okay” in the dialog this method calls the different getters of the “add entry method” explained further and appends the task to the appropriates view according to the due date.

```
void todoApp::on_actionNew_Task_triggered()
{
    //create the dialog
    Task T;

    //Execute the dialog
    auto reply = T.exec();

    //if the the dialog is accepted, show the task in the appropriate table view
    if(reply == Task::Accepted)
    {
        addEnty(T.getDesc(), T.getDate(), T.getTag(), T.getStatus());

        if(T.getDate() == QDate::currentDate())
        {
            //définir la requete sur la base de données
            auto  query = QSqlQuery(db);
            QString view{"SELECT * FROM todaytasks"};
            query.exec(view);
            todayTaskModel->setQuery(query);
            ui->todayTask->setModel(todayTaskModel);
        }
        else if(T.getDate() > QDate::currentDate())
        {
            //définir la requete sur la base de données
            auto  query = QSqlQuery(db);
            QString view{"SELECT * FROM pendingtasks"};
            query.exec(view);
            pendingTaskModel->setQuery(query);
            ui->pendingTask->setModel(pendingTaskModel);
        }
        else
        {
            //définir la requete sur la base de données
            auto  query = QSqlQuery(db);
            QString view{"SELECT * FROM finishedtasks"};
            query.exec(view);
            finishedTaskModel->setQuery(query);
            ui->finishedTask->setModel(finishedTaskModel);
        }

    }

}
```

addEntry : In this method we've added a new entry taking as arguments, the description date and tag values provided by the user and according to it's due date the entry will be added either in finished pending or todays task then we have connected model to the queries

```
void todoApp::addEnty(QString desc, QDate date, QString tag, QString finished)
{

    if(date == QDate::currentDate())
    {
        //définir la requete sur la base de données
        auto query = QSqlQuery(db);

        //Ajouter les champs du dialogue à la base de données
        QString addInfo{"INSERT INTO todaytasks (description, date, tag, finished)"
                        " VALUES('%1','%2', '%3', '%4')"};

        //Executer la requete
        if(!query.exec(addInfo.arg(desc).arg(date.toString()).arg(tag).arg(finished)))
            QMessageBox::critical(this,"Info","Cannot add the Entry");

        //Connecter le modèle à la requete
        todayTaskModel->setQuery(query);
    }
    else if(date > QDate::currentDate())
    {
        auto query = QSqlQuery(db);
        QString addInfo{"INSERT INTO pendingtasks VALUES('%1','%2', '%3', '%4')"};
        if(!query.exec(addInfo.arg(desc).arg(date.toString()).arg(tag).arg(finished)))
            QMessageBox::critical(this,"Info","Cannot add the Entry");
        pendingTaskModel->setQuery(query);
    }
    else
    {
        auto query = QSqlQuery(db);
        QString addInfo{"INSERT INTO finishedtasks VALUES('%1','%2', '%3', '%4')"};
        if(!query.exec(addInfo.arg(desc).arg(date.toString()).arg(tag).arg(finished)))
            QMessageBox::critical(this,"Info","Cannot add the Entry");
        finishedTaskModel->setQuery(query);
    }
}
```

on_actionDelete_Tasks_triggered : This method allows us to delete a task from the different view using again an sql query that we insert into a Qstring then we set the query to the model.

Bellow the code for todayTask and the same applies fro pending and finished task

```
void todoApp::on_actionDelete_Tasks_triggered()
{
    //Get the selected rows in todayTasks
    QModelIndexList selectToday = ui->todayTask->selectionModel()->selectedRows();

    //Remove the selected rows from todayTaskModel
    for(int i=0; i<selectToday.count(); i++)
    {
        //Define a query on the DB
        auto query = QSqlQuery(db);

        //Define the body of the query
        QString del{"DELETE FROM todaytasks WHERE description="
                    "'"+selectToday[i].data(Qt::EditRole).toString()+"'"};

        //Execute the query
        if(!query.exec(del))
            qDebug() << "Cannot delete row!";

        //set the query to the model
        todayTaskModel->setQuery(query);

        //submit changes
        todayTaskModel->submit();

        //show changes
        auto todayQuery = QSqlQuery(db);
        QString today{"SELECT * FROM todaytasks"};
        todayQuery.exec(today);
        todayTaskModel->setQuery(todayQuery);
        ui->todayTask->setModel(todayTaskModel);

    }
```

This function allows us to edit a task, like changing the due date for instant, this function opens a new dialog that allows the user to make any modifications, and again this change has to apply to the dabaBase as well for this purpose we use an update SQL query

"UPDATE todaytasks (description, date, tag, finished)"

" VALUES('%1','%2', '%3', '%4') WHERE id = "

```
void todoApp::editTaskSlot()
{
    auto currRow = ui->todayTask->currentIndex().row();

    qDebug() << currRow;

    Task T;
    auto reply = T.exec();

    if(reply == Task::Accepted)
    {
        auto query = QSqlQuery(db);

        //Ajouter les champs du dialogue à la base de données
        QString addInfo{"UPDATE todaytasks (description, date, tag, finished)"
                        " VALUES('%1','%2', '%3', '%4') WHERE id = "};

        //Connecter le modèle à la requete
        todayTaskModel->setQuery(query);
    }

}
```