Chapitre 4

Gestion dynamique des objets de Note

Jusqu'à présent, nous avons vu que QML est un puissant langage déclaratif. Cependant, en le combinant avec le Javascript, il est possible de le rendre encore plus puissant. QML permet non seulement d'inclure des fonctions Javascript mais également d'importer des librairies Javascript complètes.

La principale fonction de NoteApp est de permettre aux utilisateurs de créer, éditer et supprimer des notes comme ils le souhaitent. L'application devrait être capable de stocker des notes automatiquement sans que l'utilisateur ait à s'en soucier.

Ce chapitre va vous guider sur l'utilisation de JavaScript pour ajouter un aspect logique au code QML, ainsi que sur l'implémentation du stockage local en utilisant « QT Quick Local Storage ».

Note : Vous trouverez l'implémentation correspondant à ce chapitre dans le fichier zip de la partie II-A en cliquant sur « Les sources sont disponibles ».

Les principaux sujets abordés dans ce chapitre sont :

- L'utilisation de JavaScript pour implémenter la fonctionnalité de la gestion dynamique des objets de note
- Comment stocker des données localement en utilisant l'API Qt Quick Database

Ce chapitre comporte les étapes suivantes :

4.1 Création et gestion des objets Note

L'utilisateur devrait pouvoir créer et supprimer des notes à la volée. Cela implique que notre code doit être capable de créer et supprimer des objets note dynamiquement. Il y a plusieurs façons de créer et gérer des objets QML. En réalité, nous en avons déjà vu une : en utilisant le type Repeater.
Pour créer un objet QML, il faut que le composant soit créé et chargé avant de créer des instances de ce composant.
Les objets QML peuvent être créés en appelant la fonction JavaScript createObject(Item parent, object properties) sur le composant.
Voyons comment créer notre notre objet parmi nos composants.

4.1.1 Création dynamique d'objets note

Nous savons qu'un objet Note se situe sur la page Composants qui est responsable de la création des objets note et également de charger les notes depuis la base de données. Comme nous l'avons déjà mentionnée, nous chargeons d'abord le composant note sur la page composants.

```
// Page.qml
...
```

```
// Chargement du composant Note
Component {
        id: noteComponent
        Note { }
}
…
```

A présent, définissons une fonction JavaScript qui va créer l'objet Note pour QML. Durant la création d'un objet QML, il faut s'assurer que l'un des arguments est le parent de cet objet. Comme nous voudrions stocker des notes dans la base de données , il est judicieux d'avoir un container sur la page composants pour gérer nos notes.

```
// Page.qml
…
// création d'un item qui nous servira de container
Item { id: container }
…
//Une fonction JavaScript qui aide à la création d'objets notes QML
function newNoteObject(args) {
        //On appelle la focntion createObject() de noteComponent
        //et l'objet container sera le parent du nouvel objet
        //et args représente les arguments
        var note = noteComponent.createObject(container, args)
        if(note == null) {
                console.log("Echec de la création de l'objet note!")
        }
}
…
```

Dans le code ci-dessus, nous voyons comment créer une nouvelle note à partir de la fonction newNoteObject. Les nouvelles notes ainsi créés appartiendront au container. Maintenant, nous allons voir comment appeler cette fonction quand l'utilisateur clique sur l'outil de création d'une nouvelle note depuis la boîte à outils qui est créé dans le fichier main.qml.
Comme le composant PagePanel est au courant de quelle est la page visible, nous pouvons créer une nouvelle propriété dans PagePanel pour stocker cette page.

Nous allons accéder à cette propriété depuis le fichier main.qml et appeler cette fonction pour créer des nouveaux objets note.

```
// PagePanel.qml
…
// Cette propriété contient la page visible active
property Page currentPage: personalpage

//Création de la liste d'états
states: [
        //Création d'un élément State avec son nom correspondant.
        State {
                name: "personel"
```

```
                PropertyChanges {
                        target: personalpage
                        opacity:1.0
                        restoreEntryValues: true
                }
                PropertyChanges {
                        target: root
                        currentPage: personalpage
                        explicit: true
                }
        },
        State {
                name: "fun"
                PropertyChanges {
                        target: funpage
                        opacity:1.0
                        restoreEntryValues: true
                }
                PropertyChanges {
                        target: root
                        currentPage: funpage
                        explicit: true
                }
        },
        State {
                name: "travail"
                PropertyChanges {
                         target: workpage
                        opacity:1.0
                        restoreEntryValues: true
                }
                PropertyChanges {
                        target: root
                        currentPage: workpage
                        explicit: true
                 }
        }
]
```

…
On modifie nos 3 états en assignant la bonne valeur de la propriété *currentPage*.
In le fichier *main.qml,* voyons comment appeler la fonction qui permet de créer des nouveaux objets note quand l'utilisateur cliqyue sur le bouton « nouvelle note ».

```
//main.qml
…
// using a Column element to layout the Tool items vertically
Column {
        id: toolbar
        spacing: 16
        anchors {
```

```
                top: window.top; left: window.left; bottom: window.bottom; topMargin: 50;
                bottomMargin: 50; leftMargin: 8
        }
// L'outil de nouvelle note, également connu comme le bouton plus
Tool {
        id: newNoteTool
        source: "images/add.png"

        // using the currentPage property of PagePanel and
        // calling newNoteObject() function without any arguments.
        onClicked: pagePanel.currentPage.newNoteObject()
        }
}
…
```

4.1.2 Supprimer des objets note

Supprimer des objets note est plus simple que d'en créer car le type objet QML fournit une fonction JavaScript destroy(). Comme nous avons déjà un objet container qui a pour enfants des objets note, il suffit d'appeler la fonction destroy() sur chacun des enfants.

Dans le composent Page, définissons une fonction qui réalisera cette opération :

```
//Page.qml
…
// Une fonction Javascript qui permet d'appliquer la fonction destroy a tous les éléments d'un
// container ( la fonction marche par itérations)

function clear() {
        for(var i=0; i<container.children.length; ++i) {
                container.children[i].destroy()
        }
}
…
```

Dans le fichier main.qml, on appelle la fonction clear() quand l'utilisateur appuie sur le bouton effacer:

```
//main.qml
..
// création d'un objet Notetoolbar (barre d'outils) qui sera liée à ses parents

NoteToolbar {
        id: toolbar
        height: 40 anchors {
        top: root.top; left: root.left; right: root.right }
         // On utilise un alias de la propriété drag pour assigner le drag.target à notre objet
note
         drag.target: root

        // Création d'un outil 'delete' pour supprimer des objets note
```

```
        Tool {
                id: deleteItem
                source: "images/delete.png"
                 onClicked: root.destroy()
        }
}
…
```

Et maintenant ?

Maintenant nous allons nous pencher sur la façon de stocker des variables dans une base de donnée locale.

4.2 Stockage et chargement de données depuis la base de données
Jusqu'à présent, nous avons implémenté la fonctionnalité de base de *NoteApp* : créer et gérer des objets note à la volée.

Dans cette étape, nous allons voir en détail comment implémenter le stockage de données dans la base. QML propose un API très accessible : QT Quick Local Storage API, qui utilise la base de données SQLite pour implémenter la fonctionnalité qui nous intéresse. La meilleure approche pour la *NoteApp* serait de charger les notes depuis à base de données lorsque l'application démarre et de les enregistrer quand on ferme l'application. De cette façon, l'utilisateur ne reçoit pas de messages et n'a rien a faire pour enregistrer les notes.

4.2.1 Définissons la base de données

La base de données pour *NoteApp* devrait être relativement simple. Elle comporte une seule table, la table note qui contient les informations de nos notes enregistrées.

En considérant la définition de la Table, essayons de déterminer (à partir des informations ci-dessus) ce que contient les composants note et si il va falloir rajouter de nouvelles informations.

> Chaque objet QML comporte des propriétés géométriques x et y. Il va être simple d'extraire ces données des objets note. Ces valeurs seront utilisées pour stocker la position des notes sur la page. *noteText* correspond au texte contenu dans la note. Nous pouvons l'obtenir grâce à l'élément *Text* de notre composant Note. Cependant, nous allons définir un alias de la propriété et le nommer *noteText*. Nous verrons cela plus tard. *noteId* et *markerId* sont des identifiants que tout objet note se doit d'avoir. noteId sera un identifiant unique demandé par la base de données alors que markerId va nous permettre de savoir à quelle page appartient l'objet note. Nous avons donc besoin de deux nouvelles propriétés dans le composant Note.

Dans l'objet note, nous définissons les nouvelles propriétés nécessaires:

```
// Note.qml


Item {
id: root
width: 200; height: 200

...

        property string markerId
        property int noteId
        property alias noteText: editArea.text

...

}
```

En considérant que c'est le composant page qui permet de créer des notes, ce composant doit aussi savoir avec quel markerId la note est associée. Quand une nouvelle note est crée (et pas récupérée depuis la base de données), la Page devrait définir la propriété markerId de l'objet note.

Utilisons une fonction Javascript pour réaliser cela:

```
//Page.qml

Item{
id:root
…
        //Cette propriété est utilisée pour stocker les objets note dans la base de données.

        property string markerId
        …
        // cette fonction Javascript est utilisée pour créer des objets notes qui ne sont pas
        //issus de la base de données, il faut définir la propriété markerId de la note

        function newNote() {
        //On appelle newNoteObject et on passe en paramètres
        //le jeu d'arguments où le markerId est défini
        newNoteObject( { "markerId": root.markerId } )
        }
        …
```

Précédemment, dans main.qml nous avons utilisé la focntion newNoteObject(), mais comme expliqué ci-dessus, cela ne nous permet plus de faire ce que nous voulons et nous allons devoir la remplacer par la fonction newNote().
Nousa vons une propriété markerId pour le composant Page qui est utilisée pour définir le markerId des objets Notes crées. lorsque nous créons des objets Page dans le composant PagePanel, nous devons nous assurer que la propriété markerId de la page est bien configurée.

```
// PagePanel.qml

Item {
id:root
…

//Création de 3 objets page liés pour remplir le parent
        Page { id: personalpage; anchors.fill: parent; markerId: "personal" }
        Page { id: funpage; anchors.fill: parent; markerId: "fun" }
        Page { id: workpage; anchors.fill: parent; markerId: "work" }

…
}
```

Jusqu'à présent, nous nous somme assurés que:

-Le lien entre les notes et les pages est correct dans la perspective d'avoir une base de données relationnelle.
-Un objet Note a un ID unique qui appartient à un identifiant page par un marqueur ID
-Les valeur de ces propriétés sont définies correctement

Maintenant, regardons comment charger et stocker de nouvelles notes.

4.2.2 Nouvelle bibliothèque JavaScript de type "Stateless"

Pour faciliter le développement, il serait pertinent de créer une interface JavaScript qui interagit avec la base de données et nous fournisse des fonction que nous pourrions appeler dans notre code QML.

Dans QTCreator, nous avons créé un nouveau fichier JavaScript: noteDB.js, et nous avons pris soin de cocher l'option "Stateless Library". .L'idée est de faire en sorte que notre fichier noteDB.js soit une bibliothèque qui nous fournisse des fonctions qui vont nous aider dans la gestion de la base de données. Ainsi, il n'y aura qu'une seule instance de ce fichier chargée et utilisée pour chaque composant QML qui utilise noteDB.js. Cela permet aussi d'être sur qu'il n'y a qu'une seule variable globale pour stocker l'instance de la base de donnée: _db.

Note: Il peut s'avérer utile d'utiliser des fonction JavaScript qui ne sont pas "stateless" lorsqu'elles sont importée dans des composants QML pour effectuer des opérations sur ce composant, et que toutes les variables sont valides dans leurs contexte uniquement. Chaque import crée des instances séparées du fichier JavaScript.

noteDB.js devrait apporter les fonctionnalités suivantes:

-Ouvrir/Créer une instance de base de données locale
-Créer les tables nécessaires
-Lire des notes depuis la base de données
-Supprimer toutes les notes

Nous allons voir plus en détails comment les implémentés les fonctions de noteDB.js lorsqu'il s'agit de de charger/sauvegarder des données dans la base. Maintenant, considérons que les fonctions suivantes sont implémentées:

• function openDB() - Crée une base de données s'il n'en existe aucune, sinon, elle en crée une.

• function createNoteTable() - Crée la table note si elle n'existe pas. Cette fonction est appelée par la fonction openDB() uniquement

• function clearNoteTable() - Supprime toutes les éléments de la table note

• function readNotesFromPage(markerId) - Cette fonction lit toutes les notes relatives au markerId passé en argument, et elle retourne les données

• function saveNotes(noteItems, markerId) - Utilisée pour sauvegarder des objets note dans la base de données. L'argument noteItems est une liste d'objets note, et markerId représente la page à laquelle correspond l'objet note.

Note: Comme toutes ces foncitons utilisnet l'API QT Quick Local Storage, il faut importer QtQuick.LocalStorage 2.0 en tant qu'SQL au début de noteDB.js

4.2.3 Charger et stocker des Notes

Maintenant que nous avons implémenté noteDB.js, nous allons utiliser ses fonctions pour stocker et charger des données.

Il faut s'habituer à initialiser ou ouvrir la base de données dans le fichier QML principal. De cette façon, on peut utiliser les fonctions de noteDB.js sans avoir à recharger la base de données.

On importe noteDB.js et Qtquick.LocalStorage 2.0 dans le fichier main.qml, mais le vrai problème est de savoir quand appeler la fonction openDB(). QML permet l'émission de signaux onCompleted() et onDestroy() qui sont émis lorsque le composant est totalement chargé ou détruit.

```
//main.qml
import QtQuick 2.0
import "noteDB.js" as NoteDB
…
//Ce signal est émis lorsque le chargement du composant est temriné
Component.onCompleted: {
        NoteDB.openDB()
```

```
}
…
```

Voici l'implémentation de la fonction openDB. Elle fait appel à la fonction openDatabaseSync() pour créer la base de données. Après cela, elle applle la fonction createNoteTable() pour créer une nouvelle table note

```
//noteDB.js
…
function openDB() {
        print("noteDB.createDB()")
        _db = openDatabaseSync("StickyNotesDB","1.0",
                "The stickynotes Database", 1000000);
         createNoteTable();
 }

function createNoteTable() {
        print("noteDB.createTable()")
         _db.transaction( function(tx) {
                tx.executeSql(
                        "CREATE TABLE IF NOT EXISTS note
                       (noteId INTEGER PRIMARY KEY AUTOINCREMENT,
                        x INTEGER,
                        y INTEGER,
                        noteText TEXT,
                        markerId TEXT)")
         })
 }
…
```

Dans le fichier main.qml, nous initialisations la base de données donc il n'y aura pas de soucis pour charger nos notes du composant page.Ci dessus, nous avons mentionné la fonction readNotesFromPage(markerId) qui retourne une liste de tableaux de données (que l'on nomme dictionnaire dans le domaine des scripts) et chaque case du tableau représente un élément de la base de donnée avec comme information les données de la note.

```
//noteDB.js
…
function readNotesFromPage(markerId) {
        print("noteDB.readNotesFromPage() " + markerId)
        var noteItems = {}
         _db.readTransaction( function(tx) {
                 var rs = tx.executeSql(
                        "SELECT FROM note WHERE markerId=?
                        ORDER BY markerid DESC", [markerId] );
                var item for (var i=0; i< rs.rows.length; i++) {
                        item = rs.rows.item(i)
                        noteItems[item.noteId] = item;
                }
        })
        return noteItems
```

```
}
```

La page composants va lire les notes et créer les objets note

```
//Page.qml

…
//Quand le compossant ets chargé, appele la fonction loadNotes()
//pour charegr les notes à aprtir de la base de données
Component .onCompleted: loadNotes()

//Une fonction JavaScript qui lit les données des  notes depuis la base de données
function loadNotes() {
        var noteItems = NoteDB.readNotesFromPage(markerId)
        for (var i in noteItems) {
                newNoteObject(noteItems[i])
        }
}
…
```

Nous pouvons voir que la fonction newNoteObject() définie précédemment dans page.qml prend un tableau de donénes en argument qui sont en fait des valeurs pour les propriétés x, y, noteText et noteID.

Note: Remarquez que dans la table note, les champs sont les mêmes que pour la propriété note. Cela nous facilite la tâche lors de l'utilisation de la base de données pour créer une note.

maintenant que nous avons implémenté une fonction pour charegr les donnes des objets note depuis la base de donénes. L'étape suivante est d'implémenter une fonction qui permet d'enregistrer des notes dan la BD (Base de Données). Dans notre code , nous savons que c'est le PagePanel qui créer les objets Page. Donc il devrait pouvoir acceder à chacune des pages et appeler saveNote() pour sauvegarder l'ensemble des notes dans la BD.

```
////noteDB.js
…

function saveNotes(noteItems, markerId) {
        for (var i=0; i<noteItems.length; ++i) {
                var noteItem = noteItems[i]
                _db.transaction( function(tx) {
                        tx.executeSql(
                                "INSERT INTO note (markerId, x, y, noteText)
                                VALUES(?,?,?,?)",
                                [markerId, noteItem.x, noteItem.y, noteItem.noteText]);
                })
        }
}
```

Dans un premier temps, on définit un alias d'une propriété qui va afficher le objets note, qui sont des enfants du container crée dans le composant page:

```
//Page.qml
Item { id: root
 …
        // Cette propriété est utilisée par le composant PagePanel
        // pour récupérer toutes les notes d'une page et les
        // stocker dans la BD
        property alias notes: container.children
…

 }
```

Dans le PagePanel, on implémente la fonctionnalité pour sauvegarder des données dans la BD.

```
// PagePanel.qml

…

Component.onDestruction: saveNotesToDB()
// Une fonction JavaScript qui sauvegarde toutes les notes des pages
function saveNotesToDB() {
        // On nettoie la BD avant de la compléter
        NoteDB.clearNoteTable();
        // On stocke les notes pour chaque pages
        NoteDB.saveNotes(personalpage.notes, personalpage.markerId)
        NoteDB.saveNotes(funpage.notes, funpage.markerId)
        NoteDB.saveNotes(workpage.notes, workpage.markerId)
}

...
```

Afin de réduire la complexité de notre code, on supprime toutes les données de la base de données avant d'enregistrer nos notes. Cela permet d'éviter d'avoir un code qui s'occupe de mettre à jour la BD.
A la fin de ce chapitre, les utilisateurs sont capables de créer et supprimer des notes et l'application va tout sauvegarder.

Et la suite?
Le chapitre suivant présente quelques animations et comment les implémenter pas à pas.

Chapitre 5
Améliorer le style et l'utilisation visuelle

L'interface utilisateur de l'application noteApp est complète en termes de fonctionnalités et dans l'interaction avec l'utilisateur. Ainsi, il existe diverses façons d'améliorer le rendu visuel de l'IG (interface Graphique). QML est un langage déclaratif mais n'oublie pas d'avoir des animations et des transitions fluides entre le différents éléments.

Dans ce chapitre, nous allons vous guider pas à pas pour rajouter des animations et rendre NoteApp plus fluide.QML propose des types dédiés à l'implémentation d'animations. Ce chapitre traite de ces nouveaux types et de leurs utilisations pour rendre l'interface utilisateur plus fluide.

Note: Vous trouverez l'implémentation relative à ce chapitre dans le fichier zip.

En résumé, ce chapitre va couvrir les sujets suivants:
-Introduction des concepts sur les animations et les transitions en QML
-Les nouveaux types en QML comme *Behavior*, *Transition* et autres *Animation elements*
-L'amélioration des composants NoteApp en QML en utilisant des animations.

Ce chapitre comporte les étapes suivantes:

5.1 Animation de la NoteToolbar

Voyons comment améliorer le composant Note et rajouter un comportement basé sur l'interaction de l'utilisateur. Le composant Note a une barre d'outils avec un outil "supprimer" pour supprimer une note. De Plus, cette barre d'outil permet de déplacer la note sur l'écran en faisant un cliqué-glissé. Une amélioration pourrait être de rendre visible le bouton supprimer seulement quand l'utilisateur en a besoin. Par exemple en rendant l'outil Delete (pour supprimer) seulement quand la barre d'outils est hovered [ndt:je ne sais pas ce que cela signifie pour le logiciel] et l'idéal serait d'utiliser des fondues comme transitions.
QML propose plusieurs approches pour implémenter cela en utilisant les types *Animation* et *Transition*. Dans ce cas spécifique, nous allons utiliser le type Behavior. Nous expliquerons pourquoi nous choisissons ce type plus tard.

5.1.1 Les types Behavior et NumberAnimation

Dans le composant NoteToolbar, on utilise le type Row pour afficher l'outil Delete, donc en modifiant l'opacité du Row de l'outil Delete, cela modifiera l'opacité du bouton supprimer.

Note: La valeur de la propriété opacity se propage de parents à enfant.

Le type behavior vous aide a définir le comportement de l'objet comme le montre le code ci-dessous:

```
// NoteToolbar.qml
…
MouseArea {
        id: mousearea
```

```
        anchors.fill: parent
        //On met la propriété hoverEnabled sur vrai
        //cela permet a la MouseArea d'avoir les
        //evènement de hover
        hoverEnabled: true
}

//Utilisation d'un élément Row  pour afficher l'outil quand on utilise la notebar
Row {
        id: layout
         layoutDirection: Qt.RightToLeft
        anchors {
                verticalCenter: parent.verticalCenter;
                left: parent.left;
                right: parent.right leftMargin: 15;
                rightMargin: 15
         }
         spacing: 20

//L'opacité dépend de si la souris se trouve dans la MouseArea

opacity: mousearea.containsMouse ? 1 : 0

//On utilise un element behavior pour spécifier le comportement de l'élément layout quand
//l'opacité change
Behavior on opacity {
                //Utilisation du NumberANimation pour animer la valeur de l'opacité
                //de 350 ms
                NumberAnimation { duration: 350 }
        }
}
…
```

Comme vous pouvez le voir dans le code ci-dessus, nous activons la propriété hoverEnabled * de MouseArea type en acceptant les événements de survol de la souris.

Ensuite, nous basculons l'opacité du type Row à 0 si le Le type Mousearea n'est pas plané et à 1 sinon. La propriété containsMouse de MouseArea est utilisé pour décider de la valeur d'opacité pour le type Row. Donc le comportement type est créé à l'intérieur du type Row pour définir son comportement en fonction de son opacité. Lorsque la valeur d'opacité change, le NumberAnimation est appliqué.

Le NombreAnimation type applique une animation basée sur des changements de valeur numérique, de sorte que nous utilisons-la pour la propriété opacity de la Row pour une durée de 350 millisecondes.

Et ensuite?

5.2 Utilisation des états et des transitions

Dans l'étape précédente, nous avons vu une approche pratique pour définir des animations simples basées sur les changements de propriété, en utilisant les types Behavior et

NumberAnimation . Certes, il y a des cas dans lesquels les animations dépendent d'un ensemble ou des changements de propriété qui pourraient être représentés par un état. Voyons comment nous pouvons améliorer l'interface utilisateur de NoteApp *.
Les éléments Marker semblent être statiques lorsqu'il s'agit d'interaction avec l'utilisateur. Et si nous pouvions ajouter des animations basées sur différents scénarios d'interaction de l'utilisateur? De plus, nous souhaitons rendre le marqueur actif actuel et la page courante plus visibles pour l'utilisateur.

5.2.1 Animation des éléments du marqueur

Si nous sommes sur le point de résumer les scénarios possibles pour améliorer l'interaction de l'utilisateur avec Marker articles, les cas d'utilisation suivants sont représentés:

• L'élément Marqueur actif en cours devrait être plus visible. Un marqueur devient actif quand l'utilisateur clique dessus. Le marqueur actif est légèrement plus grand, et il pourrait glisser de gauche à droite (juste comme un tiroir).

• Lorsque l'utilisateur passe un marqueur avec la souris, le marqueur glisse de gauche à droite mais pas autant qu'un marqueur actif glisserait.

Considérant les scénarios mentionnés ci-dessus, nous devons travailler sur le Marker et MarkerPanel Composants. En lisant la description ci-dessus sur le comportement souhaité (l'effet de glissement de gauche à à droite), la pensée immédiate que je reçois est de changer la propriété x de l'élément Marqueur tel qu'il représente la position de l'article sur l'axe X. De plus, comme l'élément marqueur devrait être au courant s'il est le marqueur actif actuel, une nouvelle propriété appelée active peut être introduite. Nous pouvons introduire deux états pour le composant marqueur qui peut représenter le comportement représenté au dessus:

• hovered - mettra à jour la propriété x du marqueur lorsque l'utilisateur le survole en utilisant la Souris

• selected - mettra à jour la propriété x du marqueur lorsque le marqueur devient actif un, c'est-à-dire quand il est cliqué par l'utilisateur.

```
//Marker.qml
…

//Cette propriété indique si si le marqueur
//est actif ou non (au début il est sur faux)
property bool active: false

//Création des deux états qui représentent les changements de propriétés
states: [
        //L'état hovered est activé lorsque l'utilisateur fait glisser la fenêtre
         State {
                name: "hovered"
                //Cette condition rend le marqueur actif
                when: mouseArea.containsMouse && !root.active
                PropertyChanges { target: root; x: 5 }
        },
        State {
```

```
                name: "selected"
                when: root.active
                PropertyChanges { target: root; x: 20 }
        }
]
//liste des transitions qui s'appliquent au changement d état
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 } } ]
…
```

Nous avons donc déclaré deux états qui représentent les changements de propriété respectifs en fonction du comportement de l'utilisateur Chaque état est lié à une condition exprimée dans la propriété when. Le type Transition est utilisé pour définir le comportement de l'élément lors du passage d'un état à un autre. Autrement dit, nous pouvons définir diverses animations sur les propriétés qui changent quand un état devient actif. Dans le composant MarkerPanel, nous devons définir la propriété active de l'élément Marqueur vrai quand il est cliqué. Reportez-vous à MarkerPanel.qml pour le code mis à jour.

5.2.2 Ajout des transitions à la PagePanel

Dans le composant PagePanel, nous utilisons des états pour gérer la navigation entre les pages. Ajouter les transitions viennent naturellement à l'esprit. Comme nous changeons la propriété d'opacité dans chaque état, nous pouvons ajouter Transition pour tous les états qui exécutent une NumberAnimation sur les valeurs d'opacité pour créer l'effet de fondu et de fondu.

```
//PagePanel.qml
…
//Création d'une liste de transitions pour les différents états du PagePanel
transitions: [
        Transition {
                //Exécute la même transition pour tous les états
                from:"     ";to "*"
                NumberAnimation { property: "opacity"; duration: 500 }
        }
]
```

Note: La valeur de l'opacité se propage aux enfants

Et ensuite? Dans la prochaine étape, nous apprendrons comment améliorer l'interface utilisateur et voir ce qui pourrait être fait de plus.

Chapitre 6

D'autres améliorations

À ce stade, nous pouvons considérer que les fonctionnalités NoteApp * sont complètes et que UI correspond également aux exigences de NoteApp. Néanmoins, il y a toujours de la place pour d'autres améliorations, qui pourraient être mineures mais qui rendent l'application plus polie et prête pour le déploiement.

Dans ce chapitre, nous allons passer en revue les améliorations mineures implémentées pour NoteApp*, mais également suggérer de nouvelles idées et fonctionnalités à ajouter.

Certainement, nous voudrions encourager tout le monde à prendre la base de code NoteApp * et à la développer davantage et peut-être refaire toute UI et introduire de nouvelles fonctionnalités.

Noté : Vous trouverez l'implémentation relative à ce chapitre dans le fichier zip fourni dans le
section get-source-code.

Voici une liste des principaux points abordés dans ce chapitre:
- Plus de Javascript utilisé pour augmenter les fonctionnalités
- Travailler avec l'ordre z des éléments QML
- Utiliser de polices locales personnalisées pour l'application

Ce chapitre comporte les étapes suivantes:

6.1 Amélioration de la fonctionnalité de Note

Une fonctionnalité astucieuse pour Note serait d'avoir la note croître que plus de texte est entré. Disons que pour des raisons de simplicité, la note va croître verticalement à mesure que le texte est saisi, alors qu'il enveloppe le texte pour l'adapter à la largeur.

Le type Text[1] a une propriété paintedHeight[2] qui nous donne la hauteur réelle du texte peint à l'écran. En fonction de cette valeur, nous pouvons augmenter ou diminuer la hauteur de la note elle-même.

1http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-text.html
2http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-text.html#paintedHeight-prop

D'abord, définissons une fonction d'assistance JavaScript qui calcule la valeur de la propriété height[3] du type Item, qui est le type de niveau supérieur du 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
        }
}
…
```

Comme la fonction updateNoteHeight () met à jour la propriété height de la racine basée sur la propriété paintedHeight4 de editArea, nous devons appeler cette fonction lors d'une modification de paintedHeight.

Note : Chaque propriété a un signal notificateur qui est émis chaque fois que la propriété change

La fonction JavaScript updateNoteHeight () modifie la propriété height afin que nous puissions définir un comportement à l'aide du type Behavior[5].

```
// Note.qml
...
// defining a behavior when the height property changes
```

```
// for the root element
Behavior on height { NumberAnimation {} }
…
```

3http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-item.html#height-prop
4http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-text.html#paintedHeight-prop
5http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-behavior.html

Et après ?
L'étape suivante montrera comment utiliser la propriété z du type Item pour ordonner correctement les notes.

6.2 Notes de commande

Notes d'une Page ne sont pas conscientes de la note sur laquelle l'utilisateur travaille actuellement. Par défaut, tous les éléments de note créés ont la même valeur par défaut pour la propriété z6 dans ce cas, QML crée une pile de commandes par défaut, basée sur l'élément créé en premier.

Le comportement souhaité serait de changer l'ordre des Notes lors de l'interaction d'un utilisateur. Lorsque l'utilisateur clique sur la barre d'outils de note ou commence à modifier la Note, la Note en cours devrait apparaître et ne pas être sous d'autres Notes. Ceci est possible en changeant la valeur z pour qu'elle soit plus élevée que le reste des Notes.

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

La propriété activeFocus[7] devient true lorsque editArea a le focus d'entrée. Ainsi, il plus sécurisé de rendre la propriété z de la racine dépendante de l'activeFocus de l'editArea. Cependant, nous devons nous assurer que l'editArea a le focus d'entrée même si la barre d'outils est cliquée par l'utilisateur. Définissons un signal pressé () dans le composant NoteToolbar qui est émis lors d'une pression sur la souris.

```
// NoteToolbar.qml
...
// this signal is emitted when the toolbar is pressed by the user
signal pressed()
...
MouseArea {
id: mousearea
...
// emitting the pressed() signal on a mouse press event
onPressed: root.pressed()
}
…
```

Dans le gestionnaire de signal onPressed8 dans le type MouseArea, nous émettons le signal pressed () de la racine de NoteToolbar.

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

6http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-item.html#z-prop
7http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-item.html#activeFocus-prop
8http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-mousearea.html#onPressed-signal

```
// 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, on définit la propriété focus[9] de l'élément editArea sur true, afin que editArea reçoive le focus d'entrée.

Et après ?

L'étape suivante montre comment charger et utiliser un fichier de police local personnalisé pour NoteApp.

6.3 Chargement *Custom Font*

Il s'agit d'une approche très courante dans les applications modernes pour utiliser et déployer des polices personnalisées et ne pas dépendre des polices système. Pour NoteApp, nous voulons faire la même chose et nous utiliserons ce que QML offre pour cela.

Le type QL de FontLoader[10] vous permettra de charger des polices(Font) par nom ou via chemin d'URL. Comme la police chargée peut être largement utilisée dans l'ensemble de

l'application, nous recommandons de charger la police dans le fichier main.qml et de l'utiliser dans le reste des composants.

```
// main.qml
Rectangle {
        // using window as the identifier for this item as
        // it will 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')
        }
}
...
}
```

9http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-item.html#focus-prop
10http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-fontloader.html

Nous avons donc créé une propriété webfont pour l'élément de la fenêtre. Cette propriété peut être utilisée en sécurité dans le reste des composants. Exemple l'utilisation pour l'editArea dans le composant Note.

```
// Note.qml
...
// creating a TextEdit item
TextEdit {
id: editArea
font.family: window.webfont.name; font.pointSize: 13
...
}
```

Pour définir la police pour editArea, nous utilisons la propriété font.family[11]. De la fenêtre, nous utilisons son
propriété webfont pour obtenir le nom de la police à définir.

Et après ?
La prochaine étape vous guidera à travers les détails de la préparation de NoteApp* pour le déploiement.

Chapitre 7

Déploiement l'application NoteApp

Nous arrivons au point qu'on souhaite maintenant rendre l'application libre et déployable pour l'environnement typic du bureau. Comme l'explication dans le chaptire 1, on a utilisé le projet Qt Quick UI dans Qt Creator pour développer l'application NoteApp. C'est à dire *qmlscene* est utilisé pour charger le fichier *main.qml* et par conséquent, on aura *NoteApp* running.

D'abord, le plus simple pour rendre NoteApp libre, c'est de créer le package qui regroupe tous les sfichier qml, qmlscence et le simple script qui charge main.qml en utilisant qmlscence. On doit baser sur la documentation de chaque plate forme destop pour savoir comment créer le petit script. Par exemple, sur la plateform Linux, vous devrez utiliser le script bash pour cela. Tant dis que pour Window utiliser un simple ficher batch. Cette approche fonctionne bien comme étant si simple, mais il se peut que vous ne souhaitiez pas envoyer le code source car votre application est
en utilisant un code propriétaire. Cette application devrait être envoyé en format binaire dans lequel inclus tous les fichier qml.

Donc, on a besoin de créer le fichier exécutable pour NoteApp* qui est facile à utiliser et à installer. Dans ce chapitre, vous aller voir comment on peut-on installer l'application Qt Quick qui regroupe le fichier qml et les images dans le fichier binaire exécutable. De plus, on va voir aussi comment utiliser Qt Resource System (Système de ressource Qt) avec QML.

Note : Vous aller trouver une implémentation de ce chapitre dans le ficher zip dans la section get-source-code.

Les pas à suivre de ce chapitre sont ainsi :

7.1 Création l'application NoteApp Qt

le but est de créer le seul fichier binaire NoteApp exécutable que l'utilisateur va charger *NoteApp.*

Regardez comment on peut utiliser Qt créateur :

7.1.1 Création l'application NoteApp Qt

D'abord, on a besoin de créer Application Qt Quick utilisant Qt Creator et assuer on a choisit *Built-in elements only* (pour tous les plateformes) dans l'application Qt Quick. On nomme cette application noteapp.

Alors maintenant, nous avons un projet nouvellement créé à partir de l'assistant et nous remarquons que le projet qtquick2applicationviewer est généré pour nous. Le projet qtquick2applicationviewer généré est une application de base 'modèle' qui charge des fichiers QML. Cette application, très générique pour déployer des applications Qt Quick sur des appareils et des cibles, inclut plusieurs codes spécifiques à la plate-forme pour chacun de ces déploiements cibles. Nous ne passerons pas par ces parties spécifiques du code, car cela ne convient pas au but de ce guide.

Néanmoins, il y a certaines particularités du qtquick2applicationviewer* qui nous limiter pour réaliser ce que nous voulons. L'application s'attend à ce que le développeur envoie les fichiers QML avec le binaire exécutable. L'utilisation du système de ressources Qt devient impossible, mais vous verrez comment surmonter ce problème pendant que nous procédons.

Pour le projet noteapp*, il existe un fichier source, main.cpp. Dans le fichier main.cpp, vous verrez
comment l'objet viewer, la classe QtQuick2ApplicationViewer, est utilisé 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();
}
```

A noter, il existe le simple fichier main.qml généré par wizard application Qt Quick qui va être remplacé par le fichier main.qml qu'on a crée pour NoteApp*.

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

noteapp- racine de répertoire de projet noteapp

- qml – ce répertoire contient tous les fichier QML
- qtquick2applicationviewer – l'application générée utiliser pour charger les fichiers QML
- noteapp.pro – ficher du projet
- main.cpp – le fichier C++ où application Qt est créé

Nous devons copier nos fichiers QML, y compris les répertoires des polices et des images, dans le répertoire qml du projet noteapp* nouvellement créé. Qt Creator identifie les modifications apportées au dossier du projet

ajouter le nouveau fichier dans le project view. Si ce n'est pas le cas, cliquer droit sur le projet et choisir Add Existing Files pour ajouter un nouveau projet.

Note : Assurez-vous d'ajouter tous les fichiers existantes incluant les images du fichier nodeDDB.js

Dans ce projet, commençons par Build et Run le projet et regarder si tout fonctionne bien avec la création de projet. Avant Build le projet noteapp*, assurer qu'on a bien réglé tous les paramètres dans notre projet. Référencer la section Configure projets dans la documentation Qt Creator.

Une fois l'application est compilée sans erreurs, un fichier exécutable en binaire institulant noteapp devrait créer dans la
racine du répertoire du projet. Si vous avez configuré Qt correctement, vous devez pouvoir Run le fichier en cliquant dessus.

7.1.2 Utilisation Qt Resource System pour stocker les fichier QML et images.
On a créé le fichier exécutable qui charge le fichier QML pour l'application pour lancer. Vous verrez dans le fichier main.cpp, l'objet viewer charge le fichier main.qml en passant le chemin relatif de ce fichier. En outre, on ouvre le fichier noteapp.pro pour comprendre le déploiement et compiler les réglages, on remarque la première ligne

```
# Add more folders to ship with the application, here  (ajouter plus de répertoires pour envoyer avec l'application ici)
folder_01.source = qml/noteapp
folder_01.target = qml
DEPLOYMENTFOLDERS = folder_01
….
```

On dirait qu'il est prévu d'expédier les fichiers QML avec le fichier exécutable, mais ce n'est pas ce que nous aimerions réaliser.

Qt fournit un Resource System3 assez intuitif qui fonctionne de manière transparente avec QML. Nous devons créer un fichier de ressources noteapp.qrc pour le projet noteapp * root afin que nous puissions ajouter notre QML

et des fichiers d'image. Reportez-vous à la documentation Création d'un fichier de ressources4 dans Qt Creator pour les étapes détaillées.

Nous devons appliquer des modifications mineures à noteapp.pro et à main.cpp afin de pouvoir utiliser le fichier de ressources nouvellement créé, noteapp.qrc.

Nous commençons par commenter les premières lignes du fichier 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, on voit que les fonctions
QtQuick2ApplicationViewer::setMainQmlFile()

sont appelées par le chemin relatif vers 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 QQuickView5, qui est une classe pratique pour le chargement et l'affichage des fichiers QML. La fonction QtQuick2ApplicationViewer :: setMainQmlFile () n'est pas optimisé pour l'utilisation des ressources car il ajuste le chemin du fichier QML avant d'appeler la fonction setSource ()6.

L'approche la plus simple pour surmonter cela serait d'appeler directement setSource () 7 sur l'objet viewer dans le fichier main.cpp, mais cette fois, nous passons le main.qml dans le fichier de ressources.

```
// 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();
}
```

7.1.3 Définition d'une icône et d'un titre d'application

Une amélioration graphique fortement recommandée consiste à définir une icône pour l'application,
qui identifie de manière unique votre application lorsqu'elle est déployée sur une plate-forme de bureau.

Dans le dossier noteapp *, vous avez peut-être remarqué quelques fichiers PNG et un fichier SVG.
Ces fichiers d'image seront utilisés pour définir l'icône de l'application en fonction de la taille de l'icône
puisque nous pouvons avoir des icônes 64x64 ou 80x80 ou vectorisées.

Pour plus de détails sur la façon dont ces fichiers d'icônes sont déployés sur diverses plates-formes,

5http://qt-project.org/doc/qt-5.0/qtquick/qquickview.html
6http://qt-project.org/doc/qt-5.0/qtquick/qquickview.html#source-prop
7http://qt-project.org/doc/qt-5.0/qtquick/qquickview.html#source-prop

vous devez regarder le fichier qtquick2applicationviewer.pri. Vous pouvez trouver des informations détaillées sur les icônes d'application dans la documentation de référence Comment définir l'application Icon8 Qt. Nous devons appeler la fonction setWindowIcon () 9 sur viwer afin de définir l'icône pour 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 window title par défaut pour notre application et nous utiliserons la fonction setWindowTitle () 10.

```
// 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!"));
…
```

Le NoteApp* est maintenant prêt à être expédié et déployé sur diverses plates-formes de bureau.

7.1.4 Déployer NoteApp

NoteApp, c'est une application Qt typique, vous devez alors décider si vous souhaitez de lier statiquement ou dynamiquement à Qt. En plus, chaque plate-forme de bureau a des configurations de liaison spécifiques à prendre en compte.

Vous pouvez trouver des informations détaillées sur Deploying Qt Applications[11] documentation de référence pour chaque cible de bureau de déploiement.

Et après?

Un résumé de ce que nous avons appris dans ce guide du développeur

Chapitre 8 Leçon apprise et lectures suppléme,ntaires

Ce guide vous a montré comment créer une application en utilisant Qt Quick et comment la préparer pour un déploiement sur un environnement de bureau. Nous avons vu comment développer l'application NoteApp * étape par étape et nous avons appris différents aspects du langage QML et son potentiel pour développer des interfaces utilisateur fluides modernes tout en gardant le code propre et simple en appliquant diverses techniques de programmation. Nous avons appris certaines des meilleures pratiques d'utilisation de différents types de QML et couvert certains sujets intéressants tels que:

- Animations et États
- Utiliser JavaScript pour améliorer les fonctionnalités
- Gestion dynamique des objets QML
- Stockage local
- Rendre l'application prête à déployer

À présent, vous devriez avoir les connaissances et la confiance nécessaires pour améliorer NoteApp * avec des fonctionnalités et améliorations de l'interface utilisateur, et explorer d’avantage de fonctionnalités de QML et Qt Quick que nous n'avons pas eu l'occasion de couvrir dans ce guide. Qt Quick est une technologie à croissance rapide qui est adoptée par divers secteurs et industries de développement de logiciels. Il serait donc utile de consulter la page Documentation Qt pour obtenir la dernière mise à jour sur cette technologie.