Qt Quick Application Developer Guide for Desktop

Release 1.0

Digia, Qt Learning

February 28, 2013

Contents

1 About this Guide **2**
- 1.1 Why Would You Want to Read this Guide? . . . 2
- 1.2 Get the Source Code and the Guide in Different Formats . . . 3
- 1.3 License . . . 3

2 Prototyping and Initial Design **5**
- 2.1 Overview of the NoteApp Application . . . 5
- 2.2 Creating a QML Component for UI Elements . . . 8
- 2.3 Anchoring QML Items and Implementing the QML Components . . . 9
- 2.4 Using Repeater and Delegate to Create List of Markers . . . 12
- 2.5 Finalize the Prototype . . . 14

3 Implementing the UI and Adding Functionality **20**
- 3.1 Creating PagePanel Component . . . 20
- 3.2 Binding Marker Item with the Page Item . . . 23
- 3.3 Adding Graphics . . . 27

4 Managing Note Objects Dynamically **34**
- 4.1 Create and Manage Note Items . . . 34
- 4.2 Store and Load Data from a Database . . . 38

5 Enhancing the Look and Feel **46**
- 5.1 Animating the NoteToolbar . . . 46
- 5.2 Using States and Transitions . . . 48

6 Further Improvements **52**
- 6.1 Enhancing the Note Item Functionality . . . 52
- 6.2 Ordering Notes . . . 54
- 6.3 Loading a Custom Font . . . 55

7 Deploying the NoteApp Application **57**
- 7.1 Creating the NoteApp Qt Application . . . 57

8 Lesson Learned and Further Reading 63

Let's learn by example!

The goal of this guide is to make you familiar with best programming practices using Qt Quick for building applications with QML. A prerequisite to this guide is to have a solid understanding of the QML language, so do read :qt5:about it<qtquick/qtquick-applicationdevelopers.html> to find out what it offers. Throughout this guide, we'll walk you through various aspects and best practices of application development with QML and how to deploy the application to a typical Desktop environment. References to other information sources are provided to make it easy for you to deepen your understanding of QML programming.

CHAPTER 1

About this Guide

1.1 Why Would You Want to Read this Guide?

This guide provides an overview of QML and Qt Quick technology with regards to developing feature-rich applications that you can deploy onto various desktop platforms.

The focus is on Qt Quick and how to use it efficiently for writing entire applications without using C++. It guides you step-by-step from the initial development environment set up to project creation to a ready-to-deploy application. We have implemented a simple application (NoteApp*) that helps users manage daily notes.

There are several chapters consisting of multiple steps. Each step describes specific features of the application, the development approach and the detailed QML code used. The application covers various aspects such as advanced UI concepts including animations, database storage and Javascript usage for application logic.

The application will not look and feel like a typical or classical desktop application, as the common UI elements used in desktop applications such as toolbars, menus, dialogs, and so on, are not used. This application is inspired by the modern fluid UIs, but the deployable target is a desktop environment.

In order to make it easy to work with the code, there is a version of NoteApp* for each chapter with all the features implemented as described in that chapter. It is recommended to refer to that code while reading the contents of this guide.

At the end of this guide, you should be able to have a solid understanding of how to develop an application using QML and Qt Quick as a technology and learn practical use of the QML language.

A screenshot of the NoteApp* application that will be developed in this guide.

1.2 Get the Source Code and the Guide in Different Formats

A .zip file that contains the source code of each chapter is provided:

Source code[1]

The guide is available in the following formats:

- PDF[2]
- ePub[3] for ebook readers.
- Qt Help[4] for Qt Assistant and Qt Creator.

1.3 License

[1] http://releases.qt-project.org/learning/developerguides/qtquickdesktop/notezapp_src.zip
[2] http://releases.qt-project.org/learning/developerguides/qtquickdesktop/QtQuickApplicationGuide4Desktop.pdf
[3] http://releases.qt-project.org/learning/developerguides/qtquickdesktop/QtQuickApplicationGuide4Desktop.epub
[4] http://releases.qt-project.org/learning/developerguides/qtquickdesktop/QtQuickApplicationGuide4Desktop.qch

What's Next?

Next, you will start to prototype the NoteApp* application and find out the power that QML offers for prototyping.

CHAPTER 2

Prototyping and Initial Design

One of the major advantages of Qt Quick and QML is that it enables you to prototype quickly. We are considering the Prototyping* phase as our first step for developing the *NoteApp* application for two reasons. First, as mentioned in the previous chapter, QML gives us the power to prototype quickly so UI designers can sketch up initial UI screens to be evaluated in no time. Second, prototyping allows you to work closely with designers, so the UI concepts are implemented in several short iterative processes.

Later on, this prototype will be used as our basis for further development of the application. In this chapter, we will guide you through the application development phase, including some UI concepts, the feature-set, UI interaction flows, and some initial QML screens as part of our prototyping.

There will be an introduction to a few main QML concepts such as creating a QML Component and how to layout QML items.

Note: You will find the implementation related to this chapter in the zip file provided in the *get-source-code* section.

Here is a brief list of the main discussion points of this chapter:

- UI Concepts and the feature-set for the *NoteApp*
- Creating QML Component using Qt Creator
- Use of Anchor and Repeater QML types for laying out UI elements

This chapter has the following steps:

2.1 Overview of the NoteApp Application

The NoteApp* application is a Post-it note[1] application that helps users create notes and store them locally. It would be easier to manage notes if they belonged to a category, so let's consider

[1] http://en.wikipedia.org/wiki/Post-it_note

having three different categories that the notes can belong to. From a visual perspective, a category can be represented by an area where notes of the same category can be placed. Let's introduce the concept of Page*. A *Page* is an area where notes will be created and placed.

The user should be able to delete notes one by one as well as all of them at once. Notes can be moved freely by the user within a Page* area. For simplicity reasons, let's define three pages and identify each page by using *Marker*. Additionally, each marker can have a different color.

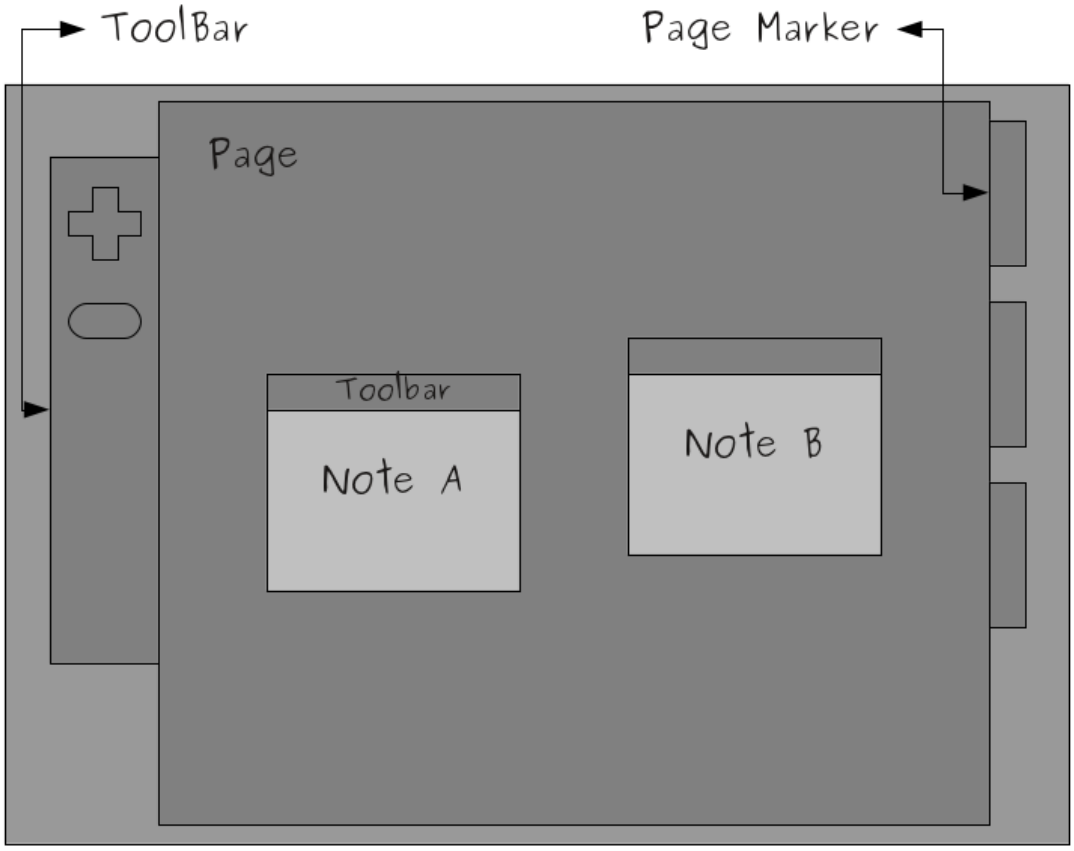
An interesting feature would be to store notes locally and perhaps have this done automatically so that we can avoid user prompts asking to save the notes.

To summarize the list of features:

- Create / Delete Note items
- Edit Notes and position them anywhere in the page
- Store notes locally
- Three different pages indicated by a page marker

2.1.1 UI Elements

Based on the requirements discussed earlier, let's start with the wire-frame design shown in the picture below. As there could be many possible designs for the *NoteApp*, let's consider the following one.



The picture above gives us a good idea of what the user expects to see from the UI perspective, but it also can help you identify possible UI elements and interactions that could be implemented later on.

2.1.2 UI Flows

As mentioned previously, there are three Pages* that can contain *Note* items. We can also see the *Marker* items on the right side and the toolbar on the left. The toolbar contains: the *New Note* tool for creating new notes and the *Clear All* tool to clear the entire page. *Note* items have a toolbar that can be used to drag notes within the page by moving the mouse while holding the left-click button down. Additionally, there is a *delete* tool in the note toolbar that enables the user to delete the note.

What's Next?

Next, we will work on identifying the QML components needed to implement our features and learn how to create them.

2.2 Creating a QML Component for UI Elements

Once we have properly defined the feature-set and UI concepts for the NoteApp* application and have identified basic UI elements, we can safely start to implement and prototype.

The prototype will be a very basic UI with no functionality whatsoever, but it will provide an overview of how the application might look when finished, as we go through implementation iterations.

In this step, you will find details about how to create your first Qt Quick UI using Qt Creator, but most importantly, how to identify and create a QML component.

2.2.1 Creating a Qt Quick UI Project in QtCreator

When working on the prototype phase, it is important to understand that creating a Qt Quick UI project with Qt Creator is the most recommended and efficient approach. In this way, prototyping, especially developing and testing each individual QML component is easier. Testing each newly created component individually is very important as you are able to spot issues right away, and using a Qt Quick UI project makes it easier.

Refer to the Creating a Qt Quick UI[2] with QtCreator for details on how to create a project.

Note: There is always a single QML file that is identified as the the main file to load and run the application. For NoteApp*, we have the *main.qml* file, which has been generated by Qt Creator.

2.2.2 Identifying QML Components as UI Elements

If you want to make an analogy with object-oriented programming, QML components could be considered as classes that are used to declare and instantiate objects. You could potentially write a simple application entirely in one big QML file, but that would certainly increase complexity and make code re-usability and maintenance quite difficult - sometimes even impossible.

QML component can be identified as a group of common UI elements. More often, it represents a single UI element with its predefined actions and properties.

Based on our UI Concept and Design, here is brief list of identified custom QML components that are obvious at first, but we may need more later on as we go to the next iteration.

Note: Each QML component sits in its own QML file (.qml) that has the same name as the component. For instance, *Note** component would be in a file named **Note.qml** .

- *Note* - that represents a note item
- *Page* - this component contains note items
- *Marker* - represents a page marker, enables users to switch between pages using makers

[2] http://qt-project.org/doc/qtcreator-2.6/quick-projects.html#creating-qt-quick-ui-projects

- *NoteToolbar* - the toolbar used on a note item to enable drag functionality and layout tools

---

Refer to Creating QML Components with QtCreator[3] for details on how to use QtCreator for creating the components mentioned above. We will go through each component in detail and learn how to implement them in the coming chapters and steps.

What's Next?

Next, we will see how to further enhance our defined components and start implementing the prototype UI.

2.3 Anchoring QML Items and Implementing the QML Components

The Rectangle QML type[4] is the natural choice to build UI blocks and the initial QML Component in the prototype phase. It is a visual type that has properties, which you can tweak to make it easier to prototype and test.

Note: It is a good practice to always give default geometry values to your defined components as it helps in testing.

Let's have a closer look at the code of our QML Components. At first, we start by implementing the *Note* component.

2.3.1 Note and NoteToolbar Component

First, as seen in the previous step, we have created the new QML files that we can use to implement the required components.

To match the given wire-frame design, the code for *NoteToolbar* could look as follows:

```
// NoteToolbar.qml

import QtQuick 2.0

// A Rectangle element with defined geometries and color.
Rectangle {
    id: root
    width: 100
    height: 62
    color: "#9e964a"
}
```

[3] http://qt-project.org/doc/qtcreator-2.6/quick-components.html
[4] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-rectangle.html

The *Note* component will have a toolbar UI element and the *NoteToolbar* component will be used for that. Additionally, there is a text input element for getting the input text from the user. We will use the TextEdit QML type[5] for this. In order to place these UI elements within the *Note* component, the *anchor* property is used. This property is inherited from the Item type[6], which is the base class that every other QML component inherits by default.

Refer to the Anchor-based Layout in QML[7] documentation for more details about anchoring and laying out QML components.

Note: Anchor-based layouts can not be mixed with absolute positioning

---

```
// Note.qml

import QtQuick 2.0

Rectangle {
    id: root
    width: 200
    height: 200
    color: "#cabf1b"

    // creating a NoteToolbar that will
    // be anchored to its parent on the top, left and right
    NoteToolbar {
        id: toolbar
        // height is required to be specified
        // since there is no bottom anchoring.
        height: 40

        // anchoring it to the parent
        // using just three anchors
        anchors {
            top: root.top
            left: root.left
            right: root.right
        }
    }

    // creating a TextEdit used for the text input from user.
    TextEdit {
        anchors {
            top: toolbar.bottom
            bottom: root.bottom
            right: root.right
            left: root.left
        }
        wrapMode: TextEdit.WrapAnywhere
    }
}
```

[5] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-textedit.html
[6] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-item.html
[7] http://qt-project.org/doc/qt-5.0/qtquick/qtquick-positioning-anchors.html

Warning: For performance reasons, you should only anchor a component to its siblings and direct parent.

2.3.2 Page

Once we have the *Note* component ready, let's work on getting the *Page* component with a couple of note items inside.

```
** $QT_END_LICENSE$
**
****************************************************************************/

// Page.qml

import QtQuick 2.0

Rectangle {
    id: root
    width: 600
    height: 400
    color: "#222525"

    // creating a Note item
    Note {
        id: note1
        // use x and y properties to specify
        // the absolute position relative to the parent
        x: 105; y: 144
    }

    Note {
        id: note2
        x: 344
        y: 83
    }
}
```

In Qt Creator, you can simply run the file above, which in fact will simply use *qmlscene** to load the *Page.qml*. The output result should look like this:

2.3.3 Marker

Same as with the rest of the components, the *Marker* component will also use a Rectangle type with predefined geometries. Later on, as described in the next chapter, we will show how the *Marker* component is used.

```
** $QT_END_LICENSE$
**
****************************************************************************/

// Marker.qml

import QtQuick 2.0

Rectangle {
    id: root
    width: 50
    height: 90
    color: "#0a7bfb"
```

What's Next?

In the next chapter, we will see how to use a Repeater QML type and a Column to manage a static list of markers.

2.4 Using Repeater and Delegate to Create List of Markers

Previously, we saw how to create QML components such as *Note*, *NoteToolbar*, *Page* and *Marker*, and how to place these QML components using anchors*.

Looking back at the previous chapters on design concepts, one thing we noticed is that there are three *Marker* elements laid out vertically. Using anchors* could also work as we can anchor UI elements with each other, but nevertheless it will increase the complexity of the code. QML

has other convenient approaches such as the layout and positioning types. The Column type[8] is one such type, enables arranging the UI elements one below the other in a column.

As we want to place the three *Marker* components within a Column type[9], let's use a nifty QML type called Repeater[10].

Now let's have a look at the code described above:

```
Column {
    id: layout
    // spacing property can set to let the item have space between them
    spacing: 10

    // a Repeater item that uses a simple model with 3 items
    Repeater {
        model: 3
        delegate:
            // using the Marker component as our delegate
            Marker { id: marker }
    }
}
```

In the code shown above, *Repeater* generates three QML components based on the model and will use a *delegate** to display them. As we want to have three *Marker* items, we simply use the *Marker* component as the delegate.

For more information about positioning, refer to Important Concepts In Qt Quick - Positioning[11] documentation.

Naturally the question "where in my qml files should I place the code shown above and how should I use it?" arises. Well, we need a separate QML Component for that which we'll call *MarkerPanel*. In short, *MarkerPanel* is simply a list of three *Marker* items that can easily be used as a UI element. We will know later on how.

Here is how the *MarkerPanel* component would look:

```
** $QT_END_LICENSE$
**
****************************************************************************/

// MarkerPanel.qml

import QtQuick 2.0

Rectangle {
    id: root
    width: 50
    height: 300

    // column type that anchors to the entire parent
```

[8] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-column.html
[9] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-column.html
[10] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-repeater.html
[11] http://qt-project.org/doc/qt-5.0/qtquick/qtquick-positioning-topic.html

```
Column {
    id: layout
    anchors.fill: parent
    spacing: 10

    Repeater {
        // just three Marker items
        model: 3
        delegate:
            Marker { id: marker }
    }
}
```

Note: It is always recommended to run and test components individually in the prototype phase because you can spot the problems right away.

If we run the *MarkerPanel* component using Qt Creator or load it with *qmlscene** then it should look like this:

What's Next?

In the next chapter, we will see how to use the components we've implemented so far to finalize our prototype.

2.5 Finalize the Prototype

We now have our QML components in place and they're ready to be used to build our prototype. Here is a list of the implemented components:

- *Note*
- *NoteToolbar*
- *Marker*

- *MarkerPanel*
- *Page*

It is very likely that more QML components might come up as we go along in later phases.

As previously mentioned, Qt Creator generates a *main.qml* file which is considered the main file to load in order to run NoteApp*. Therefore, we will start laying out our components inside the *main.qml* file to compose the prototype.

2.5.1 Composing the Prototype

Going back to the UI concepts and looking at the design provided, we start laying out the QML components. We have the panel of *Maker*, that is the *MarkerPanel* component placed on the right, and the *Page* component in the center. We haven't yet covered the toolbar so let's do that now.

The toolbar contains two tools: one for creating new note and one for clearing the page. For simplicity, we will not create a component for this, but rather define it inside the *main.qml* file.

The code could look something like this:

```
// using a Rectangle element to represent our toolbar
// it helps to align the column better with the rest of the components
Rectangle {
    id: toolbar

    // setting a width because there is no right anchoring
    width: 50

    color: "#444a4b"
    anchors {
        left: window.left
        top: window.top; bottom: window.bottom
        topMargin: 100; bottomMargin: 100
    }

    // using a Column type to place the tools
    Column {
        anchors { anchors.fill: parent; topMargin: 30  }
        spacing: 20

        // For the purpose of this prototype we simply use
        //a Repeater to generate two Rectangle items.
        Repeater {
            model: 2
            // using a Rectangle item to represent
            // our tools, just for prototype only.
            Rectangle { width: 50; height: 50;  color: "red" }
        }
    }
}
```

Now, we are ready to actually finalize our prototype. Here is how the *main.qml* file would look:

```
** $QT_END_LICENSE$
**
****************************************************************************/

// main.qml

import QtQuick 2.0

Rectangle {
    // using window as the identifier for this item as
    // it will the only window of the NoteApp
    id: window
    width: 800
    height: 600

    // creating a MarkerPanel item
    MarkerPanel {
        id: markerPanel
        width: 50
        anchors.topMargin: 20
        anchors {
            right: window.right
            top: window.top
            bottom: window.bottom
        }
    }

    // the toolbar
    Rectangle {
        id: toolbar
        width: 50
        color: "#444a4b"
        anchors {
            left: window.left
            top: window.top
            bottom: window.bottom
            topMargin: 100
            bottomMargin: 100
        }

        Column {
            anchors { fill: parent; topMargin: 30  }
            spacing: 20

            Repeater {
                model: 2
                Rectangle { width: 50;
                    height: 50;
                    color: "red"
                }
            }
```

```
        }
    }

    // creating a Page item
    Page {
        id: page1
        anchors {
            top: window.top
            bottom: window.bottom
            right: markerPanel.left
            left: toolbar.right
        }
    }
}
```

The following screen shows how the prototype looks when either running it using Qt Creator or *qmlscene**:

2.5.2 Making Note Components Draggable

So far we have managed to get a very basic prototype that will be our basis for the NoteApp UI. An interesting UI functionality we can quickly add during the prototype phase is to enable the user to drag note items within the page. To achieve this, the MouseArea QML Type[12] has a grouped property called drag[13]. We will use the *drag.target** property by setting it to the **id** of our note component.

Considering that the user should use the *NoteToolbar* to drag a note, the MouseArea* type should be inside the *NoteToolbar* component. The *NoteToolbar* component handles the dragging operation by the user, so we should set the drag.target[14] to the *Note* component.

[12] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-mousearea.html
[13] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-mousearea.html#drag.target-prop
[14] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-mousearea.html#drag.target-prop

To achieve this, we need to allow *NoteToolbar* used inside the *Note* component to bind the drag.target[15] property of *MouseArea* with the *Note* component's *id**. QML provides Property Aliases[16] to enable this.

Let's take the *NoteToolbar* component and create a property alias for the *drag** grouped property of *MouseArea*:

```
** $QT_END_LICENSE$
**
****************************************************************************/

// NoteToolbar.qml

import QtQuick 2.0

Rectangle {
    id: root
    width: 100
    height: 62
    color: "#9e964a"

    // declaring a property alias to the drag
    // property of MouseArea type
    property alias drag: mousearea.drag

    // creating a MouseArea item
    MouseArea {
        id: mousearea
        anchors.fill: parent
    }

}
```

In the code shown above, we see the *drag** property alias for *NoteToolbar*, which is bound to the **drag** property of *MouseArea*, and now we will see how to use that in our *Note* component.

```
** $QT_END_LICENSE$
**
****************************************************************************/

// Note.qml

import QtQuick 2.0

Rectangle {
    id: root
    width: 200
    height: 200
    color: "#cabf1b"
```

[15] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-mousearea.html#drag.target-prop
[16] http://qt-project.org/doc/qt-5.0/qtqml/qtqml-syntax-objectattributes.html#property-aliases

```
    // creating a NoteToolbar item that
    // will be anchored to its parent
    NoteToolbar {
        id: toolbar
        height: 40
        anchors {
            top: root.top
            left: root.left
            right: root.right
        }

        // using the drag property alias to
        // set the drag.target to our Note item.
        drag.target: root
    }

    // creating a TextEdit item
    TextEdit {
        anchors {
            top: toolbar.bottom
            bottom: root.bottom
            right: root.right
            left: root.left
        }

        wrapMode: TextEdit.WrapAnywhere
    }
}
```

Detailed information on property bindings in QML can be found on the Property Binding[17] documentation page.

What's Next?

Next, we will start implementing the UI and basic functionality based on the prototype.

[17] http://qt-project.org/doc/qt-5.0/qtqml/qtqml-syntax-propertybinding.html

CHAPTER 3

Implementing the UI and Adding Functionality

The initial prototype has introduced the basis of our UI and concepts of NoteApp* functionality, and helped us identify which QML Component is needed.

Based on the prototype, we will try to build a working application reaching towards a more complete UI with some working functionality. Now, we will start composing the application's UI using the components we have implemented so far.

This chapter will cover detailed steps about using graphics with QML, images and backgrounds, enhancing the UI, and also about adding functionality using Javascript. You will have a chance to dive deeper into some QML types and also get introduced to new ones while slightly increasing the level of code complexity as we continue to implement functionality.

Note: You will find the implementation related to this chapter in the zip file provided in the *get-source-code* section.

These are the main points covered in this chapter:

- Managing *Page* elements by introducing the *PagePanel* component and by using the *Repeater* element
- Using graphics with QML
- Inline Javascript code for enhancing the UI and adding functionality
- Diving deeper into the QML types that we are currently using

This chapter has the following steps:

3.1 Creating PagePanel Component

To keep things simple, we have only created one page* item using the *Page* component. We've also anchored it to its parent together with the rest of the items. However, the *NoteApp* con-

cepts and requirements demand three different pages and the user should be able to navigate to them using *markers* as well. We have seen how the *MarkerPanel* component helped us create and layout for three *Marker* items, so let's use the same approach for our *Page* items, thus implementing the *PagePanel* component.

3.1.1 Using an Item Type

Before we go any further with our new component, it is important to understand why using the Rectangle type[1] as a top level type in a component should be avoided from now on. The main reason we've been using the *Rectangle* element is because it helped us get visual results quickly and this is what the prototype phase requires.

Once prototyping is done, however, it would make more sense to replace the *Rectangle* type with the *Item* type wherever possible, especially when considering to use graphics as backgrounds or as UI elements, which we will see in the coming chapters.

The *Page* component would look like this:

```
import QtQuick 2.0

// Page.qml

Item {
    id: root
    ...
}
```

Warning: From now on, it is considered that the top level element for our components is an Item QML type. Refer to the source code of each chapter.

3.1.2 Using States

To continue with the implementation of the *PagePanel* component, we see that the main difference between *PagePanel* and *MarkerPanel* is that three *Marker* items should always be visible, but there should only be one *Page* item visible at a time for the *PagePanel*. This will depend on which *Marker* item the user clicks on.

There could be several ways to achieve this behavior. One would be to have an inline Javascript function that toggles the visibility of the *Page* item according to the current *Marker* clicked by the user.

In NoteApp*, we have used the State type[2] to implement the intended behavior. The *PagePanel* component will have three states and each of them is bound to make one *Page* visible. So navigating through pages will be a matter of setting the respective state for the *PagePanel*.

First, in the *Page* component, we need to set the *opacity* property to *0.0* as the default value. This is to make the pages invisible initially and make them visible based on the respective state change. The following sections describes how this is done:

[1] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-rectangle.html
[2] http://qt-project.org/doc/qt-5.0/qml-qtquick2-state.html

```
// Page.qml

Item {
    id: root
    opacity: 0.0
    ...
}
```

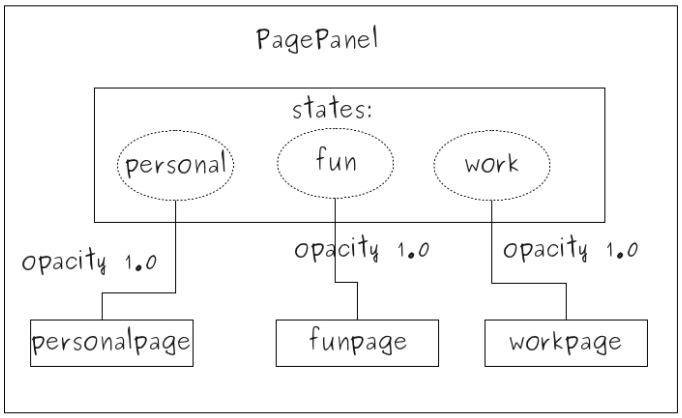
Once we have created the *PagePanel.qml* file, let's start to create the states and *Page* items. We need three states:

personal fun work

They will change the *opacity* property respectively for each of the following pages:

personalpage funpage workpage

The following image illustrates that the states are "connected" to the respective pages.



Here is the implementation in the *PagePanel* component:

```
import QtQuick 2.0

// PagePane.qml

Item {
    id: root

    // creating the list of states
    states: [

        // creating a state item with its corresponding name
        State {
            name: "personal"

            //the properties that are about to change for a defined target
            PropertyChanges {
                target: personalpage
                opacity:1.0
```

```
                restoreEntryValues: true
            }
        },
        State {
            name: "fun"
            PropertyChanges {
                target: funpage
                opacity:1.0
                restoreEntryValues: true
            }
        },
        State {
            name: "work"
            PropertyChanges {
                target: workpage
                opacity:1.0
                restoreEntryValues: true
            }
        }
    ]

    // creating three page items that are anchored to fill the parent
    Page { id: personalpage; anchors.fill: parent }
    Page { id: funpage; anchors.fill: parent }
    Page { id: workpage; anchors.fill: parent }
}
```

Note: Setting the *restoreEntryValues** property to true makes the changed property of the target to reset its default value, meaning that the *opacity* property of the page will be reset to *false* when the state changes.

Looking at the code shown above, we see the three *Page* items created and the states that change the *opacity* property of these items. In this step, we managed to create a new component named *PagePanel* that will help us switch between pages using the three available states.

What's Next?

In the next step, it will be shown how to change the state of the *PagePanel* item using a *Marker* item.

3.2 Binding Marker Item with the Page Item

Earlier we saw the implementation of the *PagePanel* component, which uses three states to toggle the opacity[3] property of the *Page* component. In this step, we will see how to use the *Marker* and the *MarkerPanel* components to enable page navigation.

During the prototype phase, we saw that the *MarkerPanel* component is quite basic and doesn't

[3] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-item.html#opacity-prop

have any functionality. It uses a Repeater type[4] that generates three QML Items and the *Marker* component is used as the delegate.

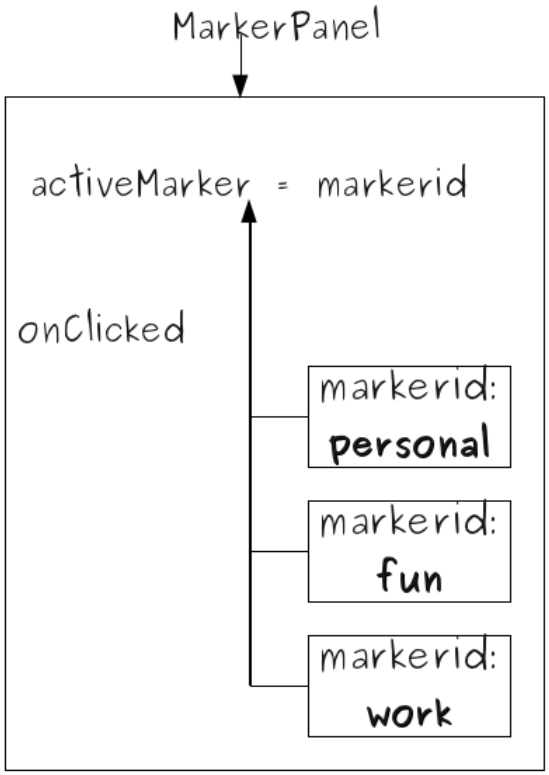
MarkerPanel should store the current active marker, which is the marker clicked by the user. Based on the active marker of *MarkerPanel*, *PagePanel* will update its state property. We need to bind the PagePanel* **state** property with the a new property of *MarkerPanel* that holds the current active marker.

Let's define a *string* property in *MarkerPanel* and call it *activeMarker**.

```
// MarkerPanel.qml

import QtQuick 2.0

Item {
    id: root
    width: 150; height: 450

    // a property of type string to hold
    // the value of the current active marker
    property string activeMarker: "personal"
    ...
```

We could have a *markerid** value stored, which we can use to uniquely identify the marker items. In this way, *activeMarker* will take the value of the **markerid** of the marker item that is clicked by the user.

[4] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-repeater.html



We have a *Repeater* element that generates three marker* items based on a model, so we can use a model to store our *markerid* values and use it in the *Repeater*.

```
// MarkerPanel.qml

import QtQuick 2.0

Item {
    id: root
    width: 150; height: 450

    // a property of type string to hold the value of
    // the current active marker
    property string activeMarker: "personal"

    // a list for holding respective data for a Marker item.
    property variant markerData: [
        { markerid: "personal" },
        { markerid: "fun" },
        { markerid: "work" }
    ]

    Column {
        id: layout
        anchors.fill: parent
        spacing: 5
```

```
        Repeater {
            // using the defined list as our model
            model: markerData
            delegate:
                Marker {
                id: marker
                // handling the clicked signal of the Marker item,
                // setting the currentMarker property
                // of MarkerPanel based on the clicked Marker
                onClicked: root.activeMarker = modelData.markerid
            }
        }
    }
}
```

In the code shown above, we set the *activeMarker* property in the *onClicked* signal handler. This means that we have defined a *clicked()* signal in the *Marker* component to get notified when the user performs a mouse click on the marker item.

Here is how the *Marker* component looks:

```
// Marker.qml

Item {
    id: root
    width: 50; height: 90
    signal clicked()

    MouseArea {
        id: mouseArea
        anchors.fill: parent
        // emitting the clicked() signal Marker item
        onClicked: root.clicked()
    }
}
```

Currently, we have achieved having a *PagePanel* component that manages the pages using the *state* property so the *MarkerPanel* component that helps identify the active marker, and therefore, toggles the visibility of the respective page by changing the *opacity* property of that page.

Let's see how to use the *activeMarker* property to update the state of *PagePanel* correspondingly.

In the *main.qml* file, where we already have a *Page* item and a *MarkerPanel* anchored, we will create and use a *PagePanel* item instead anchor that respectively.

```
// creating a MarkerPanel item

MarkerPanel {
    id: markerPanel
    width: 50
    anchors.topMargin: 20
    anchors {
        right: window.right
        top: window.top
```

```
        bottom: window.bottom
    }
}
...

// creating a PagePanel item
PagePanel {
    id: pagePanel
    // binding the state of PagePanel to the
    // activeMarker property of MarkerPanel
    state: markerPanel.activeMarker
    anchors {
        right: markerPanel.left
        left: toolbar.right
        top: parent.top
        bottom: parent.bottom
        leftMargin: 1
        rightMargin: -50
        topMargin: 3
        bottomMargin: 15
    }
}
```

In the code shown above, we see how the property binding[5] feature of QML helps in binding the *state** property with the **activeMarker** property. This means that whatever value **activeMarker** will have during user's selection, the same value is also assigned to the **state** property of the *PagePanel*, thus toggling the visibility of the respective page.

What's Next?

The next step will give us details on how to use graphics for our components and items to enhance our application's UI.

3.3 Adding Graphics

Due to the nature of QML, developers and designers are entitled to work closely throughout the development cycle. Nowadays, using graphics does make a big difference to the user experience and how the application is perceived by users.

QML encourages the use of graphics as much as possible while implementing the UI. The collaboration between the developers with graphic designers is easier and efficient with QML, as designers can test their graphics right away on basic UI elements. This helps designers to understand what a developer requires while developing new components, and also makes the application's UI more appealing and certainly easier to maintain.

Let's start adding graphics to our components.

[5] http://qt-project.org/doc/qt-5.0/qtqml/qtqml-syntax-propertybinding.html

3.3.1 Setting Background Images to Components

The BorderImage[6] type is recommended to be used in cases where you would like to have the image scaled, but keep its borders as is. A good use case for this type is background images that has a shadowing effect for your QML item. Your item might scale at some point but you would like to keep corners untouched.

Let's see how to set a *BorderImage* type as a background for our components.

Inside the *PagePanel.qml*, we add the *BorderImage* type:

```
// PagePanel.qml

...
BorderImage {
    id: background

    // filling the entire PagePanel
    anchors.fill: parent
    source: "images/page.png"

    // specifying the border margins for each corner,
    // this info should be given by the designer
    border.left: 68; border.top: 69
    border.right: 40; border.bottom: 80
}
...
```

We do the same for the *Note* component in the *Note.qml* file:

```
// Note.qml

...
BorderImage {
    id: noteImage
    anchors { fill: parent}
    source: "images/personal_note.png"
    border.left: 20; border.top: 20
    border.right: 20; border.bottom: 20
}

// creating a NoteToolbar item that will be anchored to its parent
NoteToolbar {
    id: toolbar
...
```

Warning: Make sure that the *BorderImage* type is used in the correct order within your component implementation, as the implementation order defines the appearance order. Items with the same z value will appear in the order they are declared. Further details regarding the stack ordering of items can be found in z property[a] documentation.

[a] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-item.html#z-prop

You may be wondering by now what would be the best approach to set a background for the

[6] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-borderimage.html

Marker items when these items are created inside the *MarkerPanel* component. The approach is already present in the *MarkerPanel* component.

There is already a *markerData* list that we are using as a model for the *Repeater* to create *Marker* items, and set *markerid* as the *activeMarker* when a marker item is clicked. We can extend the *markerData* to have a url path to an image and use the Image[7] type as the top-level type for the *Marker* component.

```
** $QT_END_LICENSE$
**
****************************************************************************/

// Marker.qml

import QtQuick 2.0

// The Image type as top level is convenient
// as the Marker component simply is a graphical
// UI with a clicked() signal.
Image {
    id: root

    // declaring the clicked() signal to be used in the MarkerPanel
    signal clicked()

    // creating a MouseArea type to intercept the mouse click
    MouseArea {
        id: mouseArea
        anchors.fill: parent

        // emitting the clicked() signal Marker item
        onClicked: root.clicked()
    }
}
```

So now let's see how to enhance the *MarkerPanel* component:

```
 // MarkerPanel.qml

 ...
// for the markerData, we add the img value pointing to the image url
property variant markerData: [
    { img: "images/personalmarker.png", markerid: "personal" },
    { img: "images/funmarker.png", markerid: "fun" },
    { img: "images/workmarker.png", markerid: "work" }
]

Column {
    id: layout
    anchors.fill: parent
    spacing: 5
```

[7] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-image.html

```
    Repeater {
        // using the defined list as our model
        model: markerData
        delegate:
            Marker {
            id: marker

            // binding the source property of Marker to that
            // of the modelData's img value.
            // note that the Marker is an Image element
            source: modelData.img

            // handling the clicked signal of the Marker item,
            // setting the currentMarker property
            // of MarkerPanel based on the clicked Marker
            onClicked: root.activeMarker = modelData.markerid
        }
    }
}
```

In the code shown above, we see how the *Marker* item's *source* property is bound to our *markerData* model's *img* value.

We use the *BorderImage* type to set a background for the *NoteToolbar* component and also for the top-level type in the *main.qml* file.

Note: Always consult with the graphics designer regarding an image's border margins and how these images should be anchored and aligned.

By running the *MarkerPanel* component in Qt Creator or by simply loading it using *qmlscene*, you should be able to get the following:

Now let's see how to use graphics to enhance the toolbar as proposed during the prototype phase.

3.3.2 Creating the Tool Component

Taking code re-usability into consideration, defining a new component to be used in the toolbar for the *New Note* and *Clear All* tools would make sense. This is why we have implemented a *Tool* component that uses an Image type[8] as a top-level type and handles the mouse click from the user.

The Image type[9] is often used as an UI element on its own, regardless of whether it's a static or animated image. It is laid out in a pixel perfect way and well defined by design requirements.

```
** $QT_END_LICENSE$
**
****************************************************************************/

// Tool.qml

import QtQuick 2.0

// Use Image as the top level type
Image {
    id: root

    // defining the clicked signal
    signal clicked()

    // using a MouseArea type to capture
    // the mouse click of the user
    MouseArea {
        anchors.fill: parent

        // emitting the clicked() signal of the root item
        onClicked: root.clicked()
    }
}
```

Now that we have the *Tool* component lets use it to create the toolbar. We modify the code from the prototype phase in order to use *tool* items instead of the *Rectangle* element.

```
// main.qml

...
// toolbar background
Rectangle {
    anchors.fill: toolbar
    color: "white"
    opacity: 0.15
```

[8] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-image.html
[9] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-image.html

```
    radius: 16
    border { color: "#600"; width: 4 }
}

// using a Column element to layout
// the Tool items vertically
Column { // sidebar toolbar
    id: toolbar
    spacing: 16
    anchors {
        top: window.top
        left: window.left
        bottom: window.bottom
        topMargin: 50
        bottomMargin: 50
        leftMargin: 8
    }

    // new note tool
    Tool {
        id: newNoteTool
        source: "images/add.png"
    }

    // clear page tool
    Tool {
        id: clearAllTool
        source: "images/clear.png"
    }
}
...
```

Now that we have all the graphics set for our components, the application should have a more appealing look and a more defined UI.

What's Next?

In the next chapter, there will be a detailed step on how to create and manage *Note* items dynamically and how to store them locally in a database.

CHAPTER 4

Managing Note Objects Dynamically

We have so far seen that QML is quite a powerful declarative language, but combining it with with JavaScript makes it even more powerful. QML not only enables us to have inline JavaScript[1] functions, but also to import entire JavaScript libraries or files.

Part of the core functionality of NoteApp* is to enable users to create, edit, and delete notes as they like, but the application should also be able to store notes automatically without prompting them.

This chapter will guide you about using JavaScript to add logic to QML code, and implementing local storage using Qt Quick Local Storage[2].

Note: You will find the implementation related to this chapter in the zip file provided in the *Get the Source Code and the Guide in Different Formats* (page 3) section.

The main topics covered in this chapter are:

- Using JavaScript for implementing the functionality of Dynamic Object Management
- How to store data locally using the Qt Quick Database API

This chapter has the following steps:

4.1 Create and Manage Note Items

The user should be able to create and delete notes on the fly and this means that our code should be able to dynamically create and delete Note* items. There are several ways to create and manage QML objects. In fact, we have seen one already using the Repeater[3] type. Creating a QML object means that the component has to be created and loaded before creating instances of that component.

[1] http://qt-project.org/doc/qt-5.0/qtqml/qtqml-javascript-expressions.html#custom-methods
[2] http://qt-project.org/doc/qt-5.0/qtquick/qmlmodule-qtquick-localstorage2-qtquick-localstorage-2.html
[3] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-repeater.html

QML objects can be created by calling the createObject(Item parent, object properties)[4] JavaScript function on the component. Refer to Dynamic Object Management in QML[5] for further details.

Let's see how we can create note item objects in our components.

4.1.1 Note Object Creation Dynamically

We know that a *Note* item belongs to the *Page* component, which is responsible for the note object creation as well as to loading notes from the database.

As mentioned before, we first load the *Note* component in the *Page* component:

```
 // Page.qml

 ...
 // loading the Note Component
Component {
    id: noteComponent
    Note { }
}
 ...
```

Now let's define a Javascript function that will creates QML *Note* objects. While creating a QML object, we must ensure that one of the arguments is the parent of the *this* object. Considering to have a *Note* item container within the *Page* component would be a good idea for managing our note objects, as we would like to save these notes in a database.

```
 // Page.qml

...
// creating an Item element that will be used as a note container
Item { id: container }

 ...
// a Javascript helper function for creating QML Note objects
function newNoteObject(args) {
    // calling the createObject() function on noteComponent item
    // and the container item will be the parent of the new
    // object and args as the set of arguments
    var note = noteComponent.createObject(container, args)
    if(note == null) {
        console.log("note object failed to be created!")
    }
}
...
```

So in the code shown above, we see how a new note* item object can be created in the *newNoteObject()* function. The newly created note objects will belong to the *container* item.

Now we would like to call this function when the new note* tool is pressed on the toolbar, which is created in the *main.qml* file. As the *PagePanel* component is aware of the current

[4] http://qt-project.org/doc/qt-5.0/qtqml/qml-qtquick2-component.html#createObject-method
[5] http://qt-project.org/doc/qt-5.0/qtqml/qtqml-javascript-dynamicobjectcreation.html

visible *page* item, we can create a new property in *PagePanel* to store that page.

We will access this property in the *main.qml* file and call the function to create new note items.

```
// PagePanel.qml
...

// this property holds the current visible page
property Page currentPage: personalpage

// creating the list of states
states: [
    // creating a State item with its corresponding name
    State {
        name: "personal"
        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: "work"
        PropertyChanges {
            target: workpage
            opacity:1.0
            restoreEntryValues: true
        }
        PropertyChanges {
            target: root
            currentPage: workpage
            explicit: true
        }
    }
]
...
```

We modify our three states for setting the appropriate value for the *currentPage* property.

In the *main.qml* file, let's see how to call the function for creating new note objects when the

new note* tool is clicked:

```
// 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
    }

    // the new note tool, also known as the plus icon
    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 Deleting Note Objects

Deleting the *Note* objects is a more straightforward process because the QML Item[6] type provides a JavaScript function called destroy()[7]. As we already have a container item whose childrens are *Note* items, we can simply iterate through the list of children and call destroy()[8] on each of them.

In the *Page* component, let's define a function to perform this operation for us:

```
// Page.qml
...

// a JavaScript helper function for iterating through the children elements of the
// container item and calls destroy() for deleting them
function clear() {
    for(var i=0; i<container.children.length; ++i) {
        container.children[i].destroy()
    }
}
...
```

In the *main.qml* file, we call the *clear()* function when the *clear* tool is pressed:

[6] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-item.html

[7] http://qt-project.org/doc/qt-5.0/qtqml/qtqml-javascript-dynamicobjectcreation.html#deleting-objects-dynamically

[8] http://qt-project.org/doc/qt-5.0/qtqml/qtqml-javascript-dynamicobjectcreation.html#deleting-objects-dynamically

```
// main.qml
...

// the 'clear' tool
Tool {
    id: clearAllTool
    source: "images/clear.png"
    onClicked: pagePanel.currentPage.clear()
}
...
```

In order to allow the user to delete each note individually, we add a tool in the *NoteToolbar* component to our *Note* component. We can use the *Tool* component that was implemented earlier for this:

```
// Note.qml
...

// creating a NoteToolbar item that will be anchored to its parent
NoteToolbar {
    id: toolbar
    height: 40
    anchors { top: root.top; left: root.left; right: root.right }
    // using the drag property alias to set the drag.target
    // to our Note item.
    drag.target: root

    // creating the 'delete' tool for deleting the note item
    Tool {
        id: deleteItem
        source: "images/delete.png"
        onClicked: root.destroy()
    }
}
...
```

What's Next?

Next, there will be a detailed step on how store the note items in a database locally.

4.2 Store and Load Data from a Database

So far we have implemented the core functionality of NoteApp*: creating and managing note items on the fly.

In this step, we will go through a detailed implementation of storing notes in a local database. QML offers a simple Qt Quick Local Storage API[9], which uses the SQLite database for implementing our required feature.

[9] http://qt-project.org/doc/qt-5.0/qtquick/qmlmodule-qtquick-localstorage2-qtquick-localstorage-2.html

The preferred approach for the NoteApp* would be to load notes from the database on application start-up and save them when the application is closed. The user is not prompted at all.

4.2.1 Defining the Database

The database for NoteApp* should be a simple one. It has just one table, the *note* table, that will contain the information for our saved notes.

Looking at the Table definition, let's try to figure out from the information stated above what the *Note* component already has and if any new data should be introduced.

> The *x* and *y* are geometrical properties that each QML item has. Getting these values from the *Note* items should be straightforward. These values will be used to store the position of a note within its page. *noteText* is the actual text of the note and we can retrieve it from the *Text* element in our *Note* component, however, a property alias could be defined for this and we can call it *noteText*. This will be shown later on. *noteId* and *markerId* are identifiers that every note item should have. *noteId* will be a unique identifier required by the database while *markerId* will help us identify the page to which this note item belongs. So we would need two new properties in the *Note* component.

In the *Note* component, we define the new required properties:

```
// Note.qml

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

...

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

...

}
```

Considering that the *Page* component is responsible for creating *Note* items, it should also know which *markerId* it is associated with. When a new *Note* item is created (not loaded from the database), the *Page* should set the *Note* item's *markerId* property.

We'll use a Javascript helper function for this:

```
// Page.qml

Item {
id: root
...

    // this property is held for helping store
    // the note items in the database
```

```
    property string markerId

...

    // this Javascript helper function is used to create,
    // Note items not loaded from database so that it will set
    // the markerId property of the note.
    function newNote() {
        // calling the newNoteObject and passing the a set of
        // arguments where the markerId is set.
        newNoteObject( { "markerId": root.markerId } )
    }

...

}
```

Previously, in *main.qml*, we used the *newNoteObject()* function, but as explained above, that doesn't fit our purpose any longer so we need to replace it with the *newNote()* function.

We have a *markerId* property for the *Page* component that is used to set the *markerId* of *Note* items when created. We must ensure that a page's *markerId* property is set properly when the *Page* items are created in the *PagePanel* component.

```
// PagePanel.qml

Item {
id: root

...

    // creating three Page items that are anchored to fill the 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" }

...

}
```

We have so far ensured that the:

- relation between notes and pages is correct from a relational database perspective,
- a *Note* item has a unique ID that belongs to a page identified by a marker ID,
- and these property values are set properly.

Now let's work on how to load and store the notes.

4.2.2 Stateless JavaScript Library

To simplify the development effort, it would be a good idea to create a JavaScript interface that interacts with the database and provides us with convenient functions to use in our QML code.

In Qt Creator, we create a new JavaScript file named, *noteDB.js*, and make sure that we check the Stateless Library* option. The idea is to make the *noteDB.js* file act like a library and provide stateless helper functions. In this way, there will be just one instance of this file loaded and used for each QML component where the *noteDB.js* is imported and used. This also ensures that there's just one global variable for storing the database instance, *_db*.

Note: Non-stateless JavaScript files are useful when imported in a QML component, perform operations on that component, and all the variables are valid within that context only. Each import creates a separate instance of the JavaScript file.

The *noteDB.js* should provide the following functionality:

- Open/Create a local database instance
- Create the necessary database tables
- Read notes from the database
- Delete all notes

We will see in greater detail how the functions in *noteDB.js* are implemented when describing the implementation of loading and saving note items to the database. Now, let's consider the following functions implemented for us:

- *function openDB()* - Creates the database if one doesn't already exist and, if it does, it opens it
- *function createNoteTable()* - Creates the *note* table if one doesn't already exist. This function is only called in the *openDB()* function
- *function clearNoteTable()* - Removes all rows from the *note* table
- *function readNotesFromPage(markerId)* - This helper function reads all the notes that are related to the *markerId* specified, and returns a dictionary of data.
- *function saveNotes(noteItems, markerId)* - Used to save note items in the database. The arguments, *noteItems* represents a list of note items, and the *markerId* representing the corresponding page to which the note items belong.

Note: As all these JavaScript functions access the database using the Qt Quick Local Storage API, add the statement, *.import QtQuick.LocalStorage 2.0 as Sql*, at the beginning of *noteDB.js*.

4.2.3 Loading and Storing Notes

Now that we have the *noteDB.js* implemented, we would like to use the given functions to load and store notes.

A good practice as to when and where to initialize or open the database connection is to do so in the main* qml file. This way, we can use the JavaScript functions defined in the *noteDB.js* file without reinitializing the database.

We import the *noteDB.js* and *QtQuick.LocalStorage 2.0* in the *main.qml* file, but the question is when to call the *openDB()* function. QML offers helpful attached signals, onCompleted()[10] and onDestruction()[11], which are emitted when the component is fully loaded and destroyed respectively.

```
// main.qml

import QtQuick 2.0
import "noteDB.js" as NoteDB

...
// this signal is emitted upon component loading completion
Component.onCompleted: {
    NoteDB.openDB()
}
...
```

Here is the implementation of the *openDB* function. It calls the openDatabaseSync()[12] function for creating the database and afterwards calls the *createNoteTable()* function for creating the *note* table.

```
//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)")
    })
}
...
```

In the *main.qml* file, we initialize the database so it is pretty safe to start loading our *Note* items in the *Page* component. Above, we have mentioned the *readNotesFromPage(markerId)* function that returns a list of data arrays (referred as dictionary in the scripting world) and each array represents a row in the database with the data of a note.

[10] http://qt-project.org/doc/qt-5.0/qtqml/qml-qtquick2-component.html#onCompleted-signal
[11] http://qt-project.org/doc/qt-5.0/qtqml/qml-qtquick2-component.html#onDestruction-signal
[12] http://qt-project.org/doc/qt-5.0/qtquick/qmlmodule-qtquick-localstorage2-qtquick-localstorage-2.html#opendatabasesync

```
//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
}
```

Tthe *Page* component will read the notes and create the respective QML note* objects.

```
// Page.qml

...
// when the component is loaded, call the loadNotes()
// function to load notes from the database
Component.onCompleted: loadNotes()

// a Javascript helper function that reads the note data from database
function loadNotes() {
    var noteItems = NoteDB.readNotesFromPage(markerId)
    for (var i in noteItems) {
        newNoteObject(noteItems[i])
    }
}
...
```

We can see that the *newNoteObject()* function, defined previously in *Page.qml*, takes the array of data as arguments, which are in fact values for the *x*, *y*, *noteText*, *markerId* and *noteId* properties.

Note: Notice that the *note* table field names are same as the *Note* component properties. This helps us pass the data of a table row as arguments when creating the note QML object.

Now that we have implemented a function to load the *Note* items from the database, the next logical step is to implement a function to save notes into *DB*. In our code, we know that the *PagePanel* component is responsible for creating the *Page* items, so it should be able to access the notes on each page and call the *saveNotes()* JavaScript function from *noteDB.js* to save the notes into *DB*.

```
//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]);
        })
    }
}
```

So at first we define a property alias that will expose the *Note* items, which are childrens of the container Item* created in the *Page* component:

```
// Page.qml

Item {
id: root
...

    // this property is used by the PagePanel component
    // for retrieving all the notes of a page and storing
    // them in the Database.
    property alias notes: container.children

...

}
```

In the *PagePanel*, we implement the functionality for saving notes to DB*:

```
// PagePanel.qml

...
Component.onDestruction: saveNotesToDB()

// a JavaScript function that saves all notes from the pages
function saveNotesToDB() {
    // clearing the DB table before populating with new data
    NoteDB.clearNoteTable();

    // storing notes for each individual page
    NoteDB.saveNotes(personalpage.notes, personalpage.markerId)
    NoteDB.saveNotes(funpage.notes, funpage.markerId)
    NoteDB.saveNotes(workpage.notes, workpage.markerId)
}
...
```

In order to reduce the complexity of our code, we clear all the data in DB* before saving our notes. This avoids the need to have code for updating existing *Note* items.

By the end of this chapter, the users are able to create and delete new notes as they like while the application saves and loads notes automatically for them.

What's Next?

The next chapter will introduce some fancy animations and how these are implemented in several steps and approaches.

CHAPTER 5

Enhancing the Look and Feel

The NoteApp's* UI can be considered complete in terms of functionality and user interaction. However, there is plenty of room for improvement by making the UI more appealing to the user. QML was designed as a declarative language, keeping in mind the animations and fluid transitions of UI elements.

In this chapter, we will guide you step-by-step on how to add animations and make NoteApp* feel more fluid. QML provides a set of QML types that enable you to implement animations in a more convenient approach. This chapter introduces new types and guides how to use them in our QML Components to make the UI more fluid.

Note: You will find the implementation related to this chapter in the zip file provided in the *get-source-code* section.

In summary, the chapter will cover the following main topics:

- Introducing concepts about animations and transitions in QML
- New QML types will be covered, such as *Behavior*, *Transition* and various *Animation* elements
- Enhancing *NoteApp* QML Components using various animations

This chapter has the following steps:

5.1 Animating the NoteToolbar

Let's see how we can improve the *Note* Component and add a behavior based on user's interaction. The *Note* Component has a toolbar with a Delete* tool for deleting the note. Moreover, the toolbar is used to drag the note around by keeping the mouse pressed on it.

An improvement could be to make the *Delete* tool visible only when needed. For instance, making the *Delete* tool visible only when the toolbar is hovered, and it would be nice to do this using fade-in and fade-out effects.

QML provides several approaches to implement this by using *Animation* and *Transition* types. In this specific case, we will use the Behavior[1] QML type, and later on cover the reason behind this.

5.1.1 Behavior and NumberAnimation Types

In the *NoteToolbar* component, we use the Row[2] type to layout the Delete* tool, so changing the *opacity* property of the Row[3] type will also effect the opacity of the *Delete* tool.

Note: The value of the *opacity* property is propagated from the parent to the child items.

The Behavior[4] type helps you define the behavior of the item based on the property changes of that item as shown in the following code:

```
// 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.
    Behavior on opacity {
```

[1] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-behavior.html
[2] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-row.html
[3] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-row.html
[4] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-behavior.html

```
        // using NumberAnimation to animate
        // the opacity value in a duration of 350 ms
        NumberAnimation { duration: 350 }
    }
}
...
```

As you can see in the code above, we enable the *hoverEnabled** property of MouseArea[5] type to accept mouse hover events. Afterwards, we toggle the *opacity* of the *Row* type to *0* if the *Mousearea* type is not hovered and to *1* otherwise. The containsMouse[6] property of *MouseArea* is used to decide the opacity value for the Row type.

So the Behavior[7] type is created inside the *Row* type to define its behavior based on its *opacity* property. When the opacity value changes, the NumberAnimation[8] is applied.

The NumberAnimation[9] type applies an animation based on numerical value changes, so we use it for the *opacity* property of the *Row* for a duration of 350 milliseconds.

Note: The *NumberAnimation* type is inherited from PropertyAnimation[10], which has *Easing.Linear* as the default easing curve animation.

What's Next?

In the next step, we will see how to implement an animation using Transition* and other QML animation types.

5.2 Using States and Transitions

In the previous step, we saw a convenient approach to defining simple animations based on the property changes, using the Behavior[11] and NumberAnimation[12] types.

Certainly, there are cases in which animations depend on a set or property changes that could be represented by a State[13].

Let's see how we can further improve the UI of NoteApp*.

The *Marker* items seem to be static when it comes to user interaction. What if we could add some animations based on different user interaction scenarios? Moreover, we would like to make the current active marker and the current page more visible to the user.

[5] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-mousearea.html
[6] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-mousearea.html#containsMouse-prop
[7] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-behavior.html
[8] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-numberanimation.html
[9] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-numberanimation.html
[10] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-propertyanimation.html
[11] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-behavior.html
[12] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-numberanimation.html
[13] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-state.html

5.2.1 Animating the Marker Items

If we are about to summarize the possible scenarios for enhancing user interaction with *Marker* items, the following use cases are depicted:

- The current active *Marker* item should be more visible. A marker becomes active when the user clicks on it. The active marker is slightly bigger, and it could slide from left to right (just like a drawer).
- When the user hovers a marker with the mouse, the marker slides from left to right but not as much as an active marker would slide.

Considering the scenarios mentioned above, we need to work on the *Marker* and *MarkerPanel* components.

While reading the description above about the desired behavior (the sliding effect from left to right), the immediate thought I get is to change the *x* property of the *Marker* item as it represents the position of the item on the X-axis. Moreover, as the *Marker* item should be aware if it is the current active marker, a new property called *active* can be introduced.

We can introduce two states for the *Marker* component that can represent the depicted behavior above:

- *hovered* - will update the *x* property of the marker when the user hovers it using the mouse
- *selected* - will update the *x* property of the marker when the marker becomes an active one, that is, when it is clicked by the user.

```
// 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 }
    }
]
...
```

So we have two states declared that represent the respective property changes based on the user's behavior. Each state is bound to a condition that is expressed in the *when* property.

Note: For the *containsMouse* property of the *MouseArea* type, the *hoverEnabled* property must be set to *true*.

The Transition[14] type is used to define the behavior of the item when moving from one state to another. That is, we can define various animations on the properties that change when a state becomes active.

Note: The default state of an item is an empty string, ("")

While in the *MarkerPanel* component, we must set the *active* property of the *Marker* item to *true* when it is clicked. Refer to *MarkerPanel.qml* for the updated code.

5.2.2 Adding Transitions to PagePanel

In the *PagePanel* component, we are using states to manage navigation between pages. Adding transitions comes naturally to mind. As we change the *opacity* property in each state, we can add *Transition* for all states that run a *NumberAnimation* on the opacity values for creating the fade-in and fade-out effect.

```
// 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 }
```

[14] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-transition.html

```
    }
]
...
```

Note: The *opacity* value of an item is propagated to its child elements too

---

What's Next?

In the next step, we will learn how to further enhance the UI and see what more could be done.

CHAPTER 6

Further Improvements

At this stage, we can consider that the NoteApp* features are complete and the UI also matches *NoteApp*'s requirements. Nevertheless, there is always room for further improvements, which could be minor but they make the application more polished and ready for deployment.

In this chapter, we will go through the minor improvements that are implemented for NoteApp*, but also suggest new ideas and features to be added. Certainly, we would like to encourage everyone to take the NoteApp* code base and develop it further and perhaps redesign the entire UI and introduce new features.

Note: You will find the implementation related to this chapter in the zip file provided in the *get-source-code* section.

Here is a list of major points covered in this chapter:

- More Javascript used to increase functionality
- Working with the *z* ordering of QML Items
- Using custom local fonts for the application

This chapter has the following steps:

6.1 Enhancing the Note Item Functionality

A nifty functionality for the *Note* items would be to have the note grow as more text is entered. Let's say for simplicity reasons that the note will grow vertically as more text is entered while it wraps the text to fit the width.

The Text[1] type has a paintedHeight[2] property that gives us the actual height of the text painted on the screen. Based on this value, we can increase or decrease the height of the note itself.

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

First, let's define a JavaScript helper function that calculates the value for the height[3] property of the *Item* type, which is the top-level type for the *Note* component.

```
// 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
    }
}
...
```

As the *updateNoteHeight()* function updates the *height* property of the *root* based on the paintedHeight[4] property of *editArea*, we need to call this function upon a change on *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()
    ...
}
```

Note: Every property has a notifier signal that is emitted each time the property changes

---

The *updateNoteHeight()* JavaScript function changes the *height* property so we can define a behavior for this using the Behavior[5] type.

```
// Note.qml

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

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

What's Next?

The next step will show how to use the *z* property of the *Item* type to properly order the notes.

6.2 Ordering Notes

Notes within a *Page* are not aware which note the user is currently working on. By default, all *Note* items created have the same default value for the z[6] property and in this case QML creates a default ordering stack of items based on which item was created first.

The desired behavior would be to change the order of the notes upon a user's interaction. When the user clicks on the note toolbar or starts editing the note, the current note should come up and not be under other notes. This is possible by changing the *z* value to be higher than the rest of the notes.

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

The activeFocus[7] property becomes true when *editArea* has input focus. So, it would be safe to make the *z* property of the *root* dependent on the *activeFocus* of the *editArea*. However, we need to make sure that the *editArea* has input focus even when the toolbar is clicked by the user. Let's define a *pressed()* signal in the *NoteToolbar* component that is emitted on a mouse press.

```
// 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()
}
...
```

In the onPressed[8] signal handler in the *MouseArea* type, we emit the *pressed()* signal of the *root* of *NoteToolbar*.

The *pressed()* signal of the *NoteToolbar* component is handled in the *Note* component.

[6] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-item.html#z-prop
[7] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-item.html#activeFocus-prop
[8] http://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
    ...
}
```

In the code above, we set the focus[9] property of the *editArea* item to *true*, so that the *editArea* will receive the input focus. That is, the *activeFocus* property becomes true, which triggers its *z* property value change.

What's Next?

The next step will show how to load and use a custom local font file for *NoteApp*.

6.3 Loading a Custom Font

It is a very common approach in modern applications to use and deploy custom fonts and not depend on the system fonts. For *NoteApp*, we would like to do the same and we will use what QML offers for this.

The FontLoader[10] QML type enables you to load fonts by name or URL path. As the loaded font could be widely used in the entire application, we recommend loading the font in the *main.qml* file and use it in the rest of the components.

```
// 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')
        }
    }
    ...
}
```

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

So we have created a *webfont* property for the *window* item. This property can be safely used in the rest of the components, so let us use it for the *editArea* in the *Note* component.

```
// Note.qml

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

To set the font for *editArea*, we use the font.family[11] property. From the *window*, we use its *webfont* property to get the font name to be set.

What's Next?

The next step will take you through the details of making NoteApp* ready for deployment.

[11] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-text.html#font.family-prop

CHAPTER 7

Deploying the NoteApp Application

We have reached the point where we would like to make the application available and deployable for a typical desktop environment. As described in the first chapters, we have used a Qt Quick UI project in Qt Creator to develop the NoteApp* application. This means that *qmlscene* is used to load the *main.qml* file, and therefore, have *NoteApp* running.

At first, the simplest way of making NoteApp* available is to create a package that bundles all the qml files, *qmlscene* and a simple script that loads the *main.qml* file using *qmlscene*. You need to refer to the documentation of each desktop platform on how to create small script. For instance, under a Linux platform, you could use a small *bash* shell script for this, while for Windows a simple batch file. This approach works well as being so straightforward, but it could be the case that you would not want to ship the source code because your application is using proprietary code. The application should be shipped in a binary format, in which all the qml files are bundled. This could help make the installation process and the user experience even more pleasant.

Hence, we need to create an executable file for NoteApp* that should be fairly simple to install and use. In this chapter, you will you see how to create a Qt Quick application that bundles qml files and images in an executable binary file. Moreover, we will explore how to use the Qt Resource System with QML.

Note: You will find the implementation related to this chapter in the zip file provided in the *get-source-code* section.

This chapter has the following steps:

7.1 Creating the NoteApp Qt Application

The goal is to create a single executable NoteApp* binary file that the user will run to get *NoteApp* loaded.

Let's see how we can use Qt Creator for this.

7.1.1 Creating a Qt Quick Application

First we need to create a Qt Quick Application[1] using Qt Creator and make sure that we have selected *Built-in elements only (for all platforms)** in the Qt Quick Application wizard. Let's name the application *noteapp*.

So now we have a newly created project from the wizard and we notice that a *qtquick2applicationviewer* project is generated for us. The generated *qtquick2applicationviewer* project is a basic 'template' application that loads QML files. This application, as being a very generic one for deploying Qt Quick applications across devices and targets, includes several platform-specific code for each of those deployment targets. We will not go through these specific parts of the code because it doesn't fit the purpose of this guide.

Nevertheless, there are certain peculiarities of the qtquick2applicationviewer* that somewhat limit us to achieve what we want. The application expects the developer to ship the QML files along with with the executable binary. Using the Qt Resource System becomes impossible, but you will see how to overcome this problem as we proceed.

For the noteapp* project, there is one source file, *main.cpp*. In the *main.cpp* file, you will see how the *viewer* object, the *QtQuick2ApplicationViewer* class, is used to load the *main.qml* file by calling the *QtQuick2ApplicationViewer::setMainQmlFile()* function.

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

Note, there is a basic *main.qml* file generated by the Qt Quick Application wizard which will be replaced by the *main.qml* file we have created for NoteApp*.

The folder structure of the generated noteapp* project is very straightforward to understand.

noteapp - root folder of the *noteapp* project

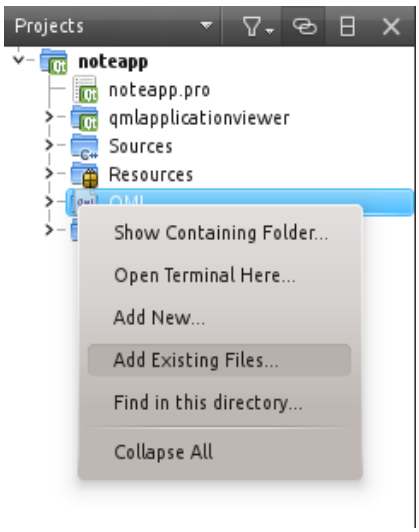
- qml - this folder contains all the QML files
- qtquick2applicationviewer - the generated application used for loading QML files

  noteapp.pro - the project file
- main.cpp - the C++ file where a Qt Application is created

We need to copy our QML files including the *fonts* and *images* directories in the *qml* directory of the newly created noteapp* project. Qt Creator identifies changes to the project folder and

[1] http://qt-project.org/doc/qtcreator-2.6/quick-projects.html#creating-qt-quick-applications

adds the new files to the project view. If it doesn't, right-click on the project and choose *Add Existing Files* to add the new files.



Note: Make sure to add all existing files including images called from the *nodeDDB.js* file

At this point, let's start to build and run the project and see if everything has gone well with the project creation. Before building the noteapp* project, let's make sure we have the right settings in place for our project. Refer to the Configure Projects[2] section in Qt Creator documentation.

Once the application is successfully built, an executable binary file called *noteapp* should be produced in the root folder of the project. If you have Qt configured properly for your system, you'll be able run the file by clicking on it.

7.1.2 Using Qt Resource System to Store QML Files and Images

We have created an executable that loads the QML file for the application to run. As you can see in the *main.cpp* file, the *viewer* object loads the *main.qml* file by passing the relative path of that file. Additionally, we open the *noteapp.pro* file to understand deployment and build settings so we notice the first lines:

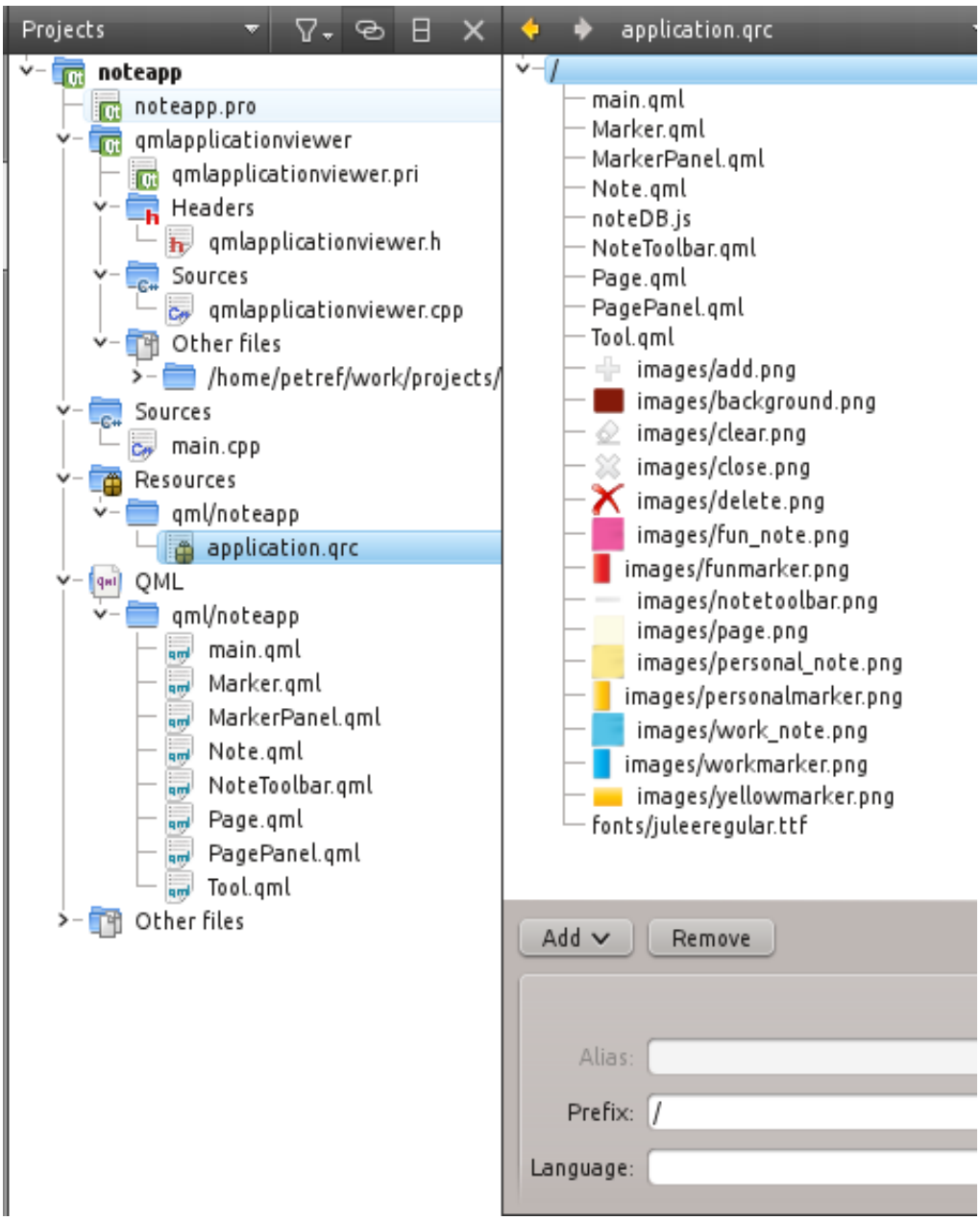
```
# Add more folders to ship with the application, here
folder_01.source = qml/noteapp
folder_01.target = qml
DEPLOYMENTFOLDERS = folder_01

....
```

Seems that it is expected to actually ship the QML files along with the executable file, but this is not what we would like to achieve.

Qt provides a quite intuitive Resource System[3] that works seamlessly with QML. We need to create a resource file, *noteapp.qrc* for the noteapp* root project so that we can add our QML

[2] http://qt-project.org/doc/qtcreator-2.6/creator-configuring-projects.html
[3] http://qt-project.org/doc/qt-5.0/qtcore/resources.html

and image files to it. Refer to the Creating a Resource File[4] in Qt Creator documentation for detailed steps.



We need to apply minor changes to the *noteapp.pro* and the *main.cpp* in order to be able to use the newly created resource file, *noteapp.qrc*.

First we comment out the first lines in the *noteapp.pro*:

```
# Add more folders to ship with the application, here
#folder_01.source = qml/noteapp
#folder_01.target = qml
#DEPLOYMENTFOLDERS = folder_01
```

....

In the *main.cpp* file, we see the *QtQuick2ApplicationViewer::setMainQmlFile()* function being

[4] http://qt-project.org/doc/qtcreator-2.6/creator-writing-program.html#creating-a-resource-file

called with the relative path to the *main.qml* file.

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

The *QtQuick2ApplicationViewer* class inherits QQuickView[5], which is a convenient class for loading and displaying QML files. The *QtQuick2ApplicationViewer::setMainQmlFile()* function is not optimized for using resources because it adjusts the path of the QML file before calling the setSource()[6] function.

The simplest approach to overcome this would be to directly call setSource()[7] on the *viewer* object in the *main.cpp* file, but this time we pass the *main.qml* as part of the resource file.

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

There is no other change to be done in the QML files where we use the image files and the font file since the path of the files is a relative one, which will point to the resource internal filesystem. So now we can build the project in Qt Creator that will produce us a binary executable file that bundles all the QML files, images and the font.

Let's try to make a build and see how that works!

7.1.3 Setting an Application Icon and Title

A graphical enhancement that is highly recommended is to set an icon for the application, which will uniquely identify your application when deployed in a desktop platform.

Inside the noteapp* folder, you may have noticed a few *PNG* files and one *SVG* file by now. These image files will be used to set the icon for the application depending on the icon size since we can have 64x64 or 80x80 icons or a vectorized one.

For more details concerning how these icon files are deployed on various platforms, you need to

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

take a close look at the *qtquick2applicationviewer.pri* file. You can find find detailed information on application icons in the How to Set the Application Icon[8] Qt reference documentation.

We need to call the setWindowIcon()[9] function on the *viewer* in order to set the icon for the application window.

```
// main.cpp
...
QScopedPointer<QApplication> app(createApplication(argc, argv));
QScopedPointer<QtQuick2ApplicationViewer> viewer(
            QtQuick2ApplicationViewer::create());

viewer->setWindowIcon(QIcon("noteapp80.png"));
...
```

We need a default window-title for our application and we will use the setWindowTitle()[10] function for this.

```
// 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!"));
...
```

The NoteApp* is now ready to be shipped and deployed onto various desktop platforms.

7.1.4 Deploying NoteApp

> NoteApp* is a typical Qt application so you need to decide whether you would like to statically or dynamically link against Qt. Additionally, every desktop platform has specific linking configurations to be considered.

You can find detailed information on Deploying Qt Applications[11] reference documentation for each deployment desktop target.

What's Next?

A summary of what we have learned in this developer guide.

[8] http://qt-project.org/doc/qt-4.8/appicon.html
[9] http://qt-project.org/doc/qt-4.8/qwidget.html#windowIcon-prop
[10] http://qt-project.org/doc/qt-4.8/qwidget.html#windowTitle-prop
[11] http://qt-project.org/doc/qt-4.8/deployment.html

CHAPTER 8

Lesson Learned and Further Reading

This guide has shown you how to create an application using Qt Quick and how to make it ready for deployment onto a desktop environment. We have seen how to develop the NoteApp* application step-by-step and we have learned various aspects of the QML language and its potential for developing modern fluid UIs while keeping the code clean, simple by applying various programming techniques.

We have learned some of the best practices of using various QML types and covered some interesting topics such as:

- Animations and States
- Using JavaScript to enhance functionality
- Dynamic QML object management
- Local database storage
- Making the application ready to deploy.

By now, you should have the necessary knowledge and confidence to further enhance NoteApp* with features and UI improvements, and explore more features of QML and Qt Quick that we haven't had a chance to cover in this guide.

Qt Quick is a fast growing technology that is being adopted by various software development industries and areas, so it would be helpful to refer to the Qt Documentation page for latest update about this technology.