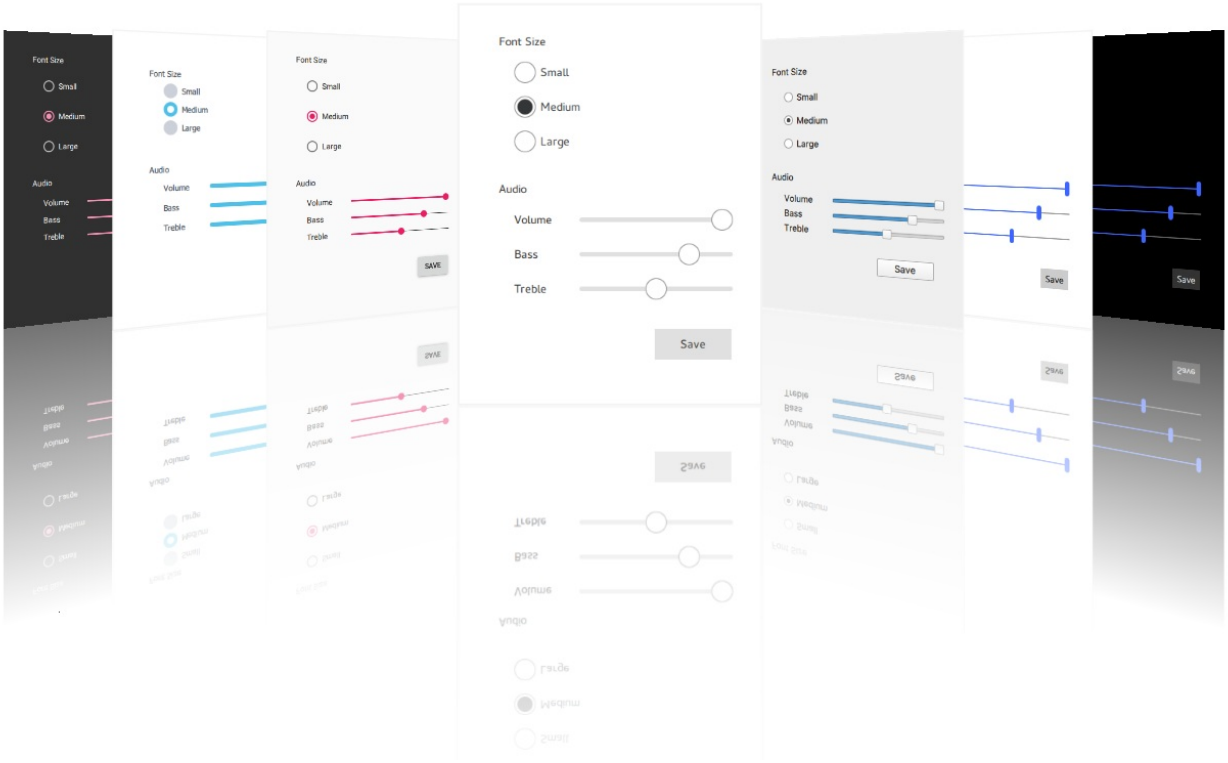
Qt pour Android : Application avec Qt Quick Controls

> Version PDF du document http://tvaira.free.fr/dev/qt-android/qt-android-controls.html
> Thierry Vaira <tvaira@free.fr>
> 2019 (rev. 1.0)

Le module Qt Quick Controls fournit un ensemble de composants pour créer des interfaces très complètes :

- Application fenêtrée : ApplicationWindow, MenuBar, StatusBar, ToolBar, Action, Page, …
- Navigation et vues : ScrollView, SplitView, StackView, TabView, TableView, TreeView, SwipeView, …
- Contrôles : Button, CheckBox, ComboBox, Label, ProgressBar, Slider, SpinBox, BusyIndicator, …
- Menus : Menu, MenuItem et MenuSeparator
- …



Liens : Qt Quick Controls 2 et Qt Quick Controls 1 + Differences between Qt Quick Controls 1

Les types QML de Qt Quick Controls 2 seront importés dans l'application en ajoutant dans le fichier `.qml` :

```
import QtQuick 2.9
import QtQuick.Controls 2.1

...
```

Pour créer un lien avec les bibliothèques C++ correspondantes, il faudra ajouter dans le fichier de projet `.pro` :

```
QT += quick quickcontrols2
```

Objectifs

Réaliser une application type à partir de **Qt pour Android** en utilisant le module **Qt Quick Controls**.

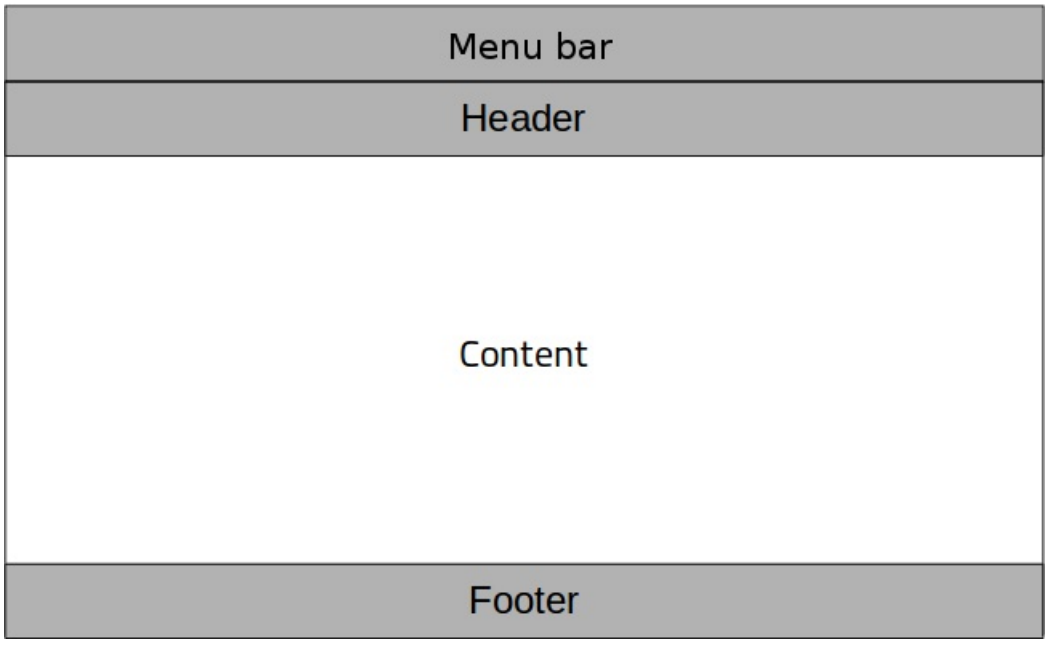
Application fenêtrée

Composants utilisés pour décrire les propriétés de base de la fenêtre d'une application :

- ApplicationWindow : fenêtre d'application
- MenuBar : barre de menu horizontale
- StatusBar : barre d'état (seulement en Qt Quick Controls 1)
- ToolBar : barre d'outils contenant des ToolButton
- Drawer : panneau latéral (tiroir) par balayage
- Action : Action d'interface utilisateur abstraite pouvant être liée à des éléments
- Page : un conteneur possédant une en-tête, un pied de page et un contenu
- …

Liens : ApplicationWindow QML Type

`ApplicationWindow` est une fenêtre qui facilite l'ajout d'une barre de menu , d'un en- tête et d'un pied de page.



On peut évidemment déclarer `ApplicationWindow` en tant qu'élément racine de l'application.

Une fenêtre

Dans le fichier `.qml` , il suffit de définir un élément `ApplicationWindow` et de fixer sa propriété `visible` à `true` . On peut définir un titre et les dimensions (`width` et `height`) de la fenêtre même si

c'est inutile pour Android :

```
import QtQuick 2.9
import QtQuick.Controls 2.1

ApplicationWindow {
    title: qsTr("Mon Application")
    //width: Screen.desktopAvailableWidth
    //height: Screen.desktopAvailableHeight
    visible: true
}
```

Lien : ApplicationWindow QML Type

Une barre d'outils et un panneau latéral

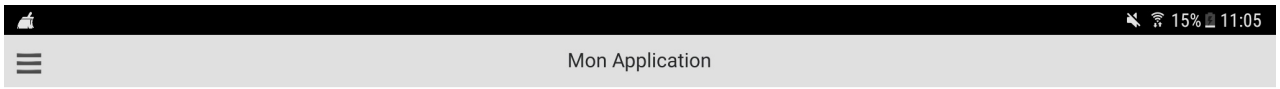
On peut ensuite ajouter une `ToolBar` associée à un panneau latéral `Drawer` dans lequel on intègrera des `ItemDelegate` pour les choix :

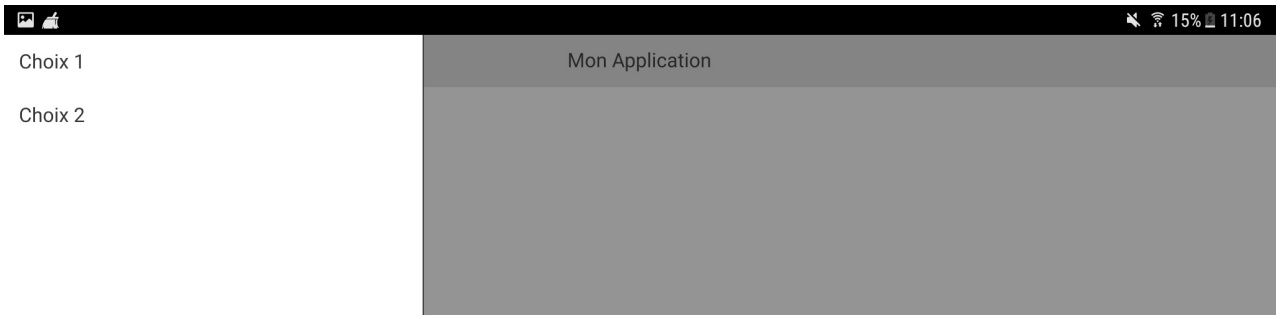
```
import QtQuick 2.9
import QtQuick.Controls 2.1

ApplicationWindow {
    id: window
    title: qsTr("Mon Application")
    visible: true
    header: ToolBar {
        contentHeight: toolButton1.implicitHeight // s'ajuste à la taille du ToolButton
        Label {
            text: qsTr("Mon Application")
            anchors.centerIn: parent
        }
        ToolButton {
            id: toolButton1
            text: "\u2630" // symbole représentant le panneau
            font.pixelSize: Qt.application.font.pixelSize * 1.6
            onClicked: {
                console.log("onClicked " + toolButton1.text)
                panneau.open() // on ouvre le tiroir
            }
        }
    }
    Drawer {
    id: panneau
    width: window.width * 0.33 // occupera 33% de la largeur
    height: window.height // et sur toute la hauteur
        Column {
        anchors.fill: parent
        ItemDelegate {
            id: choix1
            text: qsTr("Choix 1")
            width: parent.width // toute la largeur du tiroir
            onClicked: {
                console.log("onClicked " + choix1.text)
                panneau.close() // et on referme le tiroir
            }
        }
```

```
        ItemDelegate {
            id: choix2
            text: qsTr("Choix 2")
            width: parent.width
            onClicked: {
                console.log("onClicked " + choix2.text)
                panneau.close() // et on referme le tiroir
            }
        }
        }
    }
}
```

Liens : ToolBar QML Type, ToolButton QML Type et Drawer QML Type

On obtient :





Un menu

Éléments utilisés pour créer des menus :

- Menu : Fournit un composant de menu à utiliser comme menu contextuel ou dans le cadre d'une barre de menus
- MenuItem : Élément à ajouter dans un menu ou une barre de menus
- MenuSeparator : Séparateur pour les éléments d'un menu

On va maintenant ajouter un `Menu` dans la barre d'outils précédente. Un menu contient des entrées de menu `MenuItem`. Lorsque l'on sélectionne une entrée de menu, le signal `onTriggered` sera déclenché.

On se retrouve donc avec : un panneau latéral `Drawer` à gauche, un `Label` texte au centre et un `Menu` à droite. Pour organiser ces trois éléments, on va les placer "en ligne" dans un `RowLayout` :

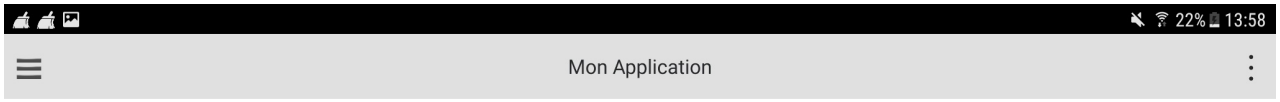
```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtQuick.Controls 2.1
```

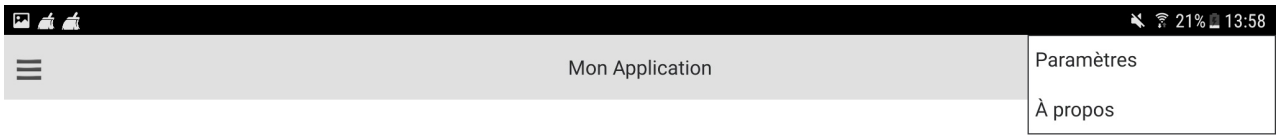
```
import QtQuick.Layouts 1.3

ApplicationWindow {
    id: window
    title: qsTr("Mon Application")
    visible: true
    header: ToolBar {
        RowLayout {
            spacing: 20
            anchors.fill: parent
            ToolButton {
                id: toolButton1
                text: "\u2630" // symbole représentant le panneau ou "\u2261"
                font.pixelSize: Qt.application.font.pixelSize * 1.6
                onClicked: {
                    console.log("onClicked " + toolButton1.text)
                    panneau.open()
                }
            }
            Label {
                text: qsTr("Mon Application")
                horizontalAlignment: Qt.AlignHCenter
                verticalAlignment: Qt.AlignVCenter
                Layout.fillWidth: true
            }
            ToolButton {
                id: toolButton2
                text: "\u22ee" // symbole représentant le menu options ou qsTr("⋮")
                onClicked: menu.open()
            }
        }
    }
    Drawer {
        ...
    }
    Menu {
        id: menu
        x: parent.width - width
        transformOrigin: Menu.TopRight
        MenuItem {
            id: parametres
            text: "Paramètres"
            onTriggered: {
                console.log("onTriggered " + parametres.text)
            }
        }
        MenuItem {
            id: about
            text: "À propos"
            onTriggered: {
                console.log("onTriggered " + about.text)
            }
        }
    }
}
```

Liens : Menu QML Type et MenuItem QML Type

Remarque : L'élément `ToolButton` accepte une icône avec sa propriété `icon`.

On obtient :

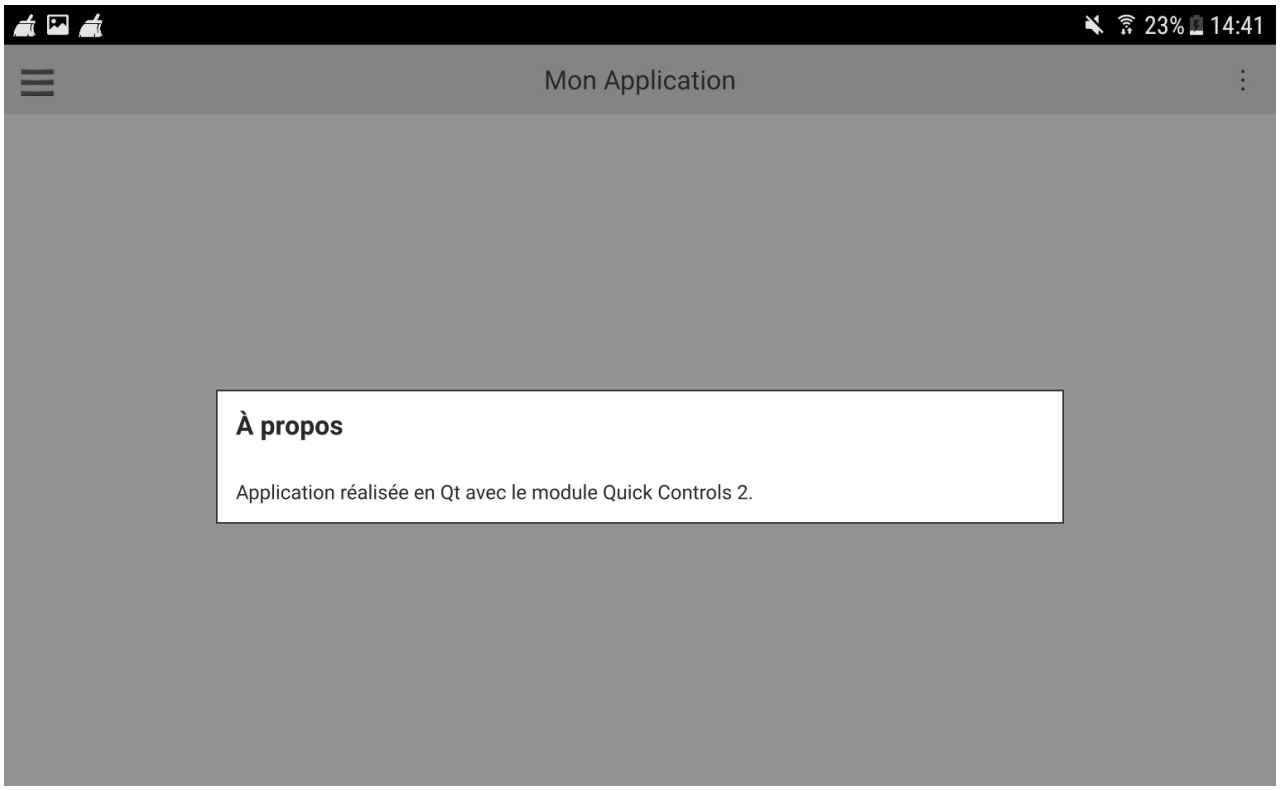




On peut associer l'affichage d'une boîte de dialogue lors de l'action sur une entrée de menu.

Exemple de boîte de dialogue pour "À propos" :

```
    ...
    MenuItem {
            id: about
            text: "À propos"
            onTriggered: {
                aPropos.open()
            }
        }
        ...

Dialog {
        id: aPropos
        modal: true
        focus: true
        title: "À propos"
        x: (window.width - width) / 2
        y: window.height / 6
        width: Math.min(window.width, window.height) / 3 * 2
        contentHeight: message.height
        Label {
            id: message
            width: aPropos.availableWidth
            text: "Application réalisée en Qt avec le module Quick Controls 2."
            wrapMode: Label.Wrap
            font.pixelSize: 12
        }
    }
```



Liens : Qt Quick Dialogs et Dialog QML Type

Un pied de page

Il est possible d'ajouter un élément dans le pied de page de l'application en utilisant la propriété `footer`, ici on va placer un simple `Label` :

```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtQuick.Controls 2.1
import QtQuick.Layouts 1.3

ApplicationWindow {
    ...
    footer: Label {
            width: parent.width
            horizontalAlignment: Qt.AlignRight
            padding: 10
            text: qsTr("© http://tvaira.free.fr")
            font.pixelSize: 14
            font.italic: true
    }
}
```

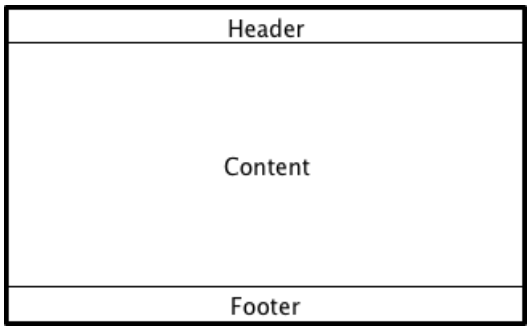
Un contenu

Il est facile d'ajouter un élément qui sera affiché dans l'espace contenu de la fenêtre :

```
import QtQuick 2.9
import QtQuick.Window 2.2
```

```
import QtQuick.Controls 2.1
import QtQuick.Layouts 1.3

ApplicationWindow {
    ...
    Item {
        id: contenu
        anchors.centerIn: parent
        Label {
                text: qsTr("Un contenu ...")
        }
    }
}
```

Evidemment, il sera plus judicieux d'utiliser des vues par Qt Quick Controls pour organiser le contenu des pages. QML propose aussi un type `Page` qui est un conteneur possédant une en-tête, un pied de page et un contenu. Par souci d'organisation, on placera ses éléments `Page` dans des fichiers `.qml` séparés.




```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtQuick.Controls 2.1
import QtQuick.Layouts 1.3

ApplicationWindow {
    ...
    Page {
        id: contenu
        title: qsTr("Ma Page")
        anchors.centerIn: parent
        Label {
                text: qsTr("Un contenu ...")
        }
    }
}
```

Lien : Page QML Type

Style

Qt Quick Controls 2 est livré avec une sélection de styles : "Default", "Fusion", "Imagine", "Material", "Universal".

On peut utiliser par exemple les styles :

- **Material** : basé sur les spécifications Google Material Design Guidelines
- **Universal** : basé sur les spécifications Microsoft Universal Design Guidelines

Il y a plusieurs moyens pour exécuter une application avec un style spécifique. Les deux plus simples sont :

- en appelant `QQuickStyle::setStyle()` dans le code C++ avant le chargement du fichier `.qml`
- en utilisant le fichier de configuration spécial `:/qtquickcontrols2.conf` à intégrer aux ressources de l'application.

Lien : Styling Qt Quick Controls 2

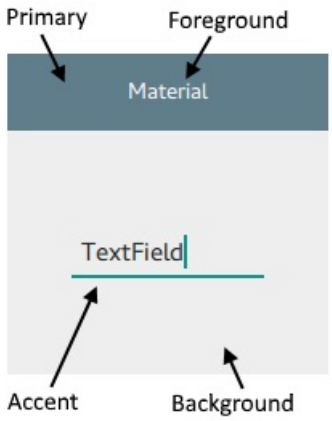
- Exemple de fichier `main.cpp` :

```
#include <QGuiApplication>
#include <QQuickStyle>
#include <QQmlApplicationEngine>
#include <QDebug>

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

    qDebug() << QQuickStyle::availableStyles(); // -> ("Default", "Fusion", "Imagine", "Material"
, "Universal")

    QQuickStyle::setStyle("Material");

    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
    if (engine.rootObjects().isEmpty())
        return -1;

    return app.exec();
}
```

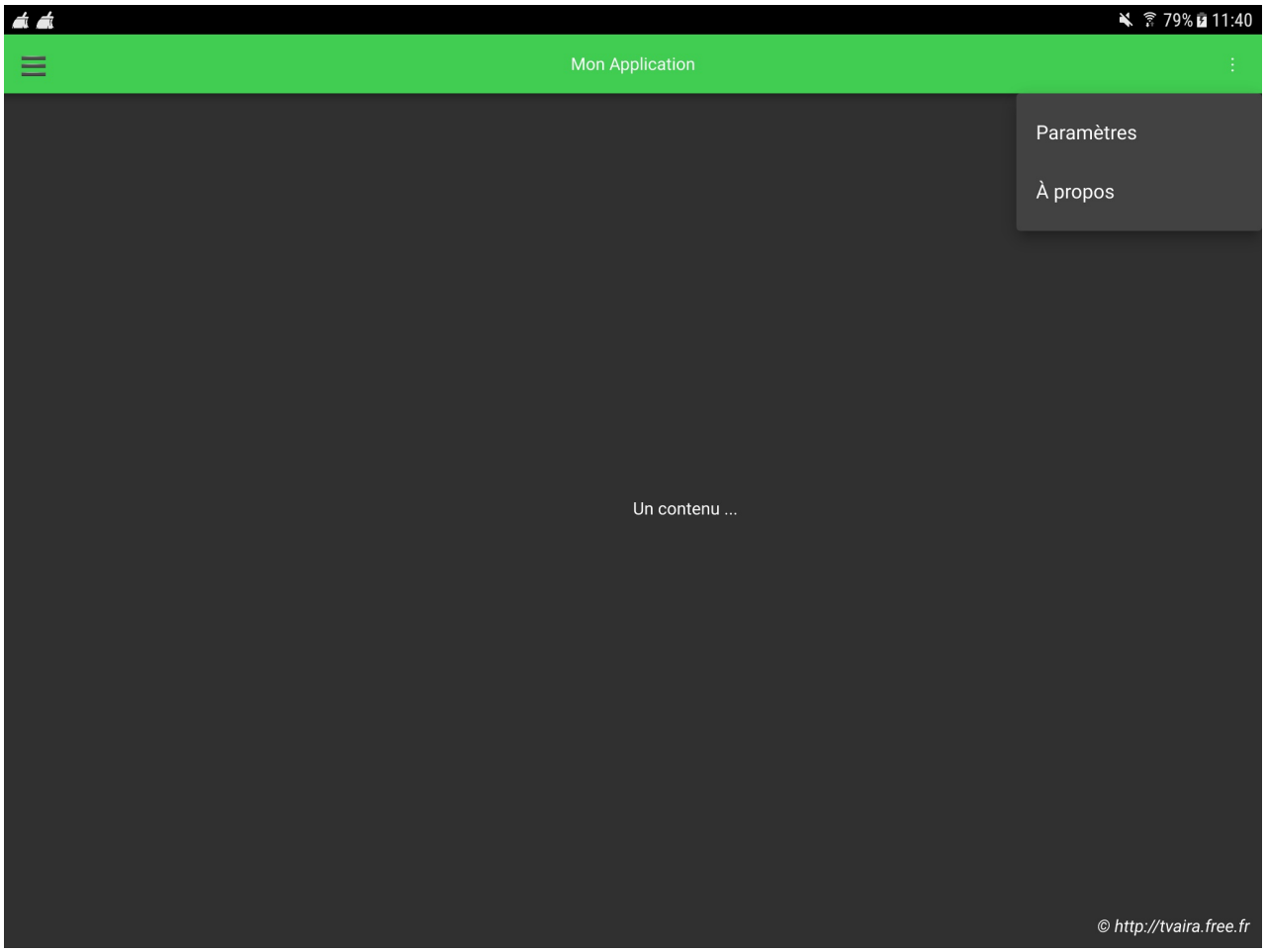
- Exemple de fichier `:/qtquickcontrols2.conf` :

```
[Controls]
Style=Material

[Material]
Primary=#41cd52
Accent=#41cd52
Theme=System
```

On peut aussi personnaliser basiquement le style choisi en modifiant quelques attributs directement dans le fichier `.qml`, par exemple pour le style Material :




```
...
import QtQuick.Controls.Material 2.12

ApplicationWindow {
    visible: true
    Material.theme: Material.Dark
    Material.primary: "#41cd52"

    ...
}
```



Navigation et vues

Les vues permettent généralement à l'utilisateur de gérer ou de présenter d'autres composants dans une mise en page. Certaines vues sont des modèles pour la navigation.

Qt Quick Controls 1 :

- SplitView : Dispose des éléments avec un séparateur déplaçable entre chaque élément
- TabView : Un contrôle qui permet à l'utilisateur de sélectionner l'un des multiples éléments empilés
- TableView : Fournit une vue de liste avec des barres de défilement, des styles et des sections d'en-tête
- TreeView : Fournit une arborescence avec des barres de défilement, des styles et des sections d'en-tête

Qt Quick Controls 1 & 2 :

- ScrollView : Fournit une vue défilante dans un autre élément (avec Qt Quick Controls 2.2)
- StackView : Fournit un modèle de navigation basé sur une pile

Et seulement en Qt Quick Controls 2 :

- SwipeView : Fournit un modèle de navigation par balayage

Liens : Navigation Controls

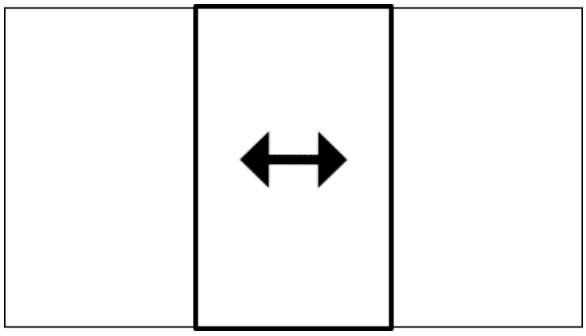
Pour organiser la navigation entre contenus, deux modèles sont intéressants :

- SwipeView : pour une navigation par balayage
- StackView : pour une navigation basée sur une pile

On peut ajouter aussi une organisation par onglets avec TabBar (ou TabView pour la version 1).

SwipeView et TabBar

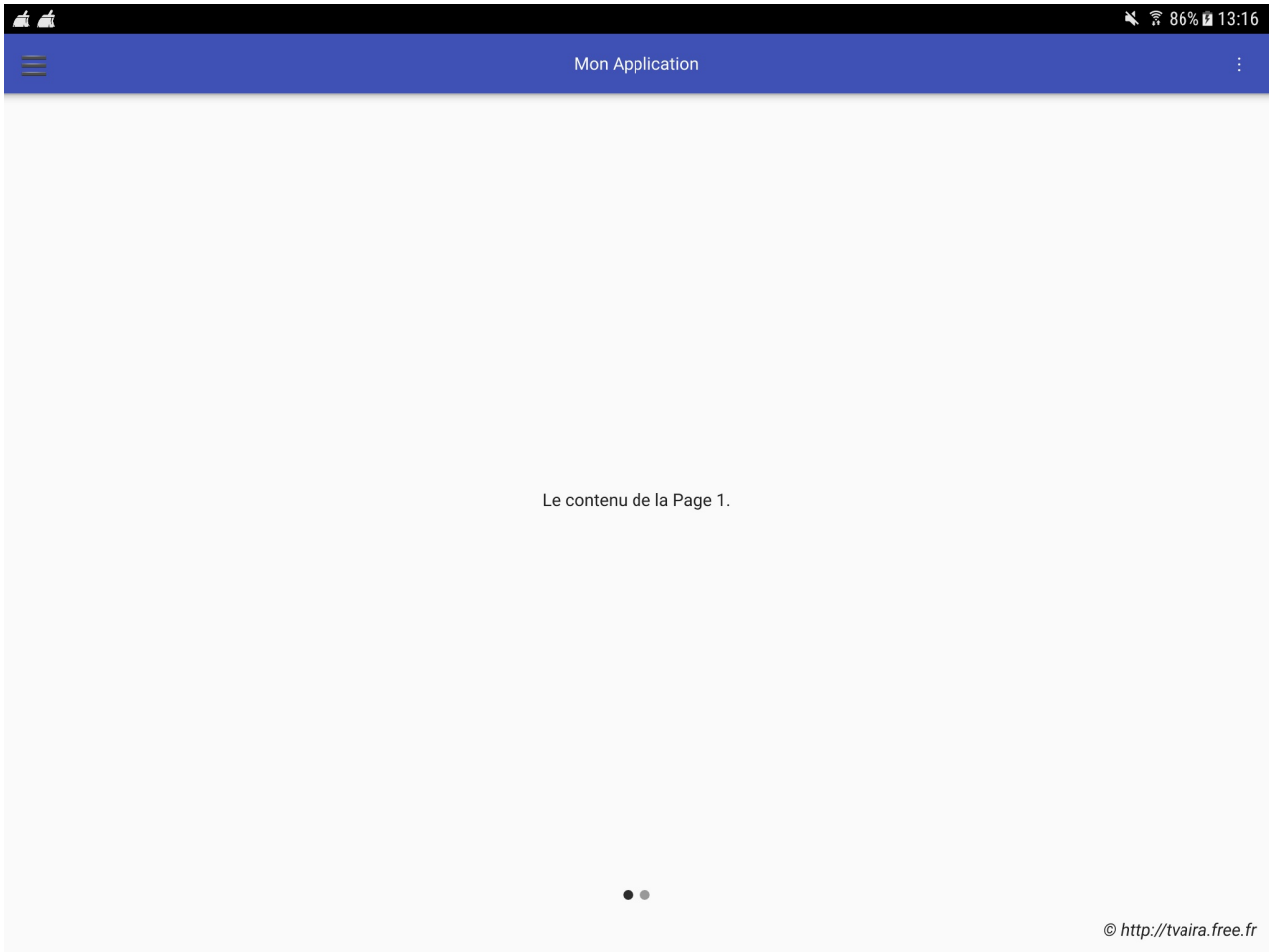
Un `SwipeView` permet à l'utilisateur de naviguer dans les pages en effectuant un balayage latéral.



Il est recommandé de le combiner avec `PageIndicator` pour donner à l'utilisateur un indice visuel de la présence de plusieurs pages.

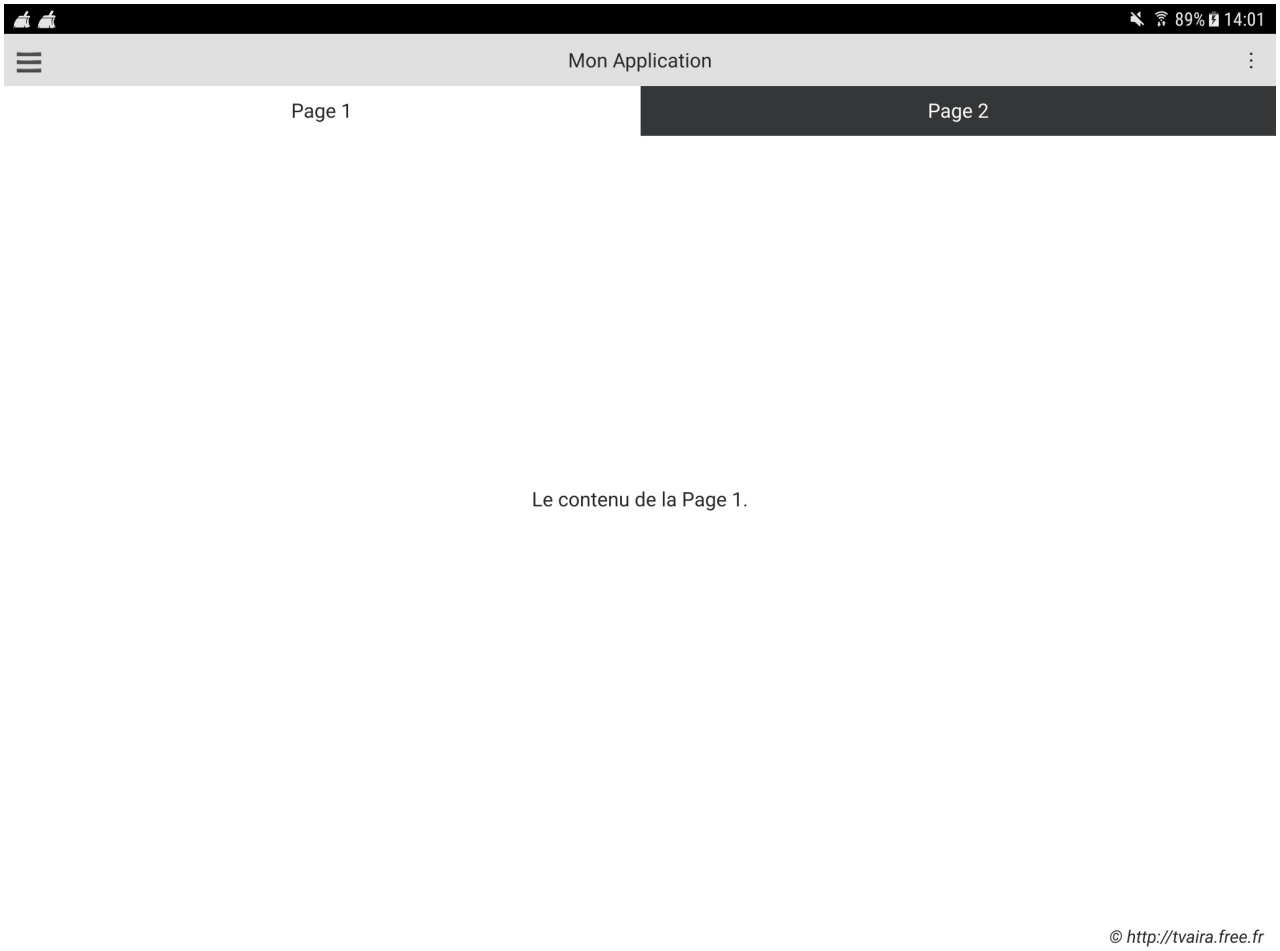
```
...

ApplicationWindow {
    visible: true
    ...
    SwipeView {
        id: vueBalayage
```

```
        currentIndex: 0
        anchors.fill: parent
        Item {
            id: page1
            Label {
                text: qsTr("Le contenu de la Page 1.")
                anchors.centerIn: parent
            }
        }
        Item {
            id: page2
            Label {
                text: qsTr("Le contenu de la Page 2.")
                anchors.centerIn: parent
            }
        }
    }
    PageIndicator {
        id: indicateur
        count: vueBalayage.count
        currentIndex: vueBalayage.currentIndex
        anchors.bottom: vueBalayage.bottom
        anchors.horizontalCenter: parent.horizontalCenter
    }
}
```



On peut remplacer le `PageIndicator` par une `TabBar` que l'on placerait au dessus par exemple. Cela modifie la gestion de l'attribut `currentIndex` :

```
...

ApplicationWindow {
    visible: true
    ...
    SwipeView {
        id: vueBalayage
        currentIndex: onglets.currentIndex
        anchors.fill: parent
        Item {
            id: page1
            Label {
                text: qsTr("Le contenu de la Page 1.")
                anchors.centerIn: parent
            }
        }
        Item {
            id: page2
            Label {
                text: qsTr("Le contenu de la Page 2.")
                anchors.centerIn: parent
            }
        }
    }
    TabBar {
        id: onglets
        width: parent.width
        anchors.top: vueBalayage.top
        anchors.horizontalCenter: parent.horizontalCenter
        currentIndex: vueBalayage.currentIndex
        TabButton {
            text: "Page 1"
            //width: implicitWidth
        }
        TabButton {
            text: "Page 2"
            //width: implicitWidth
        }
    }
}
```



Par défaut, les onglets `TabButton` se répartissent sur la largeur du conteneur. Il est possible de définir la largeur de l'onglet `TabButton` à son contenu texte avec : `width: implicitWidth`

StackView

On va ajouter un `StackView` pour l'affichage des contenus associés aux choix proposés dans le panneau latéral.

StackView étant un modèle basé sur une pile, il prend en charge trois opérations de navigation principales :

- `push()` : ajoute un élément au sommet de la pile,
- `pop()` : supprime l'élément le plus haut de la pile et
- `replace()` : remplace l'élément le plus haut donc équivalent à un *pop* suivi d'un *push*.

Pour chaque opération *push* ou *pop*, différentes animations de transition sont appliquées aux éléments entrant et sortant. Les animations peuvent être personnalisées en affectant différentes transitions aux propriétés : `pushEnter`, `pushExit`, `popEnter`, `popExit`, `replaceEnter` et `replaceExit`.

Dans l'exemple, on modifie :

- le `ToolButton` pour assurer le retour d'une vue
- le `Drawer` pour faire un *push* lors d'une sélection

```
...
```

```
ApplicationWindow {
    visible: true
    ...
    header: ToolBar {
        RowLayout {
            spacing: 20
            anchors.fill: parent

            ToolButton {
                id: toolButton1
                text: vuePile.depth > 1 ? "\u25C0" : "\u2630"
                font.pixelSize: Qt.application.font.pixelSize * 1.6
                onClicked: {
                    if (vuePile.depth > 1)
                    {
                        vuePile.pop()
                    }
                    else
                    {
                        panneau.open()
                    }
                }
            }
            ...
        }
    }
    Drawer {
        id: panneau
        width: window.width * 0.33
        height: window.height
        Column {
            anchors.fill: parent
            ItemDelegate {
                id: choix1
                text: qsTr("Choix 1")
                width: parent.width
                onClicked: {
                    vuePile.push(contenu1)
                    panneau.close()
                }
            }
            ItemDelegate {
                id: choix2
                text: qsTr("Choix 2")
                width: parent.width
                onClicked: {
                    vuePile.push(contenu2)
                    panneau.close()
                }
            }
        }
    }
    ...
    StackView {
        id: vuePile
        anchors.fill: parent
        initialItem: contenu1
        // test
        pushEnter: Transition {
            PropertyAnimation {
```

```
                property: "opacity"
                from: 0
                to: 1
                duration: 2000
            }
        }
    }
    Item {
        id: contenu1
        visible: false
        Label {
            text: qsTr("Choix 1 ...")
            anchors.centerIn: parent
        }
    }
    Item {
        id: contenu2
        visible: false
        Label {
            text: qsTr("Choix 2 ...")
            anchors.centerIn: parent
        }
    }
}
```

Evidemment, il sera préférable de définir ses contenus dans des fichiers `.qml` indépendants. Dans ce cas, on assurera le *push* de cette manière : `vuePile.push("MaPage.qml")`.

ListView

`ListView` permet une vue en liste des éléments fournis par un `model`. L'affichage des éléments de la liste sont pris en charge par un `delegate`.

Le squelette pour intégrer un `ListView` sera :

```
ListView {
    id: vueListe
    model: myModel
    delegate: myDelegate
}
```

Lien : ListView QML Type

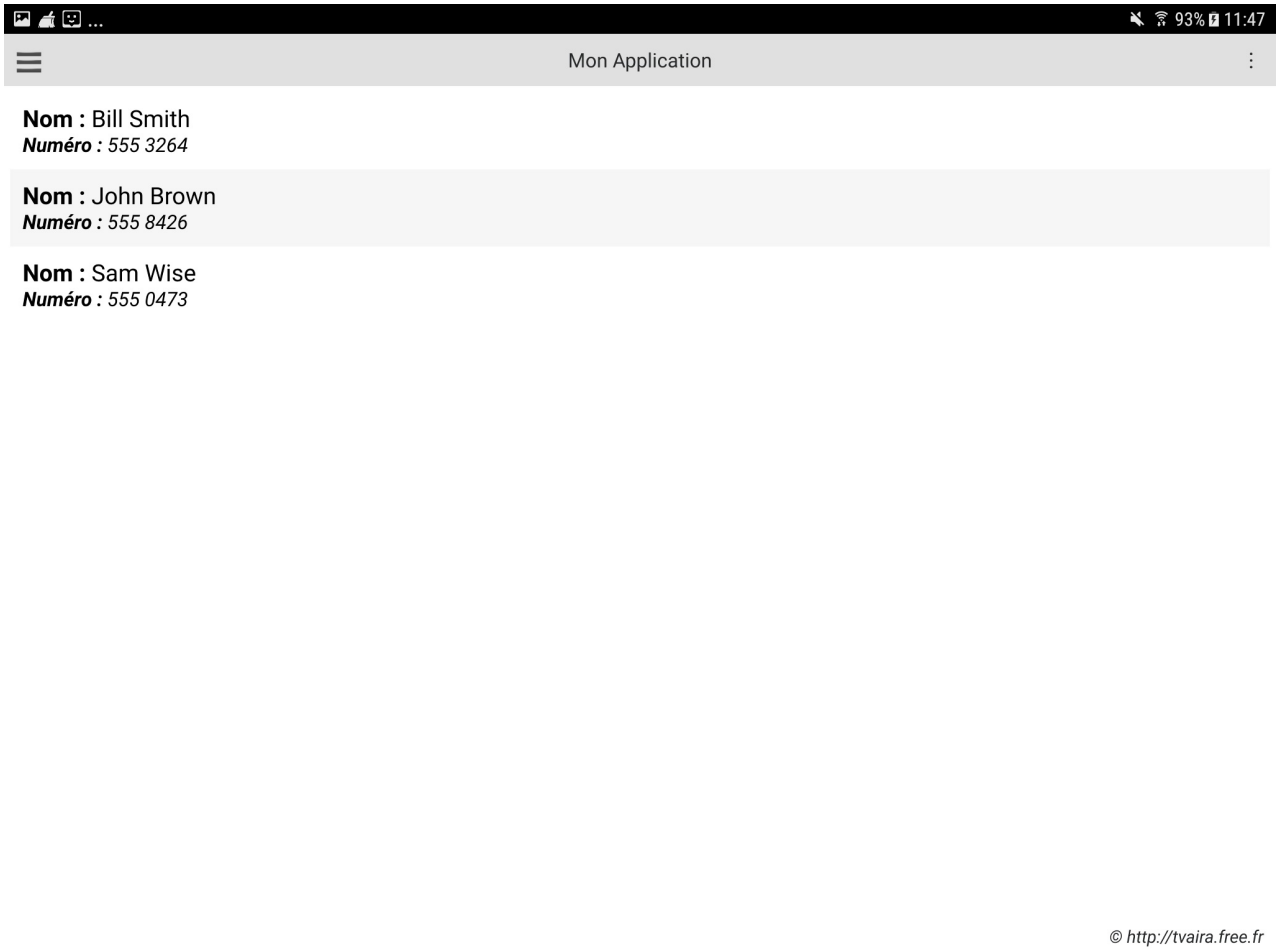
Le modèle et le délégué peuvent être déclarés à l'intérieur de la `ListView` ou séparément. Les données du `model` peuvent aussi provenir du code C++. En QML, on peut créer un modèle avec `ListModel` et ajouter des données à ce modèle avec `ListElement`.

Le délégué fournit un modèle définissant chaque élément instancié par la `ListView`. Le type `ItemDelegate` est l'élément de vue standard. En utilisant sa propriété `highlighted`, on peut mettre l'élement sélectionné en surbrillance.

La `ListView` étant défilable dans la vue, il peut être nécessaire de lui ajouter un `ScrollIndicator`.

```
...
ListView {
    id: vueListe
```

```
    anchors.fill: parent
    model: myModel
    delegate: myDelegate
    focus: true
    ScrollIndicator.vertical: ScrollIndicator { }
}

ListModel {
    id: myModel
    ListElement { nom: "Bill Smith"; numero: "555 3264" }
    ListElement { nom: "John Brown"; numero:  "555 8426" }
    ListElement { nom: "Sam Wise"; numero: "555 0473" }
}

Component {
    id: myDelegate
    ItemDelegate {
        width: parent.width
        height: colonne.implicitHeight
        highlighted: ListView.isCurrentItem
        onClicked: {
            vueListe.currentIndex = index
        }
        Column {
            id: colonne
            padding: 10
            Text { text: '<b>Nom :</b> ' + model.nom; font.pixelSize: Qt.application.font.pixelSi
ze * 1.2 }
            Text { text: '<b>Numéro :</b> ' + model.numero; font.italic: true }
        }
    }
}
```



On pourrait utiliser une `ListView` pour la liste des choix du menu de notre panneau latéral `Drawer` :

```
...
Drawer {
        id: panneau
        width: window.width * 0.33
        height: window.height
        interactive: vuePile.depth === 1

        ListView {
            id: listeChoix
            focus: true
            currentIndex: -1
            anchors.fill: parent

            delegate: ItemDelegate {
                width: parent.width
                text: model.title
                highlighted: ListView.isCurrentItem
                onClicked: {
                    listeChoix.currentIndex = index
                    vuePile.push(model.source)
                    panneau.close()
                }
            }
```

```
            model: ListModel {
                ListElement { title: "Vue balayage"; source: "pageChoix1.qml" }
                ListElement { title: "Vue liste"; source: "pageChoix2.qml" }
            }

            ScrollIndicator.vertical: ScrollIndicator { }
        }
}
```

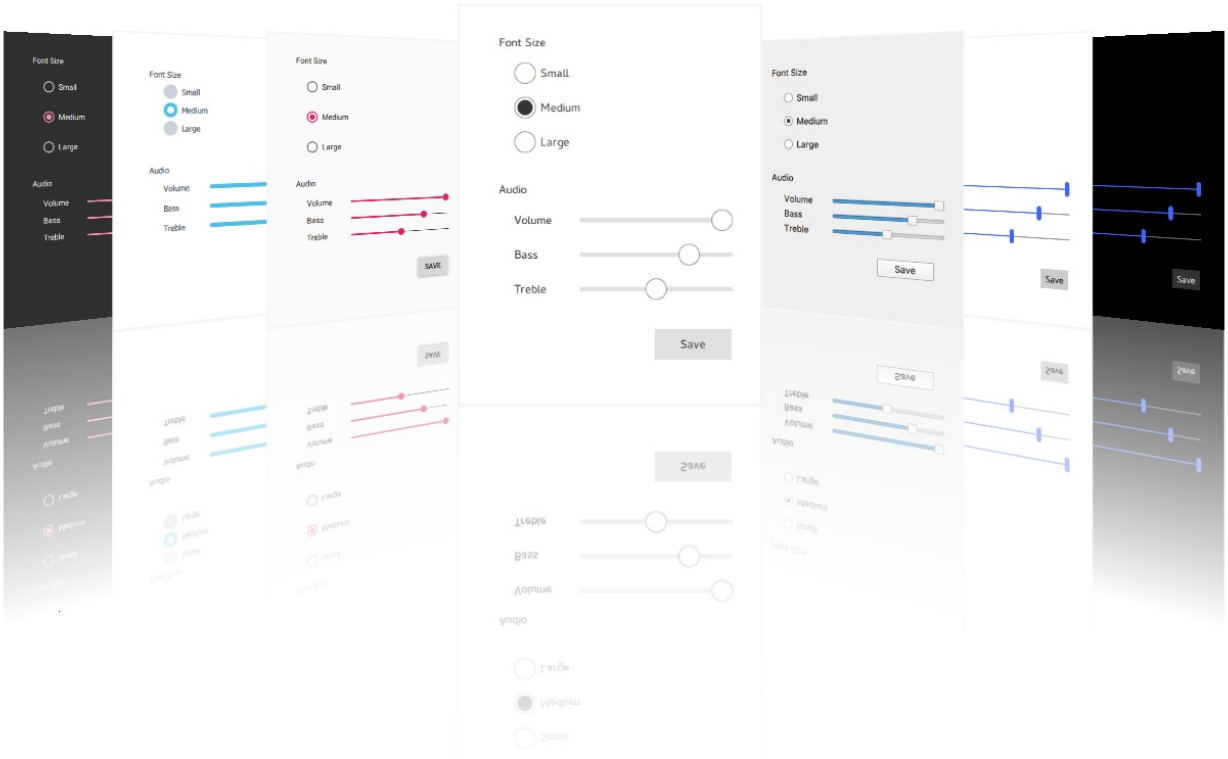
Ici le modèle et le délégué ont été intégrés directement dans la déclaration de la `ListView`.

Contrôles

Les contrôles sont généralement des éléments utilisés pour présenter ou recevoir des entrées de l'utilisateur :

- Button : un bouton avec du texte
- CheckBox : Une case à cocher avec du texte
- ComboBox : Une liste déroulante
- Label : du texte
- ProgressBar : une barre de progression
- RadioButton : Un bouton radio avec une du texte
- Slider : un curseur vertical ou horizontal

  …



Lien : Qt Quick Controls QML Types

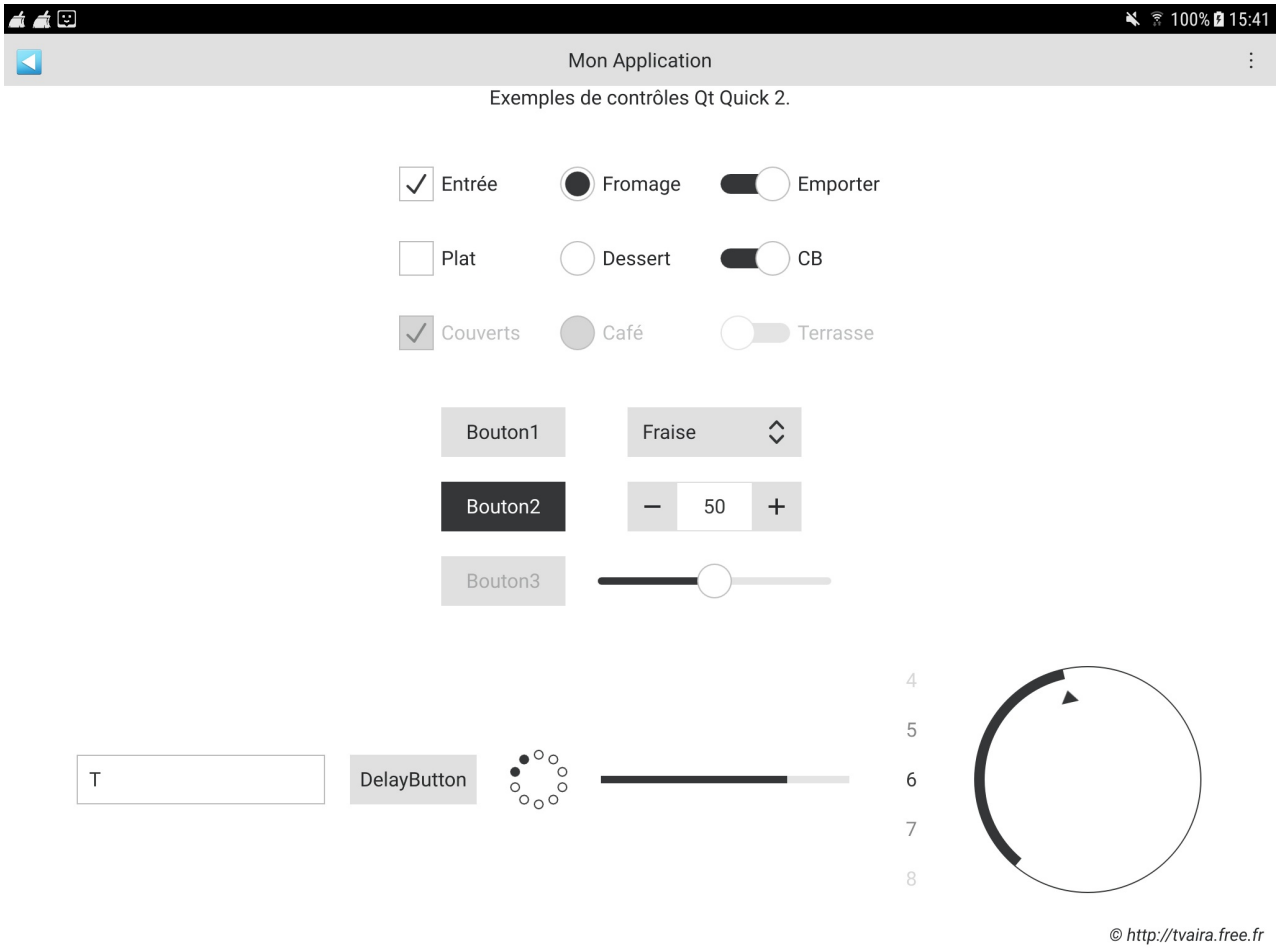
Quelques contrôles de base :

```
import QtQuick 2.9
import QtQuick.Controls 2.2
```

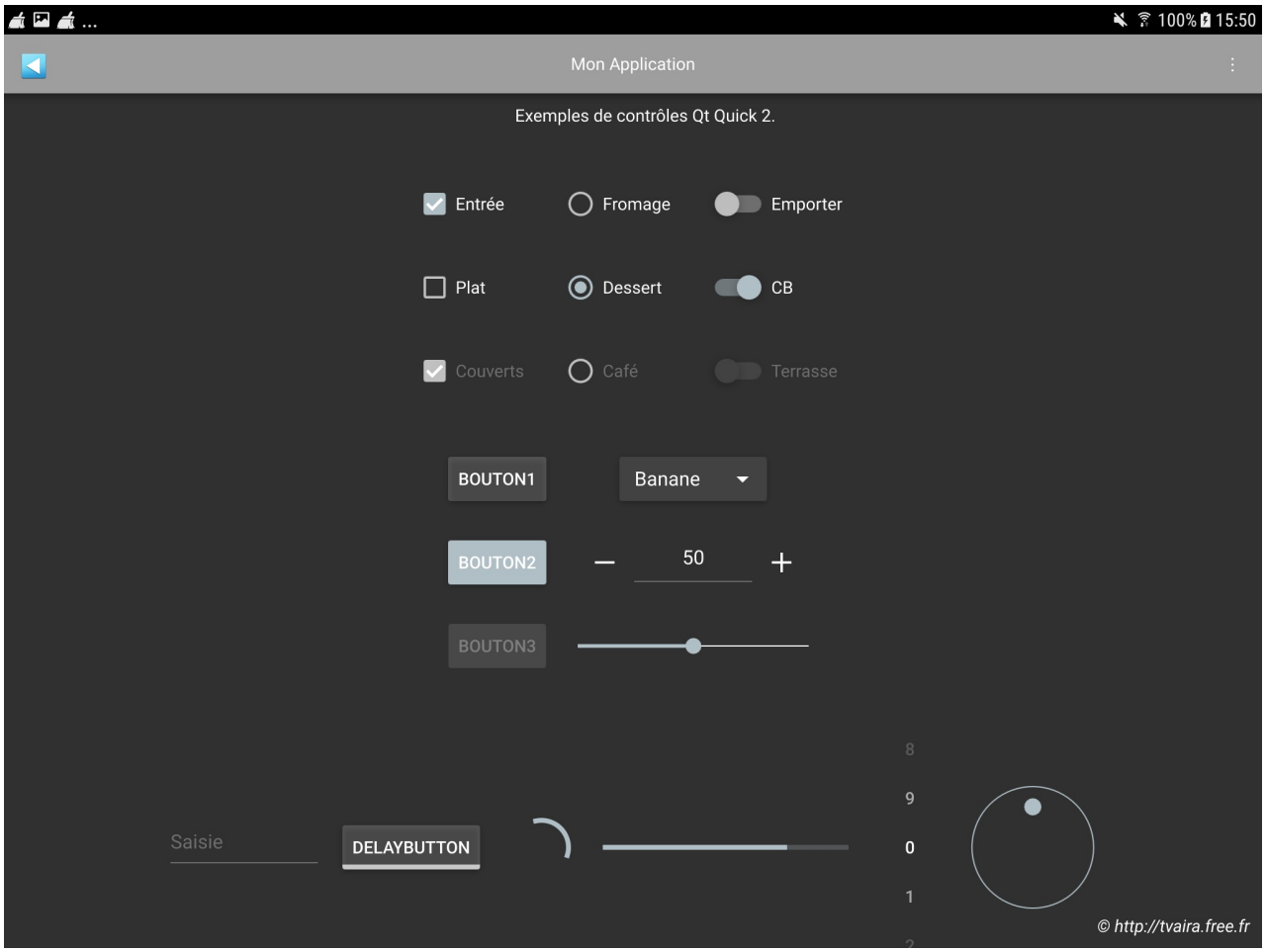
```
Item {
    Column {
        padding: 10
        spacing: 40
        width: parent.width
        Label {
            width: parent.width
            wrapMode: Label.Wrap
            horizontalAlignment: Qt.AlignHCenter
            text: "Exemples de contrôles Qt Quick 2."
        }
        Row {
            anchors.horizontalCenter: parent.horizontalCenter
            spacing: 20
            Column {
                spacing: 20
                CheckBox {
                    text: "Entrée"
                    checked: true
                }
                CheckBox {
                    text: "Plat"
                }
                CheckBox {
                    text: "Couverts"
                    checked: true
                    enabled: false
                }
            }
            Column {
                spacing: 20
                RadioButton {
                    text: "Fromage"
                }
                RadioButton {
                    text: "Dessert"
                    checked: true
                }
                RadioButton {
                    text: "Café"
                    enabled: false
                }
            }
            Column {
                spacing: 20
                Switch {
                    text: "Emporter"
                }
                Switch {
                    text: "CB"
                    checked: true
                }
                Switch {
                    text: "Terrasse"
                    enabled: false
                }
            }
        }
        Row {
```

```
            anchors.horizontalCenter: parent.horizontalCenter
            spacing: 20
            Column {
                spacing: 20
                Button {
                    text: "Bouton1"
                }
                Button {
                    id: button
                    text: "Bouton2"
                    highlighted: true
                }
                Button {
                    text: "Bouton3"
                    enabled: false
                }
            }
            Column {
                spacing: 20
                ComboBox {
                    model: ["Banane", "Fraise", "Orange"]
                    anchors.horizontalCenter: parent.horizontalCenter
                }
                SpinBox {
                    id: box
                    value: 50
                    anchors.horizontalCenter: parent.horizontalCenter
                    editable: true
                }
                Slider {
                    id: slider
                    value: 0.5
                    anchors.horizontalCenter: parent.horizontalCenter
                }
            }
        }
        Row {
            anchors.horizontalCenter: parent.horizontalCenter
            spacing: 20
            Row {
                spacing: 20
                TextField {
                    id: field
                    placeholderText: "Saisie"
                    anchors.verticalCenter: parent.verticalCenter
                }
                DelayButton {
                    text: "DelayButton"
                    anchors.verticalCenter: parent.verticalCenter
                }
                BusyIndicator {
                    anchors.verticalCenter: parent.verticalCenter
                }
                ProgressBar {
                    id: bar
                    value: 0.75
                    anchors.verticalCenter: parent.verticalCenter
                }
                Tumbler {
                    model: 10
```

```
                    anchors.verticalCenter: parent.verticalCenter
                }
                Dial {
                    value: 0.5
                    anchors.verticalCenter: parent.verticalCenter
                }
            }
        }
    }
}
```



Avec un thème *Dark* du style **Material** :



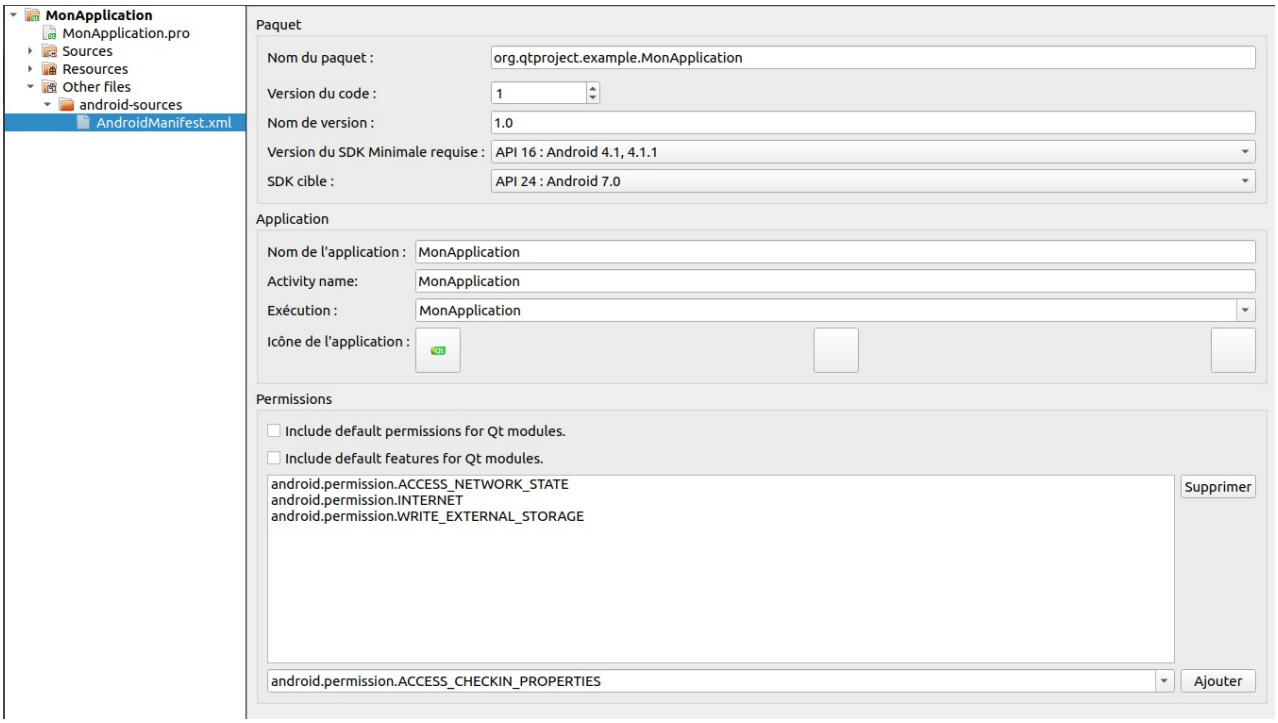
AndroidManifest.xml

Il est possible de modifier paramétrer son application. Pour cela, il faudra éditer le fichier `AndroidManifest.xml` dans Qt Creator.

Il faut tout d'abord copier le fichier `AndroidManifest.xml` généré par Qt depuis le répertoire `android-build` de votre répertoire de *build* dans un répertoire `android-sources` de votre répertoire de projet.

Ensuite vous pouvez ajouter le fichier `AndroidManifest.xml` à votre projet avec le bouton droit "Ajouter des fichiers existants …".

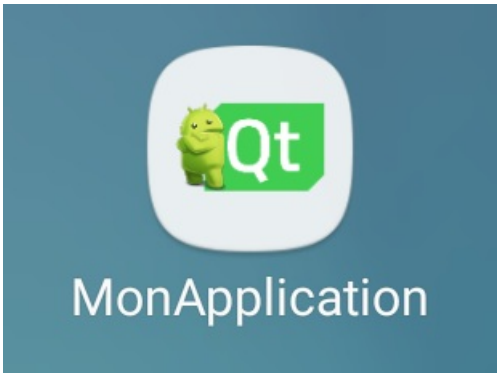
En double-cliquant dessus, vous pouvez l'éditer directement dans Qt Creator et insérer une nouvelle icône par exemple :



Pour terminer, il faut ajouter à votre fichier de projet `.pro` :

```
...
ANDROID_PACKAGE_SOURCE_DIR = $$PWD/android-sources
```

Redéployer l'application sur votre terminal Android.



Code source

Lien : MonApplication.zip

Voir aussi

Qt pour Android :

- Base de données SQLite
- Base de données MySQL
- Dessiner des graphiques

http://tvaira.free.fr/