Development/Tutorials/D-Bus/CustomTypes

Using Custom Types with DBus

Tutorial Series	D-Bus
Previous	Creating Interfaces, Intermediate DBus
What's Next	n/a
Further Reading	n/a

DBus-based IPC has been intregrated nicely in the Qt framework. Qt also provides Adaptors and Interfaces to communicate with remote objects, so you don't have to manually construct QDBusMessages. These Adaptors and Interfaces are very similar to the Stubs and Proxies used in other RPC mechanisms like Corba and RMI.

So, Qt provides a way to talk to objects in a different process and it contains Adaptors and Interfaces to do so in a nice, object-oriented way. A number of types can be marshaled and unmarshaled by default, (QString, QPoint, QRect, etc.), so you can just use those in DBus-enabled classes without extra effort.

However, using a custom type is somewhat more complicated, as you need to tell Qt how to handle it. That is what this tutorial is about.

An example was created to go along with this tutorial. It is a very basic chat application that communicates through the session DBus.

目录

Write a class

Write the class you would like to publish, complete with signals, slots and properties, using custom types as you go along.

The class being published in the example is the following:

```
class Chat : public QObject
{
  Q_OBJECT

  Q_CLASSINFO("D-Bus Interface", "demo.Chat")
  Q_PROPERTY( QStringList users READ users)

signals:
  void userAdded(const QString& user);
  void userRemoved(const QString& user);

  void messageSent(const Message &message);

public slots:
  void addUser(const QString &user);
  void removeUser(const QString &user);

  void sendMessage(const Message &message);

public:
  Chat(QObject* parent = 0);
  virtual ~Chat();

  QStringList users() const;
private:
   QStringList m_users;
};
```

It contains simple user management and provides the means to send a message. The Message class is the custom type in the example. It contains only a user and a text message. There is no reason why the methods in the Chat class couldn't just take 2 QString parameters, but then Qt would be able to do everything automatically and we need something irregular for this tutorial.

To show that Qt supports a lot of types right out of the box, a QStringList was used. This will become clear further on.

Q_CLASSINFO

The Q_CLASSINFO declaration provides a means to specify an interface name that will be take into account by the xml tools.

Typically, the company name is included in the name, resulting in declarations like:

```
Q_CLASSINFO("D-Bus Interface", "com.firm.department.product.interface")
```

Generate XML

Generate the xml description of the class using the qdbuscpp2xml tool provided by Qt.

If you run **qdbuscpp2xml** on the Chat.hpp file in the example, you will get the following output:

```
PE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN" "http://www.freedesktop.org/sta
<node>
  <interface name="demo.Chat">
    <property name="users" type="as" access="read"/>
    <signal name="userAdded">
      <arg name="user" type="s" direction="out"/>
    </signal>
    <signal name="userRemoved">
      <arg name="user" type="s" direction="out"/>
    </signal>
    <method name="addUser">
      <arg name="user" type="s" direction="in"/>
    </method>
    <method name="removeUser">
      <arg name="user" type="s" direction="in"/>
    </method>
  </interface>
</node>
```

Note that this xml file contains all methods from the Chat class using standard Qt types. If you were to generate Adaptor and Interface classes based on this xml, you would be able to add and remove users, get the list of users and receive the userAdded and userRemoved signals. Even the QStringList type is handled automatically.

The methods dealing with the Message type, however, are not available. To generate an Adaptor and an Interface capable of dealing with the Message type, you need to modify the XML.

> **Tip**
> You may want to generate the Adaptor and Interface using only the automatically generated xml. It might be helpful to compare them to the versions we will generate further on.

Edit the XML

The **qdbusxml2cpp** tool needs to be told about the custom types, so you need to add some type information to the XML.

This is done by specifying annotations:

```
<annotation name="org.qtproject.QtDBus.QtTypeName"
value="*customType*"/>
```

The syntax differs slightly depending on wheter you're using it in a signal/method or in a property (the {{{.In0}}} is omitted for properties), but it is fairly straightforward. The following is an example dealing with the QRect type (it has no relation with the example):

```
<property name="rectangle" type="(iiii)" access="readwrite">
 <annotation name="org.qtproject.QtDBus.QtTypeName" value="QRect"/>
</property>

<signal name="rectangleChanged">
 <arg name="rect" type="(iiii)" direction="out"/>
 <annotation name="org.qtproject.QtDBus.QtTypeName.In0" value="QRect"/>
</signal>

<method name="changeRectangle">
 <arg name="message" type="(iiii)" direction="in"/>
 <annotation name="org.qtproject.QtDBus.QtTypeName.In0" value="QRect"/>
</method>
```

Don't worry too much about what type to specify; any type that is somewhat too complex to get marshaled/unmarshaled by default handlers will be processed using the custom type, so it really doesn't matter what type you use. You could even use {{{"(iiii)"}}} for everything.

If any of your methods returns a complex result, you need to add an annotation for org.qtproject.QtDBus.QtTypeName.**Out**0 as well.

> **Note**
> The XML above uses QRects. QRect is supported by default, so the code above would be generated for you by **qdbuscpp2xml**.

The complete interface used for the Chat interface in the example is as follows:

```
<!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN" "http://www.freedeskto
<node>
```

```
    <interface name="demo.Chat">
     <property name="users" type="as" access="read"/>
     <signal name="userAdded">
      <arg name="user" type="s" direction="out"/>
     </signal>
     <signal name="userRemoved">
      <arg name="user" type="s" direction="out"/>
     </signal>
     <signal name="messageSent">
      <arg name="message" type="a(ii)" direction="out"/>
      <annotation name="org.qtproject.QtDBus.QtTypeName.In0" value="Message"/>
     </signal>
     <method name="addUser">
      <arg name="user" type="s" direction="in"/>
     </method>
     <method name="removeUser">
      <arg name="user" type="s" direction="in"/>
     </method>
     <method name="sendMessage">
      <arg name="message" type="a(ii)" direction="in"/>
      <annotation name="org.qtproject.QtDBus.QtTypeName.In0" value="Message"/>
     </method>
    </interface>
   </node>
```

Generate Adaptor and Interface classes

Now that the XML contains all the information **qdbusxml2cpp** needs, it's time to generate the Adaptor and Interface classes.

Since custom types are involved, you will need to specify some extra includes, so the Adaptor and Interface classes can find all the type information.

For the DBusChat example, the Adaptor and Interface classes were created by calling

```
qdbusxml2cpp Chat.xml -i Message.hpp -a ChatAdaptor
qdbusxml2cpp Chat.xml -i Message.hpp -p ChatInterface
```

Qt Meta object magic

Even though we now have an Adaptor to wrap an object and an Interface to talk to it, you will not be able to compile them as some extra things are necessary for the Qt Meta object system to be able to process the custom type.

Register the type

Declare the type as a Qt meta type by adding a

```
 Q_DECLARE_METATYPE
```

statement to the header file containing

your custom type definition.

Add qRegisterMetaType and qDBusRegisterMetaType calls to enable the framework to work with the custom type.

In the example's Message class, a static method is included to do this:

```
void Message::registerMetaType()
{
  qRegisterMetaType<Message>("Message");

  qDBusRegisterMetaType<Message>();
}
```

Important
You need to register the type both in the application publishing the object and in the application using it, since both applications need to be able to handle the custom type.

Also take care to register the custom types before calling methods on the Adaptor/Interface that need them.

Qt will show error output if you are using types it cannot (yet) handle, but you should be aware of it nonetheless.

Provide QDBusArgument streaming operators

When a DBus call is executed that uses a custom type, the Qt framework will need to marshal (*serialize*) or unmarshal (*unserialize*) the instance.

This is done by using the QDBusArgument stream operators, so you will need to implement those operators for your custom types, as explained here.

For the Message type from the example, these operators are very simple, since it only contains 2 strings:

```
QDBusArgument &operator<<(QDBusArgument &argument, const Message& message)
{
  argument.beginStructure();
  argument << message.m_user;
  argument << message.m_text;
  argument.endStructure();

  return argument;
}

const QDBusArgument &operator>>(const QDBusArgument &argument, Message &message)
{
  argument.beginStructure();
  argument >> message.m_user;
  argument >> message.m_text;
  argument.endStructure();

  return argument;
}
```

QDBusArgument provides functions to handle much more complex types as well.

Use the adaptor and interface classes

You are now ready to start using the Adaptor and Interface classes and have your custom type handled automatically.

Publishing an object using an Adaptor

To publish an object, you should instantiate an Adaptor for it and then register the object with the DBus you are using.

This is the code that publishes a Chat object in the DBusChat example:

```
Chat* pChat = new Chat(&a);
ChatAdaptor* pChatAdaptor = new ChatAdaptor(pChat);

if (!connection.registerService(CHAT_SERVICE))
{
  qFatal("Could not register service!");
}

if (!connection.registerObject(CHAT_PATH, pChat))
{
  qFatal("Could not register Chat object!");
}
```

The ChatAdaptor instance will process any incoming DBus requests for the Chat object. When a method is invoked, it will call the matching slot on the Chat object. When the Chat object emits a signal, the ChatAdaptor will transmit it across DBus.

Talking to a remote object using an Interface

To talk to a remote object, you need only instantiate the matching Interface and pass the correct service name and object path.

In the DBusChat example, a connection is made with a remote Chat object like this:

```
demo::Chat chatInterface(CHAT_SERVICE, CHAT_PATH, connection);
```

Using the remote object in your application

Once you have an instance of the Interface class, you should be able to interact with the remote object like you would with any other QObject.

The ChatWindow class in the DBusChat example, for instance, adds a user simply by calling

```
m_chatInterface.addUser(m_userName);
```

Furthermore, ChatWindow is able to respond to signals emitted by the Chat object by connecting to its Interface class just like with any other QObject:

```
connect(&m_chatInterface, SIGNAL(userAdded(QString)), SLOT(chat_userAdded(QString)));
connect(&m_chatInterface, SIGNAL(userRemoved(QString)), SLOT(chat_userRemoved(QString)));
connect(&m_chatInterface, SIGNAL(messageSent(Message)), SLOT(chat_messageSent(Message)));
```

Varia

Automation

We're not too wild about having to do some of this stuff manually, but using the Adaptors and Interfaces is much more convenient than writing code that manually constructs dbus calls and processes the replies.

One might hope **qdbuscpp2xml** and **qdbusxml2cpp** will be extended to support custom types by analyzing an entire code tree instead of just the header files passed to them, so they would be able to recognize the custom types and add methods/signals/properties to the XML accordingly. One potential strategy for extending **qdbuscpp2xml** is with plugins, and this is presented at Cpp2XmlPlugins.

Alternatively, a feature could be added to Qt Creator to support the kind of solution presented here in a automated fashion. Since Qt Creator already needs to be aware of custom types used in a project, this might be more feasible (or at least easier) than modifying the qdbus tools.

Adventurous serialization of enumerations

If you happen to use a lot of enumerations and you need them to be exported across DBus, you will likely end up with QDBusArgument stream operators for every enumeration. A basic implementation could simply cast the enum to and from an int, resulting in code looking like this:

```
QDBusArgument &operator<<(QDBusArgument &argument, const EnumerationType& source)
{
   argument.beginStructure();
   argument << static_cast<int>(source);
   argument.endStructure();
   return argument;
}

const QDBusArgument &operator>>(const QDBusArgument &argument, EnumerationType &dest)
{
   int a;
   argument.beginStructure();
   argument >> a;
   argument.endStructure();

   dest = static_cast<EnumerationType>(a);

   return argument;
}
```

You will notice that this code will be much the same for all enumerations. The only part that will differ is the "EnumerationType".

Thankfully, C++ knows how to handle templates, so we should be able to write just the one template-based implementation. However, we need that one implementation to handle **only** enumeration types, not the other custom types we want to send across DBus. Otherwise, any custom type would be cast from and to an integer value, which is probably not what you want for all types, especially not for complex ones.

This is where the boost library comes to our aid. With some magic, it supports conditionals within template definitions.

That way, we can provide QDBusArgument marshaling and unmarshalling implementations that will be used only in situations when such a conditional is true.

This is what the marshaling and unmarshaling code actually looks like:

```
template<typename T, typename TEnum>
class QDBusEnumMarshal;

template<typename T>
class QDBusEnumMarshal<T, boost::true_type>
{
public:
    static QDBusArgument& marshal(QDBusArgument &argument, const T& source)
    {
        argument.beginStructure();
        argument << static_cast<int>(source);
        argument.endStructure();
        return argument;
    }

    static const QDBusArgument& unmarshal(const QDBusArgument &argument, T &source)
    {
        int a;
        argument.beginStructure();
        argument >> a;
        argument.endStructure();

        source = static_cast<T>(a);

        return argument;
    }
}
```

The marshal implementation can now be called by invoking:

```
QDBusEnumMarshal<T, typename boost::is_enum<T>::type>::marshal(argument, source);
```

with T being the type we want to marshal.

Some further glue code is needed to link the implementation to the QDbusArgument stream operators:

```
template<typename T>
```

```
QDBusArgument& operator<<(QDBusArgument &argument, const T& source)
{
   return QDBusEnumMarshal<T, typename boost::is_enum<T>::type>::marshal(argument, source);
}

template<typename T>
const QDBusArgument& operator>>(const QDBusArgument &argument, T &source)
{
   return QDBusEnumMarshal<T, typename boost::is_enum<T>::type>::unmarshal(argument, source);
}
```

Put all that in a header file and include it where you declare the enumerations that need to pass across DBus. The compiler is now able to find the streaming operators all on its own.

You do still need to declare the enumeration as a metatype and register it with the Qt meta object system though.

> **Note**
> You only need a boost header file to do this (**boost/type_traits/is_enum.hpp**, to be exact). There is no need to have the entire library installed, which is rather nice on an embedded platform, since the boost library takes up quite a bit of space.

enumDBus.hpp

This is the entire header file needed to marshal / unmarshal any enumeration:

```
#ifndef _ENUM_DBUS_HPP
#define _ENUM_DBUS_HPP

#include <QtDBus/QDBusArgument>

#include <boost/type_traits/is_enum.hpp>

using namespace std;

template<typename T, typename TEnum>
class QDBusEnumMarshal;

template<typename T>
class QDBusEnumMarshal<T, boost::true_type>
{
public:
    static QDBusArgument& marshal(QDBusArgument &argument, const T& source)
    {
        argument.beginStructure();
        argument << static_cast<int>(source);
        argument.endStructure();
        return argument;
    }

    static const QDBusArgument& unmarshal(const QDBusArgument &argument, T &source)
    {
        int a;
        argument.beginStructure();
        argument >> a;
        argument.endStructure();

        source = static_cast<T>(a);
        return argument;
    }
};

template<typename T>
QDBusArgument& operator<<(QDBusArgument &argument, const T& source)
{
    return QDBusEnumMarshal<T, typename boost::is_enum<T>::type>::marshal(argument, source);
}

template<typename T>
const QDBusArgument& operator>>(const QDBusArgument &argument, T &source)
{
    return QDBusEnumMarshal<T, typename boost::is_enum<T>::type>::unmarshal(argument, source);
}

#endif //_ENUM_DBUS_HPP
```

Example

This section contains the sources for the chat example. The wiki does not allow the upload of files other than images, so they've been posted as text.

Chat is the interface being published. ChatAdaptor and ChatInterface were generated using Chat.xml.

Put all these files in a directory and run

```
qmake

make
```

Chat.hpp

```
#ifndef CHAT_HPP
#define CHAT_HPP

#include <QObject>
#include <QStringList>

class Message;

class Chat : public QObject
{
  Q_OBJECT

  Q_CLASSINFO("D-Bus Interface", "demo.Chat")
  Q_PROPERTY( QStringList users READ users)

signals:
  void userAdded(const QString& user);
  void userRemoved(const QString& user);

  void messageSent(const Message &message);

public slots:
  void addUser(const QString &user);
  void removeUser(const QString &user);
```

```
    void sendMessage(const Message &message);

public:
    Chat(QObject* parent = 0);
    virtual ~Chat();

    QStringList users() const;

private:
    QStringList m_users;
};
#endif // CHAT_HPP
```

Chat.cpp

```
#include "Chat.hpp"

Chat::Chat(QObject* parent) :
        QObject(parent)
{
}

Chat::~Chat()
{
}

void Chat::addUser(const QString &user)
{
    if (!m_users.contains(user))
    {
        m_users.append(user);

        emit userAdded(user);
    }
}

void Chat::removeUser(const QString &user)
{
    if (m_users.contains(user))
    {
        m_users.removeOne(user);

        emit userRemoved(user);
    }
}

void Chat::sendMessage(const Message &message)
{
    Q_EMIT messageSent(message);
}

QStringList Chat::users() const
{
    return m_users;
}
```

Chat.xml

```
<!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN" "http://www.freedesktop.or
<node>
```

```
 <interface name="demo.Chat">
   <property name="users" type="as" access="read"/>
   <signal name="userAdded">
     <arg name="user" type="s" direction="out"/>
   </signal>
   <signal name="userRemoved">
     <arg name="user" type="s" direction="out"/>
   </signal>
   <signal name="messageSent">
     <arg name="message" type="a(ii)" direction="out"/>
     <annotation name="com.trolltech.QtDBus.QtTypeName.In0" value="Message"/>
   </signal>
   <method name="addUser">
     <arg name="user" type="s" direction="in"/>
   </method>
   <method name="removeUser">
     <arg name="user" type="s" direction="in"/>
   </method>
   <method name="sendMessage">
     <arg name="message" type="a(ii)" direction="in"/>
     <annotation name="com.trolltech.QtDBus.QtTypeName.In0" value="Message"/>
   </method>
 </interface>
</node>
```

ChatAdaptor.h

```
/*
 * This file was generated by qdbusxml2cpp version 0.7
 * Command line was: qdbusxml2cpp Chat.xml -i Message.hpp -a ChatAdaptor
 *
 * qdbusxml2cpp is Copyright (C) 2010 Nokia Corporation and/or its subsidiary(-ies).
 *
 * This is an auto-generated file.
 * This file may have been hand-edited. Look for HAND-EDIT comments
 * before re-generating it.
 */

#ifndef CHATADAPTOR_H_1270658274
#define CHATADAPTOR_H_1270658274

#include <QtCore/QObject>
#include <QtDBus/QtDBus>
#include "Message.hpp"
class QByteArray;
template<class T> class QList;
template<class Key, class Value> class QMap;
class QString;
class QStringList;
class QVariant;

/*
 * Adaptor class for interface demo.Chat
 */
class ChatAdaptor: public QDBusAbstractAdaptor
{
    Q_OBJECT
    Q_CLASSINFO("D-Bus Interface", "demo.Chat")
    Q_CLASSINFO("D-Bus Introspection", ""
"  <interface name=\"demo.Chat\">\n"
"    <property access=\"read\" type=\"as\" name=\"users\"/>\n"
"    <signal name=\"userAdded\">\n"
"      <arg direction=\"out\" type=\"s\" name=\"user\"/>\n"
"    </signal>\n"
"    <signal name=\"userRemoved\">\n"
"      <arg direction=\"out\" type=\"s\" name=\"user\"/>\n"
"    </signal>\n"
"    <signal name=\"messageSent\">\n"
"      <arg direction=\"out\" type=\"a(ii)\" name=\"message\"/>\n"
"      <annotation value=\"Message\" name=\"com.trolltech.QtDBus.QtTypeName.In0\"/>\n"
"    </signal>\n"
"    <method name=\"addUser\">\n"
"      <arg direction=\"in\" type=\"s\" name=\"user\"/>\n"
"    </method>\n"
"    <method name=\"removeUser\">\n"
"      <arg direction=\"in\" type=\"s\" name=\"user\"/>\n"
"    </method>\n"
"    <method name=\"sendMessage\">\n"
"      <arg direction=\"in\" type=\"a(ii)\" name=\"message\"/>\n"
"      <annotation value=\"Message\" name=\"com.trolltech.QtDBus.QtTypeName.In0\"/>\n"
"    </method>\n"
"  </interface>\n"
        "")
public:
    ChatAdaptor(QObject *parent);
    virtual ~ChatAdaptor();

public: // PROPERTIES
    Q_PROPERTY(QStringList users READ users)
    QStringList users() const;

public Q_SLOTS: // METHODS
    void addUser(const QString &user);
    void removeUser(const QString &user);
    void sendMessage(Message message);
Q_SIGNALS: // SIGNALS
    void messageSent(Message message);
    void userAdded(const QString &user);
    void userRemoved(const QString &user);
};

#endif
```

ChatAdaptor.cpp

```
/*
 * This file was generated by qdbusxml2cpp version 0.7
 * Command line was: qdbusxml2cpp Chat.xml -i Message.hpp -a ChatAdaptor
 *
 * qdbusxml2cpp is Copyright (C) 2010 Nokia Corporation and/or its subsidiary(-ies).
 *
 * This is an auto-generated file.
 * Do not edit! All changes made to it will be lost.
 */

#include "ChatAdaptor.h"
#include <QtCore/QMetaObject>
#include <QtCore/QByteArray>
#include <QtCore/QList>
#include <QtCore/QMap>
#include <QtCore/QString>
#include <QtCore/QStringList>
#include <QtCore/QVariant>
```

```
/*
 * Implementation of adaptor class ChatAdaptor
 */

ChatAdaptor::ChatAdaptor(QObject *parent)
    : QDBusAbstractAdaptor(parent)
{
    // constructor
    setAutoRelaySignals(true);
}

ChatAdaptor::~ChatAdaptor()
{
    // destructor
}

QStringList ChatAdaptor::users() const
{
    // get the value of property users
    return qvariant_cast< QStringList >(parent()->property("users"));
}

void ChatAdaptor::addUser(const QString &user)
{
    // handle method call demo.Chat.addUser
    QMetaObject::invokeMethod(parent(), "addUser", Q_ARG(QString, user));
}

void ChatAdaptor::removeUser(const QString &user)
{
    // handle method call demo.Chat.removeUser
    QMetaObject::invokeMethod(parent(), "removeUser", Q_ARG(QString, user));
}

void ChatAdaptor::sendMessage(Message message)
{
    // handle method call demo.Chat.sendMessage
    QMetaObject::invokeMethod(parent(), "sendMessage", Q_ARG(Message, message));
}
```

ChatInterface.h

```
/*
 * This file was generated by qdbusxml2cpp version 0.7
 * Command line was: qdbusxml2cpp Chat.xml -i Message.hpp -p ChatInterface
 *
 * qdbusxml2cpp is Copyright (C) 2010 Nokia Corporation and/or its subsidiary(-ies).
 *
 * This is an auto-generated file.
 * Do not edit! All changes made to it will be lost.
 */

#ifndef CHATINTERFACE_H_1270658265
#define CHATINTERFACE_H_1270658265

#include <QtCore/QObject>
#include <QtCore/QByteArray>
#include <QtCore/QList>
#include <QtCore/QMap>
#include <QtCore/QString>
#include <QtCore/QStringList>
```

```
#include <QtCore/QVariant>
#include <QtDBus/QtDBus>
#include "Message.hpp"

/*
 * Proxy class for interface demo.Chat
 */
class DemoChatInterface: public QDBusAbstractInterface
{
   Q_OBJECT
public:
   static inline const char *staticInterfaceName()
   { return "demo.Chat"; }

public:
   DemoChatInterface(const QString &service, const QString &path, const QDBusConnection &connection, QObje

   ~DemoChatInterface();

   Q_PROPERTY(QStringList users READ users)
   inline QStringList users() const
   { return qvariant_cast< QStringList >(property("users")); }

public Q_SLOTS: // METHODS
   inline QDBusPendingReply<> addUser(const QString &user)
   {
      QList<QVariant> argumentList;
      argumentList << qVariantFromValue(user);
      return asyncCallWithArgumentList(QLatin1String("addUser"), argumentList);
   }

   inline QDBusPendingReply<> removeUser(const QString &user)
   {
      QList<QVariant> argumentList;
      argumentList << qVariantFromValue(user);
      return asyncCallWithArgumentList(QLatin1String("removeUser"), argumentList);
   }

   inline QDBusPendingReply<> sendMessage(Message message)
   {
      QList<QVariant> argumentList;
      argumentList << qVariantFromValue(message);
      return asyncCallWithArgumentList(QLatin1String("sendMessage"), argumentList);
   }

Q_SIGNALS: // SIGNALS
   void messageSent(Message message);
   void userAdded(const QString &user);
   void userRemoved(const QString &user);
};

namespace demo {
  typedef ::DemoChatInterface Chat;
}
#endif
```

ChatInterface.cpp

```
/*
 * This file was generated by qdbusxml2cpp version 0.7
 * Command line was: qdbusxml2cpp Chat.xml -i Message.hpp -p ChatInterface
 *
 * qdbusxml2cpp is Copyright (C) 2010 Nokia Corporation and/or its subsidiary(-ies).
 *
 * This is an auto-generated file.
 * This file may have been hand-edited. Look for HAND-EDIT comments
 * before re-generating it.
 */

#include "ChatInterface.h"

/*
 * Implementation of interface class DemoChatInterface
 */

DemoChatInterface::DemoChatInterface(const QString &service, const QString &path, const QDBusConnection &
    : QDBusAbstractInterface(service, path, staticInterfaceName(), connection, parent)
{
}

DemoChatInterface::~DemoChatInterface()
{
}
```

ChatWindow.hpp

```
#ifndef CHATWINDOW_HPP
#define CHATWINDOW_HPP

#include <QtGui/QMainWindow>
#include <QtGui/QStringListModel>

#include "ChatInterface.h"

namespace Ui
{
    class ChatWindow;
}

class QCloseEvent;

class ChatWindow : public QMainWindow
{
    Q_OBJECT

public:
    ChatWindow(demo::Chat& chatInterface, QWidget *parent = 0);
    virtual ~ChatWindow();

protected:
    virtual void closeEvent(QCloseEvent *event);

private:
    ChatWindow(const ChatWindow &other);
    ChatWindow& operator=(const ChatWindow &other);

    Ui::ChatWindow *ui;

    QString m_userName;

    QStringListModel m_users;
    demo::Chat &m_chatInterface;

private slots:
    void sendMessage();

    void chat_userAdded(const QString &user);
    void chat_userRemoved(const QString &user);
    void chat_messageSent(const Message &message);
};

#endif // CHATWINDOW_HPP
```

ChatWindow.cpp

```
clude "ChatWindow.hpp"
clude "ui_ChatWindow.h"

clude <QtGui/QCloseEvent>
clude <QtGui/QInputDialog>
clude <QtGui/QMessageBox>

clude "Message.hpp"

atWindow::ChatWindow(demo::Chat& chatInterface, QWidget *parent) :
MainWindow(parent),
i(new Ui::ChatWindow),
_userName(),
_users(),
_chatInterface(chatInterface)

i->setupUi(this);

StringList userList = chatInterface.users();
_users.setStringList(userList);
_users.sort(0);

onnect(&m_chatInterface, SIGNAL(userAdded(QString)), SLOT(chat_userAdded(QString)));
onnect(&m_chatInterface, SIGNAL(userRemoved(QString)), SLOT(chat_userRemoved(QString)));
onnect(&m_chatInterface, SIGNAL(messageSent(Message)), SLOT(chat_messageSent(Message)));

onnect(ui->lneMessage, SIGNAL(returnPressed()), SLOT(sendMessage()));
onnect(ui->btnSend, SIGNAL(released()), SLOT(sendMessage()));

ool ok;
_userName = QInputDialog::getText(this, "Username?", "User name:", QLineEdit::Normal, QDir::home().dirName

 (!ok)

  QApplication::exit();
  qFatal("Cancelled");


 (userList.contains(m_userName))

  QMessageBox::critical(this, "Invalid Username", "Username " + m_userName + " is already taken");
  QApplication::exit();
  qFatal("Duplicate username");


_chatInterface.addUser(m_userName);

etWindowTitle("Chat -- " + m_userName);
```

```
i->lstUsers->setModel(&m_users);


atWindow::~ChatWindow()

elete ui;


d ChatWindow::closeEvent(QCloseEvent *event)

MainWindow::closeEvent(event);

 (event->isAccepted())

  m_chatInterface.removeUser(m_userName);



d ChatWindow::sendMessage()

lessage message(m_userName, ui->lneMessage->text());
n_chatInterface.sendMessage(message);

i->lneMessage->clear();


d ChatWindow::chat_userAdded(const QString &user)

StringList users = m_users.stringList();
sers.append(user);
sers.sort();
n_users.setStringList(users);


d ChatWindow::chat_userRemoved(const QString &user)

StringList users = m_users.stringList();
sers.removeOne(user);
n_users.setStringList(users);


d ChatWindow::chat_messageSent(const Message &message)

i->txtChat->appendPlainText(message.getUser() + " : " + message.getText());
```

ChatWindow.ui

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>ChatWindow</class>
 <widget class="QMainWindow" name="ChatWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>600</width>
    <height>400</height>
   </rect>
  </property>
```

```xml
<property name="windowTitle">
 <string>ChatWindow</string>
</property>
<widget class="QWidget" name="centralWidget">
 <layout class="QVBoxLayout" name="verticalLayout">
  <item>
   <widget class="QWidget" name="widget_2" native="true">
    <layout class="QHBoxLayout" name="horizontalLayout_2">
     <item>
      <widget class="QListView" name="lstUsers">
       <property name="maximumSize">
        <size>
         <width>150</width>
         <height>16777215</height>
        </size>
       </property>
       <property name="focusPolicy">
        <enum>Qt::NoFocus</enum>
       </property>
       <property name="editTriggers">
        <set>QAbstractItemView::NoEditTriggers</set>
       </property>
       <property name="selectionMode">
        <enum>QAbstractItemView::NoSelection</enum>
       </property>
      </widget>
     </item>
     <item>
      <widget class="QPlainTextEdit" name="txtChat">
       <property name="focusPolicy">
        <enum>Qt::NoFocus</enum>
       </property>
      </widget>
     </item>
    </layout>
   </widget>
  </item>
  <item>
   <widget class="QWidget" name="widget" native="true">
    <layout class="QHBoxLayout" name="horizontalLayout">
     <item>
      <widget class="QLineEdit" name="lneMessage"/>
     </item>
     <item>
      <widget class="QPushButton" name="btnSend">
       <property name="text">
        <string>Send</string>
       </property>
      </widget>
     </item>
    </layout>
   </widget>
  </item>
 </layout>
</widget>
<widget class="QMenuBar" name="menuBar">
 <property name="geometry">
  <rect>
   <x>0</x>
   <y>0</y>
   <width>600</width>
   <height>25</height>
  </rect>
 </property>
```

```
  </widget>
  <widget class="QToolBar" name="mainToolBar">
   <attribute name="toolBarArea">
    <enum>TopToolBarArea</enum>
   </attribute>
   <attribute name="toolBarBreak">
    <bool>false</bool>
   </attribute>
  </widget>
  <widget class="QStatusBar" name="statusBar"/>
 </widget>
 <layoutdefault spacing="6" margin="11"/>
 <resources/>
 <connections/>
</ui>
```

Message.hpp

```
#ifndef MESSAGE_HPP
#define MESSAGE_HPP

#include <QtDBus>

class Message
{
public:
    Message();
    Message(const QString& user, const QString &message);
    Message(const Message &other);
    Message& operator=(const Message &other);
    ~Message();

    friend QDBusArgument &operator<<(QDBusArgument &argument, const Message &message);
    friend const QDBusArgument &operator>>(const QDBusArgument &argument, Message &message);

    QString getUser() const;
    QString getText() const;

    //register Message with the Qt type system
    static void registerMetaType();

private:
    QString m_user;
    QString m_text;
};

Q_DECLARE_METATYPE(Message)

#endif // MESSAGE_HPP
```

Message.cpp

```
#include "Message.hpp"

Message::Message() :
        m_user(),
        m_text()
{
}

Message::Message(const QString &user, const QString &text) :
      m_user(user),
      m_text(text)
{
}

Message::Message(const Message &other) :
      m_user(other.m_user),
      m_text(other.m_text)
{
}

Message& Message::operator=(const Message &other)
{
   m_user = other.m_user;
   m_text = other.m_text;

   return *this;
}

Message::~Message()
{
}

QString Message::getUser() const
{
   return m_user;
}

QString Message::getText() const
{
   return m_text;
}

void Message::registerMetaType()
{
   qRegisterMetaType<Message>("Message");

   qDBusRegisterMetaType<Message>();
}

QDBusArgument &operator<<(QDBusArgument &argument, const Message& message)
{
   argument.beginStructure();
   argument << message.m_user;
   argument << message.m_text;
   argument.endStructure();

   return argument;
}

const QDBusArgument &operator>>(const QDBusArgument &argument, Message &message)
{
   argument.beginStructure();
   argument >> message.m_user;
   argument >> message.m_text;
   argument.endStructure();

   return argument;
}
```

main.cpp

```
#include <QtDBus/QDBusConnection>
#include <QtDBus/QDBusConnectionInterface>
#include <QtGui/QApplication>

#include "Chat.hpp"
#include "ChatAdaptor.h"
#include "ChatInterface.h"
#include "ChatWindow.hpp"
#include "Message.hpp"

#define CHAT_SERVICE "demo.dbus.chat"
#define CHAT_PATH "/chat"

int main(int argc, char *argv[])
{
  /*
    Register the Message type first thing, so Qt knows how to handle
    it before an Adaptor/Interface is even constructed.

    It should be ok to register it further down, as long as no
    Message marshaling or unmarshaling takes place, but this is
    definitely the safest way of doing things.
  */
  Message::registerMetaType();

  QApplication a(argc, argv);
  Chat* pChat = NULL;
  ChatAdaptor* pChatAdaptor = NULL;

  /*
    Create a Chat instance and register it with the session bus only if
    the service isn't already available.
  */
  QDBusConnection connection = QDBusConnection::sessionBus();
  if (!connection.interface()->isServiceRegistered(CHAT_SERVICE))
  {
    pChat = new Chat(&a);
    pChatAdaptor = new ChatAdaptor(pChat);

    if (!connection.registerService(CHAT_SERVICE))
    {
      qFatal("Could not register service!");
    }

    if (!connection.registerObject(CHAT_PATH, pChat))
    {
      qFatal("Could not register Chat object!");
    }
  }

  demo::Chat chatInterface(CHAT_SERVICE, CHAT_PATH, connection);

  ChatWindow w(chatInterface);
  w.show();

  int ret = a.exec();

  //cleanup
  if (pChat)
  {
    delete pChat;
  }
  if (pChatAdaptor)
  {
    delete pChatAdaptor;
  }

  return ret;
}
```

DBusChat.pro

```
# -------------------------------------------------
# Project created by QtCreator 2010-04-06T21:08:22
# -------------------------------------------------
QT += dbus
TARGET = DBusChat
TEMPLATE = app
SOURCES += main.cpp \
  ChatWindow.cpp \
  Chat.cpp \
  Message.cpp \
  ChatAdaptor.cpp \
  ChatInterface.cpp
HEADERS += ChatWindow.hpp \
  Chat.hpp \
  Message.hpp \
  ChatAdaptor.h \
  ChatInterface.h
FORMS += ChatWindow.ui
```

来自“http://techbase.kde.org/index.php?title=Development/Tutorials/D-Bus/CustomTypes&oldid=81404”
1个分类： Tutorial

- KDE Links
  - Homepage
  - News
  - Community Forums
  - UserBase Wiki
  - Community Wiki
  - Documentation
  - Planet
  - Bugzilla
  - Developer Blogs