Модель / Представление

Язык QML, предоставляет механизм отделения данных от отображения.

Концепция Model/View

Самый простой способ отделить данные от представления это использовать ***Repeater***. ***Repeater*** использует модель (model), которая может быть любой.

```
import QtQuick

Column {
    spacing: 2
    Repeater {
        model: 10
        BlueBox {
            width: 120
            height: 32
            text: index
        }
    }
}
```

```
import QtQuick

Column {
    spacing: 2
    Repeater {
        model: 10
        delegate: BlueBox {
            width: 120
            height: 32
            text: index
        }
    }
}
```

```
import QtQuick

Column {
    spacing: 2
    Repeater {
        model: ["Alfa", "Betta", "Gamma", "Delta"]
        BlueBox {
            width: 120
            height: 32
            radius: 3
            text: modelData
        }
    }
}
```

```
import QtQuick

Column {
    spacing: 2
    Repeater {
        model: ListModel {
            ListElement { name: "Mercury"; surColor: "gray"}
            ListElement { name: "Venus"; surColor: "yellow"}
            ListElement { name: "Earth"; surColor: "blue"}
            . . .
        BlueBox {
            width: 120
            height: 32
            radius: 3
            text: name
            Box { anchors. Left: parent.left;
                  anchors.verticalCenter;
                  anchors.leftMargin: 4
                  width: 16; height: 16; radius: 8;
                  color: surColor
            }
        }
    }
}
```

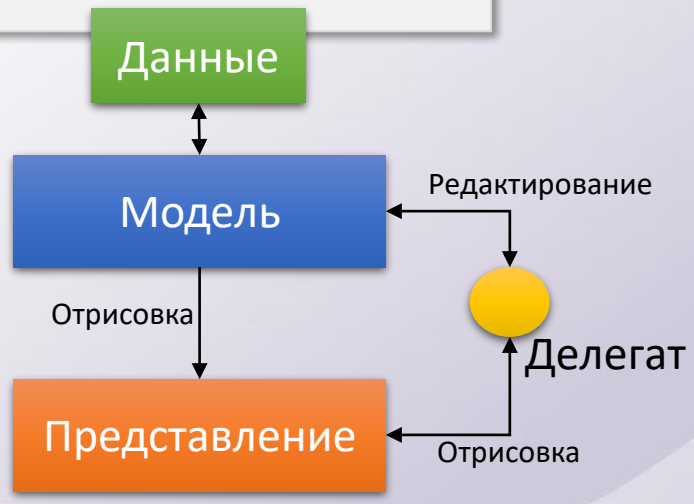
Модели

Модель - это элемент, который предоставляет интерфейс для обращения к данным, в отдельных случаях этот элемент может также содержать и сами данные, но это совсем необязательно. Типичными моделями *QML*, являются модели ***ListModel*** и ***XmlListModel, ObjectModel*** .

Элементы моделей не располагают информацией о том, как отображать их данные, элементы моделей отвечают за поставку данных. За отображение и редактирование данных отвечают **представления** и **делегаты**.

В *QML* в качестве модели может выступать:

- Целое число,
- Число вещественного типа,
- Массивы,
- данные в формате *JSON,*
- модель, которая реализована на C++



Модель списка (ListModel)

Модель списка представлена элементом ***ListModel*** и содержит последовательности элементов модели ***ListElement*** в виде:

```
ListModel {
    ListElement { ... }
    ListElement { ... }
     . . .
}
```

Элемент ***ListModel*** - это динамический список элементов. Элементы этого списка могут быть добавлены, вставлены, удалены и перемещены при помощи интегрированных в элемент ***ListModel*** методов:
append(), **insert ()**, **remove ()** и **move ()**.

Каждый элемент ***ListElement*** содержит одно или более свойств для данных. Элемент ***ListElement*** не содержит ни одного предопределенного свойства, а все они задаются пользователем.

Пример модели списка

*Модель реализована в отдельном файле «CDs.qml». Если разместить ее в одном и том же файле вместе с представлением, то для того чтобы иметь возможность к ней обратиться, необходимо при помощи свойства ***id*** снабдить ее идентификатором.

*Код для удаления текущего элемента может выглядеть так:

CDs.remove(view.currentindex)

«CDs.qml»

```
import QtQuick 2.15
    ListModel {
     ListElement {
          artist: "Arnaranthe"
          alburn: "Arnaranthe"
          year: 2011
          cover: "qrc:/covers/Arnaranthe.jpg“
     }
     ListElement {
          artist: "Dark Princess"
          alburn: "Without You"
          year: 2005
          cover: "qrc: /covers/WithoutYou. jpg“
     }
    ListElement {
        artist: "Within Ternptation"
        alburn: "The Unforgiving"
        year: 2011
        cover: "qrc: /covers/TheUnforgi ving. jpg“
    }
}
```

XML-модель

Элемент ***XmlListModel*** - это тоже модель списка и используется для *XML*-данных. Модель ***XmlListModel*** задействует для заполнения данными опросы XPath и присваивает данные свойствам.

CDs.qmt

```
import QtQuick 2.15
import QtQuick.XffilListModel 2.0
XmlListModel {
    source: "qrc: / / /CDs. xml"
    query: "/CDs/CD"
    XmlRole {name: "artist"; query: "artist/string()“}
    XmlRole {name: "album"; query: "album/string()“}
    XmlRole {name: "year"; query: "year/string()“}
    XmlRole {name: "cover"; query: "cover/string () “}
}
```

Данные могут быть из Интнрнета

модель, которая работает с данными из файла «CDs.xml»

CDs.xml

```
<?xml version = "1.0"?>
<CDs>
   <CD>
      <artist>Amaranthe</artist>
      <album>Amaranthe</albшn>
      <year>2011</year>
      <cover>qrc:/covers/Amaranthe.jpg</cover>
   </CD>
   <CD>
      <artist>Dark Princess</artist>
      <album>Without You</album>
      <year>2005</year>
      <cover>qrc:/covers/WithoutYou.jpg</cover>
   </CD>
   <CD>
      <artist>Within Temptation</artist>
      <album>The Unforgiving</albшn>
      <year>2011</year>
      <cover>qrc:/covers/TheUnforgiving.jpg</cover>
   </CD>
</CDs>
```

*Чтобы иметь возможность получать данные, нужен механизм ***XPath***. Он позволяет легко запросить нужную нам информацию. Этот механизм вы можете представить как механизм запросов к базе данных, которая представлена *XML*-файлом.

JSON-модель

В качестве данных модели могут выступать данные, полученные приложением от веб-сервисов. Благодаря тому, что в *QML* интегрирован *JavaScript*, данные в формате ***JSON*** могут использоваться напрямую, то есть вы можете считать ***JSON*** данные в переменную и использовать ее в качестве модели данных.

CDs.js

```
var jsonModel = [
  {
    artist: "Amaranthe",
    album: "Amaranthe",
    year: 2011,
    cover: "qrc: /covers/Amaranthe.jpg",
  },
  {
    artist: "Dark Princess",
    album: "Without You",
    year: 2005,
    cover: "qrc: /covers/WithoutYou.jpg",
  },
  {
    artist: "Within Temptation",
    album: "The Unforgiving",
    year: 2011,
    cover: "qrc: /covers/TheUnforgiving. jpg",
  }
  . . .
]
```

Представления

Повторители(***Repeater***) хорошо работают с ограниченными и статическими наборами данных, в реальном мире модели крупнее и сложнее для этого требуются более специализированные решения.

Для отображения данных моделей *QML* предоставляет три основных элемента:

- ***ListView*** - показывает классический список элементов, расположенных в горизонтальном или вертикальном порядке;
- ***GridView*** - отображает элементы в виде таблицы подобно тому, как это делается в обозревателе в режиме отображения значков;
- ***PathView*** - отображает элементы в виде замкнутой ленты.

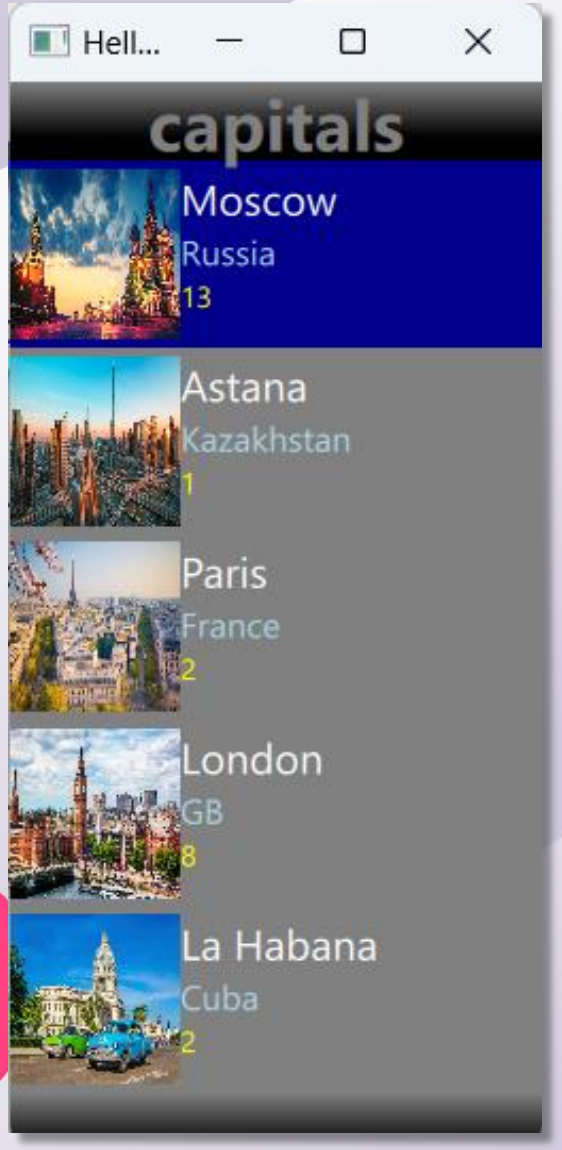
*Все эти элементы базируются на элементе ***Flickable,*** поэтому пользователь может перемещаться по большому набору данных. В то же время они ограничивают кол-во одновременно создаваемых экземпляров делегатов.

Представление для модели списка (ListView)

За отображение данных в виде столбца или строки отвечает элемент ***ListView.***

За отображение каждого элемента списка в отдельности всегда отвечает элемент ***делегата***. За основу для делегата берем элемент ***Component*** и присваиваем ему идентификатор ***delegate***, который имеет свойство ***id***.

Импортируем ***JS***-файл с данными *JSON* модели и указываем идентификатор для пространства имен CDs, чтобы получить доступ к переменной, которой эта модель присвоена. Далее задаем элемент верхнего уровня ***Rectangle***, присваиваем ему серый цвет (свойство *color*) и размеры 200x360.



Пример для ListView

```
import QtQuick 2.15
import "qrc:/CDs.js" as CDs
Rectangle {
    id: mainrect
    color: "gray"
    width: 200
    height: 360
    Component {
        . . .
    }
   ListView {
      . . .
    }
}
```

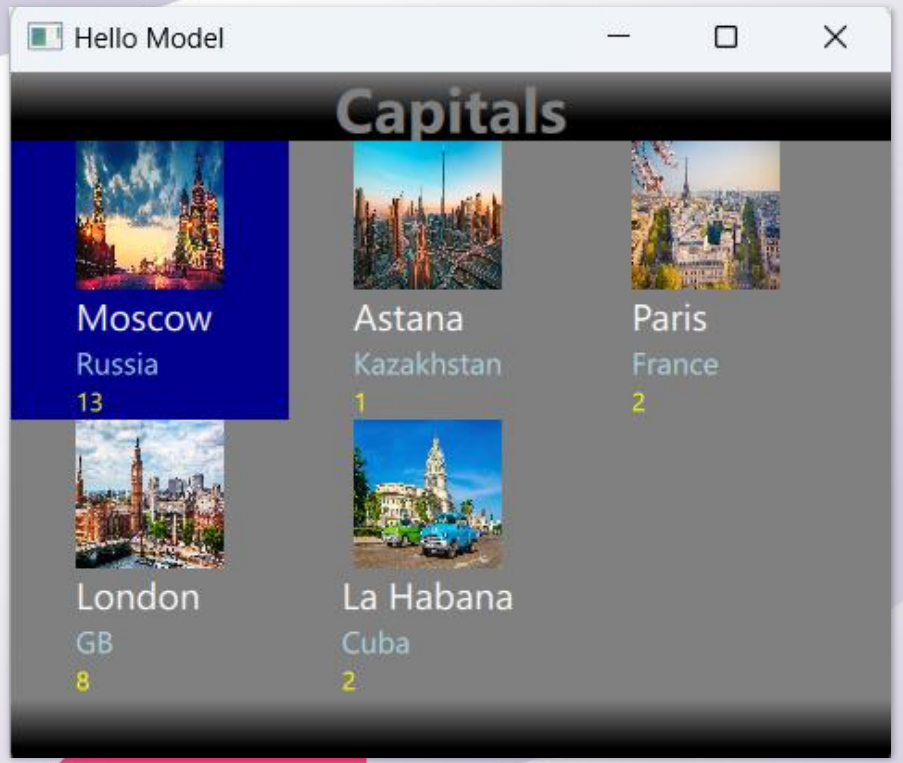
```
Component {
    id: delegate
    Item {
        width: mainrect.width ;  height: 70
        Row {
          anchors.verticalCenter: parent.verticalCenter
          Image {
              width: 64 ; height: 64
              source: modelData.cover
              smooth: true
           }
          Column {
              Text { color: "white"
                     text: modelData.artist ; font.pointSize: 12 }
              Text { color: "lightblue"
                     text: modelData.album; font.pointSize: 10 }
             Text { color: "yellow"
                    text: modelData.year; font.pointSize: 8 }
          }
       }
    }
}
```

```
ListView {
        focus: true
        header: Rectangle {
              width: parent.width ; height: 30
              gradient: Gradient {
                 GradientStop {position: 0; color: "gray“}
                 GradientStop {position: 0.7; color: “black“}
              Text{
                   anchors.centerin: parent;
                   color: "gray";  text: "CDs"
                   font.bold: true;
                   font.pointSize: 20
              }
       }
        footer: Rectangle {
                width: parent.width ; height: 30
                gradient: Gradient {
                   GradientStop {position: 0; color: "gray“}
                   GradientStop {position: 0.7; color: “black“}
                }
        }
        highlight: Rectangle {
                width: parent.width
                color: "darkblue"
        }
         anchors.fill: parent
         orientation: ListView.Horizontaldelegate: delegate
         model: CDs.jsonModel
         delegate: delegate
}
```

Табличное представление (GridView)

Элемент ***GridView*** автоматически заполняет всю область отображаемыми элементами в табличном порядке , поэтому нет необходимости устанавливать количество столбцов и строк.

*Использование элемента табличного размещения ***GridView*** практически идентично использованию элемента ***ListView***



Продолжение GridView

```
import QtQuick 2.15
Rectangle {
    id: mainrect
    color: "gray"
    width: 380
    height: 420
    Component {
      . . .
    }
    GridView {
    . . .
    }
}
```

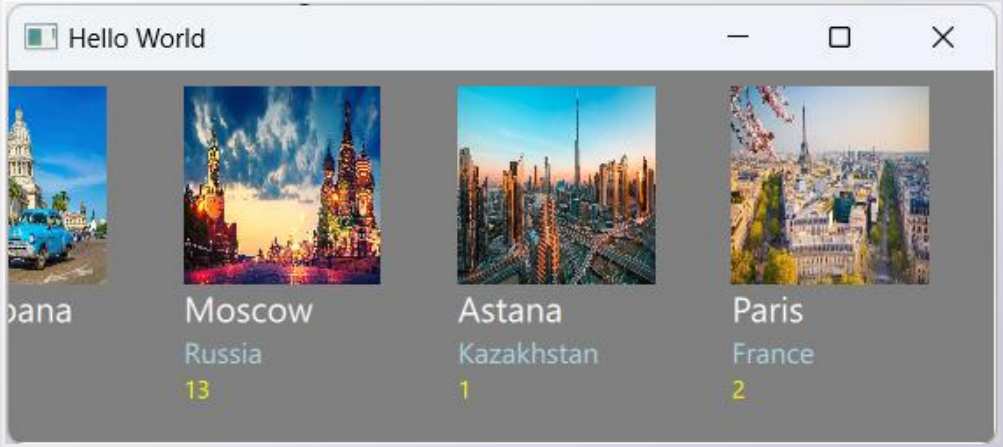
```
Component {
    id: delegate
    Item {
        width: 120; height: 120
        Column {
            anchors.centerIn: parent
            Image {
                anchors.horizontalCenter: parent.horizontalCenter
                width: 64; height: 64
                source: cover; smooth: true
            }
            Text { color: "white"; text: artist; font.pointSize: 12}
            Text { color: "lightblue"; text: album; font.pointSize: 10}
            Text { color: "yellow"; text: year; font.pointSize: 8}
        }
    }
}
```

```
GridView {
    cellHeight: 120 ; cellWidth: 120 ; focus: true
    header: Rectangle {
        width: parent.width ; height: 30
        gradient: Gradient {
            GradientStop {position: 0; color: "gray“}
            GradientStop {position: 0.7; color: "ьlack“}
        }
        Text {
            anchors.centerIn: parent;  color: "gray";  text: "CDs";
            font.bold: true;
            font.pointSize: 20
        }
    }
    footer: Rectangle {
        width: parent.width; height: 30
        gradient: Gradient {
            GradientStop {position: 0; color: "gray“}
            GradientStop {position: 0.7; color: “black“}
        }
    }
    highlight: Rectangle {
        width: parent.width; color: "darkblue"
        anchors.fill: parent
        model: CDs { }
        delegate: delegate
    }
}
```

Представление в виде замкнутой линии (PathView)

Элемент ***PathView*** показывает элементы в виде замкнутой линии. Т.е, пользователь может до бесконечности прокручивать элементы в определенную сторону, и они будут просто повторяться.

Делегат идентичен делегатам предыдущих примеров. Ключевой момент заключается в создании элемента ***Path***. Этот элемент задает форму замкнутой линии. В нашем случае мы определяем горизонтальную линию от 0 до 500 с одинаковым удалением сверху 80. Если бы, например, в элементе *Path* мы присвоили бы свойству *startY* значение 0, то все элементы пошли бы по наклонной линии. Созданный элемент *Path* устанавливается в представлении ***PathView*** при помощи свойства **path**. В завершение мы устанавливаем свойством ***pathItemCount*** количество элементов, которые должны быть одновременно видимы.



Пример использования PathView

```
import QtQuick 2.15
Rectangle {
    color: "gray"
    width: 450
    height: 170
    Component {
      . . .
    }
    Path {
      . . .
    }
    PathView {
     . . .
    }
}
```

```
Component {
    id: delegate
    Item {
      width: item.width
      height: item.height
      Column {
           id: item
           Image {
              width: 90; height: 90
              source: cover
              smooth: true
           }
          Text {color: "white"; text: artist; font.pointSize: 12}
          Text {color: "lightblue"; text: album; font.pointSize: 10}
          Text {color: "yellow"; text: year; font.pointSize: 8}
      }
    }
}
```

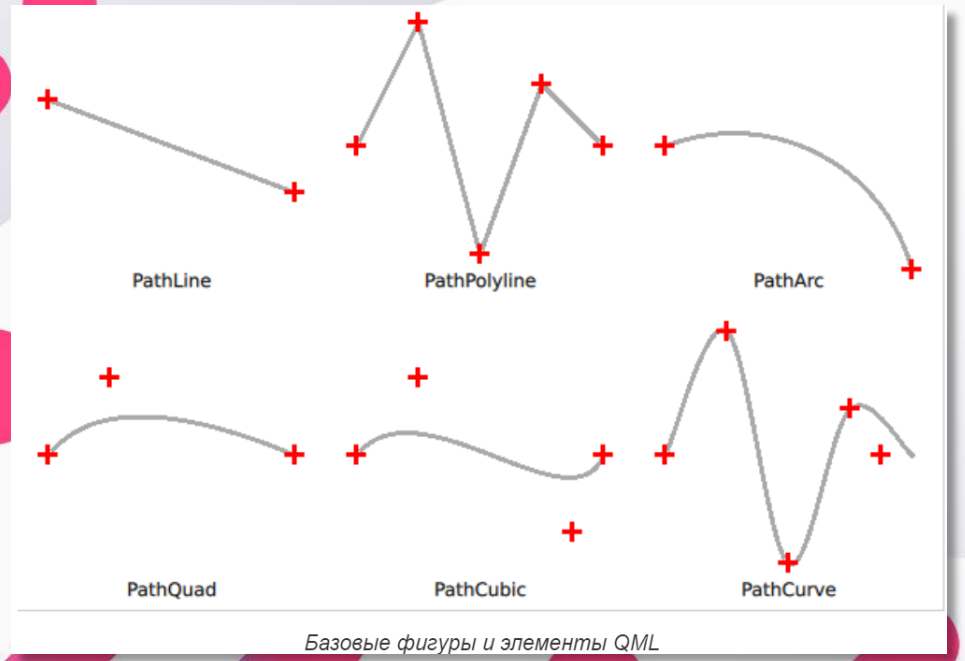
```
Path {
    id: itemsPath
    startX: 0
    startY: 80
    PathLine {x: 500; y: 80}
}
 ////////////////

PathView {
    id: itemsView
    anchors.fill: parent
    model: CDs { }
    delegate: delegate
    path: itemsPath
    pathItemCount: 4
}
```

3D-карусель

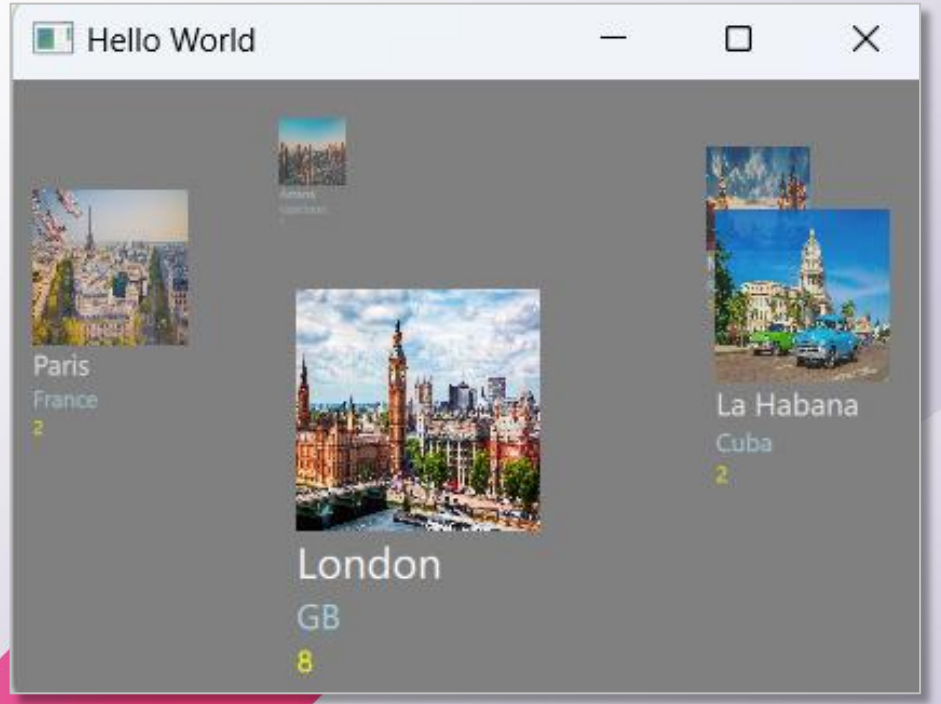
В элементе ***Path*** добавился элемент ***PathQuad***, который задает дугу. Для создания окружности нам потребуется два таких элемента: первый - для правой половины окружности, а второй - для левой. В зависимости от расположения элементов, отображаемых на дуге, мы при помощи элементов ***PathAttribute*** выполняем изменение их прозрачности и размера. Эти действия и создают иллюзию ***3D***.

main.qml

```
Path {
    id: itemsPath
    startX: 150
    startY: 150
    PathAttribute {name: "iconScale"; value: 1.0}
    PathAttribute {name: "iconOpacity"; value: 1.0}
    PathQuad {x: 150; y: 25; controlX: 460; controlY: 75}
    PathAttribute { name: "iconScale"; value: 0. 3}
    PathAttribute {name: "iconOpacity"; value: 0.3}
    PathQuad {x: 150; y: 150; controlX: -80; controlY: 75}
}
```

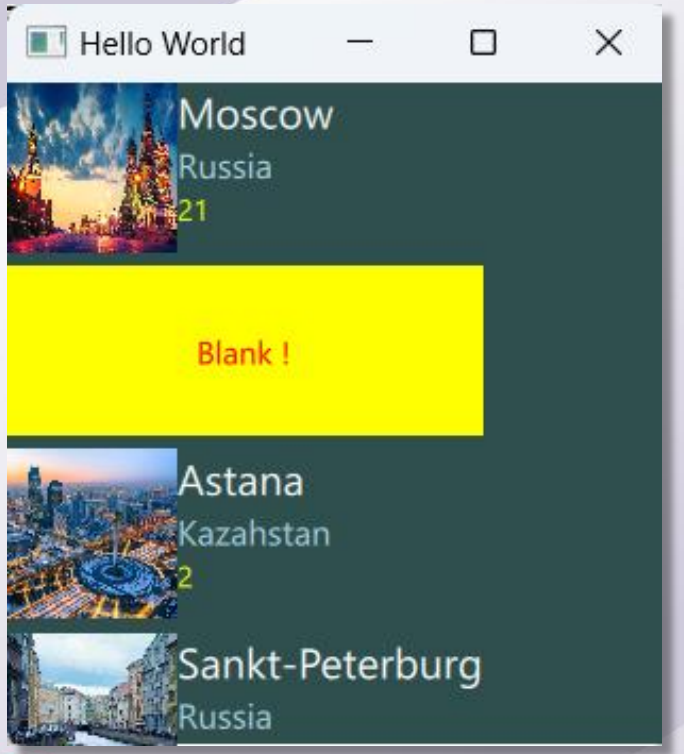


Базовые фигуры и элементы QML



Визуальная модель

Рассмотрим особый вид модели, который сам отвечает за представление своих данных и не нуждается в делегате. Этот вид модели может очень пригодиться, в тех случаях, когда каждый из элементов нужно отображать в индивидуальной манере.



Каждый элемент этой модели содержит полную реализацию своего представления, включая размещения составных элементов, а также размеры их шрифтов и цвета. Для всех элементов нашего примера мы используем одинаковые цвета и размеры шрифтов. На самом деле это совсем не обязательно, потому что визуальная модель позволяет каждый из элементов представить полностью по-своему.

Пример

CDs.qml

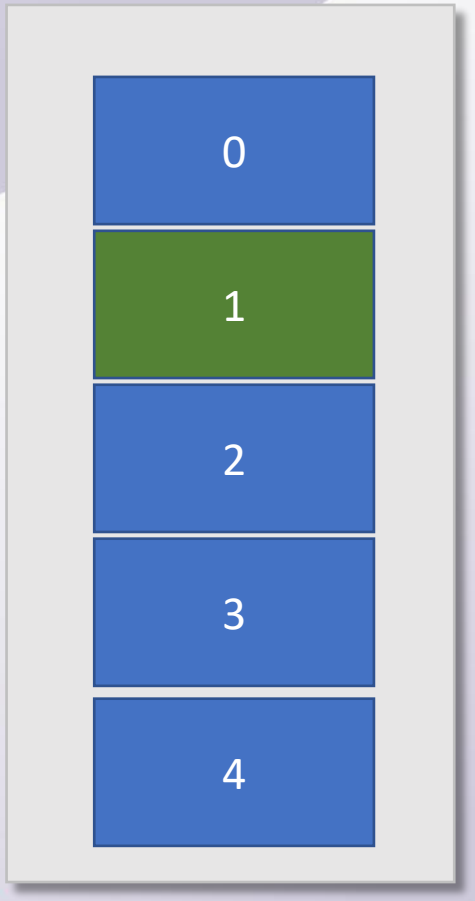
```
import QtQuick 2.15
ObjectModel {
   Row {
      Image {
          width: 64; height: 64 ; source: "qrc:/covers/fallen.jpg" ; smooth: true
      }
      Column {
        Text {color: "white"; text: "Evanescence"; font.pointSize: 12}
        Text {color: "lightblue"; text: "fallen"; font.pointSize: 10}
        Text {color: "yellow"; text: "2003"; font.pointSize: 8}
      }
  }
   Rectangle {
        width: parent.width ; height: 64; color: "Yellow"
        Text {anchors.centerln: parent ; color: "Red“; text: "Blank !" }
  }
  Row {
        Image  {  width: 64 ;  height: 64 ; source: "qrc: /covers/Rubicon. jpg"
                        smooth: true  }
        Column {
                Text { color: "whi te"; text: "Tristania"; font. pointSize: 12}
                Text {color: "lightblue"; text: "Rubicon"; font.pointSize: 10}
                Text {color: "yellow"; text: "2010"; font.pointSize: 8}
        }
   }
}
```

maln.qml

```
import QtQuick 2.8
   Rectangle {
      width: 250; height: 250;
      color: "DarkSlateGray"
      Flickable {
        id: view
        width: 250; height: 500
        contentWidth: 250
        contentHeight: column.height
        anchors.fill: parent
        Column {
            id: column
            anchors.fill: view
            spacing: 5
            Repeater {
              model: CDs{ }
            }
        }
    }
}
```

Домашка #13

1. Создать JSON-модель и XML-модель в отдельном файле.
2. Разработка делегата в соответствии с идеей приложения
3. Навигация по модели с помощью клавиатуры.
4. Выделение текущего элемента модели.
5. Анимация добавления и удаления элементов.



Может пригодиться

- https://qmlbook.github.io/ch07-modelview/modelview.html
- https://stuff.mit.edu/afs/athena/software/texmaker_v5.0.2/qt57/doc/qtquick/qtquick-modelviewsdata-modelview.html