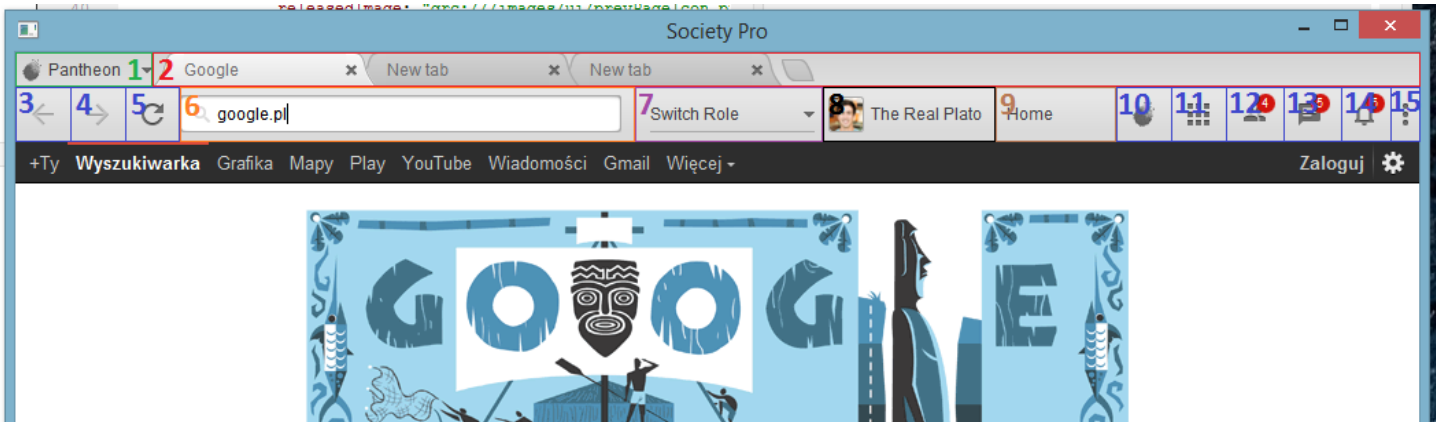
In your project, there are only 4 things that you need to know:

0) project structure
1) how to expose your data into QML (useful in handling app settings/data models)
2) how to make a connection between QML signal and C++ slot
3) how to make a connection between C++ signal and QML slot

Starting from the beginning, application consists mainly of components that we placed into "/components" directory of resources file. Most of them you can see below:



Each component are represented by:
1 – PantheonDropdown.qml
2 – TabViewHeader.qml
3 – ToolButton.qml
4 – ToolButton.qml
5 – ToolButton.qml
6 – LocationBar.qml
7 – RolesDropdown.qml
8 – RoleInfo.qml
9 – HomeButton.qml
10 – CreationStackButton.qml (inherits from ToolButton.qml)
11 – ApplicationsButton.qml (inherits from ToolButton.qml)
12 – PeersButton.qml (inherits from ToolButton.qml)
13 – MessagesButton.qml (inherits from ToolButton.qml)
14 – NotificationsButton.qml (inherits from ToolButton.qml)
15 – MoreButton.qml (inherits from ToolButton.qml)

Most of these Items exists as a content of some Tab. Tabs are created dynamically so for example LocationBar/RolesDropdown instances can exists more than once in the whole application. We are guessing that the data model used in MessagesButton.qml (for example) will be always the same. Remember that you don't need to make an individual instances of your list model for each Tab in application. You need to create only one (using singleton is good idea here) and pass it to QML as (for example) global object.

So let's say we want to store application width and height in some settings file. You can do that with all 3 ways that was mentioned at the beginning. But the best idea for this kind of problem will rely on the appropriate singleton created on c++ side.

In main.qml we have:

```
import QtQuick 2.3
import QtQuick.Controls 1.2 as QuickControls
import "qrc:///components/ui"

QuickControls.ApplicationWindow {
    id: appWindow
    objectName: "appWindow"
    visible: true
    minimumWidth: 1024
    width: 1024
    height: 748
    title: qsTr("Society Pro")
```

We can replace it by:

```
import QtQuick 2.3
import QtQuick.Controls 1.2 as QuickControls
import "qrc:///components/ui"

QuickControls.ApplicationWindow {
    id: appWindow
    objectName: "appWindow"
    visible: true
    minimumWidth: 1024
    width: globalThing.appWidth
    height: globalThing.appHeight
    title: qsTr("Society Pro")
```

Of course QML engine doesn't know what is that "globalThing", so we need to provide such information before engine will start loading. But before, lets create appropriate c++ class:

global.h

```
#ifndef GLOBAL_H
#define GLOBAL_H

#include <QObject>

class global : public QObject
{
    Q_OBJECT
private:
    explicit global(QObject *parent = 0);
    explicit global(const global &g) {}

private:
    int m_appWidth;
    int m_appHeight;

public:
    int appWidth() const { return m_appWidth; }
    int appHeight() const { return m_appHeight; }
    void setAppWidth(int w);
    void setAppHeight(int h);

signals:
    void appWidthChanged();
    void appHeightChanged();

    friend global &globalInstance();
};

global &globalInstance();

#endif // GLOBAL_H
```

global.cpp

```cpp
#include "global.h"

global &globalInstance() {
    static global g;
    return g;
}

global::global(QObject *parent) :
    QObject(parent)
{
    m_appWidth = 1024;
    m_appHeight = 768;
}

void global::setAppWidth(int w) {
    if(m_appWidth != w) {
        m_appWidth = w;
        emit appWidthChanged();
    }
}

void global::setAppHeight(int h) {
    if(m_appHeight != h) {
        m_appHeight = h;
        emit appHeightChanged();
    }
}

```

The code is obvious I think. Now we need to pass the instance of this object as main context property of QML engine.

```cpp
#include <QApplication>
#include <QQmlApplicationEngine>
#include <QQmlContext>
#include "global.h"
#include "filter.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    filter *eventFilter = new filter;
    app.installEventFilter(eventFilter);

    QQmlApplicationEngine engine;
    engine.rootContext()->setContextProperty("globalThing", &globalInstance());
    engine.load(QUrl(QStringLiteral("qrc:/ui/main.qml")));
    QObject *rootObject = engine.rootObjects().first();

    QObject::connect(eventFilter, SIGNAL(mousePressed(QVariant,QVariant,QVariant)),
    QObject::connect(eventFilter, SIGNAL(mouseReleased(QVariant,QVariant,QVariant)),

    return app.exec();
}

```

From this moment QML engine know what exactly is globalThing that we used in main.qml but still doesn't know anything about appWidth and appHeight properties. Of course we defined them already in global.h/global.cpp but even with this declarations/definitions engine will not recognize these properties in qml. To inform him about them, we need to use Q_PROPERTY macro as below:

modification of global.h

```
#ifndef GLOBAL_H
#define GLOBAL_H

#include <QObject>

class global : public QObject
{
    Q_OBJECT
    Q_PROPERTY(int appWidth READ appWidth WRITE setAppWidth NOTIFY appWidthChanged)
    Q_PROPERTY(int appHeight READ appHeight WRITE setAppHeight NOTIFY appHeightChanged)

private:
    explicit global(QObject *parent = 0);
    explicit global(const global &g) {}

private:
    int m_appWidth;
    int m_appHeight;

```

Read more about Q_PROPERTY here:

http://qt-project.org/doc/qt-5/properties.html

Now everything should work. From c++ side we can do:

globalInstance().setAppWidth(1600);

And app window width will automatically change. Thanks to WRITE section of Q_PROPERTY macro we can do the same thing from QML side. For example

//some component.qml
Component.onCompleted: globalThing.appWidth = 1600; //still works thanks to setAppWidth method of "global" class.

Note that you can do the same thing to store home page url or to work with data models. All you need to do, is read about QVariant class (wen we are talking about data models), especially about QVariant::fromValue method + additionally about QAbstractListModel.

In the whole applications you can find a few test models introduced only in order to see whether the interface behaves correctly. For example in MessagesButton.qml:

```
        ListModel {
            id: testModel
            ListElement { name: "Jon Peters"; content: "Continuing our earlier discusion I have attached"; time: "10:06 am" }
            ListElement { name: "Jon Peters"; content: "Continuing our earlier discusion I have attached"; time: "10:06 am" }
            ListElement { name: "Jon Peters"; content: "Continuing our earlier discusion I have attached"; time: "10:06 am" }
            ListElement { name: "Jon Peters"; content: "Continuing our earlier discusion I have attached"; time: "10:06 am" }
            ListElement { name: "Jon Peters"; content: "Continuing our earlier discusion I have attached"; time: "10:06 am" }
        }
```

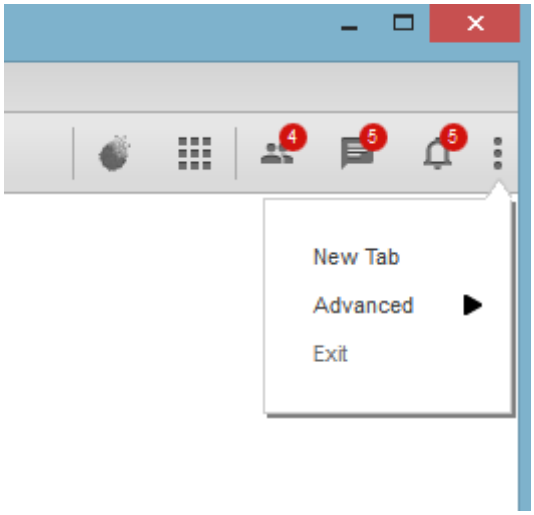
The best idea would be to create appropriate class that inherits from QAbstractListModel with the same role names as on the picture above, put the instance of it into global class, and provide necessary informations about this object thanks to Q_PROPERTY macro (remember about QVariant) The last thing you need is to modify "model" property of ListView Instance like below:

MessagesButton.qml

```
107         ListView {
108             id: listView
109             anchors.left: parent.left
110             anchors.right: parent.right
111             anchors.top: headText.bottom
112             anchors.bottom: footBar.top
113             anchors.leftMargin: 3
114             anchors.rightMargin: 3
115             spacing: 3
116             boundsBehavior: ListView.StopAtBounds
117             onContentYChanged: flickableScroll.flickableContentYChanged()
118             clip: true
119             model: globalThing.messagesModel
120
```

Our singleton doesn't solve (in an elegant way) the problem in which we need to react on menu/button clicked. It's possible to create appropriate slots/signals directly in "global" class but with this approach the whole c++ code will be placed in one class (it doesn't looks nice)

Let's say we would like to close application when we click on the last menu item:



All necessary signals are place in main.qml here:

```
36      Item {
37          id: rootItem
38          anchors.fill: parent
39
40          signal viewAllNotificationsClicked()
41          signal clearAllNotificationsClicked()
42          signal notificationItemAcceptClicked(int index, bool yes) //false means decline
43          signal notificationItemRemoveClicked(int index)
44
45          signal viewAllPeerRequestsClicked()
46          signal clearAllPeerRequestsClicked()
47          signal peerItemConfirmClicked(int index, bool yes) //false means "not now"
48
49          signal viewAllMessagesClicked()
50          signal clearAllMessagesClicked()
51          signal messageItemRemoveClicked(int index)
52
53          signal applicationsMoreButtonClicked()
54          signal roleMenuItemClicked(int index)
55          signal creationStackMenuItemClicked(int index)
56          signal pantheonMenuItemClicked(int index)
57          signal moreMenuItemClicked(int index)
58          signal moreSubMenuItemClicked(int index, bool checked) //you don't have to use checked
59          //arg if it's not necessary. Value of checked arg is not defined if menuitem has no
60          //visible checkbox
```

The only thing we need to to is to connect to them, so I think it would be nice to create another class called mainWindow

```
1   #ifndef MAINWINDOW_H
2   #define MAINWINDOW_H
3
4   #include <QObject>
5
6   class mainWindow : public QObject
7   {
8       Q_OBJECT
9   public:
10      explicit mainWindow(QObject *root);
11
12  private:
13      QObject *m_pRootItem;
14
15  private slots:
16      void moreMenuItemClicked(int index);
17  };
18
19  #endif // MAINWINDOW_H
20
```

```
1   #include "mainWindow.h"
2   #include <QCoreApplication>
3   #include <QDebug>
4
5   mainWindow::mainWindow(QObject *root) :
6       QObject(0)
7   {
8       m_pRootItem = root->findChild<QObject*>("magicItem");
9       if(!m_pRootItem)
10          qWarning()<<"I can't find necessary item!";
11
12      connect(m_pRootItem, SIGNAL(moreMenuItemClicked(int)),
13              this, SLOT(moreMenuItemClicked(int)));
14  }
15
16  void mainWindow::moreMenuItemClicked(int index) {
17      qDebug()<<"item clicked:"<<index;
18      if(index == 2) qApp->exit(0);
19  }
20
```

and use it like below (main.cpp)

```
13      QQmlApplicationEngine engine;
14      engine.load(QUrl(QStringLiteral("qrc:/ui/main.qml")));
15      QObject *rootObject = engine.rootObjects().first();
16
17      mainWindow mw(rootObject);
```

For the first time it won't work, because root->findChild method can't find object name called "magicItem" wee need to go into QML and set appropriate object name to the item that contains all signals/slots that we are looking for.

Main.qml

```
36    Item {
37        id: rootItem
38        objectName: "magicItem"
39        anchors.fill: parent
40
41        signal viewAllNotificationsClicked()
42        signal clearAllNotificationsClicked()
43        signal notificationItemAcceptClicked(int index, b
```

In the same way we can make a connection between c++ signal and QML slot. QML slot can looks like:

```
Item {
    id: rootItem
    objectName: "magicItem"
    anchors.fill: parent

    function magicSlot(arg) {
        console.log("yesss!!" + arg)
    }
```

Appropriate signal in c++ could looks like:

void magicSignal(const QVariant &arg);

Note that, in javascript all variables has no well-defined types. That's why we are forced to use QVariant agains (for example) QString. Thanks to the ability to create slots in QML, we don't need to create our own list models. You can always define appropriate slots which will work directly on ListModel elements placed in QML. More informations here:

http://qt-project.org/doc/qt-5/qml-qtqml-models-listmodel.html

Refer to the methods of this QML object.