Table of content

Filling and reading QML UI forms from Python

This PySide tutorial shows you how to create a "classic" form-based UI with the Colibri QML Components and have it filled and controlled by Python code. There are several ways to do this, and depending on your use case, there might be a better method. Please also note that in this example, the controller code knows a bit about the UI (or rather: the UI has to inform the controller which widgets are to be filled), which might not be desired.

CarAnalogy.py

Import the required modules

We need the QtCore module for QObject, the QtGui module for QApplication and the QtDeclarative module for the QML View (QDeclarativeView):

```
import sys

from PySide import QtCore, QtGui, QtDeclarative
```

Define a Car as QObject

This is simply the Python version of a normal QObject with 4 properties:

* model (String) – The car name
* brand (String) – The company who made the card
* year (int) – The year it was first produced
* inStock (bool) – If the car is still in stock at the warehouse

```
class Car(QtCore.QObject):
    def __init__(self, model='', brand='', year=0, in_stock=False):
        QtCore.QObject.__init__(self)
        self.__model = model
        self.__brand = brand
        self.__year = year
        self.__in_stock = in_stock

    changed = QtCore.Signal()

    def _model(self): return self.__model
    def _brand(self): return self.__brand
    def _year(self): return self.__year
    def _inStock(self): return self.__in_stock

    def _setModel(self, model):
        self.__model = model
        self.changed.emit()

    def _setBrand(self, brand):
        self.__brand = brand
        self.changed.emit()

    def _setYear(self, year):
        self.__year = year
        self.changed.emit()

    def _setInStock(self, in_stock):
```

```
        self.__in_stock = in_stock
        self.changed.emit()

    model = QtCore.Property(str, _model, _setModel, notify=changed)
    brand = QtCore.Property(str, _brand, _setBrand, notify=changed)
    year = QtCore.Property(int, _year, _setYear, notify=changed)
    inStock = QtCore.Property(bool, _inStock, _setInStock, notify=changed)
```

The controller to fill the form and react to events

This is another QObject that takes a list of cars as constructor parameter. It also remembers the current position in the list of cars. There are three slots that are visible to the "outside" (QML in our case):

* prev – Go to the previous item
* next – Go to the next item
* init – Show the first item

All these slots take a QObject as parameter, and from the QML file, we will pass the root object there, which has a property widgets where we save a dictionary of mappings from name to QML component. The fill function takes care of filling in the data of the current car into the QML widgets.

```
class Controller(QtCore.QObject):
    def __init__(self, lst):
        QtCore.QObject.__init__(self)
        self._lst = lst
        self._pos = 0

    def fill(self, widgets):
        widgets['model'].setProperty('text', self._lst[self._pos].model)
        widgets['brand'].setProperty('text', self._lst[self._pos].brand)
        widgets['year'].setProperty('value', self._lst[self._pos].year)
        widgets['inStock'].setProperty('checked', self._lst[self._pos].inStock)
     widgets['position'].setProperty('text', '%d/%d' % (self._pos+1, len(self._lst)))

    @QtCore.Slot(QtCore.QObject)
    def prev(self, root):
        print 'prev'
        self._pos = max(0, self._pos - 1)
        self.fill(root.property('widgets'))

    @QtCore.Slot(QtCore.QObject)
    def next(self, root):
        print 'next'
        self._pos = min(len(self._lst) - 1, self._pos + 1)
        self.fill(root.property('widgets'))

    @QtCore.Slot(QtCore.QObject)
    def init(self, root):
        print 'init'
        self.fill(root.property('widgets'))
```

Example data

Here is some example data, so that we can use our example and click through a list of cars:

```
cars = [
        Car('Model T', 'Ford', 1908),
        Car('Beetle', 'Volkswagen', 1938, True),
        Car('Corolla', 'Toyota', 1966),
        Car('Clio', 'Renault', 1991, True),
        Car('Ambassador', 'Hindustan', 1958),
        Car('Uno', 'Fiat', 1983, True),
        Car('Ibiza', 'Seat', 1984, True),
]
```

Putting it all together

We first need to create the controller, which then also knows about our cars. Then, there is some housekeeping that we need to do – create a QApplication, create the QDeclarativeView and set its resizing mode (so that the root object in QML is always as big as the window).

We then get the root context and expose the controller and the cars list to it (if you look closely, we don't really need the cars themselves). Then, we load the QML file, show the view and start the application.

```
controller = Controller(cars)

app = QtGui.QApplication(sys.argv)

view = QtDeclarative.QDeclarativeView()
view.setResizeMode(QtDeclarative.QDeclarativeView.SizeRootObjectToView)

ctx = view.rootContext()

for name in ('controller', 'cars'):
    ctx.setContextProperty(name, locals()[name])

view.setSource(__file__.replace('.py', '.qml'))
view.show()

app.exec_()
```

CarAnalogy.qml

This is the user interface of our application. We only use the controller in the UI, and we also only use it for initialization and when buttons are clicked.

```
import Qt 4.7
import "colibri"

Rectangle {
    id: page

    property variant widgets

    width: 800
    height: 480
```

```
    Grid {
        id: grid
        columns: 2
        anchors.centerIn: parent
        spacing: 10

        Row {
            CLButton { text: "←"; onClicked: { controller.prev(page) } }
            CLButton { text: "→"; onClicked: { controller.next(page) } }
        }

        Text { id: position; text: " " }

        Text { text: "Model:" }

        CLLineEdit { id: model }

        Text { text: "Brand:" }

        CLLineEdit { id: brand }

        Text { text: "Year:" }

        Column {
            spacing: 10
            CLSlider {
                id: year
                minimum: 1900
                maximum: 2010
            }
            Text {
                text: year.value
            }
        }

        Text { text: " " }

        Row {
            spacing: 10
            CLCheckBox { id: inStock }
            Text { text: "In Stock" }
        }
    }

    Component.onCompleted: {
        widgets = {
            'position': position,
            'model': model,
            'brand': brand,
            'year': year,
            'inStock': inStock,
        }
        controller.init(page)
    }
}
```

How the example app looks like

Simply start the resulting app with python CarAnalogy.py and you should get something like this: