QML Coding Conventions | Qt 5.7

doc.qt.io/qt-5/qml-codingconv…

No link preview available. Please open the link for details.

Open link

In addition:

1. Attached properties, such as Keys.onReleased or Component.onCompleted, should be placed after child objects and before states.
2. Use actual types in QML wherever possible – for example, Units.dp() should be assigned into a property of type 'real', not var.
3. We are following QML coding conventions in QML files, so braces on the same line as the element name ('Row {').
4. Properties in the root element don't need to be prefixed with the component name. We can use 'itemWidth' instead of 'appInfoBoxItemWidth'.
5. Fonts should be sized using '`font.pixelSize`' and not '`font.pointSize`', because we cannot rely on the DPI setting being correct for Atom hardware.
6. QML components should not reference properties inherited from their parent/ancestor components. Define a property at the top-level of the component and bind it to the right value at instantiation – this provides better code reuse and readability.

Software Test Friendly

1. Giving objects appropriate IDs when ever possible as the swt team try to identify most objects by this property.
2. If you have sets of similar image files, giving them a consistent naming standard will make testing easier e.g. (unknown.png, unknown_normal.png, unknown_dangerous.png)
3. In some cases a software tester will need to access back end/non UI/non Qitem objects in order to support test creation.
   Example of this would be the chart view service presenter:

```
        GOOD:
    property QtObject chartViewService: ChartViewService {
        chartAppPresenter: appPresenter
        property size mysize: Qt.size(app.width, app.height)
        ...
    }
        BAD:
        ChartViewService {
        id: chartViewService
        property size mysize: Qt.size(app.width, app.height)
        ...
    }
```

   If you don't do this we can not see it!

QML Repeaters

A common requirement is to use a QML Repeater to do things to a list or map of data. This is used show a table in a dialog, or to render a set of waypoints / AIS Targets. We define a QmlMapModel or QmlListModel of presentation sub-objects (e.g. AisDisplayData) in an owning C++ presenter (e.g. AisPresenter), and then use the repeater to process them all.

We've found the following guidelines helpful to maintain performance

1. Create a separate presentation type for the objects - we have to do Data Item formatting so often that trying to make the SystemFunction object presentation-friendly isn't possible, and dependency direction rules mean that the SystemFunction layer can't depend on the presentation layer.
2. Avoid signal handlers in the per-item code. Do the signal slot connections in C++ (either in the presentation object or the owning presenter) and then depend on the property notify signals to cause the QML to update. Then have straight bindings from the properties to the QML objects you need. Particularly avoid LatLong / MC coordinate to screen conversions in QML. Have screen coordinate properties and do the conversion in C++.
3. Make the map / list models constant. They emit their own changed signals, so after adding an element, you don't need your own map notify.
4. Sorting and / or filtering should be done with one of the QSortFilterProxyModel derivatives defined in QmlModelBase.h:
    a. SortableProxyModel does sorting by your feeding in a sort object property and a less than operator
    b. BoolPropertyFilterProxyModel does filtering by your feeding in a boolean filter property (i.e. a boolean property in the subtype - if its value is false the item is filtered out)
    c. SortableBoolPropertyFilterProxyModel does both by your feeling in a sort property, a less than operator working on it, and a boolean filter property.
5. However, you almost certainly want fine control over the scehduling of the sorts by turning off dynamic sorting on the Proxy Model, and calling sort(sortColumn()) to resort at the times you want to. See below.
6. In general, the presentation subobjects should have shared notify signals for multiple properties, mapped to how they're updated. AIS targets are updated by different static and dynamic data messages, so there should be a single notify signal for the static data, and a single notify signal for the dynamic data. It's also a good idea to do change detection to avoid unnecessary signal emission.

Sorting Tables

We have encountered a number of problems with adding sorting to tables. To overcome them:

Turn Dynamic Sorting Off

At the moment dynamic sorting is on by default (see `QmlListModel::createSortableProxyModel()` and `QmlMapModel::createSortableProxyModel()`). You should turn it off as soon as you have called `createSortableProxyModel()`. Once all existing code is fixed we should change the default to off.

The `QSortFilterProxyModel` documentation for dynamic sorting simply says that the *"proxy model is dynamically sorted and filtered whenever the contents of the source model change"*. We have found that with dynamic sorting enabled, a change to any (sortable) property in **any** cell causes a full re-sort of the list, which in turn re-renders every cell. In effect, dynamic sorting causes the entire table to created and rendered from scrach, when any property on any row changes.

With dynamic sorting off we have a problem when items are added or removed from the model. The table does not update. To overcome this we have added `SortableProxyModel::resort()`. This should be called whenever the model has changed substantially. Note that it can be expensive, causing the whole table to be sorted and re-rendered from scratch.

Changing Values in the Sorted-by Column

Turning off dynamic sorting seems to stop all automatic sorting, including when the sorted-by property of a row has changed. If you are altering the value that the table is sorted by you must ask the table to re-sort. Take care to do this after all rows have been changed, not after each value is changed. A typical example of this would be updating the range and bearing of all targets in a list.

To help with this we have added `SortableProxyModel::resortIfColumnChanged(QString changedPropertyName)`. This should be called with the name of the property you have just changed. If this happens to be the current sorted-by property, the table will be sorted, otherwise the request will be ignored.

Example

`WaypointsInGroupPresenter` has been written using these guidelines.