Porting Simple Android List Application to Symbian with Qt

This article explains how to port a simple Android list application to Symbian with Qt Quick. The article uses the example of a simple shopping list app to illustrate the main steps, and includes source code for both platforms. Note that the example is not a complete app, but contains all the main elements needed to understand how to port any list-based app.

Introduction

Android and Qt application development differ in many ways. In most cases no code can be copied as-is from one platform to the other. In addition, unlike Android, Qt is not actual a platform but a framework that several platforms support. Some, like MeeGo Harmattan, are partially built with Qt.

Of course, basic utility applications are built with similar components in most platforms, and these use common concepts even though their names may differ. As a result, this example matches the similar list components in both platforms.

Setting up the project

The only tool you need to develop Qt Quick apps for Symbian is the Qt SDK. For quick instructions on how to setup the development environment, see Getting started. Once the environment is set up, create a new project using the templates that come with the SDK. This whole process is thoroughly explained in Step 2 of Getting started, the mains steps of which are reproduced below:

1. Open Qt Creator
2. Select **File > New File or Project**
3. From the dialog select **Qt Quick Application** and click **Choose...**
4. Enter the name and the location for your new project
5. In Qt Quick Application selection dialog, select **Qt Quick Components for Symbian** and click **Next**
6. In the next dialog select targets for the project: Check **Symbian Device** and **Qt Simulator**
7. You don't need to change the automatically generated **Target UID3**. Just click **Next**
8. In Project Management dialog, click **Finish**

The equivalent of AndroidManifest.xml file in Qt is the project file: ***<your project name>*.pro**. It declares the resources used by your application e.g. capabilities such as network access, Qt modules and Qt Mobility APIs, source files etc. For this example you don't have to manually modify the project file at all; just let the Qt Creator manage it.

Now you should have your template project set up. Let's start implementing the UI in the next section.

Creating the UI

Basic Android applications are implemented using Java extended with Android APIs, and parts of the UI can be implemented with declarative XML. In Qt Quick, similar apps are implemented using a declarative language known as QML. The application logic can be implemented with JavaScript code which is included within the QML markup. It is also possible to use C++ and integrate it with your QML. While this provides great flexibility and improved performance, for simple apps the use of C++ is seldom required. In this article we do everything in QML.

In Android the views are implemented using Activity classes. For a simple list application Android has conveniently provided ListActivity class which has the ListView object already in it. In Qt Quick applications, the views are referred as Pages. There's no equivalent for `ListActivity` in Qt Quick but adding a QML ListView to a page is extremely simple.

Navigating between views in Android is done with Intents. The pages in Qt Quick apps are managed with the QML PageStack element.

Main view with list

Let's start with the main view. First, let's remove some generated bits from our **main.qml** file: erase the tool bar implementation:

```
import QtQuick 1.1
import com.nokia.symbian 1.1

PageStackWindow {
    id: window
```


(C) Copyright Nokia 2014. All rights reserved.

```
    initialPage: MainPage {}
    showStatusBar: true
    showToolBar: true
}
```

`PageStackWindow` is a Qt Quick Component that contains a page stack as a property. `initialPage` property defines the page that is pushed onto the page stack when the application is started. QML elements that you create can be accessed in the code by their filename without the **.qml** suffix. In the code snippet above we declare that the page that is pushed in the page stack on start-up is that implemented in **MainPage.qml**.

The template has created the **MainPage.qml** for us. Let's now add a QML ListView into our page. Naturally, the list view also requires a model (in this case we use QML ListModel element) which defines where to get the data:

```
import QtQuick 1.1
import com.nokia.symbian 1.1

Page {
    id: mainPage

    ListModel {
        id: shoppingListModel
    }

    ListView {
        anchors.fill: parent
        model: shoppingListModel

        delegate: ListItem {
            id: listItem

            Column {
                anchors.fill: listItem.paddingItem

                ListItemText {
                    role: "Title"
                    text: name
                }
                ListItemText {
                    role: "SubTitle"
                    text: count
                }
            }
        }
    }
}
```

As you can see, in QML we define IDs for our elements for accessing and referring to them in different parts in the code. Defining an ID is not necessary; if you don't need to refer to the element, you can just leave it out. For the content in the model you can declare properties by freely selecting the name of the property. In our shopping list model, we expect the items in the model to have two properties: `name` and `count`.

In Android, you can use the ready-made list items or create your own e.g. with Adapters. The list items in QML ListView are called delegates. In the snippet above we use QML ListItem component to implement a delegate. We could alternatively build a custom delegate with any QML element.

Let's now add a tool bar to our main page. Add the following code:

(C) Copyright Nokia 2014. All rights reserved.

```
Page {
    ...

    tools: ToolBarLayout {
        ToolButton {
            iconSource: "toolbar-back"
            onClicked: Qt.quit(); // Terminates the application
        }
        ToolButton {
            iconSource: "toolbar-add"

            onClicked: {
                // We'll add pushing the edit/add page to the stack here.
            }
        }
    }

    // This is Page component level here. A QML file can have only one root level
element!
}
```

Add a header for the `ListView`:

```
    ListView {
        ...
        header: ListHeading {
            id: listHeading

            ListItemText {
                anchors.fill: listHeading.paddingItem
                role: "Heading"
                text: "ShoppingList"
            }
        }
        ...
```

At this point we can see how our UI looks in the Simulator or on a device. Just add some hard coded elements to the model to see how the list view works:

```
    ListModel {
        id: shoppingListModel

        ListElement { name: "Item 1"; count: "2" }
        ListElement { name: "Item 2"; count: "4" }
        ListElement { name: "Item 3"; count: "6" }
        ListElement { name: "Item 4"; count: "8" }
        ListElement { name: "Item 5"; count: "10" }
    }
```

Clicking **Run** button should show us something like this:

(C) Copyright Nokia 2014. All rights reserved.

View for adding and editing list items

So you've already created few views for Android apps before, right? The usual way is to create `.xml` file which declares the layout and the IDs for components in it. Then implement an `Activity` derived class to manage the view.

In Qt we create a new QML file, in which the main (root) element implements a `Page` component. It will also contain the layout for the page. QML provides the usual layout suspects, namely: Column, Grid and Row. In addition you can layout the items using anchors. These tools help you to build any scalable layout you can imagine.

Here's the code for our `EditPage.qml`:

```
import QtQuick 1.1
import com.nokia.symbian 1.1

Page {
    property bool editing: true
    property string name
    property string count

    ListHeading {
        id: listHeading

        anchors {
            top: parent.top
            left: parent.left
            right: parent.right
        }

        z: 1 // Make sure that the heading is always on top
```

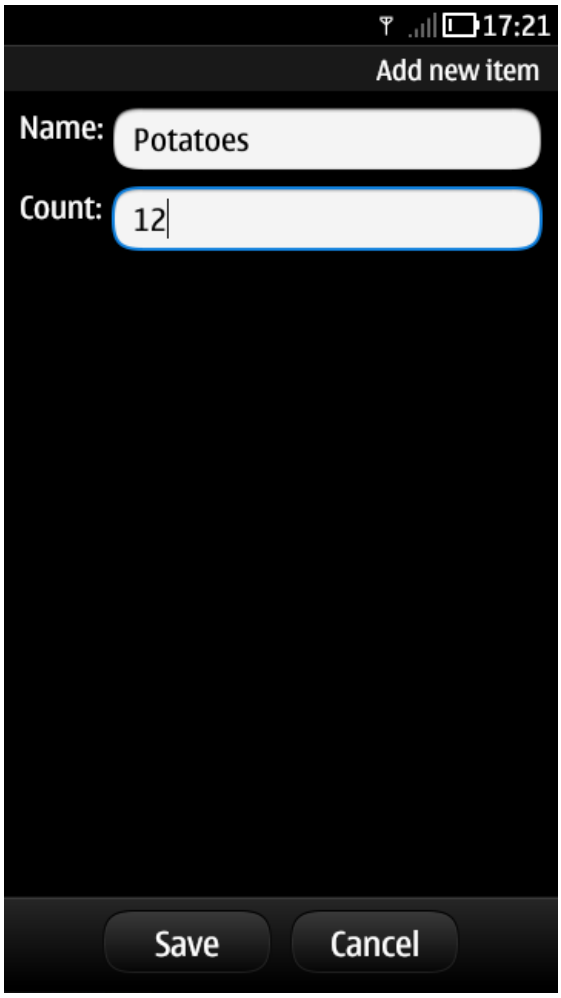

(C) Copyright Nokia 2014. All rights reserved.

```
        ListItemText {
            anchors.fill: listHeading.paddingItem
            role: "Heading"
            text: editing ? "Edit item" : "Add new item";
        }
    }

    Flickable {
        anchors {
            top: listHeading.bottom
            left: parent.left
            right: parent.right
            bottom: parent.bottom
        }

        Grid {
            anchors {
                top: parent.top
                left: parent.left
                right: parent.right
                margins: 10
            }

            rows: 2
            columns: 2
            spacing: 10

            Label {
                id: nameLabel
                width: 50
                height: nameTextField.height
                text: "Name:"
            }
            TextField {
                id: nameTextField
                width: parent.width - nameLabel.width - 10
            }
            Label {
                width: nameLabel.width
                height: nameLabel.height
                text: "Count:"
            }
            TextField {
                id: countTextField
                width: nameTextField.width
                inputMethodHints: Qt.ImhDigitsOnly
            }
        }
    }

    tools: ToolBarLayout {
        ToolButton {
            text: "Save"
            enabled: nameTextField.text.length && countTextField.text.length

            onClicked: {
                ...
```


(C) Copyright Nokia 2014. All rights reserved.

```
                pageStack.pop();
            }
        }
        ToolButton {
            text: "Cancel"
            onClicked: pageStack.pop();
        }
    }
}
```

And this is what it looks like (not too pretty but there's plenty of time to decorate it):



Context menu and other actions

In Android context menus are pretty straightforward to implement. Simplified, the steps are as folllows:

1. Create `.xml` file where you declare the menu items
2. In your `Activity` class implement `onCreateContextMenu()` and `onContextItemSelected()` methods, and finally
3. Register your list view for the context menu using `registerForContextMenu()` method.

In Qt Quick things are even more simple: you just have to fill your declarative bits with a touch of JavaScript logic. QML provides a convenient ContextMenu element which we use to create our context menu:

```
    ContextMenu {
        id: contextMenu

        content: MenuLayout {
            MenuItem {
                text: "Edit"
```


(C) Copyright Nokia 2014. All rights reserved.

```
                onClicked: {
                    if (selectedIndex >= 0) {
                        pageStack.push(Qt.resolvedUrl("EditPage.qml"),
                                       { editing: true,
                                         name:
shoppingListModel.get(selectedIndex).name,
                                         count:
shoppingListModel.get(selectedItem).count });
                    }
                }
            }
            MenuItem {
                text: "Delete"

                onClicked: {
                   ...
                }
            }
        }
    }
```

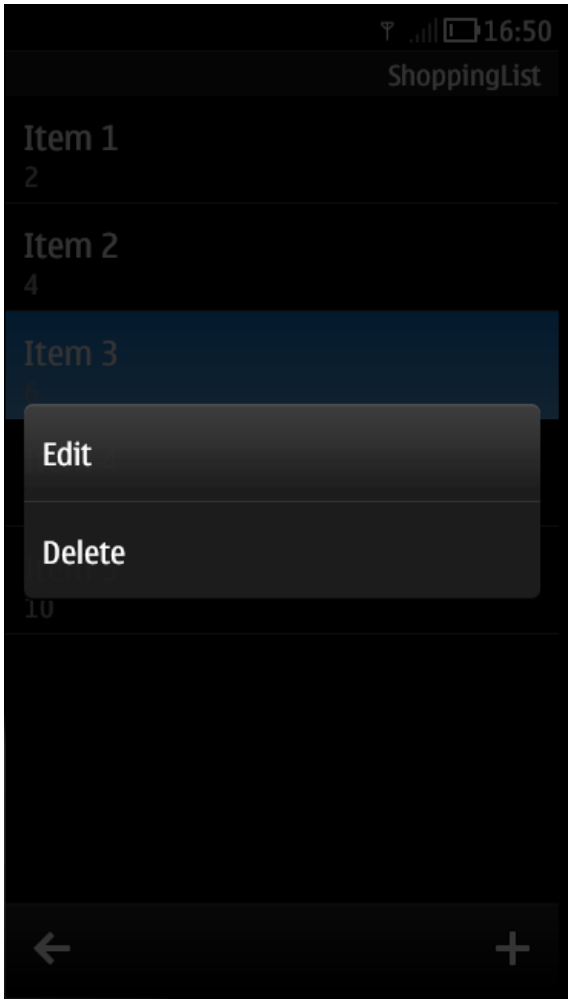
As you can see, in our `onClicked()` signal handler we push the edit page into the page stack and initialize some of its properties. The transition to the new page is done automatically by the framework. `selectedIndex` in the snippet above is declared on the main level of the page. It's a member where we store the selected item reference when an item is long pressed and that is also the place where we show the context menu:

```
    ListView {
        ...
        delegate: ListItem {
            id: listItem
            ...

            onPressAndHold: {
                selectedIndex = index;
                contextMenu.open();
            }
    ...
```

The context menu is automatically closed when a menu item is tapped or when the user taps outside the menu.

And this is how the context menu looks on Symbian:

(C) Copyright Nokia 2014. All rights reserved.



Now let's finally add functionality into the "add" button. In Android applications new pages are showed using `Intent` objects in the following fashion:

```
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        if (item.getItemId() == R.id.addMenuItem) {
            Intent intent = new Intent(this, EditView.class);
            startActivityForResult(intent, ADD_ITEM_REQUEST);
        }

        return true;
    }
```

In Android, if you simply want to invoke another activity then you have to use startActivity(Intent) but if you want some result from the called activity when it gets finish then you can use startActivityForResult(Intent, Int). So, in above written piece of code, you could also have called `startActivity()` in place of `startActivityForResult()` based on your requirements.

In QML we simply implement `onClicked()` signal handler of the main page tool bar:

```
    tools: ToolBarLayout {
        ToolButton {
            iconSource: "toolbar-back"
            onClicked: Qt.quit(); // Terminates the app
        }
        ToolButton {
            iconSource: "toolbar-add"

            onClicked: {
```

 (C) Copyright Nokia 2014. All rights reserved.

```
                // Show the edit page for adding a new item
                pageStack.push(Qt.resolvedUrl("EditPage.qml"), { editing: false });
            }
        }
    }
```

When we push the page into the page stack, we also initialize one of the properties on that page, boolean property named editing in this case. This is similar to equipping `Intent` objects with extras but with the difference that the `editing` property must be declared in the page code.

Now that our initial UI is good to go, let's, in the next section, proceed with the persistent storage for the actual data.

Persistent data storage

In Android, Content providers provide the means to store persistent application data. Regardless of the actual back-end, content providers enforce to use SQL-like query language. This favors SQL databases to be selected as the actual storage.

Qt provides many different means to store persistent data e.g. SQLite, files (including serialized objects), QSettings for application settings. In addition, Qt Mobility provides modules like Organizer and Publish and subscribe which also relate to persistent data but are meant for certain use-cases.

For the sake of simplicity, we use SQLite for this example. The database can be easily managed with JavaScript code. The whole implementation is encapsuled in a single JavaScript file, `ShoppingListDb.js`:

```
var DatabaseName = "ShoppingListDb";
var DatabaseDescription = "Shopping list items";
var TableName = "ShoppingList";


function createDb() {
    var db = openDatabaseSync(DatabaseName, "1.0", DatabaseDescription, 1000000);

    db.transaction(
        function(createDbTransaction) {
            createDbTransaction.executeSql('CREATE TABLE IF NOT EXISTS '
                                           + TableName + '(name TEXT, count INT)');
        }
    );
}


function addItem(name, count) {
    var db = openDatabaseSync(DatabaseName, "1.0", DatabaseDescription, 1000000);

    db.transaction(
        function(addItemTransaction) {
            addItemTransaction.executeSql('INSERT INTO ' + TableName
                                          + ' VALUES(?, ?)', [ name, count ]);
        }
    );
}


function updateModel(model) {
    var db = openDatabaseSync(DatabaseName, "1.0", DatabaseDescription, 1000000);

    db.transaction(function(getRowsTransaction) {
        var rows = getRowsTransaction.executeSql('SELECT * FROM ' + TableName);
```


(C) Copyright Nokia 2014. All rights reserved.

```
        model.clear();

        for (var i = 0; i < rows.rows.length; i++) {
            var item = rows.rows.item(i);
            model.append({"name": item.name,
                          "count": item.count});
        }
    });
}
```

Then import the JavaScript file in the QML files where needed and call the functions. In `EditPage.qml` we add a new item when the **Save** button is tapped:

```
import QtQuick 1.1
import com.nokia.symbian 1.1
import "ShoppingListDb.js" as ShoppingListDb

Page {
    ...
    property ListModel model
    ...
    tools: ToolBarLayout {
        ToolButton {
            text: "Save"
            enabled: nameTextField.text.length && countTextField.text.length

            onClicked: {
                ShoppingListDb.addItem(nameTextField.text, countTextField.text);

                if (model) {
                    ShoppingListDb.updateModel(model);
                }

                pageStack.pop();
            }
        }
        ToolButton {
            text: "Cancel"
            onClicked: pageStack.pop();
        }
    }
...
```

Summary

That's it, now we've successfully ported the main features of a basic list app from Android to Qt. To learn more about porting follow the links in the next Related documentation section.

Related documentation

- API Mapping Tool: Android to Qt
- Porting guide for Android developers
- Qt code examples

(C) Copyright Nokia 2014. All rights reserved.

(C) Copyright Nokia 2014. All rights reserved.