

L'interface de NoteApp peut être considérée comme complète en terme de nombre de fonctionnalités et d'interactions avec l'utilisateur.

Cependant, il y a encore une grosse marge de progression en pour rendre l'interface plus attrayante pour l'utilisateur.

Le QML a été considéré comme un langage déclaratif, qui se rappelle des animations et des transitions fluides des éléments de l'interface.

Le chapitre va couvrir les thèmes principaux suivants :

- Introduire des concepts à propos des animations et des transitions en QML

- De nouveaux types de QML seront présentés, comme le *Behaviour*, *Transition* et plusieurs *Animations* d'éléments

Le chapitre suit les étapes suivantes :

## 5.1 Animer la NoteToolbar

Voyons comment nous pouvons améliorer le composant *Note* et ajouter un comportement basé sur l'interaction avec l'utilisateur. Le composant *Note* a une toolbar avec un bouton *supprimer\** pour supprimer la note. De plus, la toolbar est utilisé pour déplacer la note en gardant le clic de la souris appuyé dessus.

Une amélioration pourrait être de rendre le bouton *Supprimer* visible seulement si nécessaire. Par exemple, en rendant le bouton *Supprimer* visible quand la toolbar est survolée, et il serait agréable d'utiliser des effets de fade-in et de fade-out.

Le QML fournit plusieurs approche pour implémenter ceci en utilisant les types *animation* et *transition*. Dans ce cas spécifique, nous utiliserons le type *Behaviour* du QML, et nous expliquerons plus tard pourquoi.

### 5.1.1 Behaviour et type NumberAnimation

Dans le composant *Notetoolbar*, on utilise le type *Row* pour disposer le bouton *Supprimer\**, donc en changeant la propriété *opacité* du type *Row* va aussi affecter l'opacité du bouton *Supprimer*.

Remarque : La valeur de la propriété *opacité* est propagé des items parents aux items enfants.

Le type *Behaviour* aide à définir le *Behaviour* de l'item basé sur les changements de propriété de cet item comme montré sur le code suivant :

```
// NoteToolbar.qml
...
MouseArea {
  id: mousearea
  anchors.fill: parent
  // setting hoverEnabled property to true
  // in order for the MouseArea to be able to get
```

```

// hover events
hoverEnabled:
true
}
// using a Row element for laying out tool
// items to be added when using the NoteToolBar
Row {
id: layout
layoutDirection: Qt.RightToLeft
anchors {
verticalCenter: parent.verticalCenter;
left: parent.left;
right: parent.right
leftMargin: 15;
rightMargin: 15
}
spacing: 20
// the opacity depends if the mousearea
// has the cursor of the mouse.
opacity: mousearea.containsMouse ? 1 :
0
// using the behavior element to specify the
// behavior of the layout element
// when on the opacity changes.
// using NumberAnimation to animate
// the opacity value in a duration of 350 ms
NumberAnimation { duration: 350 }
}
}
...

```

Comme vous pouvez le voir sur le code ci-dessus, on active la propriété `hoverEnabled*` du type `MouseArea` pour accepter les événements de survol de la souris. Ensuite, on bascule l'opacité du type `Row` à 0 si le type `Mousearea` n'est pas survolé et à 1 sinon. La propriété `containsMouse` de `MouseArea` est utilisé pour décider de la valeur de l'opacité pour le type `Row`.

Le type `Behaviour` est créé à l'intérieur du type `Row` pour définir son comportement basé sur sa propriété `opacité`. Quand la valeur de l'opacité change, `NumberAnimation` est appliquée.

Le type `NumberAnimation` applique une animation basée sur des changements de valeurs numériques, nous l'utilisons donc sur la propriété `opacité` du `Row` pour une durée de 350 millisecondes.

Remarque : Le type `NumberAnimation` est hérité de `PropertyAnimation`, qui a `Easing.Linear` comme animation de la courbe d'accélération par défaut.

Et ensuite ?

Dans la prochaine étape, nous verrons comment implémenter une animation en utilisant *Transition\** et d'autres types d'animation QML.

## 5.2 Utiliser états et des transitions

Dans l'étape précédente, nous avons vu une approche pratique pour définir de simples animations basées sur les changements de propriétés, en utilisant les types *Behaviour* et *NumberAnimation*.

Evidemment, il y a des cas dans lesquels l'animation dépend d'une palette de changements de propriétés qui pourraient être représentés par un *State*.

Voyons comment nous pouvons aller plus loin dans l'amélioration de l'interface du NoteApp\*.

Les items *Marker* paraissent statiques quant on en vient à l'interaction de l'utilisateur. Et si on voulait ajouter quelques animations basées sur plusieurs scénarios d'interaction de l'utilisateur ?

De plus, nous voudrions rendre le marqueur actuelle actif et la page actuelle plus visible pour l'utilisateur.

### 5.2.1 Animer les items *Marker*

Si nous voulons résumer les scénarios possible pour améliorer les interactions de l'utilisateur avec des items *Marker*, les cas d'usage suivants sont décrits :

Le *Marker* actif actuel devrait être plus visible. Un marker devient actif quand l'utilisateur clique dessus. Le marker actif est un peu plus gros, et il pourrait glisser de gauche à droite (tout comme un curseur).

Quand un utilisateur survole un marker avec une souris, le marqueur glisse de gauche à droite mais pas autant qu'un marqueur actif le ferait.

En considérant les scénarios mentionnés ci-dessus, nous devons travailler sur les composants *Marker* et *MarkerPanel*.

En lisant la descriptions ci-dessus à propos du comportement désiré (l'effet de glissement de gauche à droite), je pense en premier lieu à changer la propriété  $x$  de l'item *Marker* comme il représente la position de l'item sur l'axe X. De plus, comme l'item marqueur doit savoir si c'est le marqueur actif actuelle, une nouvelle propriété appelée *active* peut être introduite.

On peut introduire 2 états pour le composant *Marker* qui peuvent représenter le comportement décrit ci-dessus :

*hovered*- qui va mettre à jour la propriété  $x$  du marqueur quand l'utilisateur le survole en utilisant la souris.

*Selected*- qui va mettre à jour la propriété *x* du marqueur quand le marqueur devient actif, ce qui signifie, quand il est cliqué par l'utilisateur.

```
// Marker.qml

...
// this property indicates whether this marker item
// is the current active one. Initially it is set to false
property bool active:
false
// creating the two states representing the respective
// set of property changes
states: [
// the hovered state is set when the user has
// the mouse hovering the marker item.
State {
name: "hovered"
// this condition makes this state active
when: mouseArea.containsMouse && !root.active
PropertyChanges { target: root; x: 5 }
},
State {
name: "selected"
when: root.active
PropertyChanges { target: root; x: 20 }
}
]
// list of transitions that apply when the state changes
transitions: [
Transition {

to: "hovered"
NumberAnimation { target: root; property: "x"; duration: 300 }
},
Transition {
to: "selected"
NumberAnimation { target: root; property: "x"; duration: 300 }
},
Transition {
to: ""
NumberAnimation { target: root; property: "x"; duration: 300 }
}
]
...
```

On a donc états déclarés qui représentent les changements respectifs de propriétés basés sur le comportement de l'utilisateur. Chaque état est lié à une condition exprimée dans la propriété *when*

Remarque : Pour la propriété *containsMouse* du type *MouseArea*, la propriété *hoverEnabled* doit être mise à *true*.

Le type *Transition* est utilisé pour définir le comportement de l'item quand il change d'un état à un autre. Cela signifie que l'on peut définir plusieurs animations grâce aux propriétés qui changent quand un état devient actif.

Remarque: L'état par défaut d'un item est une chaîne de caractère vide, ("")

Pendant que nous sommes dans le composant *MarkerPanel*, nous devons régler la propriété *active* de l'item *Marker* sur *true* lorsque l'on clique dessus. Se référer à *MarkerPanel.qml* pour le code mis à jour.

## 5.2.2 Ajouter des transitions à PagePanel

Dans le composant *PagePanel*, nous utilisons des états pour gérer la navigation entre les pages. Ajouter des transitions nous vient naturellement à l'esprit. Comme nous changeons la propriété *opacité* dans chaque état, nous pouvons ajouter *Transition* pour tous les états qui contrôlent un *NumberAnimation* sur les valeurs de l'opacité pour créer les effets fade-in et fade-out.

```
// PagePanel.qml

...
// creating a list of transitions for
// the different states of the PagePanel
transitions: [
  Transition {
    // run the same transition for all states
    from: " "; to: "
    *
    "
    NumberAnimation { property: "opacity"; duration: 500 }
  }
]
...
```

Remarque: La valeur *opacity* d'un item est propagé à ces éléments enfants aussi

Et ensuite ?

Dans la prochaine étape, nous allons apprendre à améliorer plus en détails l'interface utilisateur et voir ce que l'on peut faire de plus.

## Amélioration en profondeur

A ce niveau, nous pouvons considérer que les fonctionnalités de NoteApp\* sont complètes et que l'interface utilisateur correspond aux spécifications de *NoteApp*. Néanmoins, on peut toujours trouver plus d'améliorations, celles-ci peuvent être mineures, mais elle rendent l'application finie et prête à l'emploi.

Dans ce chapitre, nous verrons les petites améliorations qui sont implémentées pour NoteApp\*, mais aussi suggérer de nouvelles idées et fonctionnalités qui pourraient être ajoutées. Bien entendu, nous souhaiterions encourager tout le monde à prendre le code source de NoteApp\* et le développer plus en profondeur en redesignant éventuellement l'entièreté de l'interface utilisateur et en ajoutant de nouvelles fonctionnalités.

Voici une liste des thèmes principaux abordés dans ce chapitre :

- Plus de javascript utilisé pour améliorer la fonctionnalités.

- Travailler avec le classement en z des items QML

- Utiliser des polices customisés pour l'application

### 6.1 Améliorer la fonctionnalité de l'item Note

Une chouette fonctionnalité pour les items *Note* serait d'avoir la note qui grandit au fur et à mesure que du texte est tapé.

Disons juste, pour des raisons de simplicité, que la note va grandir verticalement tant que du texte est entré et que la note entoure le texte pour tenir en largeur.

Le type *Text* a une propriété *paintedHeight* qui nous donne la taille actuelle du texte affiché sur l'écran. A partir de cette valeur, nous pouvons augmenter ou diminuer la hauteur de la note.

Premièrement, définissons une fonction Javascript helper qui calcule la valeur de la propriété hauteur du type *Item*, qui est le type de plus haut niveau pour le composant *Note*.

```
// Note.qml
```

```
...
// JavaScript helper function that calculates the height of
// the note as more text is entered or removed.
function
updateNoteHeight() {
// a note should have a minimum height
var
```

```

noteMinHeight = 200
var
currentHeight = editArea.paintedHeight + toolbar.height +40
root.height = noteMinHeight
if
(currentHeight >= noteMinHeight) {
root.height = currentHeight
}
}
...

```

Tant que la fonction *updateNoteHeight()* met à jour la propriété *height* de *root* basée sur la propriété *paintedHeight* de *editArea*, nous devons appeler cette fonction via un changement sur *paintedHeight*.

```

// Note.qml

...

// creating a TextEdit item

TextEdit {

id: editArea

...

// called when the painterHeight property changes

// then the note height has to be updated based

// on the text input

onPaintedHeightChanged: updateNoteHeight()

...

}

```

Remarque : Toutes les propriétés émettent un signal de notification à chaque fois qu'une propriété changeant

La fonction JavaScript *updateNoteHeight()* change la propriété *height*, on peut donc définir un comportement pour cela en utilisant le type *Behavior*.

```
// Note.qml
...
// defining a behavior when the height property changes
// for the root element
Behavior on height { NumberAnimation {} }
```

Et ensuite ?

La prochaine étape montre comment utiliser la propriété *z* du type *Item* pour arranger convenablement les notes.

## 6.2 Arranger les Notes

Les notes à l'intérieur d'une page ne savent pas sur quel note l'utilisateur est en train de travailler. Par défaut, tous les objets *Notes* créés ont la même valeur par défaut pour la propriété *z* et dans ce cas-ci, le QML crée un arrangement par défaut des objets basés sur celui qui a été créé en premier.

Le comportement désiré serait de changer l'ordre des notes selon une interaction de l'utilisateur.

Quand l'utilisateur clique sur la note toolbar ou commence à éditer une note, la dite note devrait ressortir pour ne pas être en dessous d'autres notes. Cela est possible en changeant la valeur *z* pour qu'elle soit plus haute que les autres notes.

```
// Note.qml
Item {
id: root
...
// setting the z order to 1 if the text area has the focus
z: editArea.activeFocus ? 1:0
...
}
```

Dans le gestionnaire de signal *onPressed* dans le type *MouseArea*, on émet le signal *pressed()* du *root* de la *NoteToolbar*.

Le signal *pressed()* du composant *NoteToolbar* est géré dans le composant *Note*.

```
// Note.qml
...
// creating a NoteToolbar item that will be anchored to its parent
NoteToolbar {
    id: toolbar
    ...
    // setting the focus on the text area when the toolbar is pressed
    onPressed: editArea.focus = true
    ...
}
```

Dans le code ci-dessus, la propriété *focus* de l'objet *editArea* est mise à *true*, de manière à ce que *editArea* reçoive l'entrée *focus*. C'est pour cela que *activeFocus*, qui devient *true*, déclenche le changement de la propriété de la valeur *z*.

Et ensuite ?

La prochaine étape va expliciter comment charger et utiliser un fichier de police d'écriture en local, pour *NoteApp*

### 6.3 Chargement d'une police personnalisée

Utiliser et répandre des polices personnalisées plutôt que des polices implémentées par défaut est devenu une méthode commune de nos jours. Pour *NoteApp*, nous souhaitons faire de même et nous utiliserons ce que les fonctionnalités que le QML peut nous offrir.

Le *FontLoader* QML vous permet de charger les polices par leur nom ou leur adresse URL. Comme la police importée pourra être largement utilisée dans toute l'application, nous recommandons de charger la police dans le fichier *main.qml* et de l'utiliser dans le reste dans sous-parties.

```

// main.qml

Rectangle {

// using window as the identifier for this item as
// it will be the only window of the NoteApp

id: window

...

// creating a webfont property that holds the font
// loading using FontLoader

property variant webfont: FontLoader {

source: "fonts/juleeregular.ttf"

onStatusChanged: {

if (webfontloader.status == FontLoader.Ready)

console.log('Loaded')

}

}

...

}

```

Par conséquent, nous avons créé une propriété *webfont* pour l'objet *window*. Cette propriété peut être utilisée sans risque dans le reste du corps, utilisons la donc pour *editArea* dans le composant *Note*.

```

// Note.qml

...

// creating a TextEdit item

TextEdit {

id: editArea

```

```
font.family: window.webfont.name; font.pointSize: 13
...
}
```

Pour régler la police de *editArea*, on utilise la propriété *font.family*. Depuis la *window*, on utilise sa propriété *webfont* pour avoir le nom de la police de réglé.

Et ensuite ?

La prochaine étape vous guidera afin de rendre NoteApp\* paré au déploiement.

## Déployer l'application NoteApp

Nous sommes arrivés au point où nous voudrions rendre l'application disponible et déployable pour un environnement de bureau classique. Comme décrits dans les premiers chapitres, nous avons utilisé un projet Qt Quick UI dans Qt Creator pour développer l'application NoteApp\*. Cela signifie que *qmlscene* est utilisé pour charger le fichier *main.qml* et ainsi, lancer *NoteApp*.

En premier lieu, le moyen le plus facile de rendre NoteApp disponible est de créer un packet qui rassemble tous les fichiers qml, *qmlscene* et un script simple qui charge le fichier *main.qml* en utilisant *qmlscene*. Vous devrez vous référer à la documentation de chaque plateforme de bureau pour voir comment écrire un tel script. Par exemple, sur une plateforme linux, vous devrez utiliser un petit script shell *bash* alors que sur Windows vous aurez besoin d'un simple fichier batch. Cette technique fonctionne bien car elle est très simple, mais peut-être que vous ne voudriez pas envoyer le code source car votre application utilise du code propriétaire. L'application devra être envoyé au format binaire, dans lequel tous les fichiers qml sont empaquetés. Ceci aidera à rendre l'installation et l'expérience utilisateur plus plaisantes.

Ensuite, nous devons créer un fichier exécutable pour NoteApp\* qui devra être simple à installer et à utiliser. Dans ce chapitre, nous verrons comment créer une application Qt Quick qui rassemble les fichiers qml et les images dans un fichier binaire exécutable. De plus, nous verrons comment utiliser les Qt Resource System avec QML.

## 7.1 Créer l'application Noteapp Qt

Le but est de créer un seul fichier binaire Noteapp\* exécutable que l'utilisateur exécutera pour charger *NoteApp*.

Voyons comment nous pouvons utiliser Qt Creator pour cela.

## 7.1.1 Créer une application Qt Quick

Premièrement, nous avons besoin de créer une *Qt Quick Application* en utilisant *Qt Creator* et vérifier que nous avons bien sélectionné des éléments *Built-in* seulement (pour toutes les plateformes) dans l'assistant Qt Quick Application. Nommons l'application *noteapp*.

Nous avons donc maintenant un nouveau projet créé depuis l'assistant et on remarque qu'un projet *qtquick2applicationviewer* a été généré en même temps. Le projet *qtquick2applicationviewer* est un 'modèle' basique qui charge les fichiers QML.

Cette application, qui est très commune pour déployer des applications Qt Quick à travers des appareils, inclut plusieurs code spécifique aux plateformes pour chacune des cibles de déploiement.

Nous ne parlerons pas de ces parties spécifiques du code car ce n'est pas l'objectif de ce guide.

Néanmoins, il y a quelques particularités de *qtquick2applicationviewer\** qui nous limitent pour réaliser ce que l'on souhaite. L'application attend du développeur qu'il envoie les fichiers QML avec le fichier exécutable en binaire. Utiliser le Qt Resource System devient impossible, mais nous verrons comment passer outre ce problème.

Pour le projet *noteapp\**, il y a un fichier source, *main.cpp*. Dans le fichier *main.cpp*, nous verrons comment l'objet *viewer*, la classe *QtQuick2ApplicationViewer*, est utilisée pour charger le fichier *main.qml* en appelant la fonction *QtQuick2ApplicationViewer::setMainQmlFile()*.

```
// main.cpp
...

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);

    QtQuick2ApplicationViewer viewer;

    viewer.setMainQmlFile(QStringLiteral("qml/noteapp/main.qml"));

    viewer.showExpanded();

    return app.exec();
}
```

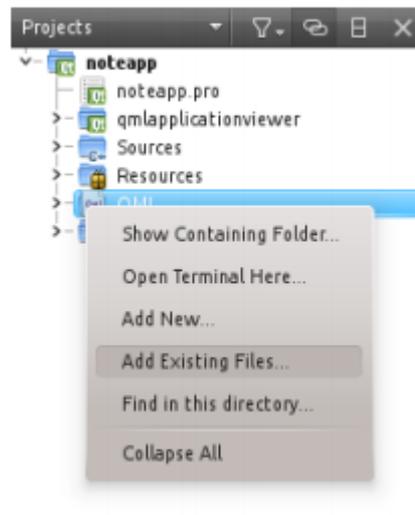
Notez qu'il y a un fichier basique *main.qml* généré par l'assistant Qt Quick Application qui va être remplacé par le fichier *main.qml* que nous avons créé pour NoteApp\*.

La structure du projet noteapp\* généré est très simple à comprendre.

noteapp - fichier racine du projet *noteapp*

- *qml* – ce fichier contient tout les fichiers QML
  - *qtquick2applicationviewer* – l'application générée utilisé pour charger les fichiers QML
- notapp.pro* – le fichier projet
- *main.cpp* – le fichier C++ où est créé l'application Qt

Nous devons copier nos fichiers QML avec les répertoires *polices* et *images* dans le répertoire *qml* du projet noteapp\* nouvellement créé. Qt Creator identifie les changements du dossier projet et ajoute les nouveaux projets dans la vue du projet. S'il ne le fait pas, clic-droit sur le projet et *Add Existing Files* pour ajouter les nouveaux fichiers.



Remarque : Vérifiez que vous avez bien ajouter tous les fichiers existants et les images appelés dans le fichier *nodeDDB.js*

Désormais, nous pouvons compiler et lancer le projet pour voir si tout s'est passé comme prévu lors de la création du projet. Avant de compiler le projet noteapp\*, vérifions que nous avons les bons paramètres en place pour notre projet. Se référer à la section *Configure Projects* dans la documentation de Qt Creator.

Une fois l'application correctement compilée, un fichier binaire exécutable nommé *noetapp* devrait

apparaître dans le dossier racine du projet. Si Qt est correctement configuré pour votre système vous pourrez lancer le fichier en cliquant dessus.

### 7.1.2 Utiliser Qt Resource System pour stocker des fichiers QML et des images

Nous avons créés un exécutable qui charge le fichier QML pour que l'application se lance. Comme vous pouvez le voir dans le fichier `main.cpp`, l'objet `viewer` charge le fichier `main.qml` en passant par le chemin relatif de ce fichier. De plus, nous ouvrons le fichier `noteapp.pro` pour comprendre le déploiement et les paramètres de compilations que l'on peut voir sur les premières lignes :

```
# Add more folders to ship with the application, here

folder_01.source = qml/noteapp

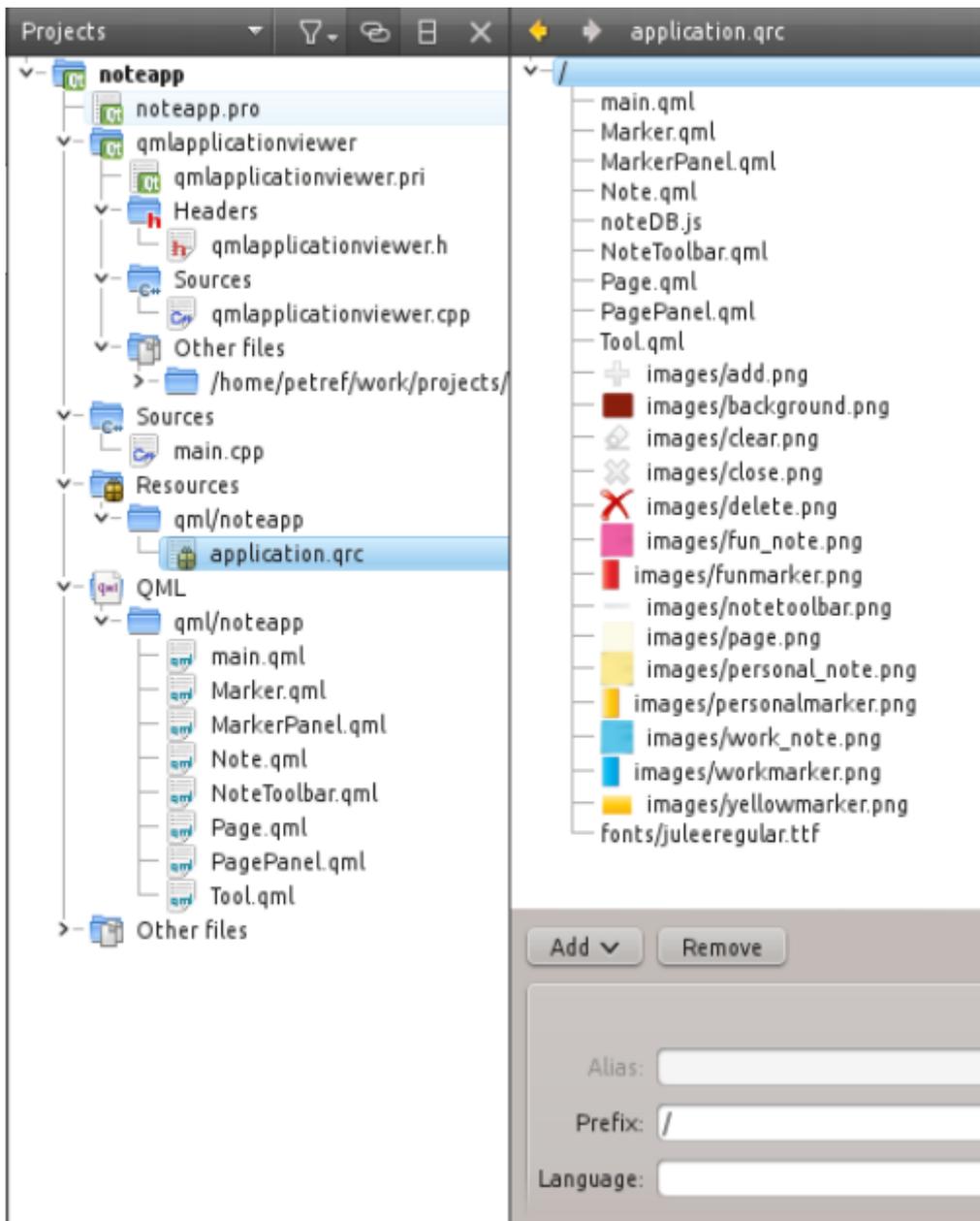
folder_01.target = qml

DEPLOYMENTFOLDERS = folder_01

....
```

Apparemment il est attendu qu'on envoie les fichiers QML avec le fichier exécutable, mais ce n'est pas ce que l'on veut faire.

Qt fournit un *Resource System* plutôt intuitif qui marche parfaitement avec QML. Nous devons créer un fichier ressource, `noteapp.qrc` pour le projet racine `noteapp*` afin que nous puissions y ajouter nos fichiers images et QML. Se référer à *Creating a Resource File* dans la documentation Qt Creator pour une explication plus détaillée.



Nous devons appliquer de petits changements aux fichiers *noteapp.pro* et *main.cpp* afin de pouvoir utiliser le fichier ressource nouvellement créé, *noteapp.pro* :

```
# Add more folders to ship with the application, here
```

```
#folder_01.source = qml/noteapp
```

```
#folder_01.target = qml
```

```
#DEPLOYMENTFOLDERS = folder_01
```

....

Dans le fichier *main.cpp*, nous voyons que la fonction `QtQuick2ApplicationViewer::setMainQmlFile()` est appelée avec le chemin relatif dans le fichier *main.qml*

```
// qtquick2applicationviewer.cpp
...
void QtQuick2ApplicationViewer::setMainQmlFile(const QString &file)
{
d->mainQmlFile = QtQuick2ApplicationViewerPrivate::adjustPath(file);
setSource(QUrl::fromLocalFile(d->mainQmlFile));
}
...
```

La classe *QtQuick2ApplicationViewer* hérite de *QQuickView*, qui est une classe pratique pour charger et afficher les fichiers QML. La fonction `QtQuick2ApplicationViewer::setMainQmlFile()` n'est pas optimisée pour utiliser les ressources parce qu'elle ajuste le chemin du fichier QML `qvqnt` d'appeler la fonction `setSource()`.

L'approche la plus simple pour passer outre serait d'appeler directement `setSource` sur l'objet *viewer* dans le fichier *main.cpp*, mais cette fois-ci on passe le *main.qml* en tant que fichier ressource.

```
// main.cpp
...
int main(int argc, char *argv[])
{
QGuiApplication app(argc, argv);
QtQuick2ApplicationViewer viewer;
viewer.setMainQmlFile(QStringLiteral("qml/noteapp/main.qml"));
viewer.showExpanded();
return app.exec();
}
```

Il n'y a pas d'autre changement à faire dans les fichiers QML où nous utilisons les fichiers image et police car le chemin des fichiers est relatif, ce qui dirigera vers le système de fichiers interne de la ressource.

Essayons de compiler et de voir si cela fonctionne !

### 7.1.3 Ajouter une icône et un titre pour l'application

Il est hautement recommandé d'ajouter une icône pour l'application, ceci permettra d'identifier votre application lorsqu'elle sera déployé sur un bureau.

Dans le dossier `noteapp*`, vous avez peut-être remarqué quelques fichiers *PNG* et u fichier *SVG*.

Ces fichiers images seront utilisées pour régler l'icône de l'application. En fonction de la taille de l'icône nous aurons des icônes 64x64 ou 80x80 ou des icônes vectorisés.

Pour plus de détails en ce qui concerne la façon dont ces fichiers icônes sont déployés sur plusieurs plateformes, vous devrez regarder attentivement au fichier `qtquick2applicationviewer.pri`. Vous trouverez des informations détaillées sur l'icône des applications dans la référence *How to Set the Application Icon* de la documentation Qt.

Nous devons appeler la fonction `setWindowIcon()` sur le *viewer* afin de régler régler l'icône pour la fenêtre de l'application.

```
// main.cpp
...
QScopedPointer<QApplication> app(createApplication(argc, argv));
QScopedPointer<QtQuick2ApplicationViewer> viewer(
QtQuick2ApplicationViewer::create());
viewer->setWindowIcon(QIcon("noteapp80.png"));
...
```

Nous avons besoin d'un titre de fenêtre par défaut pour notre application. Pour cela nous utiliserons la fonction `setWindowTitle()`

```
// main.cpp
...
QScopedPointer<QApplication> app(createApplication(argc, argv));
QScopedPointer<QtQuick2ApplicationViewer> viewer(
QtQuick2ApplicationViewer::create());
viewer->setWindowIcon(QIcon("noteapp80.png"));
viewer->setWindowTitle(QString("Keep Your Notes with NoteApp!"));
...
```

La NoteApp\* est maintenant prête à être envoyée et déployée sur plusieurs plateformes.

## 7.1.4 Déployer NoteApp

NoteApp\* est une application Qt classique, vous devrez alors décider si vous voudrez la lier statiquement ou dynamiquement à Qt. En outre, toutes les plateformes de bureaux ont des configurations de liens spécifiques à considérer.

Vous pourrez trouver des informations plus détaillées dans la référence *Deploying Qt Applications* de la documentation pour chaque cible de déploiement.

Et ensuite ?

Un résumé de ce que nous avons appris dans ce guide pour développeur.

Ce que nous avons appris et pour aller plus loin

Ce guide vous a montré comment créer une application avec Qt Quick et comment la déployer sur un environnement de bureau. Nous avons vu comment développer une application NoteApp\* étape par étape et nous avons appris divers aspects du langage QML et de son potentiel pour développer des interfaces utilisateurs fluides et modernes, tout en gardant le code propre et simple en appliquant plusieurs techniques de programmation.

Nous avons appris quelques-unes des meilleures utilisations des types de QML et avons traité des sujets intéressants comme :

- Les états et les animations
- Utiliser le JavaScript pour améliorer la fonctionnalité
- La gestion dynamique des objets QML
- Le stockage local des bases de données
- Rendre l'application prête au déploiement

Maintenant, vous devriez avoir les compétences nécessaires pour améliorer NoteApp\* avec des fonctionnalités, des améliorations au niveau de l'interface utilisateur et en apprendre plus sur les différentes fonctionnalités de QML et Qt Quick que nous n'avons pas pu traiter dans ce guide.

Qt Quick est une technologie qui grandit vite et qui est en train d'être adoptée par différentes entreprises de développement logiciels. Il serait donc utile de se référer à la page de documentation de Qt pour connaître les dernières mises à jour de cette technologie.

