Après avoir installé correctement **QtCreator** avec les différentes librairies de **Qt** ainsi que tout l'environnement Android, je vous propose dans cette étude, sans être exhaustif, de comprendre le fonctionnement de **QML**. Nous verrons également comment mettre en relation la **vue** proposer par **QML**, et les différents codages nécessaires en **C++** à la fois pour la partie **contrôleur** et la partie **modèle** (relation **BDD**).

Nous reprenons le même projet que nous avons déjà mis en place lors de la phase finale de l'installation complète de l'outil de développement. Nous verrons ainsi plus en détail le code de QML et aussi comment le concevoir à l'aide du mode « design ».

Retour sur la première application Qt Quick

Ce premier projet affiche un simple texte de bienvenue et qui, lorsque nous cliquons dessus, change son texte et nous dit au-revoir, l'application se termine d'elle-même au bout de 2 secondes.

Ce type de projet est intéressant puisqu'il est structuré suivant le modèle de conception MVC (Modèle, Vue et Contrôleur). Ce type de modèle permet de bien séparer l'aspect visuel, décrit ici avec du QML, du traitement de fond (Contrôleur) qui est codé à l'aide de classes C++ qui sont en relation directe avec la ou les vues. Généralement, la partie « Modèle » permet de se mettre en relation avec des bases de données qui sera traité également à l'aide de classes C++, ce que nous appelons des entités (objets persistants).

Ce type de projet respecte cette architecture et le programme principal s'occupe de lancer la vue principale.

Dans ce projet, tout le code se situe dans la vue représentée ici par le fichier **« main.qml »** (remarquez au passage l'utilisation du mécanisme de ressources **« qml.qrc »**).

La structure d'un document QML ressemble à la fois à du CSS, mais surtout à un document JSON puisque nous remarquons des imbrications, seuls les guillemets n'apparaissent pas.

Les noms de chaque élément constituant ce document ressemble étrangement aux composants que l'on connaît avec les « Widgets ». Par exemple, au lieu d'avoir QWindow, nous avons Window, au lieu de QTimer, nous avons Timer.

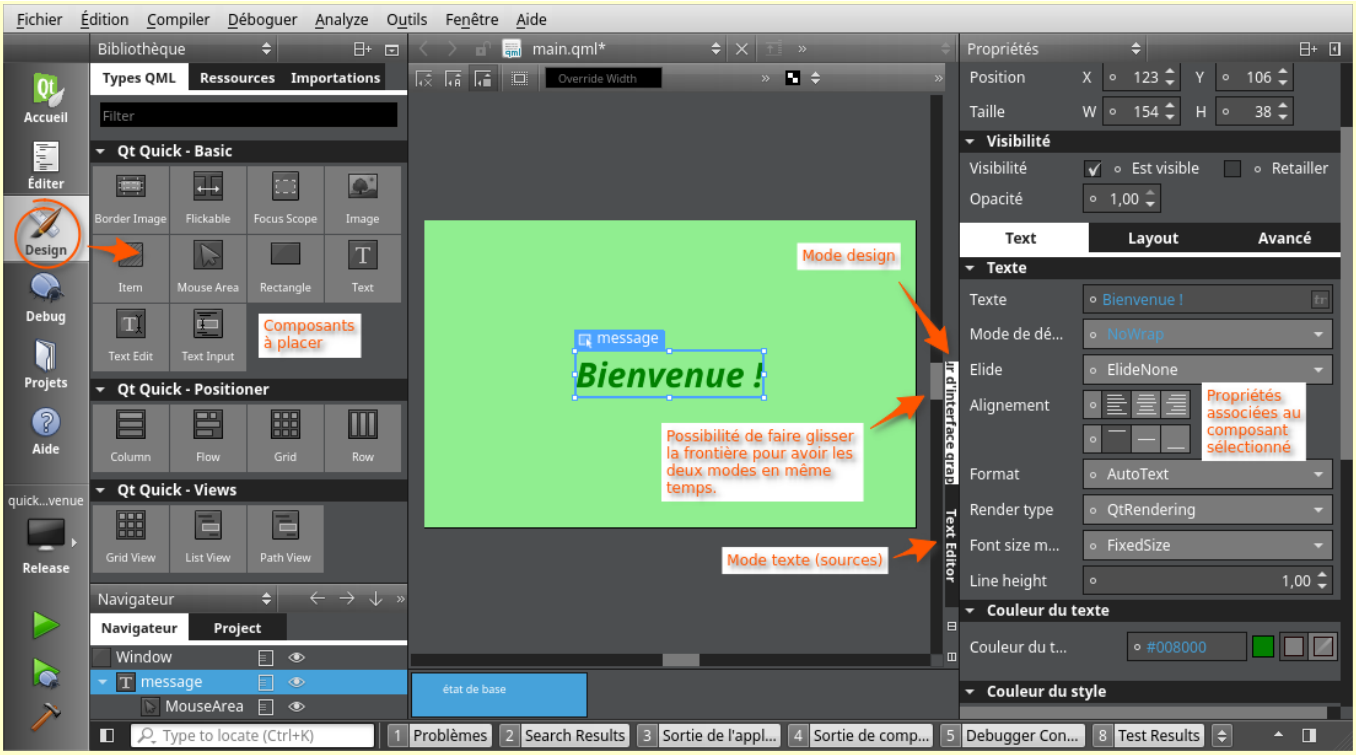
En réalité, tous ces éléments sont les classes que nous avons déjà vus dans les projets classiques et nous agissons directement sur leurs propriétés respectives (attribut avec les deux méthodes accesseur et mutateur).

Remarquez aussi la présence d'identifiant « id » qui permet de nommer l'objet afin de pouvoir l'utiliser par la suite par un autre composant. Comme pour le langage C++, chaque objet doit disposer d'un nom unique sinon une erreur de compilation se produit.

Voici les résultats obtenus suivant le type de cible. Attention, il faut que le smartphone virtuel soit déjà lancé si vous désirez utiliser l'émulateur.

Pour votre vrai smartphone, vous devez autoriser le mode débogage et le brancher sur une prise USB de libre.

Vous pouvez concevoir votre vue en mode **« Design »** comme cela vous est montré ci-dessous. Vous pouvez même avoir les deux modes de conception simultanément.



Tout se que vous faites dans l'éditeur de texte se répercute dans le mode design et vice-versa. Par exemple, vous pouvez faire vos importations directement en mode design, comme cela vous est montré ci-dessous. Faites l'importation des composants de type contrôle, vous verrez ainsi beaucoup plus de composants à votre disposition :

La difficulté dans ce type d'environnement, c'est d'arriver à connaître tous ces composants. Bien sûr, nous n'en ferons pas une étude exhaustive. Il suffit de regarder dans l'aide de QtCreator le rôle de chacun, les propriétés importantes ainsi que les différents événements associés. Cette étude portera plus sur les éléments annexes associés à ces composants.

États et transitions

Je vous propose d'ailleurs de voir un aspect intéressant avec QML que nous ne pouvons pas faire avec un développement Widgets, c'est de prévoir plusieurs états possibles pour un même composant et de prévoir aussi des transitions avec des effets lors d'un passage d'un état vers un autre.

Par exemple, en reprenant ce projet, lorsque nous cliquons sur le message bienvenue, il serait sympathique que le message « Au revoir ! » s'agrandisse automatique de plus en plus en s'estompant juste avant que l'application ne s'arrête.

Pour cela, nous prévoyons l'état de base et un état supplémentaire que nous appellerons **« fondu »** et qui sera activé juste au moment où nous cliquerons sur le message **« bienvenue ! »**.

Cet état « fondu » agit sur l'opacité du message et sur la dimension en pixel de chaque caractères.

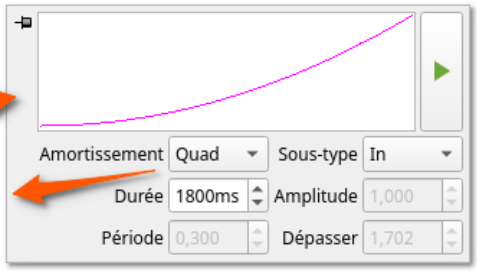
*Pour définir de nouveaux états, vous éditez la propriété **« states »**.*
*Pour définir des transitions, vous éditez la propriété **« transitions »**.*

main.qml

```
import QtQuick 2.9
import QtQuick.Window 2.2

Window {
  id: window
  visible: true
  width: 400
  height: 250
  color: "lightgreen"
  title: qsTr("Bienvenue !")

  Timer {
    id: attente
    interval: 2000
    onTriggered: Qt.quit()
  }

  Text {
    id: message
    anchors.centerIn: parent
    color: "green"
    text: "Bienvenue !"
    font {
      bold: true
      italic: true
      pixelSize: 24
    }

    MouseArea {
      id: souris
      anchors.fill: parent
      onClicked: {
        message.text = "Au revoir !"
        attente.start()
        message.state = "fondu"
      }
    }

    states: [ // les crochets indiquent qu'il peut y avoir plusieurs états
      State {
        name: "fondu"
        PropertyChanges {
          target: message
          opacity: 0
          font.pixelSize: 124
        }
      }
    ]

    transitions: [
      Transition {
        from: "*"
        to: "fondu"
        PropertyAnimation {
          target: message
          property: "opacity"
          duration: 1800
        }
        NumberAnimation { // peut être aussi PropertyAnimation
          target: message
          property: "font.pixelSize"
          duration: 2000
        }
      }
    ]
  }
}
```

```
transitions: [
  Transition {
    from: "*"
    to: "fondu"
    PropertyAnimation {
      target: message
      property: "opacity"
      easing.type: Easing.InQuad
      duration: 1800
    }
    NumberAnimation {
      target: message
      property: "font.pixelSize"
        easing.type: Easing.InOutExpo
      duration: 2000
    }
  }
]
```

Vous pouvez cliquer sur l'ampoule pour activer l'aide.



Plusieurs effets de transitions sont possibles.
Par défaut, c'est une transition linéaire.
Nous pouvons choisir une autre transition sous forme de parabole, comme ici par exemple

Modèle MVC – création de nouveaux composants – nouvelles propriétés

Nous allons maintenant réaliser un projet plus ambitieux qui va nous permettre de voir comment mettre en relation le codage **C++** qui réalise le traitement de fond, appelé **contrôleur**, avec la **vue** décrite en **QML**. Dans le modèle **MVC** (**M**odèle, **V**ue et **C**ontrôleur), la partie **modèle** n'est pas traité ici. Cette partie également en **C++** s'occupe de la connexion avec une base de données pour la persistance des valeurs importantes.

Nous en profiterons pour voir comment créer de nouveaux composants, à la fois en C++, mais aussi en QML (objets réutilisables qui fait la force de la programmation orientée objet). Pour que tous les composants puissent être en relation, que ce soit en C++ ou en QML, vous devez pouvoir les identifier de façon unique (nom de l'objet) et leurs rattacher des propriétés (attribut de la classe avec deux méthodes spécifiques accesseur et mutateur). Alternativement, côté C++, il est aussi possible d'invoquer des méthodes particulières depuis la vue, dans ce cas là elles doivent être déclarées comme telles.

Nous allons commencer par le **contrôleur** dont l'objectif sera de faire les calculs de conversion monétaire. Pour cela, nous allons créer une classe **Monnaie** qui doit impérativement hériter de la classe **Object**. Cet héritage permet de mettre en place la gestion événementielle et la création de nouvelles propriétés (qui n'existent pas dans le langage C++ de base).

monnaie.h

```
#ifndef MONNAIE_H
#define MONNAIE_H


#include <QObject>


class Monnaie : public QObject
{
  Q_OBJECT
  Q_PROPERTY(double euro READ geteuro WRITE seteuro NOTIFY euroChanged)
  Q_PROPERTY(double franc READ getfranc WRITE setfranc NOTIFY francChanged)
public:
  explicit Controleur(QObject *parent = nullptr) : QObject(parent) {}
signals:
  void euroChanged();
  void francChanged();
public:
  double geteuro() const         { return euro; }
  double getfranc() const        { return franc; }
  void seteuro(double nouveau)   { euro=nouveau; franc=euro*TAUX; }
  void setfranc(double nouveau)  { franc=nouveau; euro=franc/TAUX; }
private:
  double euro=0.0, franc=0.0;
  const double TAUX=6.55957;
};


#endif // MONNAIE_H
```

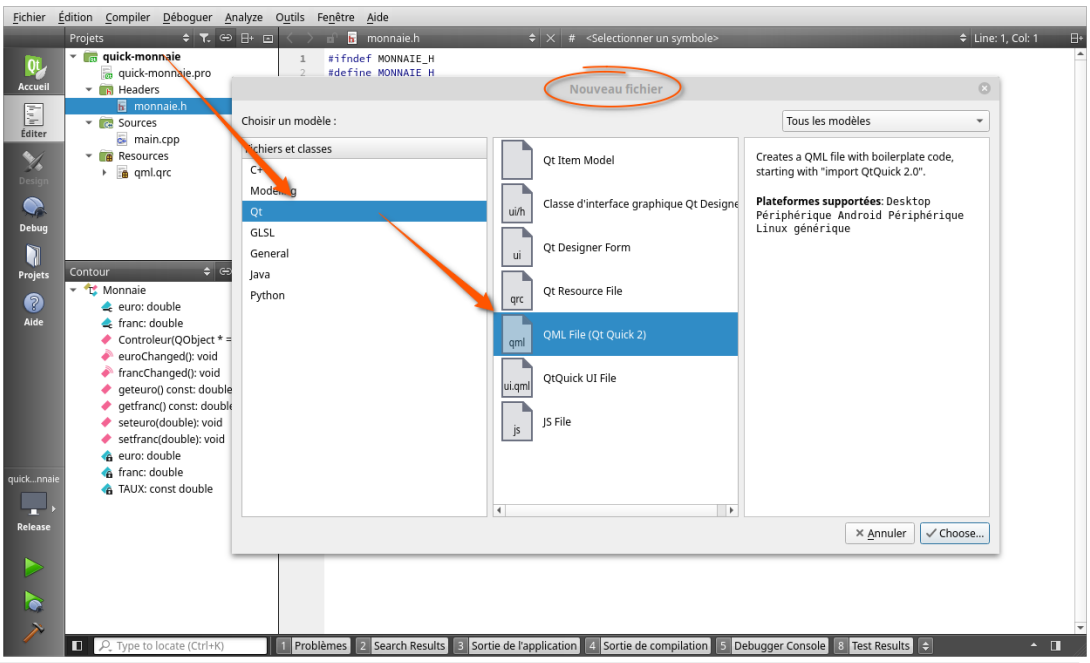
Vu le peu de traitement à faire, l'écriture complète de la classe se fait directement dans le fichier inclu avec des méthodes « inline ». Les propriétés sont réalisées grâce à la macro Q_PROPERTY. Cette déclaration permet de mettre en relation l'attribut et les méthodes de lecture et d'écriture ainsi que l'événement associé (non utile ici), tout ceci décrit dans le reste de la classe.

Pour que ce contrôleur soit pris en compte, vous devez créer un objet dans le fichier principal, dans la fonction « main() » et aussi faire en sorte que ce contrôleur soit connu de la vue « main.qml ». Pour cela, vous devez proposer un nom d'objet (texte, QML est un script) associé au véritable objet créé.

main.cpp

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QtQml>
#include <monnaie.h>


int main(int argc, char *argv[])
{
  QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
  QGuiApplication app(argc, argv);


  Monnaie monnaie;
  QQmlApplicationEngine engine;
  engine.rootContext()->setContextProperty("monnaie", &monnaie);
  engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
  if (engine.rootObjects().isEmpty()) return -1;


  return app.exec();
}
```

Ensuite nous créons un nouveau composant visuel, décrit donc en QML, qui pourra être exploité par la suite par la vue principale. Ce composant permettra la saisie de valeurs monétaires directement, un objet pour les euros, et un autre objet pour les francs. Il s'appelle « Saisie.qml ».



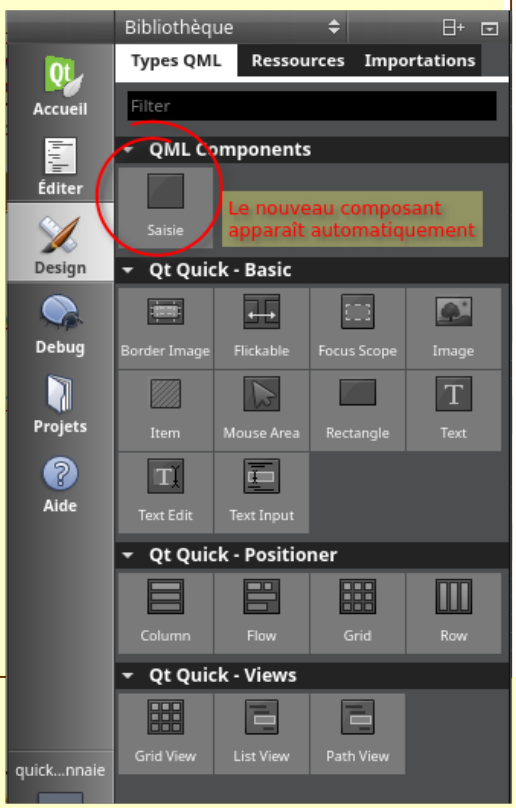
Comme pour les classes C++, il est possible de demander la création d'un nouveau composant QML, il suffit de donner le nom du fichier QML adéquat pour qu'il soit reconnu tel quel par la suite (c'est le nom du fichier qui précise le nom du composant). Ce composant est déjà un composant de haut niveau que nous connaissons bien, il s'agit d'un TextField (QTextField en C++). Le fait de rajouter des spécificités supplémentaires correspond finalement à un héritage en C++. Pour que ces spécificités soient agréables à utiliser, nous créons trois propriétés :

1. *la première s'appelle **symbole** qui nous permet de choisir le symbole monétaire que nous souhaitons prendre en compte, respectivement **« € »** et **« F »**.*
2. *La deuxième, nommée **valeur** permet en temps réel d'avoir la valeur réelle (**double**) équivalente au texte saisie.*
3. *Enfin, **nouvelleValeur** qui prend en compte la nouvelle valeur saisie, afin d'éviter d'avoir des boucles événementielles infinies. De plus, l'affichage se réajuste en donnant le symbole monétaire, propose deux chiffres après la virgule et visualise même la séparation des milliers par un espace (en France).*

Il est à noter que lorsque nous proposons de nouvelles propriétés, il est possible de prendre en compte le changement de valeur avec la syntaxe spécifique « onNouvellepropriétéChanged ».

Saisie.qml

```
import QtQuick 2.9
import QtQuick.Controls 2.2


TextField {
  property string symbole: "$"
  property real valeur: Number.fromLocaleString(Qt.locale("fr_FR"), text)
  property real nouvelleValeur: 0.0


  onNouvelleValeurChanged: {
     text = Number(nouvelleValeur).toLocaleString(Qt.locale("fr_FR"), 'f', 2)
  }


  onSymboleChanged: {
    text = "0,00"
    préfix.text = symbole
  }


  Text {
    id: préfix
    text: symbole
    font.pointSize: 12
    x: 7
    y: 3
    color: "darkred"
  }


  font.pointSize: 16
  leftPadding: 30
  inputMethodHints: Qt.ImhDigitsOnly // Clavier virtuel numérique


  anchors {
     left: parent.left
     right: parent.right
     margins: 14
  }


  validator: DoubleValidator {
     locale: "fr_FR"
     decimals: 2
     bottom: 0
  }
}
```



À la fin de ce script nous remarquons la présence d'un validateur de type double. C'est cet élément qui permet de faire les conversions automatiques entre les valeurs réelles pour faire les calculs et le texte associé avec le formatage qui suit l'écriture monétaire en France.

Lorsque vous basculez sur la vue principale, vous voyez apparaître ce nouveau composant. Tous les réglages et précisions que nous venons d'effectuer seront automatiquement pris en compte sur les deux objets euro et franc que nous allons mettre en place

Penchons nous maintenant sur la vue principale. Nous profitons de l'occasion pour prévoir une couleur dégradée de l'orange vers le jaune. Ensuite, nous rajouterons un bouton rond avec une icône pour quitter l'application, uniquement si nous sommes sur un smartphone ou sur une tablette (plateforme **android**).

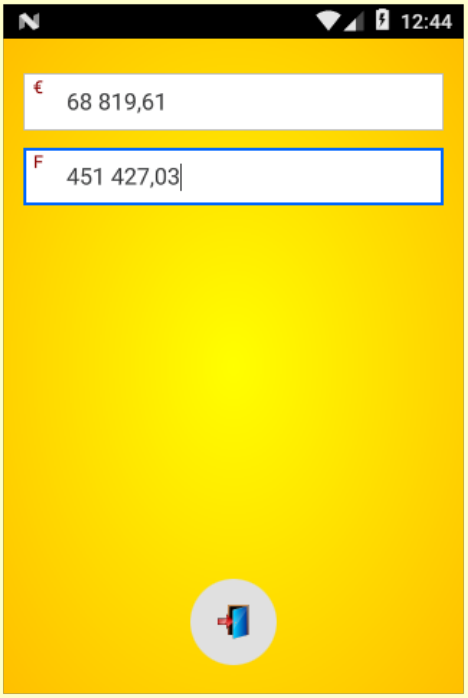
Pour cette vue principale, nous utilisons les compétences des deux composants (classes) que nous venons de créer , d'une part l'objet monnaie développé en C++, et d'autre part la vue Saisie.

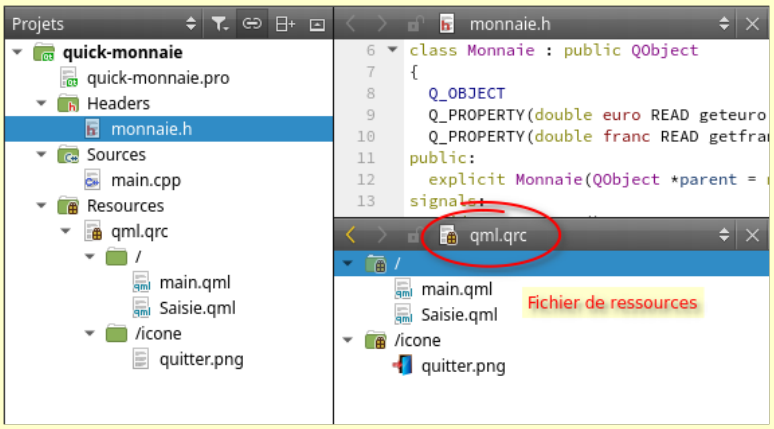
Il faut que le ou les fichiers icônes soient placés dans une ressource afin qu'ils puissent être déployés automatiquement avec l'exécutable, comme nous le proposons dans les différentes vues. Prévoyez alors un préfixe adapté à ces icônes.

main.qml

```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtGraphicalEffects 1.0
import QtQuick.Controls 2.3
```

```
Window {
  id: window
  visible: true
  width: 300
  height: 400
  title: qsTr("Conversion monétaire")

  RadialGradient {
    anchors.fill: parent
    gradient: Gradient {
      GradientStop { position: 0.0; color: "yellow" }
      GradientStop { position: 1.0; color: "orange" }
    }
  }

  RoundButton {
    visible: Qt.platform.os == "android"
    width: 60
    height: 60
    icon {
      source: "qrc:/icone/quitter.png"
      color: "transparent"
    }
    anchors {
      bottom: parent.bottom
      bottomMargin: 20
      horizontalCenter: parent.horizontalCenter
    }
    onClicked: Qt.quit()
  }

  Column {
    spacing: 12
    width: parent.width
    anchors.top: parent.top
    anchors.topMargin: 24

    Saisie {
      id: euro
      symbole: "€"
      onAccepted: {
        nouvelleValeur = valeur
        monnaie.euro = valeur
        franc.nouvelleValeur = monnaie.franc
      }
    }

    Saisie {
      id: franc
      symbole: "F"
      onAccepted: {
        nouvelleValeur = valeur
        monnaie.franc = valeur
        euro.nouvelleValeur = monnaie.euro
      }
    }
  }
}
```





La deuxième approche au niveau du **contrôleur** est juste de proposer des méthodes de traitement qui vont réaliser les calculs sans passer par les propriétés. Dans ce cas de figure, vous devez spécifier (déclarer) que ces méthodes sont **« invocables »** depuis la **vue**.

*Je vous propose de rajouter à notre projet un deuxième contrôleur **Conversion** qui prend en compte ces nouvelles spécificités. Bien entendu, nous devons créer une nouvelle classe qui hérite de **QObject** pour les mêmes raisons que précédemment. Dans votre programme principal, pensez de nouveau à créer un objet de cette classe et rattachez lui un nom simple pour la vue principale.*

main.cpp

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QtQml>
#include <monnaie.h>
#include <conversion.h>

int main(int argc, char *argv[])
{
  QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
  QGuiApplication app(argc, argv);

  Monnaie monnaie;
  Conversion conversion;
  QQmlApplicationEngine engine;
  engine.rootContext()->setContextProperty("monnaie", &monnaie);
  engine.rootContext()->setContextProperty("conv", &conversion);
  engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
  if (engine.rootObjects().isEmpty()) return -1;

  return app.exec();
}
```

La vue connaît maintenant un nouvel objet de conversion nommé « conv ».

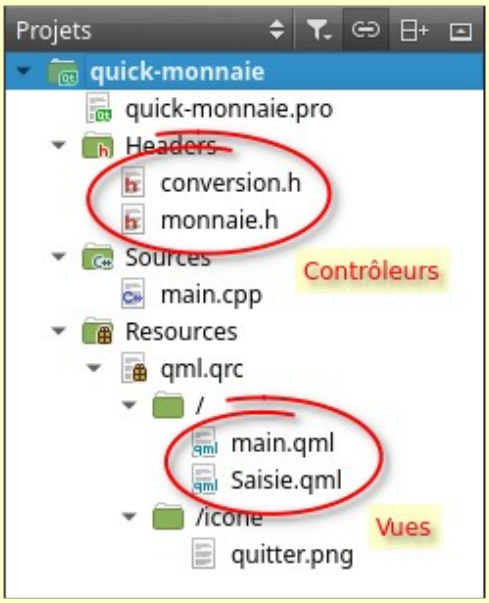
Conversion.h

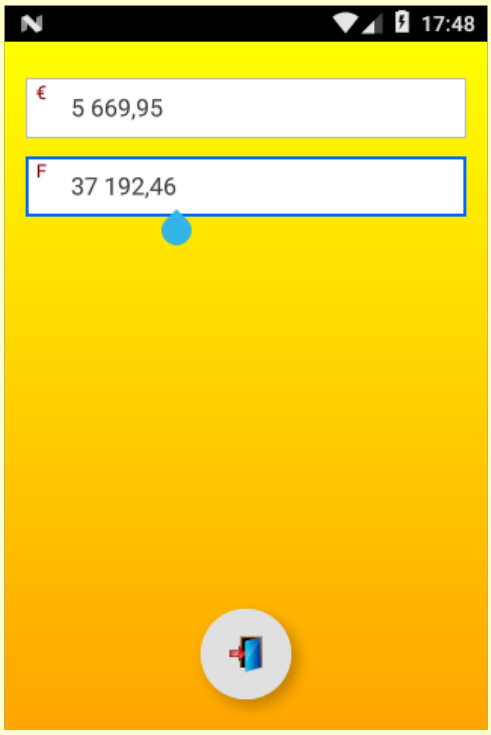
```
#ifndef CONVERSION_H
#define CONVERSION_H


#include <QObject>


class Conversion : public QObject
{
  Q_OBJECT
public:
  explicit Conversion(QObject *parent = nullptr) : QObject(parent) {}
  Q_INVOKABLE double euroFranc(double valeur)  { return valeur*TAUX; }
  Q_INVOKABLE double francEuro(double valeur)  { return valeur/TAUX; }
private:
  const double TAUX = 6.55957;
};


#endif // CONVERSION_H
```

Modifions la vue principale en conséquence avec cette fois-ci un dégradé linéaire et une ombre portée sur le bouton de clôture.

main.qml

```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtQuick.Controls 2.3
import QtGraphicalEffects 1.0


Window {
  visible: true
  width: 300
  height: 400
  title: qsTr("Conversion monétaire")


  LinearGradient {
    anchors.fill: parent
    gradient: Gradient {
      GradientStop { position: 0.0; color: "yellow" }
      GradientStop { position: 1.0; color: "orange" }
    }
  }


  DropShadow {
    visible: Qt.platform.os == "android"
    source: bouton
    anchors.fill: bouton
    horizontalOffset: 4
    verticalOffset: 4
    radius: 8
    color: "#40000000"
  }


  RoundButton {
    id: bouton
    visible: Qt.platform.os == "android"
    width: 60
    height: 60
    icon {
      source: "qrc:/icone/quitter.png"
      color: "transparent"
    }
    anchors {
      bottom: parent.bottom
      bottomMargin: 20
      horizontalCenter: parent.horizontalCenter
    }
    onClicked: Qt.quit()
  }


  Column {
    spacing: 12
    width: parent.width
    anchors.top: parent.top
    anchors.topMargin: 24


    Saisie {
      id: euro
      symbole: "€"
      onAccepted: {
        nouvelleValeur = valeur
        franc.nouvelleValeur = conv.euroFranc(valeur)
      }
    }


    Saisie {
      id: franc
      symbole: "F"
      onAccepted: {
        nouvelleValeur = valeur
        euro.nouvelleValeur = conv.francEuro(valeur)
      }
    }
  }
}
```





Une deuxième solution pour la classe Conversion, la plus simple à mon avis puisque nous connaissons déjà bien le principe des « signaux » et des « slots », est de définir ces méthodes de rappel euroFranc() et francEuro() comme de simples « slots » comme cela vous est montré ci-dessous.

Conversion.h

```
#ifndef CONVERSION_H
#define CONVERSION_H


#include <QObject>


class Conversion : public QObject
{
  Q_OBJECT
public:
  explicit Conversion(QObject *parent = nullptr) : QObject(parent) {}
public slots:
  double euroFranc(double valeur)  { return valeur*TAUX; }
  double francEuro(double valeur)  { return valeur/TAUX; }
private:
  const double TAUX = 6.55957;
};


#endif // CONVERSION_H
```

Prise en compte des événements issus des contrôleurs

Nous allons voir maintenant comment prendre en compte dans les différentes **vues** des événements déclenchés par les **contrôleurs**. Reprenons la première partie du projet précédent, avec le **contrôleur Monnaie**.

Dans la première écriture que nous avons faites, les signaux ont bien été déclarés, mais ils n'ont jamais été déclenchés par le contrôleur lui-même. Je vous propose de remédier à ce manque. Voici la modification du code à apporter pour cela :

monnaie.h

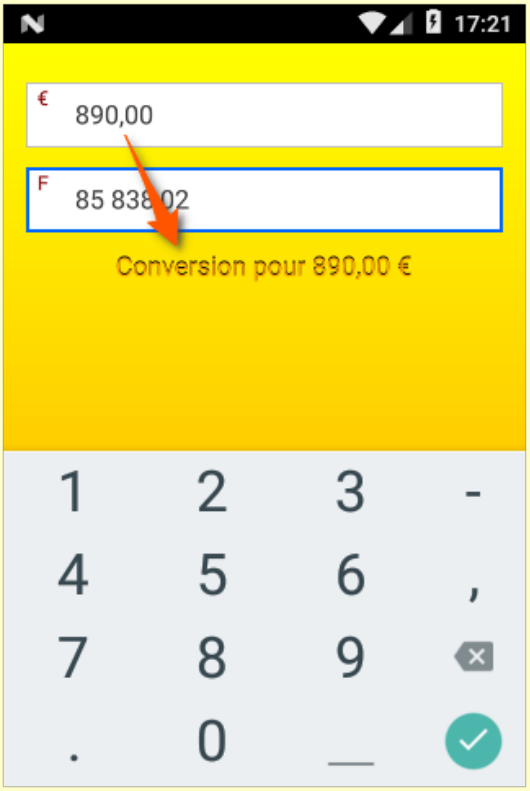
```
#ifndef MONNAIE_H
#define MONNAIE_H


#include <QObject>


class Monnaie : public QObject
{
  Q_OBJECT
  Q_PROPERTY(double euro READ geteuro WRITE seteuro NOTIFY euroChanged)
  Q_PROPERTY(double franc READ getfranc WRITE setfranc NOTIFY francChanged)
public:
  explicit Monnaie(QObject *parent = nullptr) : QObject(parent) {}
signals:
  void euroChanged();
  void francChanged();
public:
  double geteuro() const         { return euro; }
  double getfranc() const        { return franc; }
  void seteuro(double nouveau)   { euro=nouveau; franc=euro*TAUX; euroChanged(); }
  void setfranc(double nouveau)  { franc=nouveau; euro=franc/TAUX; francChanged(); }
private:
  double euro=0.0, franc=0.0;
  const double TAUX=6.55957;
};


#endif // MONNAIE_H
```

Lorsque vous êtes avec des applications avec des Widgets, vous passez par la méthode connect() pour mettre en relation les différents signaux des slots à activer. Dans le cas de QML, vous devez passer par la commande « Connections ». Vous devez alors spécifier la source des événements au moyen de la propriété « target ».

main.qml

```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtQuick.Controls 2.3
import QtGraphicalEffects 1.0
import QtQuick.Layouts 1.3


Window {
  id: window
  visible: true
  width: 300
  height: 400
  title: qsTr("Conversion monétaire")


  LinearGradient {
    anchors.fill: parent
    gradient: Gradient {
      GradientStop { position: 0.0; color: "yellow" }
      GradientStop { position: 1.0; color: "orange" }
    }
  }
```

```
  DropShadow {
    visible: Qt.platform.os == "android"
    source: bouton
    anchors.fill: bouton
    horizontalOffset: 4
    verticalOffset: 4
    radius: 8
    color: "#40000000"
  }

  RoundButton {
    id: bouton
    visible: Qt.platform.os == "android"
    width: 60
    height: 60
    icon {
      source: "qrc:/icone/quitter.png"
      color: "transparent"
    }
    anchors {
      bottom: parent.bottom
      bottomMargin: 20
      horizontalCenter: parent.horizontalCenter
    }
    onClicked: Qt.quit()
  }

  Column {
    spacing: 12
    width: parent.width
    anchors {
      top: parent.top
      topMargin: 24
    }

    Saisie {
      id: euro
      symbole: "€"
      onAccepted: {
        nouvelleValeur = valeur
        monnaie.euro = valeur
        franc.nouvelleValeur = monnaie.franc
      }
    }

    Saisie {
      id: franc
      symbole: "F"
      onAccepted: {
        nouvelleValeur = valeur
        monnaie.franc = valeur
        euro.nouvelleValeur = monnaie.euro
      }
    }

    Text {
      id: calcul
      text: "Événements"
      anchors.horizontalCenter: parent.horizontalCenter
      style: Text.Sunken  // Mise en place d'un texte en relief
      font.pixelSize: 16
      color: "orange"
    }
  }

  Connections {
    target: monnaie
    onEuroChanged: calcul.text = "Conversion pour "+euro.text+" €"
    onFrancChanged: calcul.text = "Conversion pour "+franc.text+" F"
  }
}
```



Un fonctionnalité intéressante pour les textes est de pouvoir ajouter un effet d'ombrage autour des caractères grâce à la propriété style. Ouline ajoute l'effet autour des caractères, Raised au dessus des caractères et Sunken en dessous des caractères.

Manipulation d'images avec gestion du glisser-déposer

Dans ce chapitre, je vous propose de voir comment visualiser des photos uniquement côté vue sans contrôleur. Nous prévoyons d'afficher la photo sélectionnée de deux façons différentes, soit de telle sorte que nous puissions voir la globalité de l'image, soit en mode taille réelle. Dans ce dernier cas, il sera alors possible de déplacer la photo pour voir une partie de l'image, soit avec la souris pour une application sur PC, soit avec le doigt pour une application « **android** ». Enfin, sur le PC classique (sous Linux), vous pourrez lancer un sélecteur de fichier pour choisir une autre photo à visualiser.

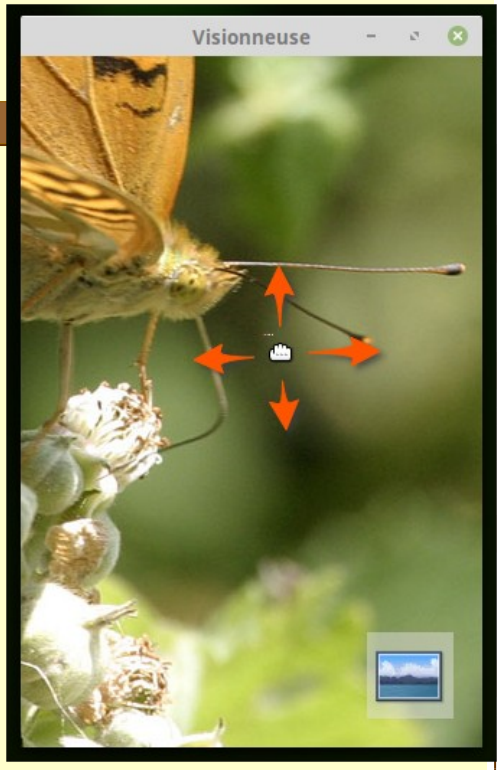
Un clic sur l'image permet de basculer entre le mode taille réelle ou visualisation complète de la photo suivant les dimensions actuelles de la fenêtre de l'application (ou de l'activité sous android).

Une pression prolongée sur le bouton de la souris (ou le doigt) permet de déplacer la photo pour visualiser la partie qui nous intéresse (uniquement en mode taille réelle). Attention, pour que cela puisse fonctionner, il est impératif que l'image ne soit pas placée par des ancres, sinon comme son nom l'indique, l'image reste figée.

Pour le sélecteur de fichier, nous retrouvons le composant FileDialog qui est tout à fait similaire à celui utilisé avec les Widgets (QFileDialog).

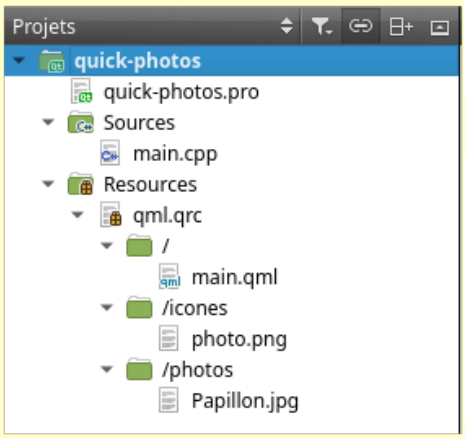
La gestion de la photo se fait avec le composant Image, similaire à Qimage. Une propriété importante dans ce composant est la propriété fillMode qui permet de choisir comment afficher l'image par rapport à son conteneur :

1. *Stretch : l'image est étirée pour prendre tout l'espace possible.*
2. *PreserveAspectFit : l'image est étirée pour avoir la plus grande taille possible en respectant les proportions.*
3. *PreserveAspectCrop : l'image est étirée au maximum pour avoir la plus grande taille possible (en coupant les bords si nécessaire) en respectant les proportions.*
4. *Tile : L'image est dupliquée horizontalement et verticalement.*
5. *TileVertically : L'image est dupliquée verticalement et étirée horizontalement.*
6. *TileHorizontally : L'image est dupliquée horizontalement et étirée verticalement.*
7. *Pad : La taille de l'image n'est pas modifiée.*



main.qml

```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtQuick.Dialogs 1.2
import QtQuick.Controls 2.2

Window {
  visible: true
  width: 320
  height: 480
  title: qsTr("Visionneuse")

  FileDialog {
    id: fichier
    title: "Votre photo"
    folder: shortcuts.pictures
    nameFilters: ["Photos (*.jpg *.png)", "Tous les fichiers (*)"]
    onAccepted: photo.source = fileUrl
  }

  Image {
    id: photo
    width: parent.width
    height: parent.height
    source: "qrc:/photos/Papillon.jpg"
    fillMode: Image.PreserveAspectCrop

    MouseArea {
      id: souris
      property bool enEntier: true
      anchors.fill: parent
      drag.target: enEntier ? souris : parent
      onClicked: {
        enEntier = !enEntier
        parent.width = enEntier ? photo.parent.width : photo.sourceSize.width
        parent.height = enEntier ? photo.parent.height : photo.sourceSize.height
        if (enEntier) { photo.x = 0; photo.y = 0 }
        else {
          photo.x = photo.parent.width/2 - photo.sourceSize.width/2
          photo.y = photo.parent.height/2 - photo.sourceSize.height/2
        }
      }
      cursorShape: drag.active ? Qt.SizeAllCursor : Qt.ArrowCursor
    }
  }

  ToolButton {
    visible: Qt.platform.os == "linux"
    icon {
      width: 48
      height: 48
      source: "qrc:/icones/photo.png"
      color: "transparent"
    }
    anchors {
      bottom: parent.bottom
      right: parent.right
      bottomMargin: 20
      rightMargin: 20
    }
    onClicked: fichier.open()
  }
}
```

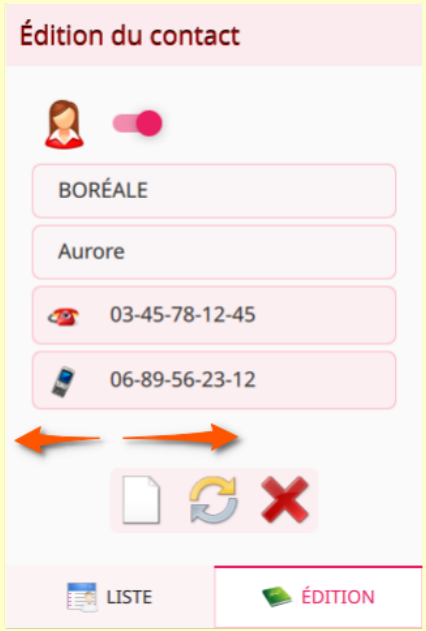


Modèle MVC

Je vous propose de clôturer cette étude préliminaire en réalisant un projet complet qui nous permettra de bien comprendre le modèle **MVC** dans sa globalité. Pour cela, nous mettons en œuvre une application qui permet de gérer les contacts téléphoniques.

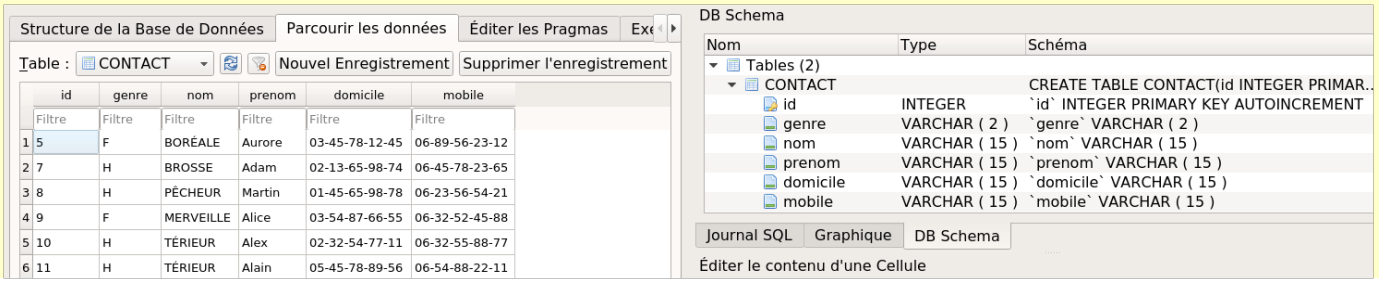
Le modèle de conception MVC (Modèle, Vue et Contrôleur) permet de bien séparer l'aspect visuel, décrit en QML, du traitement de fond (Contrôleur) qui est codé à l'aide d'une classe C++ qui est en relation directe avec les différentes vues. Ici, le « Modèle » également codé avec une classe C++ correspond à une table d'une base de données (à chaque table correspond une classe spécifique). Tous les objets issues de cette classe sont les différents enregistrements effectués (lignes de la table dans la base). Les classes dans ce contexte s'appellent des entités (objets persistants). Enfin, le contrôleur sert d'intermédiaire entre les vues et le modèle. Les différents requêtes SQL seront ainsi exprimées directement dans des méthodes spécifiques du contrôleur.

Ce projet nous permet également de découvrir de nouveaux concepts que nous n'avons pas encore eu le temps de voir, par exemple, la possibilité de présenter plusieurs vues dans une même application, ce que nous voyons souvent dans les smartphones, notamment le « swipe » (faire glisser sur les côtés pour passer d'une vue à l'autre).

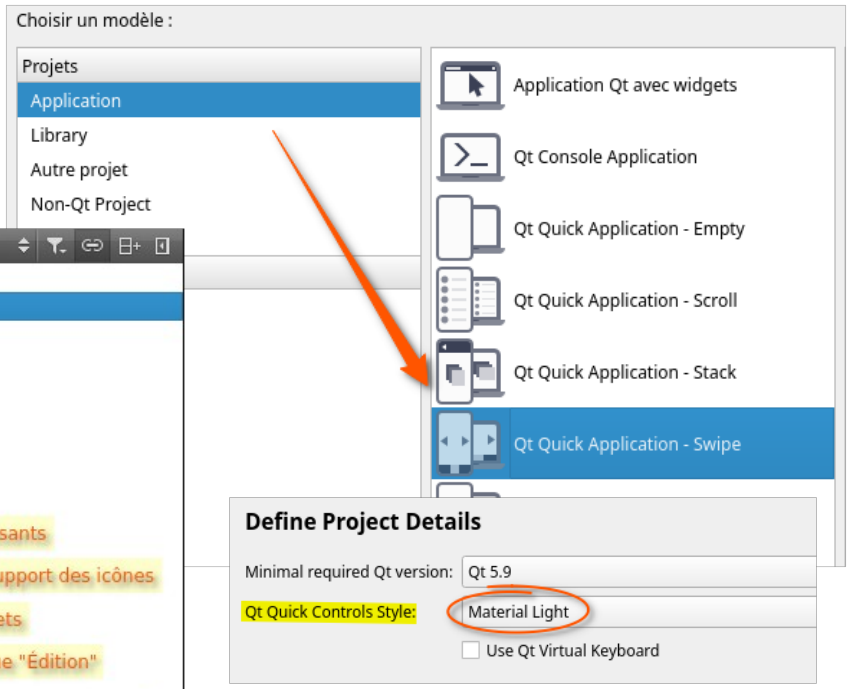


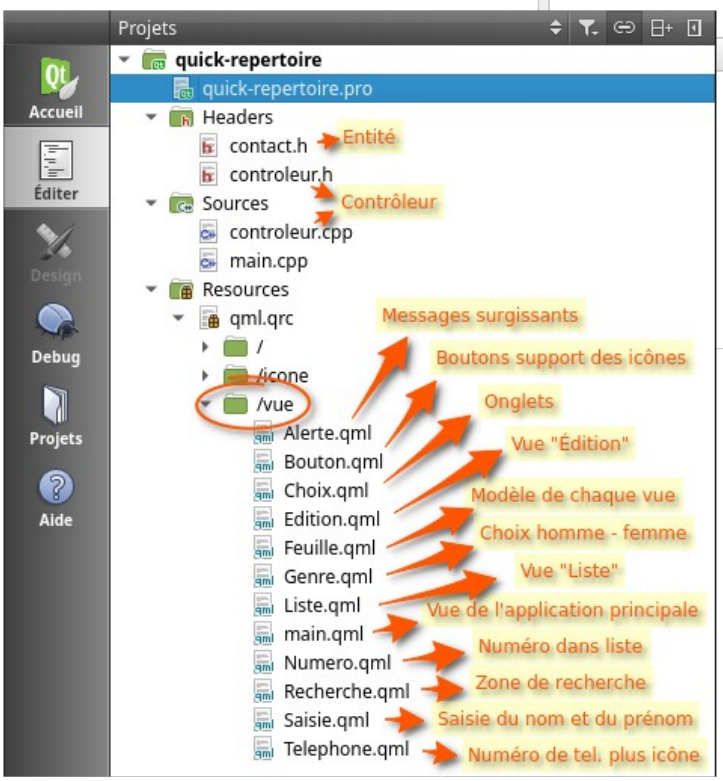
Nous verrons ensuite : comment construire des prototypes de composants qui seront ensuite complétés suivant le besoin, comment créer une liste de données, comment mettre en œuvre un ascenseur vertical pour parcourir ces données, comment les entités sont reconnues automatiquement par les différentes vues, etc.

Vous avez ci-dessous la structure de la base de données sachant qu'en réalité, il s'agit d'un seul fichier (mieux pour les smartphones) puisque nous utilisons SQLITE.



Lorsque vous créez une application, vous pouvez sélectionner le type de projet qui vous intéresse sachant qu'un certain nombre sont déjà prédéfinis. Vous remarquez justement un projet qui prend en compte dès le départ le « **Faire glisser** ».





Vous voyez ci-contre l'ossature de notre projet. Nous avons **l'entité** (**le modèle**) dont la classe s'appelle « **Contact** » qui représente la seule table de la base de données, qui s'appelle elle-même « **CONTACT** ». Nous avons le **contrôleur** qui implémente l'ensemble des requêtes SQL nécessaires avec les événements associés. Enfin, nous avons toutes les **vues**, souvent nombreuses dans les projets, sachant qu'il s'agit de créer de nouveaux composants visuels qui sont utilisés par la suite dans les vues principales et évite ainsi beaucoup de copier-coller.

Nous allons voir maintenant l'ensemble du code de cette application. Le mieux c'est de commencer par implémenter les classes C++, en premier lieu la ou les **entités**, et ensuite le **contrôleur**. Mais d'abord voyons le fichier de projet dans lequel vous devez indiquez que vous rajoutez le module « **sql** » qui nous permet d'utiliser toutes les classes spécifiques à l'exploitation des requêtes **SQL**.

quick-repertoire

```
QT += quick sql
CONFIG += c++11
DEFINES += QT_DEPRECATED_WARNINGS

SOURCES += main.cpp controleur.cpp
HEADERS += controleur.h contact.h
RESOURCES += qml.qrc

# Additional import path used to resolve QML modules in Qt Creator's code model
QML_IMPORT_PATH =

# Additional import path used to resolve QML modules just for Qt Quick Designer
QML_DESIGNER_IMPORT_PATH =

# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

contact.h

```
#ifndef CONTACT_H
#define CONTACT_H
#include <Qobject>

class Contact : public QObject
{
  Q_OBJECT
  Q_PROPERTY(QString nom READ getNom WRITE setNom)
  Q_PROPERTY(QString prenom READ getPrenom WRITE setPrenom)
  Q_PROPERTY(int identifiant MEMBER id)
  Q_PROPERTY(QString genre MEMBER genre NOTIFY genreChanged)
  Q_PROPERTY(QString domicile MEMBER domicile NOTIFY domicileChanged)
  Q_PROPERTY(QString mobile MEMBER mobile NOTIFY mobileChanged)
  Q_PROPERTY(QString identite READ identite NOTIFY identiteChanged)
public:
  Contact(QObject *parent = nullptr) : QObject(parent) {}
  QString getNom() const               { return nom; }
  QString getPrenom() const            { return prenom; }
  void setNom(const QString& nom)      { this->nom = nom.toUpper(); }
  void setPrenom(const QString& prenom)
  {
    this->prenom = prenom.toLower();
    this->prenom[0] = prenom[0].toUpper();
  }
signals:
  void genreChanged();
  void domicileChanged();
  void mobileChanged();
  void identiteChanged();
public slots:
  QString identite() const { return nom+" "+prenom; }
private:
  QString nom, prenom;
public:
  int id;
  QString domicile, mobile, genre;
};

#endif // CONTACT_H
```

Bien entendu, les entités doivent posséder toutes les propriétés nécessaires pour être correctement exploités dans les différentes vues. Certaines passent par des méthodes accesseurs et mutateurs (get et set) pour, par exemple, avoir automatiquement les noms en majuscule, d'autres sont plus directes puisqu'elles n'ont pas besoin de traitement spécifique. Dans ce dernier cas, les attributs doivent être publics.

controleur.h

```
#ifndef CONTROLEUR_H
#define CONTROLEUR_H

#include <QObject>
#include <QtQml/QQmlListProperty>
#include <contact.h>

class Controleur : public QObject
{
  Q_OBJECT
  Q_PROPERTY(bool etat MEMBER etat)
  Q_PROPERTY(QQmlListProperty<Contact> liste READ getContacts NOTIFY listeChanged)
public:
  explicit Controleur(QObject *parent = nullptr) : QObject(parent)
  {
    ouvertureBD();
    creationTable();
    listeContacts();
  }
  QQmlListProperty<Contact> getContacts()
  {
    return QQmlListProperty<Contact>(this, contacts);
  }
signals:
  void enregistrementChanged();
  void modificationChanged();
  void suppressionChanged();
  void contactChanged();
  void listeChanged();
public slots:
  void enregistrer(Contact *contact);
  void modifier(Contact *contact);
  void supprimer(int identifiant);
  void getId(Contact *contact);
  void listeContacts(const QString& critere="");
  void choisirContact(Contact *contact, int identifiant);
private:
  void ouvertureBD();
  void creationTable();
  QList<Contact *> contacts;
public:
  bool etat;
};
```

Pour gérer les listes de contacts, vous devez passer par les classe QList et QQmlListProperty afin que les contacts puissent être exploités directement dans les vues. Comme pour un projet QWidget, tous les objets créés dans l'interface graphique doivent systématiquement être des objets dynamiques (utilisation de pointeur et de l'opérateur new), c'est pour cela que nous devons créer une liste de pointeur de contacts.

controleur.cpp

```
#include <QSqlDatabase>
#include <QSqlQuery>
#include <QSqlResult>
#include <controleur.h>

void Controleur::ouvertureBD()
{
  QSqlDatabase bd = QSqlDatabase::addDatabase("QSQLITE");
  bd.setDatabaseName("contacts.bd");
  bd.open();
}

void Controleur::creationTable()
{
  QSqlQuery requete;
  requete.exec("CREATE TABLE IF NOT EXISTS CONTACT"
          "(id INTEGER PRIMARY KEY AUTOINCREMENT,"
          "genre VARCHAR(2),"
          "nom VARCHAR(15),"
          "prenom VARCHAR(15),"
          "domicile VARCHAR(15),"
          "mobile VARCHAR(15))");
}

void Controleur::enregistrer(Contact *contact)
{
  QSqlQuery requete;
  requete.prepare("INSERT INTO CONTACT (genre, nom, prenom, domicile, mobile) VALUES"
                  " (:genre, :nom, :prenom, :domicile, :mobile)");
  requete.bindValue(":genre", contact->genre);
  requete.bindValue(":nom", contact->getNom());
  requete.bindValue(":prenom", contact->getPrenom());
  requete.bindValue(":domicile", contact->domicile);
  requete.bindValue(":mobile", contact->mobile);
  etat = requete.exec();
  listeContacts();
}

void Controleur::getId(Contact *contact)
{
  QSqlQuery requete;
  requete.prepare("SELECT * FROM CONTACT WHERE nom = :nom AND prenom = :prenom");
  requete.bindValue(":nom", contact->getNom());
  requete.bindValue(":prenom", contact->getPrenom());
  if (etat = requete.exec())
    if (requete.next()) {
      contact->id = requete.value("id").toInt();
      contact->genre = requete.value("genre").toString();
      contact->setNom(requete.value("nom").toString());
      contact->setPrenom(requete.value("prenom").toString());
      contact->domicile = requete.value("domicile").toString();
      contact->mobile = requete.value("mobile").toString();
    }
  else contact->id = 0;
}

void Controleur::modifier(Contact *contact)
{
  QSqlQuery requete;
  requete.prepare("UPDATE CONTACT SET "
                  "genre=:genre, nom=:nom, prenom=:prenom, domicile=:domicile, mobile=:mobile "
                  "WHERE id=:id");
  requete.bindValue(":id", contact->id);
  requete.bindValue(":genre", contact->genre);
  requete.bindValue(":nom", contact->getNom());
  requete.bindValue(":prenom", contact->getPrenom());
  requete.bindValue(":domicile", contact->domicile);
  requete.bindValue(":mobile", contact->mobile);
  etat = requete.exec();
  modificationChanged();
}

void Controleur::supprimer(int identifiant)
{
  QSqlQuery requete;
  requete.prepare("DELETE FROM CONTACT WHERE id = :id");
  requete.bindValue(":id", identifiant);
  etat = requete.exec();
  suppressionChanged();
}

void Controleur::listeContacts(const QString &critere)
{
  contacts.clear();
  QString motif("SELECT * FROM CONTACT WHERE nom LIKE '");
  motif += critere;
  motif += "%' ORDER BY nom, prenom";
  QSqlQuery requete(motif);
  while (requete.next()) {
    Contact* contact = new Contact();
    contact->id = requete.value("id").toInt();
    contact->genre = requete.value("genre").toString();
    contact->setNom(requete.value("nom").toString());
    contact->setPrenom(requete.value("prenom").toString());
    contact->domicile = requete.value("domicile").toString();
    contact->mobile = requete.value("mobile").toString();
    contacts.append(contact);
  }
  listeChanged();
}

void Controleur::choisirContact(Contact *contact, int identifiant)
{
  QSqlQuery requete;
  requete.prepare("SELECT * FROM CONTACT WHERE id = :id");
  requete.bindValue(":id", identifiant);
  if (etat = requete.exec())
    if (requete.next()) {
      contact->id = requete.value("id").toInt();
      contact->genre = requete.value("genre").toString();
      contact->setNom(requete.value("nom").toString());
      contact->setPrenom(requete.value("prenom").toString());
      contact->domicile = requete.value("domicile").toString();
      contact->mobile = requete.value("mobile").toString();
      contactChanged();
    }
```

```
  else contact->id = 0;
}
```

La grosse partie de ce code consiste à manipuler des requêtes SQL au travers de classes spécifiques qui nous permettent de simplifier le texte associé sans avoir besoin de gérer les guillemets («») dans les chaînes de caractères qui posent toujours des problèmes.

*La première classe **QSqlDataBase** s'occupe ce créer la base de données en choisissant les bons pilotes pour être en connexion avec n'importe quel serveur de bases de données. La deuxième classe **QSqlQuery** représente un bon moyen d'exécuter directement des instructions **SQL** et de gérer leurs résultats. Avant d'exécuter des requêtes **SQL**, nous devons tout d'abord établir une connexion avec une base de données à l'aide de la classe **QSqlDataBase**.*

*Méthodes de la classe **QSqlDataBase** :*

- ***addDataBase()** : Cette méthode statique permet de créer l'objet représentant le serveur de la base de données utilisée.*
- ***setDataBaseName()** : Spécifie le nom de cette base de données à créer où à atteindre.*
- ***open()** : Nous établissons la connexion effective à l'aide de cette méthode qui renvoie une valeur booléenne précisant si l'opération d'ouverture s'est bien déroulée ou non.*

*Méthodes de la classe **QSqlQuery** :*

- ***QSqlQuery(requête)** : Exécute immédiatement la requête exprimée en argument du constructeur, dès la phase de création de l'objet.*
- ***exec(requête)** : Exécute la requête exprimée en argument de la méthode après que l'objet ait été créé.*
- ***next()** : Cette méthode nous permet de parcourir l'ensemble des enregistrements issus de la requête demandée.*
- ***prepare(requête)** : Exécute une requête contenant des emplacements réservés liés par des valeurs qui seront insérées par la suite.*
- ***bindValue(clé, valeur)**: Après avoir spécifié la requête préparée à l'aide de la méthode **prepare()** dans laquelle vous devez désigner les variables à prendre en compte, à l'aide du préfixe « : », vous les enregistrez ensuite successivement au travers de cette méthode. Lorsque toutes les variables sont correctement renseignées, il suffit de faire appel ensuite à la méthode **exec()** sans paramètre.*
- ***value()** : Cette méthode retourne la valeur d'un champ en tant que **QVariant**. La classe **QVariant** peut contenir de nombreux types **Qt** et **C++**, dont **int** et **QString**. Les différents types de données susceptibles d'être stockés dans une base de données sont transformés en types **Qt** et **C++** correspondants et stockés en tant que **QVariant**. Par exemple, un **VARCHAR** est représenté en tant que **QString** et un **DATETIME** en tant que **QDateTime**.*
- ***numRowsAffected()** : Permet de connaître le nombre de lignes affectées par la requête **SQL**.*

Au démarrage de l'application, il faut créer la base de données si elle n'existe pas ou l'ouvrir tout simplement. Cela se fait automatiquement grâce à la méthode setDataBaseName() de la classe QSqlDataBase. Dans le même ordre d'idée, la première démarche consiste à créer la table (là aussi si elle n'existe pas).

main.cpp

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QtQml>
#include <controleur.h>
#include <contact.h>

int main(int argc, char *argv[])
{
  QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
  QGuiApplication app(argc, argv);

  qmlRegisterType<Contact>("Repertoire", 1, 0, "Contact");
  Controleur controleur;

  QQmlApplicationEngine engine;
  engine.rootContext()->setContextProperty("controleur", &controleur);
  engine.load(QUrl(QStringLiteral("qrc:/vue/main.qml")));
  if (engine.rootObjects().isEmpty()) return -1;

  return app.exec();
}
```

L'objet correspondant au contrôleur, comme pour tous les projets QML, doit être créer dès le départ pour être exploité directement par l'ensemble des vues. Par contre, pour l'entité, il peut y avoir un nombre quelconque d'objets, un seul pour la vue « Edition » et toute une liste dans la vue « Liste ». Il n'est donc pas possible de savoir combien nous en avons besoin et surtout le nombre varie suivant le cours d'utilisation de l'application.

Il est donc nécessaire que les vues connaissent la classe Contact représentant l'entité grâce à la fonction qmlRegisterType<Contact>() avec pour paramètres : le nom donné pour l'importation dans la vue, avec son numéro de version et enfin le nom du composant tel qu'il sera utilisé par la vue (il est plutôt judicieux de choisir le même nom que la classe elle-même).

Alerte.qml

```
import QtQuick 2.0

Rectangle {
  id: fond
  property alias texte: alerte.text

  anchors.fill: parent
  anchors.margins: 5
  radius: 5
  state: "nonactif"

  Text {
    id: alerte
    anchors.centerIn: parent
  }

  states: [
    State {
      name: "nonactif"
      PropertyChanges {
        target: fond
        visible: false
      }
    },
    State {
      name: "valide"
      PropertyChanges {
        target: fond
        visible: true
        color: "lightgreen"
      }
      PropertyChanges {
        target: alerte
        color: "darkgreen"
      }
    },
```

```
    State {
      name: "nonvalide"

      PropertyChanges {
        target: fond
        visible: true
        color: "red"
      }

      PropertyChanges {
        target: alerte
        color: "darkred"
      }
    }
  ]
}
```

Ce composant nous permet de visualiser temporairement un message surgissant qui nous indique si les opérations d'enregistrement de contact, de modification et de suppression se sont passées correctement ou pas avec un choix des couleurs adapté suivant le résultat.

Il est possible de proposer de nouvelles propriétés qui sont de simples alias qui permettent d'avoir un code plus sobre lors de leurs utilisations.

Bouton.qml

```
import QtQuick 2.0
import QtQuick.Controls 2.2

ToolButton {
  id: bouton
  property alias icone: bouton.icon.source
  property int taille: 38

  icon {
    color: "transparent"
    width: taille
    height: taille
  }
  highlighted: true
}
```

Création d'un nouveau composant Bouton utilisé dans la feuille d'édition pour effacer tous les champs, pour enregistrer, pour modifier et pour supprimer un contact, dans la vue « Edition », mais également pour l'ensemble des composants utilisant des icônes.

Choix.qml

```
import QtQuick 2.0
import QtQuick.Controls 2.2

TabButton {
  id: choix
  property alias icone: choix.icon.source
  background: Rectangle {
    color: parent.checked ? "white" : "#20FFC0CB"
    border.color: parent.checked ? "pink" : "transparent"
  }
  icon.color: "transparent"
}
```

Nouveau composant Choix utilisé pour les onglets visibles sur la partie basse de l'application principale de l'application.

Genre.qml

```
import QtQuick 2.0
import QtQuick.Controls 2.2

Item {
  property alias checked: choix.checked
  height: 55
  anchors {
    left: parent.left
    right: parent.right
  }

  Row {
    Bouton {
      id: bouton
      icone: "qrc:/icone/homme.png"
    }

    Switch {
      id: choix
      onCheckedChanged: bouton.icone = checked ? "qrc:/icone/femme.png" : "qrc:/icone/homme.png"
    }
  }
}
```

Nouveau composant Genre qui permet de choisir entre un homme ou une femme dans la vue « Edition ».

Saisie.qml

```
import QtQuick 2.9
import QtQuick.Controls 2.2

TextField {
  font.pointSize: 14
  placeholderText: "Choix"
  leftPadding: 20

  anchors {
    left: parent.left
    right: parent.right
  }

  background: Rectangle {
    anchors {
      fill: parent
      bottomMargin: 5
    }

    border.color: "pink"
    color: "#10FFC0CB"
    radius: 7
  }
}
```

Nouveau composant Saisie qui permet de saisir le nom et le prénom du contact dans la vue « Edition ».

Numero.qml

```
import QtQuick 2.9
import QtQuick.Controls 2.2

Rectangle {
  property alias texte: numero.text
  property alias icone: bouton.icone
  height: 25
  color: "#10FFC0CB"
  anchors {
    left: parent.left
    right: parent.right
  }

  Bouton {
    id: bouton
    width: 30
    height: 30
    anchors {
      left: parent.left
      top: parent.top
    }
  }

  Text {
    id: numero
    anchors {
      left: bouton.right
      top: bouton.top
      topMargin: 5
    }
  }
}
```

Nouveau composant Numero qui permet de visualiser le numéro de téléphone du contact avec le choix de l'icône adaptée suivant le type de téléphone dans la vue « Liste ».

Recherche.qml

```
import QtQuick 2.9
import QtQuick.Controls 2.2

TextField {
  id: recherche
  property alias texte: recherche.text
  property alias icone: bouton.icone
  property alias rayon: fond.radius
  property alias couleur: fond.color

  placeholderText: "Recherche de contacts"
  font.pointSize: 14
  leftPadding: 60
  height: 50
  anchors {
    left: parent.left
    right: parent.right
  }

  background: Rectangle {
    id: fond
    anchors {
      fill: parent
      bottomMargin: 5
    }
    border.color: "pink"
    color: "#FFC0CB"
  }

  Bouton {
    id: bouton
    icone: "qrc:/icone/recherche.png"
    taille: 32
  }
}
```

Nouveau composant Recherche qui permet de cibler les contacts dans la vue « Liste ».

Telephone.qml

```
import QtQuick 2.9
import QtQuick.Controls 2.2

Recherche {
  id: telephone
  property alias numero: telephone.text
  inputMask: "99-99-99-99-99"
  inputMethodHints: Qt.ImhDigitsOnly
  rayon: 7
  couleur: "#30FFC0CB"
}
```

Nouveau composant Telephone qui hérite du composant Recherche qui rajoute un masque de saisie lors de l'ajout du numéro de téléphone du contact dans la vue « Edition ».

Feuille.qml

```
import QtQuick 2.9
import QtQuick.Controls 2.2

Page {
  property alias titre: titre.text
  property alias contenu: contenu.sourceComponent

  width: 320
  height: 400
  header: Label {
    id: titre
    text: qsTr("Titre")
    font.pixelSize: 20
    padding: 10
    color: "#990000"
    background: Rectangle { color: "#40FFC0CB" }
    style: Text.Sunken
  }

  Item {
    id: zone
    anchors.fill: parent
```

```
    Loader {
      id: contenu
      anchors {
        fill: parent
        margins: 20
      }
    }
  }
}
```

La particularité ici, c'est que nous créons une page générique Feuille (modèle de page) qui représente une des pages associées au « glisser-déposer » de l'application principale. L'objectif est de réaliser tous les réglages communs aux vues « Edition » et « Liste ».

Dans les pages définitives qui se servent de ce modèle, vous rajouterez toutes les particularités de la page dans une zone prévue à cet effet (il peut y en avoir autant que vous voulez).

Comme pour tout modèle de page, vous avez donc une partie fixe et une partie variable dans laquelle vous pouvez rajouter tout ce qui vous intéresse. Cette partie variable est délimité par le composant Loader (chargement).

Pour que cela fonctionne sans problème, vous devez identifier obligatoirement cette zone variable et prévoir un alias avec un nom à votre convenance, sachant qu'il existe une propriété fondamentale dans ce composant Loader qui s'appelle sourceComponant.(sources spécifiques de votre page réelle).

Liste.qml


```
import QtQuick 2.11
import QtQuick.Controls 2.2
import QtQuick.Layouts 1.3
import Repertoire 1.0 // Très important

Feuille {
  id: contacts
  titre: qsTr("Tous les contacts")
  signal contactSelected(int identifiant)

  onActiveFocusChanged: recherche.text = ""

  Recherche {
    id: recherche
    onTextChanged: controleur.listeContacts(text)
  }

  contenu: ScrollView {
    anchors {
      fill: parent
      topMargin: 50
    }

    ListView {
      spacing: 10
      width: parent.width
      model: controleur.liste

      delegate: Rectangle {
        width: parent.width
        height: 80
        radius: 7
        color: "#20FFC0CB"
        border.color: "pink"

        MouseArea {
          anchors.fill: parent
          onClicked: contacts.contactSelected(modelData.identifiant)
        }

        Column {
          anchors {
            fill: parent
            margins: 5
          }
          Text {
            text: modelData.identite
            font {
              bold: true
              pixelSize: 16
            }
            color: "maroon"
          }

          Numero {
            texte: modelData.domicile
            icone: "qrc:/icone/telephone.png"
          }

          Numero {
            texte: modelData.mobile
            icone: "qrc:/icone/mobile.png"
          }
        }

        Bouton {
          taille: 48
          icone: modelData.genre == "H" ? "qrc:/icone/homme.png" : "qrc:/icone/femme.png"
          anchors {
            right: parent.right
            rightMargin: 10
            top: parent.top
            topMargin: 10
          }
        }
      }
    }
  }
}
```

Nous retrouvons la première page principale de l'application, la vue correspondant à la liste des contacts. Pour afficher une liste d'éléments, il existe le composant ListView prévu à cet effet.

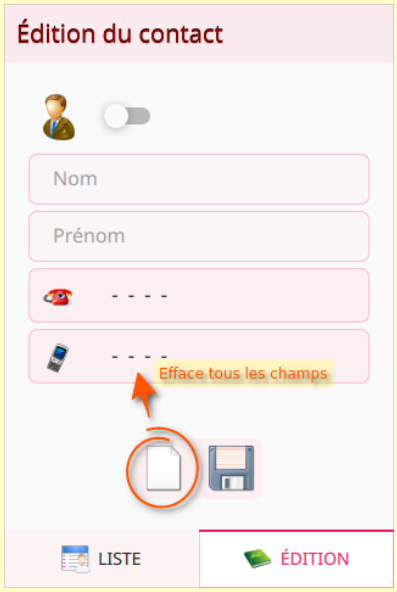
Ce composant possède une propriété importante nommée model (modèle). C'est grâce à cette propriété que vous mettez en relation la liste (de type QQmlListProperty<Contact>) générée dans le contrôleur.

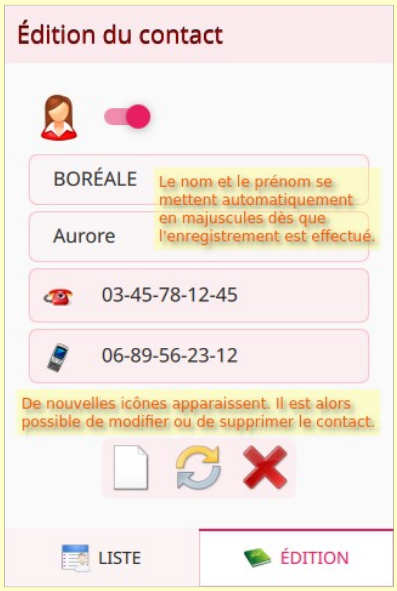
Ce composant possède également une deuxième propriété importante nommée delegate. Cette propriété permet de récupérer chaque élément séparé de la liste, donc ici chaque contact individuel. Pour cela, il existe un terme générique « modelData » (donnée récupérée du modèle) représentant cet objet individuel (l'objet modelData est donc ici un objet de type Contact).

Au niveau des vues, il est également possible de créer de nouveau signaux grâce à la commande « signal ». Ici, le signal « contactSelected » est envoyé lorsque nous cliquons sur un des contacts. L'objectif est de changer de vue et de passer en mode d'Édition pour éventuellement supprimer ou changer les caractéristiques du contact sélectionné.

Edition.qml

```
import QtQuick 2.11
import QtQuick.Controls 2.2
import QtQuick.Layouts 1.3
import Repertoire 1.0  // Très important

Feuille {
  titre: qsTr("Édition du contact")
  property alias nouveaucontact: contact

  Contact { id: contact }

  contenu: Column {
    signal raz()
    signal sauvegarde()
    signal enregistrement()
    signal modification()

    onRaz: {
      nom.text = ""
      prenom.text = ""
      domicile.text = ""
      mobile.text = ""
      contact.identifiant = 0
      enregistrer.visible = true
      modifier.visible = false
      supprimer.visible = false
    }

    onSauvegarde: {
      contact.genre = civilite.checked ? "F" : "H"
      contact.nom = nom.text
      contact.prenom = prenom.text
      contact.domicile = domicile.numero
      contact.mobile = mobile.numero
      nom.text = contact.nom
      prenom.text = contact.prenom
    }

    onEnregistrement: {
      controleur.enregistrer(contact)
      controleur.getId(contact)
      enregistrer.visible = false
      modifier.visible = true
      supprimer.visible = true
    }

    onModification: controleur.modifier(contact)

    Connections {
      target: controleur

      onEnregistrementChanged: {
        attente.start()
        alerte.texte = controleur.etat ? "Contact enregistré" : "Problème d'enregistrement"
        alerte.state = controleur.etat ? "valide" : "nonvalide"
      }

      onModificationChanged: {
        attente.start()
        alerte.texte = controleur.etat ? "Contact modifié" : "Modification non effectuée"
        alerte.state = controleur.etat ? "valide" : "nonvalide"
      }

      onSuppressionChanged: {
        attente.start()
        alerte.texte = controleur.etat ? "Contact supprimé" : "Suppression non effectuée"
        alerte.state = controleur.etat ? "valide" : "nonvalide"
      }

      onContactChanged: {
        civilite.checked = contact.genre == "F"
        nom.text = contact.nom
        prenom.text = contact.prenom
        domicile.text = contact.domicile
        mobile.text = contact.mobile
        enregistrer.visible = false
        modifier.visible = true
        supprimer.visible = true
      }
    }

    Genre { id: civilite }
    Saisie { id: nom ; placeholderText: "Nom" }
    Saisie { id: prenom ; placeholderText: "Prénom" }
    Telephone { id: domicile ; icone: "qrc:/icone/telephone.png" }
    Telephone { id: mobile ; icone: "qrc:/icone/mobile.png" }

    Rectangle {
      width: parent.width
      height: 40
      color: "transparent"
      Alerte { id: alerte }
    }

    Timer {
      id: attente
      interval: 2000
      onTriggered: alerte.state = "nonactif"
    }

    ToolBar {
      anchors.horizontalCenter: parent.horizontalCenter

      background: Rectangle {
        color: "#30FFC0CB"
        radius: 7
      }

      RowLayout {
        anchors.fill: parent

        Bouton {
          id: effacer
          icone: "qrc:/icone/nouveau.png"
          onClicked: clicked.connect(raz)
        }

        Bouton {
          id: enregistrer
          icone: "qrc:/icone/enregistrer.png"
          onClicked: {
            clicked.connect(sauvegarde)
            clicked.connect(enregistrement)
```





```
        }
      }

      Bouton {
        id: modifier
        visible: false
        icone: "qrc:/icone/synchronise.png"
        onClicked: {
          clicked.connect(sauvegarde)
          clicked.connect(modification)
        }
      }

      Bouton {
        id: supprimer
        visible: false
        icone: "qrc:/icone/supprimer.png"
        onClicked: {
          controleur.supprimer(contact.identifiant)
          clicked.connect(raz)
        }
      }
    }
  }
}
```

il s'agit de la deuxième vue principale de l'application. Nous retrouvons tous les composants ainsi que le modèle de feuille que nous venons de créer. Cette façon de procéder est vraiment importante. Elle évite d'avoir énormément de lignes de code pour une même page. Ainsi, si un défaut se présente, il suffit d'aller directement sur le composant concerné.

Bien entendu, nous retrouvons le contrôleur avec lequel nous travaillons pour réaliser toutes les opérations de base pour la gestion avec une base de données, savoir : enregistrer, modifier ou supprimer un contact.

À ce sujet, nous créons directement dans cette vue un objet de type Contact (remarquez l'importation « Repertoire ») dont le nom de l'objet est ici « contact ». c'est grâce à cet unique objet que nous réalisons toutes les opérations nécessaires pour le bon fonctionnement de cette partie de l'application (mode Édition).

Une autre solution très intéressante, c'est qu'il est possible de connecter des signaux entre eux, directement dans la vue, comme nous le faisons éventuellement dans le codage C++. Pour cela, vous utilisez la méthode connect() de l'objet clicked dans l'événement initiateur en spécifiant en paramètre le deuxième signal. L'intérêt ici, c'est de factoriser un ensemble d'actions qui peut être déclenché par plusieurs événement différents.

main.qml

```
import QtQuick 2.9
import QtQuick.Controls 2.2

ApplicationWindow {
  visible: true
  width: 360
  height: 580
  title: qsTr("Contacts")

  SwipeView {
    id: faireGlisser
    anchors.fill: parent
    currentIndex: tabBar.currentIndex

    Liste {
      onContactSelected: {
        choixEdition.toggle()
        controleur.choisirContact(editionContact.nouveaucontact, identifiant)
      }
    }

    Edition { id: editionContact }
  }

  footer: TabBar {
    id: tabBar
    currentIndex: faireGlisser.currentIndex

    Choix {
      id: choixListe
      text: qsTr("Liste")
      icone: "qrc:/icone/liste-contact.png"
      onToggled: controleur.listeContacts()
    }

    Choix {
      id: choixEdition
      text: qsTr("Édition")
      icone: "qrc:/icone/livre.png"
    }
  }
}
```

Nous retrouvons enfin notre application principale (vue principale) qui possède très peu de lignes de code puisque tout est écrit dans les vues antérieures.

Nous voyons ici la prise en compte du signal que nous avons créé dans la vue précédente « onContactSelected ». C'est ici que nous basculons sur la vue « Édition » et que nous récupérons toutes les coordonnées du contact sélectionné.