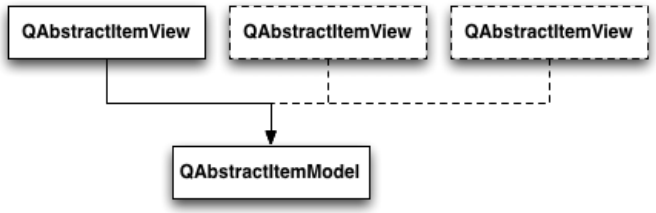
Incremental Search with Qt

Incremental search is the current standard user interface paradigm for implementing search. The search is performed while the user is typing. Classic search approaches require the user to complete a search request in a form before sending it. In the worst case, the user has to wait for the search to be finished then. Many popular applications implement some sort of incremental search, for example the iTunes jukebox.

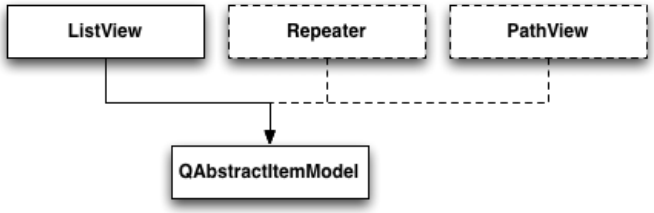
28 Nov 2010

Implementing incremental search in Qt

The typical architecture of an application that displays a number of records is shown below. One or multiple views show the data of a common data source. The datasource is a subclass of QAbstractItemModel and the views are subclasses of QAbstractItemView. For a general overview, see Model/View Programming in the Qt reference.



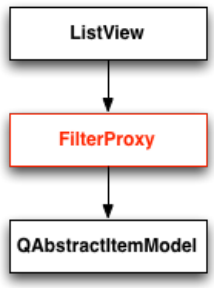
If you are using QML, you will have at one or more views in a QDeclarativeView that show the data of a common data source. The datasource is still based on QAbstractItemModel.



Note: Although QML-views can use other data sources, for the full functionality and especially for incremental search, a model based on QAbstractItemModel is required.

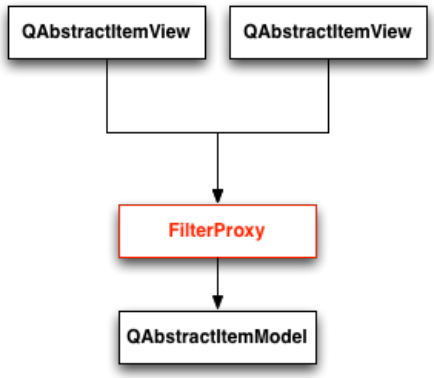
QSortFilterProxyModel

The key to incremental search is to have another party between your model and views: The so called QSortFilterProxyModel.

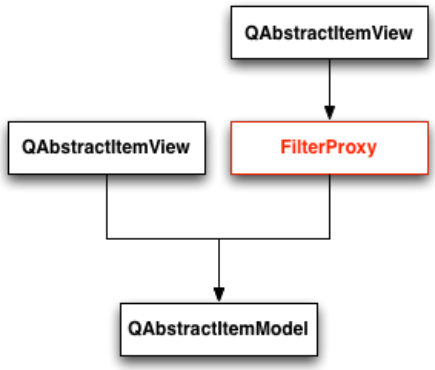


You can filter the data for all your views:

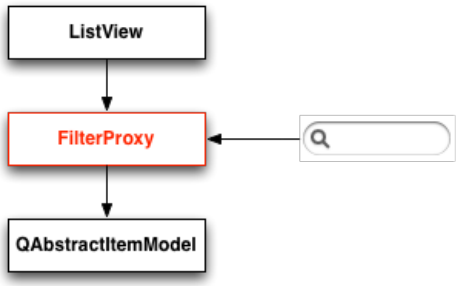
(C) Copyright Nokia 2014. All rights reserved.



or just filter one view and show all information in the other view.



When searching for something, we need of course the information what to search. This information is usually provided by a lineedit:

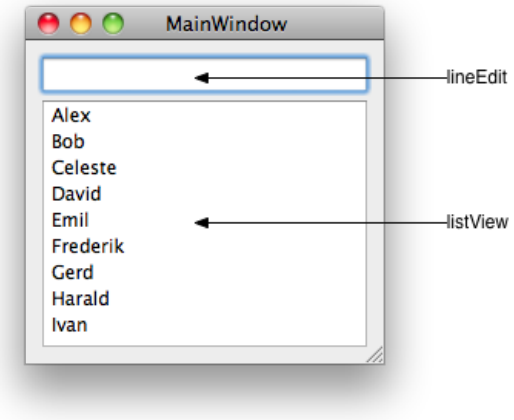


QSortFilterProxyModel can filter for a regular expression but for a lineedit, searching for a fixed string is more appropriate. Call QSortFilterProxyModel::setFilterFixedString to search for fixed string instead of a regular expression. If your search condition cannot be expressed as a regular expression, you will have to subclass as explained below.

An example using QWidgets

We design a form with a QLineEdit and a QListView. These are laid out using a QVBoxLayout. The form is based on QMainWindow but could be based on any other of Qt's widgets.

(C) Copyright Nokia 2014. All rights reserved.




```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

class QStringListModel;      // Forward declare of our model class
class QSortFilterProxyModel; // Forward declare of the filter proxy

namespace Ui {
    class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;
    QStringListModel* m_model;  // Instance variable for the model
    QSortFilterProxyModel* m_filter; // Instance variable for the filter proxy
};

#endif // MAINWINDOW_H
```

In the constructor for the MainWindow, we initialize our members and connect set the datamodel as source model for the QSortFilterProxyModel. We also connect the lineEdit to the proxy and set the proxy as model for the listView.

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    QStringList data;
    data << "Alex" << "Bob" << "Celeste" << "David"
         << "Emil" << "Frederik" << "Gerd" << "Harald" << "Ivan";
```


(C) Copyright Nokia 2014. All rights reserved.

```
    m_model = new QStringListModel(this);
    m_model->setStringList(data); // Fill model with fake data

    m_filter = new QSortFilterProxyModel(this);
    m_filter->setSourceModel(m_model); // Set complete model as source model to the
proxy

    // Set the string to search whenever the text in the lineEdit was changed
    connect(ui->lineEdit, SIGNAL(textChanged(QString)),
            m_filter, SLOT(setFilterFixedString(QString)));

    m_filter->setFilterCaseSensitivity(Qt::CaseInsensitive);

    ui->listView->setModel(m_filter); // Finaly set the filtered model as data source
for the listView
}
```

Tip: If your model data changes dynamically, you have to set dynamicSortFilter for the filter proxy to update when your model changes

Download the QWidget-based example

Download the complete example File:Filterproxy.zip

An example using Qt Quick and QML/declarative

When using QtQuick for the user interface, we need some modifications to our application. Both the lineEdit and the listView need to be removed in favor of a QDeclarativeView. Instead of calling a slot from our QDeclarativeScene to set the search string, we bind the lineEdit in the QDeclarativeScene to a property that contains the search string. Because the QMainWindow is the only class that is subclassed, we put the property there and call it "searchPredicate". We have a getter and setter for the property as well a notify-signal:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

class QStringListModel;
class QSortFilterProxyModel;

namespace Ui {
    class MainWindow;
}

class MainWindow : public QMainWindow
{
Q_OBJECT
Q_PROPERTY(QString searchPredicate READ searchPredicate WRITE setSearchPredicate NOTIFY
searchPredicateChanged)

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

    QString searchPredicate() const; // Getter
```


(C) Copyright Nokia 2014. All rights reserved.

```
public slots:
    void setSearchPredicate(const QString & nSearchPredicate); // Setter

signals:
    void searchPredicateChanged(const QString & nSearchPredicate); // Notify-signal

private:
    Ui::MainWindow *ui;
    QStringListModel* m_model;
    QSortFilterProxyModel* m_filter;
    QString m_searchPredicate; // Variable to store our search predicate
};

#endif // MAINWINDOW_H
```

In the implementation, we need to publish both the MainWindow because of the property to the context as well as the model. A nicer way from the software engineering point of view would be subclassing the proxy model and add the property there. After publishing both objects to the context, we load the QML-scene from a resource file. In the setter of the property, we check whether the new value is actually different and only then, we set the fixed string of the proxy and emit the notify-signal.

```
#include "MainWindow.h"
#include "ui_MainWindow.h"

#include <QStringListModel>
#include <QSortFilterProxyModel>

#include <QDeclarativeContext>

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    QStringList data;
    data << "Alex" << "Bob" << "Celeste" << "David"
         << "Emil" << "Frederik" << "Gerd" << "Harald" << "Ivan";

    m_model = new QStringListModel(this);
    m_model->setStringList(data);

    m_filter = new QSortFilterProxyModel(this);
    m_filter->setSourceModel(m_model);

    m_filter->setFilterCaseSensitivity(Qt::CaseInsensitive);

    ui->declarativeView->rootContext()->setContextProperty("controller", this);
    ui->declarativeView->rootContext()->setContextProperty("filteredModel", m_filter);

    ui->declarativeView->setSource(QUrl("qrc:/main.qml"));
}
...
QString MainWindow::searchPredicate() const
{
    return m_searchPredicate;
}
```


(C) Copyright Nokia 2014. All rights reserved.

```
void MainWindow::setSearchPredicate(const QString & nSearchPredicate)
{
    if (nSearchPredicate != m_searchPredicate) {
        m_searchPredicate = nSearchPredicate;
        m_filter->setFilterFixedString(searchPredicate());
        emit searchPredicateChanged(searchPredicate());
    }
}
```

In the QML-File, a textInput and a listView are declared. The text-property of the textInput is bind to the searchPredicate-property of the MainWindow. The model-property of the listView is bind to the proxy-model that was published as filteredModel. Note that we need an explicit binding back to the C++-code because QML-bindings usually only work in one direction: From C++ to the QML-scene.

```
import Qt 4.7

Rectangle {
    width: 240
    height: 320

    TextInput {
        id: textInput
        x: 0
        y: 0
        anchors.top: parent.top
        anchors.left: parent.left
        anchors.right: parent.right
        height: 20
        text: controller.searchPredicate
        anchors.rightMargin: 1
        anchors.leftMargin: 0
        anchors.topMargin: 0
        selectByMouse: true
 focus: true
    }

    ListView {
        id: listView
        model: filteredModel
        anchors.left: parent.left
        anchors.top: textInput.bottom
        anchors.right: parent.right
        anchors.bottom: parent.bottom
        delegate: Text {
            text: display
        }
    }

    Binding {
        target: controller; property: "searchPredicate"; value: textInput.text
    }
}
```

Download the QML-Example

(C) Copyright Nokia 2014. All rights reserved.

Subclassing QSortFilterProxyModel

There is cases where a regular expression based decision whether to show a row or not is not possible. For example, if you are implementing a music player and that music player has the ability to rate songs. Now you only want to show songs with 3 or more stars. This can be easily done by subclassing QSortFilterProxyModel. Consider the ranking of a song to be in column 1 of your model. Instead of harcoding a 3, we use a variable called m_minrank to store our minimum star limit.

Reimplement filterAcceptsRow as follows:

```
bool RankingFilter::filterAcceptsRow(int source_row, const QModelIndex &source_parent)
const
{
    return sourceModel()->data(sourceModel()->index(source_row,
SongModel::RatingColumn)).toInt() >= m_minrank;
}
```

We need another slot to dynamically adjust the limit. We first check whether the new minimum ranking is actually different from the previous one and if yes, store that value for comparison and call invalidateFilter to tell the filter to reevaluate all filtered rows.

```
void RankingFilter::setMinimumRanking(int nminrank)
{
    if (m_minrank != nminrank) {
        m_minrank = nminrank;
        invalidateFilter();
    }
}
```

Download the example

File:Subclass filter.zip

Mapping indices

The sourcemodel and the filtered model have a different set of QModelIndices. If you take for example the currently selected row from a view that has a filtered model as source, you will have to map this index first before it becomes valid in context of the original model. QSortFilterProxyModel has the two methods mapToSource and mapFromSource for exactly this task.

Search boxes

Incremental search usually uses a search box that shows a gray string that either says "Search" or more specific where to search or which search engine is to be used. This text disappears when the user enters text but a clear-button appears. There is currently no stock component in Qt for such a special search box but a number of implementations exist:

- A native Qt search box: http://labs.qt.nokia.com/2007/06/06/lineedit-with-a-clear-button/
- A native carbon or cocoa search box for the Mac example in the Qt distribution: demos/mainwindow/macmainwindow
- A QML implementation as example in the Qt distribution: examples/declarative/ui-components/searchbox

See Also

- http://doc.qt.nokia.com/4.7/itemviews-customsortfiltermodel.html
- Basic Sort/Filter Model Example from the Qt distribution: itemviews/basicsortfiltermodel

(C) Copyright Nokia 2014. All rights reserved.

(C) Copyright Nokia 2014. All rights reserved.