
Qt Quick Application Developer Guide for Desktop

Release 1.0

Nokia, <http://qt.nokia.com/learning>

November 25, 2011

CONTENTS

1	About this Guide	3
1.1	Why Would You Want to Read this Guide?	3
1.2	Get the Source Code and the Guide in Different Formats	4
1.3	Help Us Help You	5
1.4	License	5
2	Prototyping and Initial Design	7
2.1	Overview of the NoteApp Application	7
2.2	Creating a QML Component for UI Elements	10
2.3	Anchoring QML Items and Implementing the QML Components	11
2.4	Using Repeater and Delegate to Create List of Markers	14
2.5	Finalize the Prototype	16
3	Implementing the UI and Adding Functionality	23
3.1	Creating PagePanel Component for Managing Pages	23
3.2	Bind Marker Item with the Page Item in Repeater	27
3.3	Adding Graphics, Image vs BorderImage Element	30
4	Managing Note Objects Dynamically and Store Notes Locally	37
4.1	Create and Manage Note Item Objects Dynamically	37
4.2	Store and Load Data from a Database Using Javascript	42
5	Adding Animations and Enhancing the Look & Feel	51
5.1	Using the Behavior Element to Animate NoteToolbar	51
5.2	Using States and Transitions to Animate Pages and Page Markers	54
6	Further Improvements of the Application	57
6.1	Using Javascript to Enhance the Note Item Functionality	57
6.2	Ordering Notes Upon User Interaction	59
6.3	Loading a Custom Font	60
7	Deploying the NoteApp Application	63
7.1	Creating the NoteApp Qt Application	63
8	Lesson Learned and Further Reading	71

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 and also use the Qt Quick Application Development Primer as your first guide on how to use Qt Quick for application development. 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.

ABOUT THIS GUIDE

1.1 Why Would You Want to Read this Guide?

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

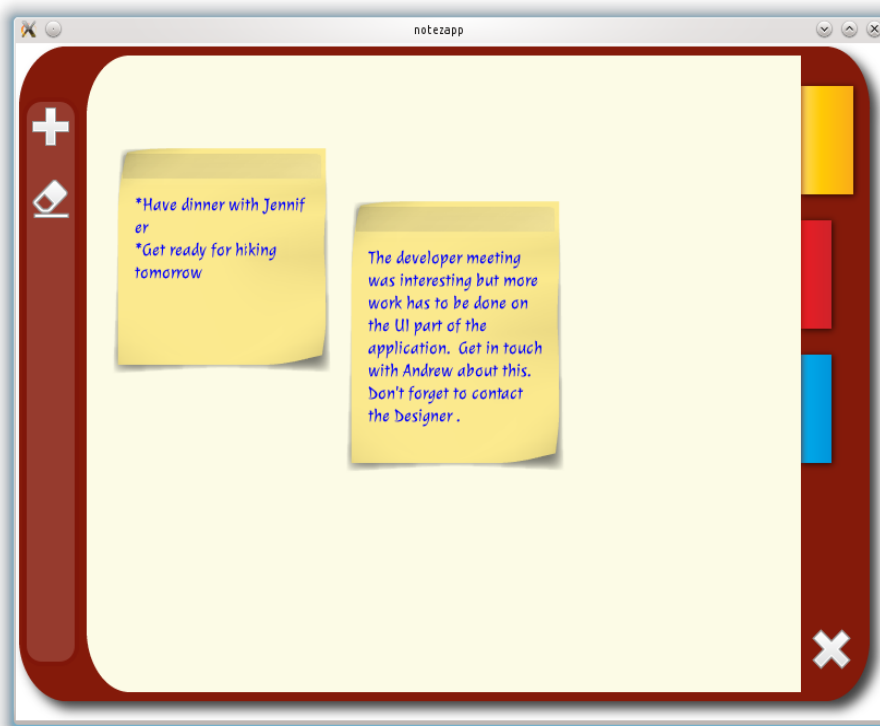
The focus is on Qt Quick and how to use it efficiently for writing entire applications without any C++. It will guide you step by step from the initial setting up of the development environment to project creation to a ready-to-deploy application. We have implemented a simple application that helps users manage daily notes. This application will hereon be referred to as *NoteApp*.

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, meaning that common well-known UI Elements used in desktop apps such as toolbars, menus, dialogs etc will not be present. 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 up to that chapter. It is recommended to always 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. Further details can be found [here](#)⁴.
- [Qt Help](#)⁵ for Qt Assistant and Qt Creator. In Qt Assistant under the [Preferences Dialog](#)⁶, in the *Documentation* tab (in a collapsable menu for Mac users), you click on the Add button in order to add this guide in the .qch format. We do the same in Qt Creator, under the *Options* dialog, in the *Help* section you, can add this guide in the *Documentation* tab.

¹http://get.qt.nokia.com/developerguides/qtquickdesktop/notezapp_src.zip

²<http://get.qt.nokia.com/developerguides/qtquickdesktop/QtQuickApplicationGuide4Desktop.pdf>

³<http://get.qt.nokia.com/developerguides/qtquickdesktop/QtQuickApplicationGuide4Desktop.epub>

⁴http://en.wikipedia.org/wiki/EPUB#Software_reading_systems

⁵<http://get.qt.nokia.com/developerguides/qtquickdesktop/QtQuickApplicationGuide4Desktop.qch>

⁶<http://doc.qt.nokia.com/assistant-details.html#preferences-dialog>

1.3 Help Us Help You

We would greatly appreciate any feedback or comments from you that can help us improve the content of this guide.

Please use the form at the link below to give us your feedback:

- <http://qt.nokia.com/forms/feedback/qtquickdesktop-guide>

1.4 License

Copyright (c) 2008-2011, Nokia Corporation and/or its subsidiary(-ies). All rights reserved.

This work, unless otherwise expressly stated, is licensed under a Creative Commons Attribution-ShareAlike 2.5.

The full license document is available from <http://creativecommons.org/licenses/by-sa/2.5/legalcode>.

The Nokia, Qt and their respective logos are trademarks of Nokia Corporation in Finland and/or other countries worldwide. All other products named are trademarks of their respective owners.

What's Next?

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

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 mainly for two reasons. First, as mentioned in the previous chapter, QML gives us the power to prototype quickly so UI designers can easily and quickly sketch up initial UI screens to be evaluated. Secondly, prototyping allows you to work closely with designers, and therefore, 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 step by step through the concepts of the application 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 the Source Code and the Guide in Different Formats* (page 4) 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 elements 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](http://en.wikipedia.org/wiki/Post-it_note)¹ application that will help users create notes and be able store them locally. It would be easier to manage notes if they belonged to a category so for simplicity, let's consider having three different categories that the notes can belong to. From a visual perspective, however, a category can be represented by an area where notes of the same

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

category can be placed. Let's introduce the concept of the *Page* element. 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 suppose that there are just three predefined pages and each page is indicated by a *Marker*. Additionally, each marker can have a different color.

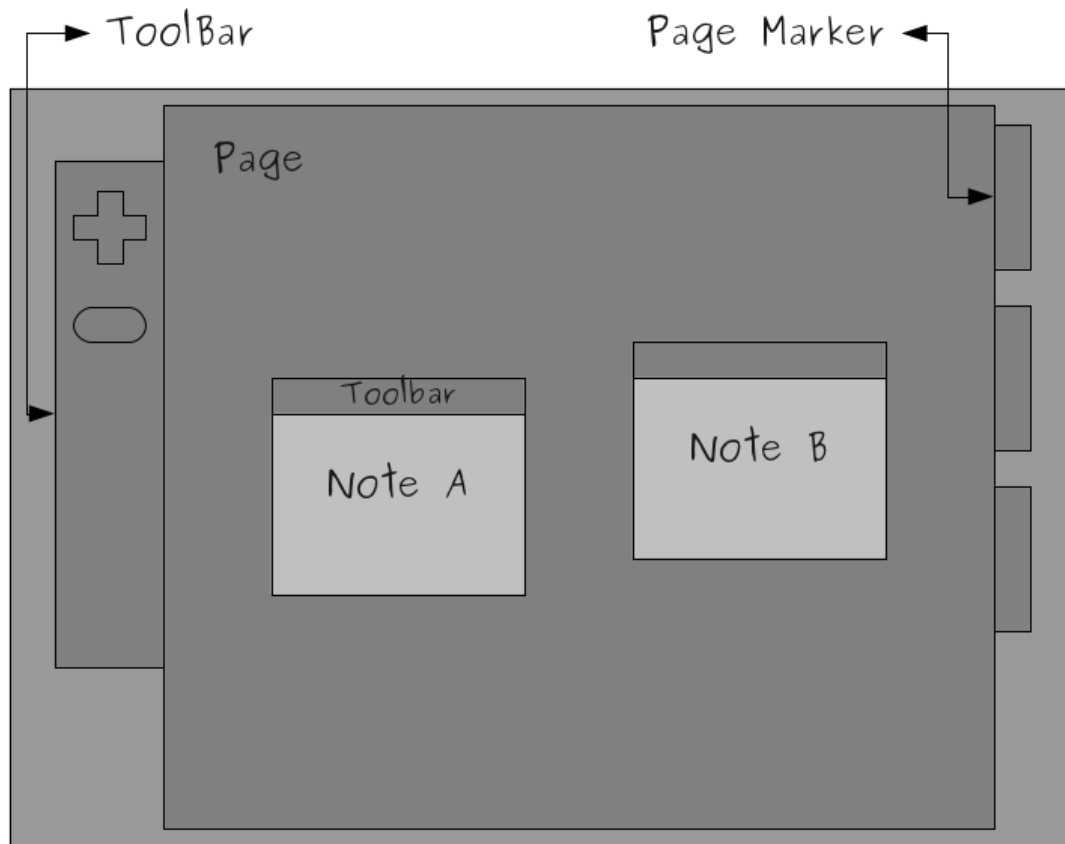
An interesting feature would be to store notes locally and perhaps have this done automatically for the user then so no user prompt would be needed.

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 *NoteApp* concepts discussed above and feature requirements, let's start with the wire-frame design shown in the picture below. Since there could be many possible designs for the *NoteApp*, let's consider the following one as given.



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 contain *Note* items. We can also see the *Marker* items on the right side. There is a toolbar on the left in *NoteApp*. It contains two tools: the *New Note* tool for creating new notes and the *Clear All* for clearing the entire page. The page markers that enable the user to switch between pages are on the right. *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 we have also identified basic initial UI elements, we can safely start to implement and prototype some initial work.

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, becomes so much 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.

Please refer to the [Creating a Qt Quick UI²](#) 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 in order to start the application. For *NoteApp*, we have the *main.qml* file, which has in fact been generated by Qt Creator.

2.2.2 Identifying QML Components as UI Elements

If you want to make an analogy with object oriented programing, then 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 components that are obvious at first and come naturally in mind, but more will come later on as we go in 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

²<http://doc.qt.nokia.com/qtcreator-snapshot/quick-projects.html#creating-qt-quick-ui-projects>

- *Page* - this component contains note items
- *Marker* - represents a page marker, enables users to switch between pages using makers
- *NoteToolbar* - the toolbar used on a note item to enable drag functionality and lay out tools

Please refer to [Creating QML Components with QtCreator³](#) 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 Element⁴](#) is the natural choice to use for building UI blocks and the initial QML Component in the prototype phase. They are visual items that have properties that you can tweak to make it easier to prototype and test your UI concepts, which makes testing so much easier.

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 close 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 with Qt Creator that we can work with for implementing the required components.

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

```
// NoteToolbar.qml

import QtQuick 1.1

// A Rectangle element with defined geometries and color.
Rectangle {
    id: root
```

³<http://doc.qt.nokia.com/qtcreator-snapshot/quick-components.html>

⁴<http://doc.qt.nokia.com/qml-rectangle.html>

```
width: 100
height: 62
color: "#9e964a"
}
```

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 [TextEdit QML Element](#)⁵ for this. In order to layout these items within the *Note* component, the *anchor* property will be used. This property is inherited from [Item Element](#)⁶, which is the base class that every other QML Component inherits by default.

Please refer to the [Anchor-based Layout in QML](#)⁷ documentation for more details about anchoring and laying out QML items.

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

```
// Note.qml

import QtQuick 1.1

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

    // creating a NoteToolbar item 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 item used for the text input from user.
    TextEdit {
        anchors {
```

⁵<http://doc.qt.nokia.com/qml-textedit.html>

⁶<http://doc.qt.nokia.com/qml-item.html>

⁷<http://doc.qt.nokia.com/qml-anchor-layout.html>


```
        top: toolbar.bottom
        bottom: root.bottom
        right: root.right
        left: root.left
    }
    wrapMode: TextEdit.WrapAnywhere
}
}
```

Warning: For performance reasons, you should only anchor an item 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.

```
// Page.qml

import QtQuick 1.1

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 **qmlviewer** 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 be just a rectangle element with predefined geometries. Later on, as described in the next chapter, we will show how the *Marker* component is used.

```
// Marker.qml

import QtQuick 1.1

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 Element 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 layout these QML items 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 items with each other, but nevertheless it will increase the complexity of the code. QML has other convenient approaches, such as layout and positioning elements like [Column Element](http://doc.qt.nokia.com/qml-column.html)⁸.

⁸<http://doc.qt.nokia.com/qml-column.html>

Since we want to lay out three *Marker* items within a [Column Element](#)⁹, let's use a nifty element QML provides 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* will generate three QML Items based on the model and will use a **delegate** to display them. Since we want to have three *Marker* items, we simply use the *Marker* component as the delegate.

Further detailed reference on *Column* and *Repeater* can be found in [Using QML Positioner and Repeater Items](#)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 later on how.

Here is how the *MarkerPanel* component would look:

```
// MarkerPanel.qml

import QtQuick 1.1

Rectangle {
    id: root
    width: 50
    height: 300

    // column element that anchors to the entire parent
    Column {
        id: layout
        anchors.fill: parent
        spacing: 10

        Repeater {
            // just three Marker items
        }
    }
}
```

⁹<http://doc.qt.nokia.com/qml-column.html>

¹⁰<http://doc.qt.nokia.com/qml-repeater.html>

¹¹<http://doc.qt.nokia.com/qml-positioners.html>

```
        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 **qmlviewer** then it should look as follows:



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 actually 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 in order 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 items, so we have the panel of *Maker* items, the *MarkerPanel* component that is, laid out on the right and the *Page* component in the center. We haven't yet covered the toolbar UI element so let's do that now.

The toolbar contains two tools: one for creating new note items and one for clearing up the page. For simplicity, we will not create a component for this, but rather declare the items 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 items
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 element to layout the tools of the toolbar
    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:

```
// main.qml

import QtQuick 1.1

Rectangle {
    // using window as the identifier for this item as
    // it will be 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 **qmlviewer**:



2.5.2 Making Note Items Draggable Using the MouseArea QML Element

So far we have managed to get a very basic prototype running 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 Element](http://doc.qt.nokia.com/qml-mousearea.html)¹² has a grouped property called **drag**¹³. We will use the **drag.target** property by setting it to the **id** of our note item.

Considering that the user should use the *NoteToolbar* item for dragging the note item, the *MouseArea* Element should be inside the *NoteToolbar* component. The *NoteToolbar* component will handle the dragging operation from the user so we should set the **drag.target**¹⁴ to the *Note* item.

¹²<http://doc.qt.nokia.com/qml-mousearea.html>

¹³<http://doc.qt.nokia.com/qml-mousearea.html#drag.target-prop>

¹⁴<http://doc.qt.nokia.com/qml-mousearea.html#drag.target-prop>

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

Let's take the *NoteToolbar* component we have created so far and create a property alias for the **drag** grouped property of *MouseArea*:

```
// NoteToolbar.qml

import QtQuick 1.1

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

    // declaring a property alias to the drag
    // property of MouseArea element
    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.

```
// Note.qml

import QtQuick 1.1

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

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

¹⁵<http://doc.qt.nokia.com/qml-mousearea.html#drag.target-prop>

¹⁶<http://doc.qt.nokia.com/propertybinding.html#property-aliases>


```
        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](http://doc.qt.nokia.com/propertybinding.html)¹⁷ documentation page.

What's Next?

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

¹⁷<http://doc.qt.nokia.com/propertybinding.html>

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 needed to be created.

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 to compose 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 the chance to dive deeper into some QML elements 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 the Source Code and the Guide in Different Formats* (page 4) 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 elements that we are currently using

This chapter has the following steps:

3.1 Creating PagePanel Component for Managing Pages

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. The *NoteApp* concepts and requirements, however, 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 lay out 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 QML Element as the Top Level Element for Components

Before we go any further with our new component, it is important to understand why using the *Rectangle Element*¹ as a top level element 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* element with the *Item* element 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 1.1

// Page.qml

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

Warning: From now on, it is considered that the top level element for our components is an Item QML Element. Please 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 *State Element*² to implement the intended behavior. The *PagePanel* component will have three states and each of them is bound to make one *Page* item visible accordingly. So navigating through pages will be a matter of setting the respective state for the *PagePanel* so we need to have three states and three *Page* items inside the *PagePanel* component.

¹<http://doc.qt.nokia.com/qml-rectangle.html>

²<http://doc.qt.nokia.com/qml-state.html>

First, in the *Page* component, we need to set the *opacity* property to *0.0* as the default value so the *Page* items will only be visible when the respective state is set to the *PagePanel* item because the respective state will change the *opacity* value of the page as shown below:

```
// Page.qml

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

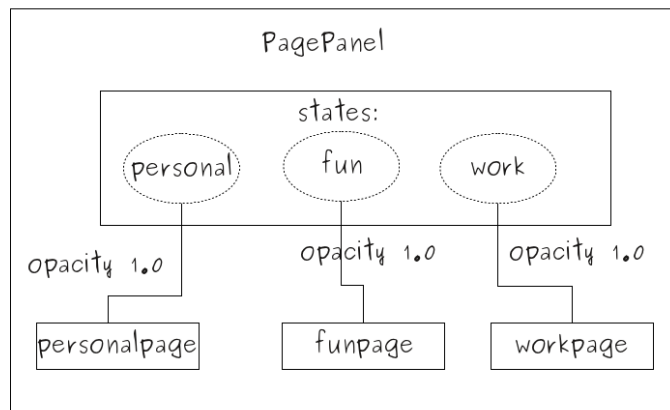
Once we have created the *PagePanel.qml* file, let's start to create the states and *Page* items. We need three states:

- *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 1.1

// PagePanel.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 will make the changed property of the target reset to its default value, meaning that the *opacity* property of the *Page* item will be reset to *false* as initially set 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 Bind Marker Item with the Page Item in Repeater

Previously we saw the implementation of the *PagePanel* component, which uses three states to toggle the `opacity`³ property of the *Page* items. 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 have any functionality. It uses a *Repeater Element*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 1.1

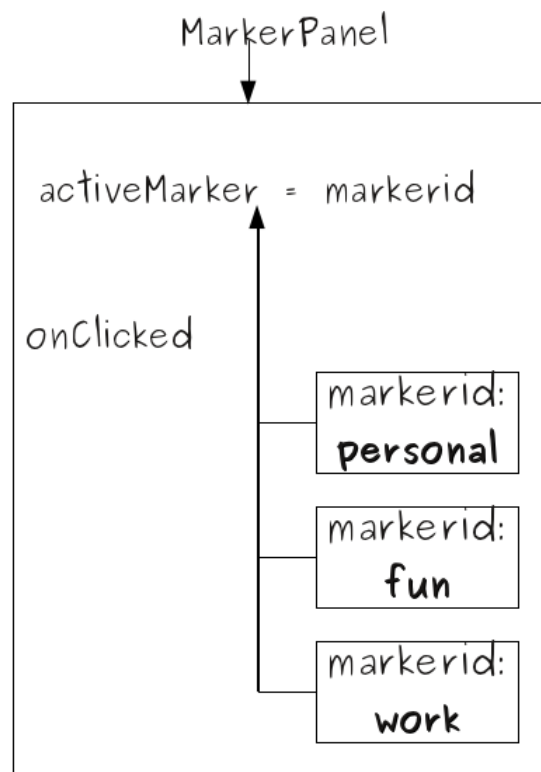
Item {
    id: root
    width: 150; height: 450

    // 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.

³<http://doc.qt.nokia.com/qml-item.html#opacity-prop>

⁴<http://doc.qt.nokia.com/qml-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 1.1
```

```
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](http://doc.qt.nokia.com/propertybinding.html)⁵ feature of QML helps by setting, or rather binding, the **state** property with the **activeMarker**. This means that whatever value **activeMarker** will have during user's selection, then this value will be 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 in order to shape up our application's UI.

3.3 Adding Graphics, Image vs BorderImage Element

Due to the nature of QML, developers and designers are entitled to work closely throughout the development process. Nowadays, using graphics does make a big difference to the UX and

⁵<http://doc.qt.nokia.com/propertybinding.html>

how the application is perceived by users.

QML encourages the use of graphics as much as possible when implementing the UI. The collaboration of developers with graphic designers becomes more efficient with QML because designers are able to test their graphics right away on basic UI elements, but they can also understand from a technical perspective what a developer requires when developing new components. This helps the process of making the UI of the application more appealing and certainly easier to maintain.

Let's start adding graphics to our components.

3.3.1 Setting Background Images to Components

The `BorderImage`⁶ element is recommended to be used in cases in which you would like to have the image scaled depending on user interaction and UI flow, but keep its borders unscaled. A good example of using the `BorderImage`⁷ element is for having background images that have a shadowing effect for your QML item while it might be expected that this item will scale at some point and you would like to keep corners untouched.

Let's see how to set a `BorderImage` element as a background for our components.

Inside the `PagePanel.qml` we add the `BorderImage` element:

```
// 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"
```

⁶<http://doc.qt.nokia.com/qml-borderimage.html>

⁷<http://doc.qt.nokia.com/qml-borderimage.html>

```
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* element is used in the correct order within your component implementation because the arrangement of elements within a component does affect their 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](http://doc.qt.nokia.com/qml-item.html#z-property)^a documentation.

^a<http://doc.qt.nokia.com/qml-item.html#z-property>

You may be wondering by now what would be the best approach to set a background for the *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 *Repeater* in order 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*⁸ element as the top level element of the *Marker* component.

```
// Marker.qml

import QtQuick 1.1

// The Image element 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 element to intercept the mouse click
    MouseArea {
        id: mouseArea
        anchors.fill: parent

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

⁸<http://doc.qt.nokia.com/qml-image.html>

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

    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* element for setting a background for the *NoteToolbar* component and also for setting a background for our most top level element 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 in *qmlviewer*, 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

When taking into consideration code re-usability then 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 will use an *Image Element*⁹ as a top level element and simply handle the mouse click from the user.

The *Image Element*¹⁰ is often used as an UI element on its own regardless of whether it's a static or animated image, but in any case it is laid out in a pixel perfect way and well defined by design requirements.

```
// Tool.qml

import QtQuick 1.1

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

    // defining the clicked signal
    signal clicked()

    // using a MouseArea element to capture
    // the mouse click of the user
```

⁹<http://doc.qt.nokia.com/qml-image.html>

¹⁰<http://doc.qt.nokia.com/qml-image.html>

```
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

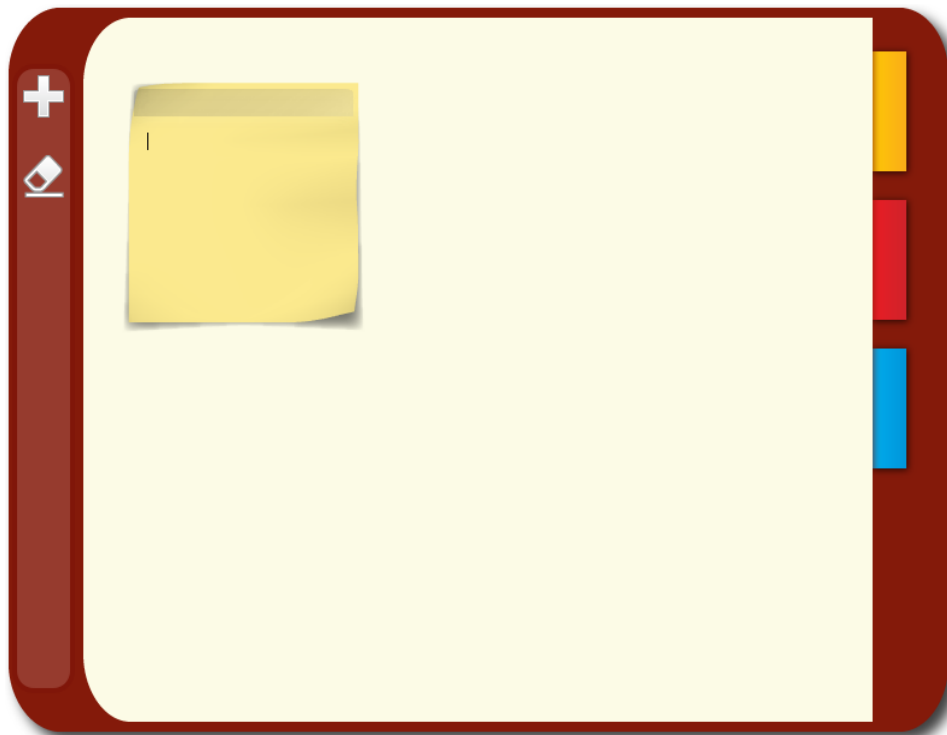
...
// toolbar background
Rectangle {
    anchors.fill: toolbar
    color: "white"
    opacity: 0.15
    radius: 16
    border { color: "#600"; width: 4 }
}

// 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.

MANAGING NOTE OBJECTS DYNAMICALLY AND STORE NOTES LOCALLY

We have seen so far that QML is quite a powerful declarative language, but combining it with Javascript makes it even more powerful. QML not only enables us to have [inline Javascript¹](#) functions, but also to import entire Javascript libraries or files as we will see in the detailed steps of this chapter.

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 in detailed steps how to use Javascript to add logic to QML code and how local storage can be implemented with [Qt Quick Database API²](#).

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 4) 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 Item Objects Dynamically

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 [Repeater³](#) element.

¹<http://doc.qt.nokia.com/qdeclarativejavascript.html#inline-javascript>

²<http://doc.qt.nokia.com/qdeclarativeglobalobject.html>

³<http://doc.qt.nokia.com/qml-repeater.html>

Creating a QML object means that the component has to be created and loaded before creating instances of that component.

QML objects can be created by calling the `createObject(Item parent, object properties)`⁴ Javascript function on the component, which will create the object for us. Please refer to [Dynamic Object Management in QML](http://doc.qt.nokia.com/qdeclarativedynamicobjects.html)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 a *Page* item so the *Page* component is responsible for note object creation as well as for loading notes from the database as we will see in the next step.

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 help us create QML *Note* objects. We need to take into consideration that, when creating a QML object, a set of arguments is passed and one of them 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 - especially for later on 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!")
    }
}
```

⁴<http://doc.qt.nokia.com/qml-component.html#createObject-method>

⁵<http://doc.qt.nokia.com/qdeclarativedynamicobjects.html>

```
}  
...
```

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. Since the *PagePanel* component is aware of the current visible *page* item, we can create a new property in *PagePanel* that stores 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 element provides a Javascript function called *destroy()*⁷. As we already have a container item whose children are all *Note* items, we can simply iterate through the list of children and call *destroy()*⁸ on each of them.

⁶<http://doc.qt.nokia.com/qml-item.html>

⁷<http://doc.qt.nokia.com/qdeclarativedynamicobjects.html#deleting-objects-dynamically>

⁸<http://doc.qt.nokia.com/qdeclarativedynamicobjects.html#deleting-objects-dynamically>

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

```
// Page.qml
...

// a Javascript helper function for iterating through the children elements of t
// 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:

```
// 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 and 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 to store the note items in a database locally.

4.2 Store and Load Data from a Database Using Javascript

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 how to store notes in a local database. QML offers a simple [Qt Quick Database API](#)⁹, which uses the SQLite database for implementing our required feature.

The preferred approach would be for the *NoteApp* to load notes from the database on application start-up and save them when the application is closed so no prompting is done for the user whatsoever.

4.2.1 Defining the Database and Required Data Fields

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

Field	Type	Description
<i>noteId</i>	INTEGER (PRIMARY KEY AUTOINCREMENT)	Unique identifier for the note item
<i>x</i>	INTEGER	The <i>x</i> property of the note item holding the position on the horizontal axis
<i>y</i>	INTEGER	The <i>y</i> property of the note item holding the position on the vertical axis
<i>note-Text</i>	TEXT	For the note item text
<i>markerId</i>	TEXT	<i>markerId</i> represents the page to which this note item belongs

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.

⁹<http://doc.qt.nokia.com/qdeclarativeglobalobject.html>

- *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

    ...
}
```

Now considering that this *Page* component is responsible for creating *Note* items, it should also know which *markerId* it is associated with. When a new *Note* item will be 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 (see above in the code snippet).

We have a *markerId* property for the *Page* component that is used to set the *markerId* of *Note* items when created, however, first we need to make sure that a page's *markerId* property is set properly and we know that *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" }

    ...
}
```

To summarize what we have done so far, we have ensured that the relation for 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. In our QML code, we make sure that these values are set properly.

Now let's work on how to load and perform the storing of 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 declarative 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 will also ensure 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, it 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 from the database, but for 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 from the database that have their *markerId* column as *markerId* and returns a dictionary of data
- ***function saveNotes(noteItems, markerId)*** Used to save note items in the database. *noteItems* represents a list of note items and *markerId* is the common value for the *markerId* column.

4.2.3 Loading and Storing Notes From DB

Now that we have the *noteDB.js* implemented, we would like to use the given functions to perform the loading and storing of our 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. In this way, we can simply use the Javascript functions defined in the *noteDB.js* file without reinitializing the database.

We import the *noteDB.js* 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 upon destruction respectively.

```
// main.qml

import QtQuick 1.1
import "noteDB.js" as NoteDB

...
// this signal is emitted upon component loading completion
Component.onCompleted: {
```

¹⁰<http://doc.qt.nokia.com/qml-component.html#onCompleted-signal>

¹¹<http://doc.qt.nokia.com/qml-component.html#onDestruction-signal>

```
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 as the *Page* component is the one responsible for managing notes. Above, we have mentioned the *readNotesFromPage(markerId)* function that returns a list of data arrays (a dictionary otherwise referred to the scripting world) and each array represents a row in the database with the data of a note.

```
//noteDB.js
```

```
...
function readNotesFromPage(markerId) {
    print("noteDB.readNotesFromPage() " + markerId)
    var noteItems = {}
    _db.readTransaction( function(tx) {
        var rs = tx.executeSql(
            "SELECT * FROM note WHERE markerId=?
            ORDER BY markerid DESC", [markerId] );
```

¹²<http://doc.qt.nokia.com/qdeclarativeglobalobject.html#db-opendatabasesync-identifier-version-description-estimated-size-callback-db>

```
        var item
        for (var i=0; i< rs.rows.length; i++) {
            item = rs.rows.item(i)
            noteItems[item.noteId] = item;
        }
    })

    return noteItems
}
```

The *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 of the database uses the same names as the *Note* component properties for the column names. This helps us pass the data of a table row as arguments when creating the note QML object.

Now that we have the loading of *Note* items from the database implemented, the next logical step is to implement the saving of notes into *DB*. In our code, we know that the *PagePanel* component is responsible for creating the *Page* items, meaning it should be able to access the notes per each page and use the *saveNotes()* Javascript function from *noteDB.js* should save them 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 in fact are children elements of the *Container Item* element 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 for the purpose of this guide, we simply clear all the data in *DB* just 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.

ADDING ANIMATIONS AND ENHANCING THE LOOK & FEEL

NoteApp's UI can be considered complete in terms of functionality and user interaction. There is, however, plenty of room for improvement by making the UI more appealing to the user. QML as a declarative language was implemented with animations and fluid transitions of UI elements in mind.

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 elements that enable you to implement animations in a more convenient approach. This chapter will introduce new elements and how these elements will be used in our QML Components in order to make the UI more fluid.

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 4) section.

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

- Introducing concepts about animations and transitions in QML
- New QML Elements 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 Using the Behavior Element to Animate NoteToolbar

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

An improvement could be to make the *Delete* tool of the *Note Toolbar* visible only when needed. For instance, what we could show the *Delete* tool only when the toolbar is hovered and it would be nice to do this in a transitional approach, meaning using fading in and out effects.

QML provides several approaches to implement this by using *Animation* and *Transition* elements as we will see later on. For this specific case, however, we use the [Behavior¹](#) QML Element and later on cover the reason behind this.

5.1.1 Behavior and NumberAnimation QML Elements

In the *NoteToolbar* component, we use the [Row²](#) element to layout the *Delete* tool so changing the *opacity* property of the [Row³](#) element 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 [Behavior⁴](#) element helps you define the behavior of your element based on the property changes of that element as shown in the following code in *NoteToolbar.qml*:

```
// NoteToolbar.qml

...
MouseArea {
    id: mousearea
    anchors.fill: parent

    // setting hovered property to true
    // in order for the MouseArea to be able to get
    // hover events
    hovered: 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
}
```

¹<http://doc.qt.nokia.com/qml-behavior.html>

²<http://doc.qt.nokia.com/qml-row.html>

³<http://doc.qt.nokia.com/qml-row.html>

⁴<http://doc.qt.nokia.com/qml-behavior.html>


```
// the opacity depends if the mousearea elements
// have 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 {
    // using NumberAnimation to animate
    // the opacity value in a duration of 350 ms
    NumberAnimation { duration: 350 }
}
...

```

As we can see from the code shown above, we enable the **hoverEnabled** property in the **MouseArea**⁵ element, which allows *MouseArea* to accept mouse hover events. Afterwards, we toggle the *opacity* of the *Row* element to *0* if the *mousearea* element is not hovered and to *1* otherwise. The **containsMouse**⁶ property of *MouseArea* is used for this, which is set to true only when *mousearea* is hovered.

So the **Behavior**⁷ element is created inside the *Row* element to define a behavior for it when the *opacity* property is toggled and therefore changed so a **NumberAnimation**⁸ element is created to animate this change.

The **NumberAnimation**⁹ element 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* element inherits from **PropertyAnimation**¹⁰, which have *Easing.Linear* as the default easing curve of the animation.

What's Next?

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

⁵<http://doc.qt.nokia.com/qml-mousearea.html>

⁶<http://doc.qt.nokia.com/qml-mousearea.html#containsMouse-prop>

⁷<http://doc.qt.nokia.com/qml-behavior.html>

⁸<http://doc.qt.nokia.com/qml-numberanimation.html>

⁹<http://doc.qt.nokia.com/qml-numberanimation.html>

¹⁰<http://doc.qt.nokia.com/qml-propertyanimation.html>

5.2 Using States and Transitions to Animate Pages and Page Markers

Previously, we saw a rather convenient approach to defining simple animations upon the property changes of an element using the [Behavior](#)¹¹ and [NumberAnimation](#)¹² elements.

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 more visible to the user, hence the current page that the user is creating notes for.

5.2.1 Animating the Marker Items using Transitions

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, in fact, 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), changing the *x* property of the *Marker* item immediately comes to mind because the *x* property 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, meaning clicked by the user

```
// Marker.qml
...
// this property indicates whether this marker item
// is the current active one. Initially it is set to false
```

¹¹<http://doc.qt.nokia.com/qml-behavior.html>

¹²<http://doc.qt.nokia.com/qml-numberanimation.html>

¹³<http://doc.qt.nokia.com/qml-state.html>

```
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* element, the *hoverEnabled* property should be set to true

The [Transition](http://doc.qt.nokia.com/qml-transition.html)¹⁴ QML Element is used to define the behavior of the item when moving from one state to another, meaning we can define various animations on the properties that change

¹⁴<http://doc.qt.nokia.com/qml-transition.html>

when a state becomes active.

Note: The default state of an item is referred to as an empty string, (“”)

While in the *MarkerPanel* component, we make sure to set the *active* property of the *Marker* item to true only when it is clicked as described in previous steps. Please check *MarkerPanel.qml* for more details.

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. Since we simply 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 }
    }
]
...
```

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.

FURTHER IMPROVEMENTS OF THE APPLICATION

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

In this chapter, we will go through some 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 the Source Code and the Guide in Different Formats* (page 4) 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 Using Javascript to Enhance 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 element has a `paintHeight`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

¹<http://doc.qt.nokia.com/qml-text.html>

²<http://doc.qt.nokia.com/qml-text.html#paintHeight-prop>

itself.

First let's define a Javascript helper function that would calculate the proper value for the `height`³ property of the *Item* element as the top-level element 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
    }
}
...
```

Since the *updateNoteHeight()* function updates the *height* property of the *root* element 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 element.

```
// Note.qml

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

³<http://doc.qt.nokia.com/qml-item.html#height-prop>

⁴<http://doc.qt.nokia.com/qml-text.html#paintedHeight-prop>

⁵<http://doc.qt.nokia.com/qml-behavior.html>

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

What's Next?

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

6.2 Ordering Notes Upon User Interaction

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, then this note should come up and not be under other notes, hence the `z` value should 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 the *editArea* element has input focus. So, it would be safe to make the `z` property of the *root* element dependent on the *activeFocus* of the *editArea*. However, we need to make sure that the *editArea* element 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 upon 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
```

⁶<http://doc.qt.nokia.com/qml-item.html#z-prop>

⁷<http://doc.qt.nokia.com/qml-item.html#activeFocus-prop>

```
        onPressed: root.pressed()
    }
    ...
```

Hence, on the `onPressed`⁸ signal handler in the *mousearea* element, we emit the *pressed()* signal of the *root* element of *NoteToolbar*.

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

```
// 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* element to *true* so that the *editArea* will receive the input focus, meaning that the *activeFocus* property becomes true, therefore the note item will update its *z* value.

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 element enables you to load fonts by name or URL path. As the loaded font could be widely used in the entire application, loading the font in the *main.qml* file and using it in the rest of the components would be the recommended approach.

```
// main.qml

Rectangle {
    // using window as the identifier for this item as
    // it will be the only window of the NoteApp
    id: window
```

⁸<http://doc.qt.nokia.com/qml-mousearea.html#onPressed-signal>

⁹<http://doc.qt.nokia.com/qml-item.html#focus-prop>

¹⁰<http://doc.qt.nokia.com/qml-fontloader.html>


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

So we have created a *webfont* property for the *window* item. This property can be safely used in the rest of the components and we would like to use it in the *Note* for setting the font of the *editArea* to the newly loaded font.

```
// 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* root element, we use its *webfont* property to get the font name to be set.

What's Next?

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

¹¹<http://doc.qt.nokia.com/qml-text.html#font.family-prop>

DEPLOYING THE NOTEAPP APPLICATION

We have reached the point where we would 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 *qmlviewer* 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, *qmlviewer* and a simple script that loads the *main.qml* file using *qmlviewer*. 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 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 the Source Code and the Guide in Different Formats* (page 4) 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 just 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 *qmlapplicationviewer* project is generated for us. The generated *qmlapplicationviewer* 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 *qmlapplicationviewer* that somewhat limit us to achieve what we want. The application expects the developer to ship the QML files along 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 just one CPP source file, *main.cpp*. In the *main.cpp* file, you will see how the *viewer* object, the *QmlApplicationViewer* class, is used to load the *main.qml* file by calling the *QmlApplicationViewer::setMainQmlFile()* function.

```
// main.cpp
...
Q_DECL_EXPORT int main(int argc, char *argv[])
{
    QScopedPointer<QApplication> app(createApplication(argc, argv));
    QScopedPointer<QmlApplicationViewer> viewer(
        QmlApplicationViewer::create());

    viewer->setOrientation(QmlApplicationViewer::ScreenOrientationAuto);
    viewer->setMainQmlFile("qml/noteapp/main.qml");
    viewer->showExpanded();

    return app->exec();
}
```

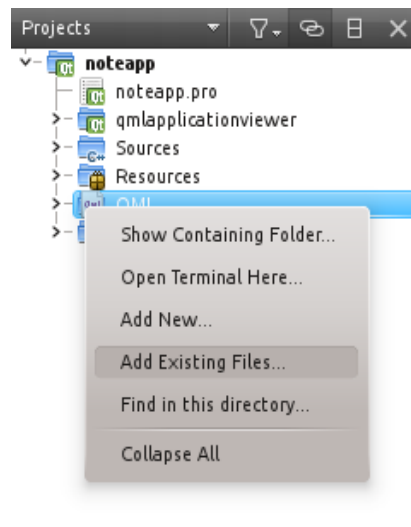
Note, there is a basic *main.qml* file generated by the Qt Quick Application wizard which will be replaced by the *main.qml* file we have created for *NoteApp*.

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
 - *qmlapplicationviewer* - the generated application used for loading QML files
 - *noteapp.pro* - the project file
 - *main.cpp* - the C++ file where a Qt Application is created that will produce a binary file for us

¹<http://doc.qt.nokia.com/qtcreator-snapshot/quick-projects.html#creating-qt-quick-applications>

We need to copy our QML files including the *fonts* and *images* directories in the *qml* directory of the newly created *noteapp* project. Additionally, we need to import these files into the project so that we will be able to work with them in Qt Creator.



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. Please refer to [Configure Projects²](#) 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 to 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
```

....

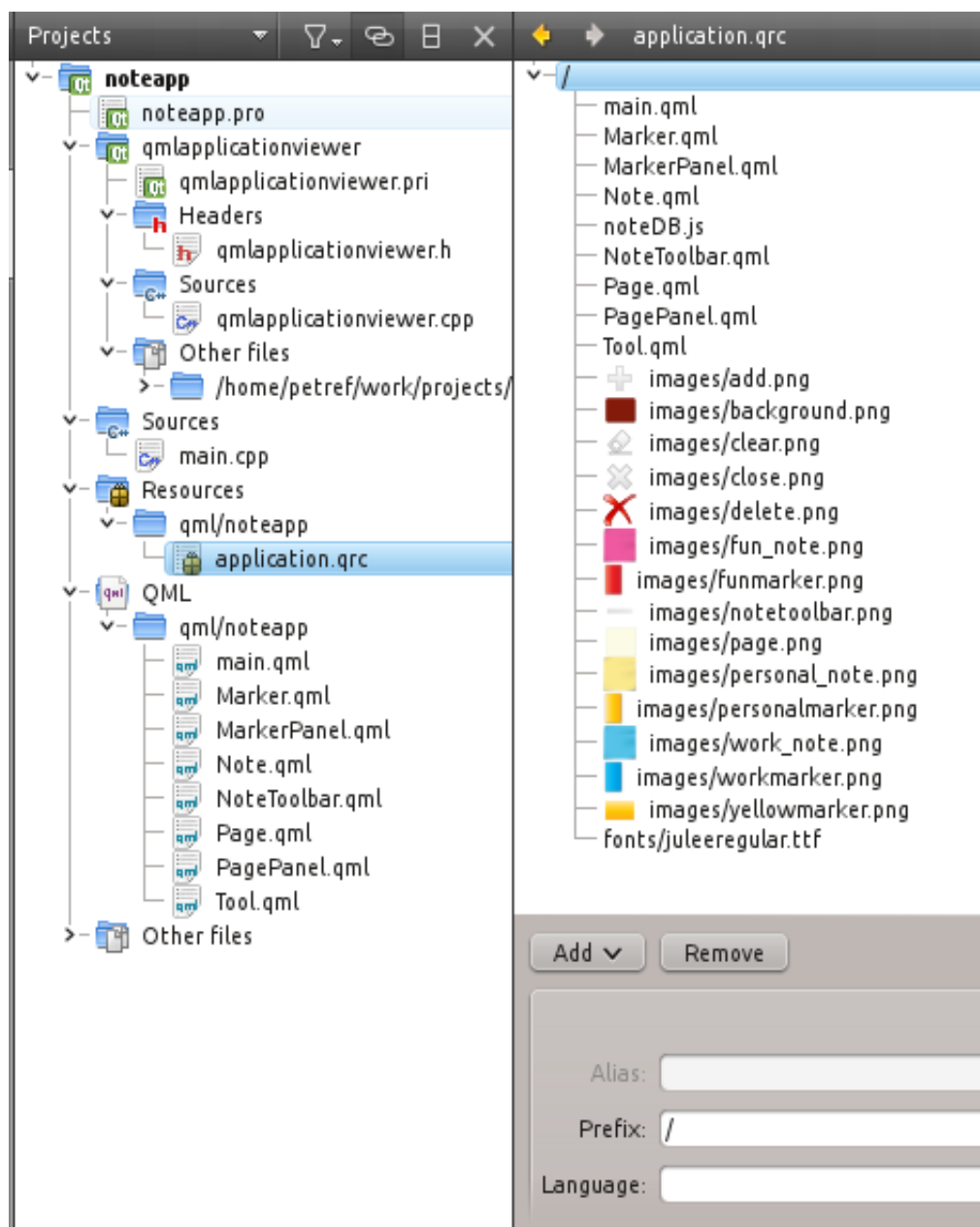
Seems that it is expected to actually ship the QML files along with the executable file, but this is not what we would like to achieve.

Qt provides a quite intuitive [Resource System³](#) that works seamlessly with QML. We need to

²<http://doc.qt.nokia.com/qtcreator-snapshot/creator-configuring-projects.html>

³<http://doc.qt.nokia.com/resources.html>

create a resource file, *noteapp.qrc* for the *noteapp* root project so that we can add our QML and image files in it. Please refer to the [Creating a Resource File⁴](#) 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
```

....

⁴<http://doc.qt.nokia.com/creator-writing-program.html#creating-a-resource-file>

In addition to that, we can remove the following lines from the *noteapp.pro* file since they are Symbian platform related.

```
...
symbian:TARGET.UID3 = 0xE68D5D88
...
# Allow network access on Symbian
symbian:TARGET.CAPABILITY += NetworkServices
...
```

In the *main.cpp* file, we see the *QmlApplicationViewer::setMainQmlFile()* function being called with the relative path to the *main.qml* file.

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

The *QmlApplicationViewer* class inherits *QDeclarativeView*⁵, which is a convenient class for loading and displaying QML files. The *QmlApplicationViewer::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
...
Q_DECL_EXPORT int main(int argc, char *argv[])
{
    QScopedPointer<QApplication> app(createApplication(argc, argv));
    QScopedPointer<QmlApplicationViewer> viewer(
        QmlApplicationViewer::create());

    viewer->setOrientation(QmlApplicationViewer::ScreenOrientationAuto);
    viewer->setSource(QUrl("qrc:/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.

⁵<http://doc.qt.nokia.com/qdeclarativeView.html>

⁶<http://doc.qt.nokia.com/qdeclarativeview.html#source-prop>

⁷<http://doc.qt.nokia.com/qdeclarativeview.html#source-prop>

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 take a close look at the *qmlapplicationviewer.pri* file. You can 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<QmlApplicationViewer> viewer(
    QmlApplicationViewer::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<QmlApplicationViewer> viewer(
    QmlApplicationViewer::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

⁸<http://doc.qt.nokia.com/appicon.html>

⁹<http://doc.qt.nokia.com/qwidget.html#windowIcon-prop>

¹⁰<http://doc.qt.nokia.com/qwidget.html#windowTitle-prop>

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 learnt in this developer guide.

¹¹<http://doc.qt.nokia.com/deployment.html>

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 QML language and its potential for developing modern fluid UIs while keeping the code clean, simple and applying various programming techniques.

We have learned some of the best practices of using various QML Elements 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 being adopted by various software development industries and areas so it would be quite helpful to refer to the Qt Documentation page for new upcoming features.

For any comments, questions related to this guide, please feel free to contact us!