Implementing an Autocomplete Line Edit component for Qt Quick

This article shows how to develop and use a simple *line-edit with autocompletion* as a reusable Qt Quick Component. The demo application in this article uses the model for autocompletion from Qt C++ code, making this article useful for learning how to use C++ code with QML as well. You can also watch the video of the demo application here and download its source code from here.

10 Apr 2011

Basic Structure

The structure for this component is relatively simple and it has two main elements.

1. Line edit
2. Autocompletion list

The line-edit would be a `BorderImage` element containing a `TextInput` element. For the Autocompletion list we would use a `ListView` and since we are making a component - we would like the model of the `ListView` and some properties of list (model, border color of list items, border image and item color) to be customizable from outside.

The QML file of the component would look like the following at a higher level,

```
// ===AutocompleteLineEdit.qml===//
Rectangle
{
 // == Properties accessible from outside == //
 property variant fetchedModel
 property string delegateItemBorderColor: "white"
 property string delegateItemColor: "white"
 property string editBarImage: "editor.png"

 // == BorderImage for Line Edit == //
 BorderImage {
  id: editBar
  source: editBarImage

  // == TextInput for Line Edit == //
  TextInput{
```


(C) Copyright Nokia 2014. All rights reserved.

```
   id: input
   }
  }
  ListView{
  id: listView
  model: fetchedModel
  delegate: Text{
     }
  }
 }
```

LineEdit and Autocomplete ListView

With the basic structure in place, it is now time to slightly align and beautify the `editBar` and the autocompletion `listView`. For the `editBar`, we add some anchors to the `TextInput`. Also note that we also expose the propery `text` of the `TextInput`, so that it can be accessed from outside.

```
BorderImage {
 id: editBar
 width: parent.width
 height: 30
 border.top: 10
 border.bottom: 10
 border.left: 10
 border.right: 10

 source: editBarImage

 // to expose to this property as text
 property alias text: input.text

   // == TextInput for Line Edit == //
 TextInput{
  id: input
  width: parent.width
  height: 30
  anchors.verticalCenter: parent.verticalCenter
  anchors.left:  parent.left
  anchors.leftMargin: 5
  font.pixelSize: 20
  }
}
```

Now for the `Listview`, we declare the delegate element as a `Rectangle` containg a `Text` element. We add number animations for the height property of the delegate with an easing curve. Adding these animations with easing curves makes the list to appear nicely, when items are added or removed from the `ListView`.

```
// == Autocomplete List == //
ListView{
 id: listView
 width: parent.width
 height: 200
 anchors.top: editBar.bottom
```

 (C) Copyright Nokia 2014. All rights reserved.

```
 model: fetchedModel
 delegate: Rectangle{
  id: delegateItem
  width: parent.width
  height:  30
  border.color: delegateItemBorderColor
  color: delegateItemColor
  Text {
   anchors.fill: parent
   anchors.leftMargin: 5
   text: modelData
  }
 }
}

ListView.onAdd: NumberAnimation { target: listView.delegate; property: "height";
duration: 500; easing.type: Easing.OutInQuad}
ListView.onRemove: NumberAnimation { target: listView.delegate; property: "height";
duration: 250; easing.type: Easing.InOutQuad }
```

Implementing the logic

We now start implementing some logic. To understand further additions more clearly, lets take an example data model that we would provide to this component in our demo application. We would be using data model from Wikipedia API (using action opensearch) in our demo application. The Line-edit in our demo application would provide autocompletion for all Wikipedia available articles.

- **Wikipedia API**: http://en.wikipedia.orghttp://developer.nokia.com/w/api.php

To fetch data from the Wikipedia API, we need to make a *HTTP get* request to the API. For example, to query the API for the word "Nokia" we need to make the following get request,

```
http://en.wikipedia.orghttp://developer.nokia.com/w/api.php?
action=opensearch&search=Nokia
```

Response from the API for the above request would be,

```
["Nokia",["Nokia","Nokia Siemens Networks","Nokia Arena, Tel Aviv","Nokia N-Gage","Nokia
N900","Nokia N95","Nokia 5800 XpressMusic","Nokia, Finland","Nokia Theatre at Grand
Prairie","Nokia 770 Internet Tablet"]]
```

We would use Qt code to make this *HTTP get* request.

```
QStringList Logic::fetchModel(QString aString)
{
    QString urlString("http://en.wikipedia.orghttp://developer.nokia.com/w/api.php?
action=opensearch&search=");
    urlString.append(aString);
    urlString.append("&format=json&callback=spellcheck");

    QUrl url(urlString);
    nam->get(QNetworkRequest(url));

    return iStringList;
}
```


(C) Copyright Nokia 2014. All rights reserved.

We can add the macro `Q_INVOKABLE` to the definition of the `fetchModel` function to allow them to be accessed from QML. However, it is not that straightforward to do so since the *HTTP get* request is asynchronous. Here comes the signals and slot in the picture!

We are going to use a couple of signal and slots here. One would be the signal when the HTTP response is finished,

```
connect(nam, SIGNAL(finished(QNetworkReply*)), this,
SLOT(finishedSlot(QNetworkReply*)));
```

We also connect a signal `updateModel` from Qt to QML. In the `finishedSlot()`, we would parse the response to our requirements and emit the updateModel signal, which would be received in QML.

```
// Conecting signal from Qt to QML
rootObject = dynamic_cast<QObject*>(viewer.rootObject());
QObject::connect(this, SIGNAL(responseReady(QVariant)), rootObject,
SLOT(updateModel(QVariant)));
```

The slot in QML would be implemented as a normal JavaScript function,

```
// == Slot for Reponse parsed ==//
function updateModel(fModel){
 lineEdit.fetchedModel= fModel
}
```

Picking up from where we stopped on the QML code, we need to add an event handler to the line-edit to make a HTTP get request from the Qt code, whenever the text in the line-edit is changed. We would simply do it by adding the following code to our line-edit.

```
onTextChanged: {
 // == Http get request when the Text is changed== //
  logic.fetchModel(editBar.text)
}
```

What is the use of an autocompletion list when the user cannot interact with it? Wouldn't it make sense to for the text in the line-edit to be set to the text that user selects from Autocompletion list? Let's add an `MouseArea` to the `ListView` to obtain this,

```
MouseArea{
anchors.fill:  parent
onClicked: {
 // == Setting text when an item is selected from the autocomplete list == //
 editBar.text = modelData
 }
}
```

The component is now ready to use in Qt Quick applications. We can use any other QML models (ListModel, etc) with this component as well.

Source code summary

To sumarize, let's glance at the what each file finally contains.

- **AutocompleteLineEdit.qml:** our QML component
- **Logic.h** and **Logic.cpp:** logic implementation for our QML component
- **main.qml:** main QML file of our demo application, where we use our QML component

 (C) Copyright Nokia 2014. All rights reserved.

AutocompleteLineEdit.qml

```
import QtQuick 1.0

Rectangle
{
    width: 360
    height: 640
    color: "transparent"

    // == Properties accessible from outside == //
    property variant customLogic
    property variant fetchedModel
    property string delegateItemBorderColor: "white"
    property string delegateItemColor: "white"
    property string editBarImage: "editor.png"


    // == BorderImage for Line Edit == //
    BorderImage {
        id: editBar
        width: parent.width
        height: 30
        border.top: 10
        border.bottom: 10
        border.left: 10
        border.right: 10

        source: editBarImage

        property alias text: input.text

        // == TextInput for Line Edit == //
        TextInput{
            id: input
            width: parent.width
            height: 30
            anchors.verticalCenter: parent.verticalCenter
            anchors.left:  parent.left
            anchors.leftMargin: 5
            font.pixelSize: 20
            onTextChanged: {
                // == Http get request when the Text is changed== //
                    logic.fetchModel(editBar.text)
            }
        }
    }

    // == Autocomplete List == //
    ListView{
        id: listView
        width: parent.width
        height: 200
        anchors.top: editBar.bottom

        model: fetchedModel
```


(C) Copyright Nokia 2014. All rights reserved.

```
        delegate: Rectangle{
            id: delegateItem
            width: parent.width
            height:  30
            border.color: delegateItemBorderColor
            color: delegateItemColor
            Text {
                anchors.fill: parent
                anchors.leftMargin: 5
                text: modelData
            }
            MouseArea{
                anchors.fill:  parent
                onClicked: {
                    // == Setting text when an item is selected from the autocomplete
list == //
                    editBar.text = modelData
                }
            }
        }

        ListView.onAdd: NumberAnimation { target: listView.delegate; property: "height";
duration: 500; easing.type: Easing.OutInQuad}
        ListView.onRemove: NumberAnimation { target: listView.delegate; property:
"height";  duration: 250; easing.type: Easing.InOutQuad }

    }
}
```

Logic.h

```
#ifndef LOGIC_H
#define LOGIC_H

#include <QObject>
#include <QStringList>
#include <QDir>
#include <QDebug>

#include <QDeclarativeContext>
#include "qmlapplicationviewer.h"
#include <QNetworkAccessManager>

class Logic: public QObject
{
    Q_OBJECT
public:
    Logic(QObject *parent = 0);
    Q_INVOKABLE QStringList fetchModel(QString aString);

public slots:
    void finishedSlot(QNetworkReply* reply);
    void ParseSpellResponse(QString);

signals:
```


(C) Copyright Nokia 2014. All rights reserved.

```
    void responseReady(QVariant);
private:
    QmlApplicationViewer viewer;
    QDeclarativeContext *ctxt;
    QNetworkAccessManager* nam;
    QObject *rootObject;
    QStringList iStringList;
    QString iQueryString;
};



#endif // LOGIC_H
```

Logic.cpp

```
#include "Logic.h"
#include <QNetworkProxyFactory>
#include <QNetworkReply>
#include <QGraphicsObject>


Logic::Logic(QObject *parent): QObject(parent)
{

    // Empty String List
    iStringList.clear();
    nam = new QNetworkAccessManager(this);
    connect(nam, SIGNAL(finished(QNetworkReply*)),
            this, SLOT(finishedSlot(QNetworkReply*)));


    viewer.setOrientation(QmlApplicationViewer::ScreenOrientationAuto);

    // Exposing Logic class to QML
    ctxt = viewer.rootContext();
    ctxt->setContextProperty("logic", this);

    // Setting QML source file
    viewer.setMainQmlFile(QLatin1String("qml/AutocompleteLE/main.qml"));

    // Conecting signal from Qt to QML
    rootObject = dynamic_cast<QObject*>(viewer.rootObject());
    QObject::connect(this, SIGNAL(responseReady(QVariant)), rootObject,
SLOT(updateModel(QVariant)));

    viewer.showFullScreen();
}

QStringList Logic::fetchModel(QString aString)
{
    iQueryString = aString;
    QString urlString("http://en.wikipedia.orghttp://developer.nokia.com/w/api.php?
action=opensearch&search=");
    urlString.append(aString);
```


(C) Copyright Nokia 2014. All rights reserved.

```
    urlString.append("&format=json&callback=spellcheck");

    QUrl url(urlString);
    nam->get(QNetworkRequest(url));

    return iStringList;
}


void Logic::finishedSlot(QNetworkReply* reply)
{
    // Reading attributes of the reply
    // e.g. the HTTP status code
    QVariant statusCodeV =
            reply->attribute(QNetworkRequest::HttpStatusCodeAttribute);
    // Or the target URL if it was a redirect:
    QVariant redirectionTargetUrl =
            reply->attribute(QNetworkRequest::RedirectionTargetAttribute);

    // no error received?
    if (reply->error() == QNetworkReply::NoError)
    {
        // read data from QNetworkReply here

        //  Reading bytes form the reply
        QByteArray bytes = reply->readAll();  // bytes
        QString string(bytes); // string

        ParseSpellResponse(string);

    }
    // Some http error received
    else
    {
        // handle errors here
    }


}

void Logic::ParseSpellResponse(QString astring)
{
    // == Parsing the received response == //

    astring = astring.right(astring.length()-QString("spellcheck(["").length());
    astring = astring.left(astring.length()-QString("]])\"").length());

    astring = astring.replace("\"", "");
    QStringList stringlist =astring.split(",");
    stringlist.removeAt(0);

    stringlist[0] = stringlist[0].right(stringlist[0].length()-1);

    iStringList = stringlist;
    if (iStringList[0]="" || (iStringList.length() == 1 && iStringList[0] ==
iQueryString))
    {
```


(C) Copyright Nokia 2014. All rights reserved.

```
        // List is empty with just one element, so clear it!
        iStringList.clear();
    }
    responseReady(iStringList);
}
```

---

main.qml

```
import QtQuick 1.0

Image {
    width: 360
    height: 640
    source: "background.png"


    // == Slot for Reponse parsed ==//
    function updateModel(fModel){
        lineEdit.fetchedModel= fModel
    }

    Text{
        width:  200
        anchors.bottom: lineEdit.top
        anchors.bottomMargin: 20
        anchors.horizontalCenter: parent.horizontalCenter
        text: "Results from Wikipedia"
        font.bold: true
        font.pixelSize: 20
    }

    // == Autocomplete LineEdit ==//
    AutoCompleteLineEdit{
        id: lineEdit
        width: parent.width-20
        height: 30
        anchors.centerIn: parent
        customLogic: logic
    }
}
```

Video demo

The video below shows the demo app that we created above in the above sections.

The media player is loading...

Exercise for the readers

I leave it as an exercise for the readers to aggregate content from this article and make a working demo - hopefully you will learn more while doing so. If you face any problems to get this running, you may refer to the source code here - File:AutocompleteLE.zip. This article explained in detail how you can create an Autocompletion QML UI component and how you can interface it with Qt code (that is, using Qt logic and engine for your Qt Quick applications).

If you have any questions while you try to use this component or while building your own components - kindly use the comments tab of this article to discuss or send me an email from this link.

 (C) Copyright Nokia 2014. All rights reserved.

(C) Copyright Nokia 2014. All rights reserved.