**Essential Steps for C++ Model Integration with QML**

**1. Create Your C++ Model Class (.h and .cpp files)**

* **Header (.h):**
  + **Inherit from QAbstractListModel:** This is the foundation.
  + **Include QObject:** (Often done automatically if inheriting QAbstractListModel, but good to remember).
  + **Q\_OBJECT Macro:** **CRITICAL!** Place Q\_OBJECT at the top of your class definition. This enables Qt's meta-object system (signals, slots, properties, QML integration).
  + **Define enum Roles (for custom data):**
    - Start your custom roles from Qt::UserRole + 1 (e.g., enum Roles { MyCustomRole = Qt::UserRole + 1 };).
    - **Q\_ENUM(Roles) Macro:** **CRITICAL!** Place Q\_ENUM(YourEnumName) *after* your enum definition to expose your enum values by name to QML.
  + **Override Essential Functions:** Declare rowCount(), data(), and roleNames() as override. Add explicit to the constructor if it's not a copy constructor.
* **Source (.cpp):**
  + **Include your model's header:** "#include "yourmodelname.h".
  + **Implement Constructor:** Initialize your internal data structure (e.g., QStringList).
  + **Implement rowCount():** Return the count of items in your internal data.
  + **Implement data():**
    - Check index.isValid() and index.row() bounds.
    - Use a switch statement on the role parameter.
    - Return appropriate QVariant data for Qt::DisplayRole and any custom roles you defined.
    - Return QVariant() for unhandled roles.
  + **Implement roleNames():**
    - Create a QHash<int, QByteArray>.
    - Map your custom enum roles (e.g., MyCustomRole) to string names (e.g., "myCustomData").
    - **Do NOT** include Qt::DisplayRole or other default Qt roles here.

**2. Register Your C++ Model in main.cpp**

* **Include Headers:**
  + #include <QQmlApplicationEngine>
  + #include <QQmlContext> **(CRITICAL for setContextProperty)**
  + #include "yourmodelname.h" (e.g., "stringlistmodel.h")
* **Create Model Instance:** Inside main(), create an instance of your C++ model class: YourModelName myModel;
* **Expose to QML:** Use engine.rootContext()->setContextProperty("qmlModelName", &myModel);
  + "qmlModelName": This is the string name you will use to refer to your model in QML. Choose a meaningful name.
  + &myModel: Pass a pointer to your model instance.

**3. Call (Use) Your Model in QML (.qml file)**

* **Import Necessary Modules:** import QtQuick, import QtQuick.Controls, import QtQuick.Layouts.
* **ListView (or GridView, Repeater):** Use a view component to display the list.
* **Connect model property:** Set the model property of your view to the name you registered in main.cpp:
  + model: qmlModelName
* **Define delegate:** Create a delegate component that defines the visual appearance of each item.
* **Access Data in delegate:**
  + Use the string names you defined in roleNames() to access your custom data (e.g., Text { text: myCustomData }).
  + For Qt::DisplayRole (if you handle it), you can often use text: model.display.
* **Important: Delegate Context:** Inside the delegate, index and the role names (like itemName) are directly available in the QML scope for the current item.

**Important Reminders & Common Pitfalls:**

* **Q\_OBJECT is not a suggestion, it's a requirement:** If you inherit from QObject (or a QObject derivative) and use meta-object features, you MUST have Q\_OBJECT. Forget it, and you'll get compilation errors.
* **Q\_ENUM for QML access:** If you want to use your C++ enum members by name in QML, use Q\_ENUM.
* **roleNames() for QML String Names:** This is how QML knows itemName in QML maps to ItemNameRole in C++.
* **Headers:** Always include the necessary headers (<QAbstractListModel>, <QQmlContext>, etc.). Qt Creator usually helps, but be aware.
* **QVariant for data():** The data() function *must* return QVariant. Qt handles the conversion for basic types like QString, bool, int.
* **Memory Management:** For models that store QObject\* items (like your initial TaskItem example), remember to delete them in the model's destructor (qDeleteAll(m\_tasks);). For simple types like QString, this isn't an issue.
* **Signals for Changes:** For editable models or models where items are added/removed, remember to emit appropriate signals (dataChanged, beginInsertRows, endInsertRows, beginRemoveRows, endRemoveRows) to notify QML views to update. (Our simplified demo doesn't do this, but the full TaskListModel does).