Practical Qt on Android JNI

Qt Con, 2016

Presented by BogDan Vatra

Material based on Qt 5.7, created on August 2016

- **What's new in Qt 5.7 for Android**
  - Android splash screen improvements
  - Helper functions to run C++ code on the Android UI thread
  - Android services
- JNI Crash Course
- Android Services with Qt

- **Android splash screen improvements**
- Helper functions to run C++ code on the Android UI thread
- Android services

Say hello to QtAndroid::hideSplashScreen()

- no more black screens until the application starts
- stays visible until QtAndroid::hideSplashScreen() is called
- one little problem, on pre Android 5 devices it flickers for a moment

How to use QtAndroid::hideSplashScreen()

Edit your **AndroidManifest.xml** file and:

- set the **android.app.splash_screen_drawable**
- enable **android.app.splash_screen_sticky**

```
<activity ...>
    <!-- ... -->
    <!-- Splash screen -->
    <meta-data android:name="android.app.splash_screen_drawable"
android:resource="@drawable/banner"/>
    <meta-data android:name="android.app.splash_screen_sticky" android:value="true"/>
    <!-- Splash screen -->
</activity>
```

How to use QtAndroid::hideSplashScreen()

From C++ code call **QtAndroid::hideSplashScreen()** after all the resources are loaded

```
 1 #include <QGuiApplication>
 2 #ifdef Q_OS_ANDROID
 3 # include <QtAndroid>
 4 #endif
 5
 6 int main(int argc, char *argv[])
 7 {
 8     //...
 9     // Next line usually takes some time to complete
10     engine.load(QUrl(QStringLiteral("qrc:/qml/main.qml")));
11 #ifdef Q_OS_ANDROID
12     QtAndroid::hideSplashScreen();
13 #endif
14     return app.exec();
15 }
```

- Android splash screen improvements
- **Helper functions to run C++ code on the Android UI thread**
- Android services

Say hello to QtAndroid::runOnAndroidThread[Sync]

- The problem:
  - Most of the Android Java methods (including the contructors) **MUST** be called on Android UI thread.
  - but we want to call them from C++ (Qt thread)!
- Qt 5.7 introduces:
  - `typedef std::function Runnable;`
  - `void QtAndroid::runOnAndroidThread(const Runnable &runnable)`
    - runs asynchronously the runable on Android UI thread
    - if it's called on the Android UI thread, it's executed immediately
    - usefull to call methods that don't return anything
  - `void QtAndroid::runOnAndroidThreadSync(const Runnable &runnable, int timeoutMs = INT_MAX)`
    - runs the runable on Android UI thread
    - waits until it's executed or until `timeoutMs` has passed
    - useful to create objects, or get properties on Android UI thread
- For pre Qt 5.7 check https://www.kdab.com/qt-android-run-c-code-android-ui-thread/

- Android splash screen improvements
- Helper functions to run C++ code on the Android UI thread
- **Android services**

Say hello to Android Services using Qt

Qt can now be used to easily create Android Services.

- What's new in Qt 5.7 for Android
- **JNI Crash Course**
  - Use case 1: access Java code from C/C++
  - Use case 2: access C/C++ code from Java
- Android Services with Qt

What is JNI and why do you need it ?

- JNI is the **Java Native Interface**. It is needed to do calls to/from Java world from/to native (C/C++) world.
- It is impossible for Qt to implement all Android features, so to use them you'll need JNI to access them.

Missing Qt APIs

- listening for Android O.S. notifications:
  - sd card notifications: bad removal, eject, mounted, unmounted, etc.
  - network notifications: network up/down.
  - battery level and charging state.
  - etc.
- accessing Android O.S. features:
  - telephony (Initiate calls, MMS, SMS, etc.)
  - contacts
  - speech (TTS and Speech Recognizer)
  - system accounts
  - system preferences
  - NFC
  - USB
  - printing (WARNING: needs API-19+)
- create own Android Activities and Services

- **Use case 1: access Java code from C/C++**
  - How to keep the screen on
  - How to use Android Toast
- Use case 2: access C/C++ code from Java

What can we do from C/C++ ?

- call static methods
- access static fields
- create Java objects
- call their methods
- access their fields

How can we do it ?

- using QAndroidExtras (QAndroidJniObject)
- using JNI functions where QAndroidExtras is not enough (e.g. access arrays)
- we need to specify their signatures

QAndroidJniObject saves the world !

- helper class useful to easily call Java code from C++
- hides all the JavaVM, JNIEnv, etc. madness
- automatically attaches/detaches to current thread
- automatically finds classes, methods & fields IDs
- caches classes, methods & fields IDs
- converts `QString` to `jstring` and vice-versa

Scalar types mapping table

Scalar types

C/C++	JNI	Java	Signature
uint8_t/unsigned char	jboolean	bool	Z
int8_t/char/signed char	jbyte	byte	B
uint16_t/unsigned short	jchar	char	C
int16_t/short	jshort	short	S
int32_t/int/(long)	jint	int	I
int64_t/(long)/long long	jlong	long	J
float	jfloat	float	F
double	jdouble	double	D
void		void	V

Java classes types

JNI	Java	Signature
jobject/jclass	JavaClass	Lfully/qualified/class/name;
jstring	String	Ljava/lang/String;
jobject/jclass	Object	Ljava/lang/Object;
jobject/jclass	Activity	Landroid/app/Activity;

- **NOTE:** use only **fully/qualified/class/name** when you need to specify only the class name (not as a type)

```
jclass stringClass = jniEnv->FindClass("java/lang/String");
```

Arrays mapping table

Arrays

JNI	Java	Signature
jbooleanArray	bool[]	[Z
jbyteArray	byte[]	[B
jcharArray	char[]	[C
jshortArray	short[]	[S
jintArray	int[]	[I
jlongArray	long[]	[L
jfloatArray	float[]	[F
jdoubleArray	double[]	[D
jarray	type[]	[Lfully/qualified/type/name;
jarray	String[]	[Ljava/lang/String;

- **NOTE:** to access arrays you'll need to use JNI functions (e.g. `GetArrayLength`, `GetObjectArrayElement`, `SetObjectArrayElement`, etc.)

Use case 1: access Java code from C/C++

- **How to keep the screen on**
- How to use Android Toast

There are two ways on Android to keep the screen on:

- by using `PowerManager` service
- by setting `WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON` flag to the activity window

we're going to use the second way in our example

How to keep the screen on from Qt

```
# Changes to your .pro file
# ....
QT += androidextras
# ....
```

```
void keepScreenOn(bool on) {
  static const int FLAG_KEEP_SCREEN_ON
    = QAndroidJniObject::getStaticField<jint>(
                                        "android/view/WindowManager$LayoutParams",
                                        "FLAG_KEEP_SCREEN_ON");

  QtAndroid::runOnAndroidThread([on, FLAG_KEEP_SCREEN_ON] {
    auto window = QtAndroid::androidActivity().callObjectMethod("getWindow",
                                                    "()Landroid/view/Window;");
    if (on)
      window.callMethod<void>("addFlags", "(I)V", FLAG_KEEP_SCREEN_ON);
    else
      window.callMethod<void>("clearFlags", "(I)V", FLAG_KEEP_SCREEN_ON);
  }
}
```

Yup, it's that simple !

How to keep the screen on from Qt

```
1 void keepScreenOn(bool on) {
2   static const int FLAG_KEEP_SCREEN_ON
3     = QAndroidJniObject::getStaticField<jint>(
4                                           "android/view/WindowManager$LayoutParams",
5                                           "FLAG_KEEP_SCREEN_ON");
6 //...
```

Use `T QAndroidJniObject::getStaticField(const char *className, const char *fieldName)` static method to get access to `FLAG_KEEP_SCREEN_ON` flag, with the following params:

- `className` is the fully/qualified/class/name$nestedName
  - for nested classes you'll need to use `$` instead of `/`
- `fieldName` is the field name

How to keep the screen on from Qt

```
1 //...
2   QtAndroid::runOnAndroidThread([on, FLAG_KEEP_SCREEN_ON] {
3     auto window = QtAndroid::androidActivity().callObjectMethod("getWindow",
4                                                                 "()Landroid/view/Window;");
5 //...
```

- use `QtAndroid::runOnAndroidThread` to run the C++ code on Android UI thread
- use `QAndroidJniObject QtAndroid::androidActivity()` to access the application activity
- use `QAndroidJniObject QAndroidJniObject::callObjectMethod(const char *methodName, const char *signature, ...) const` to get the Activity window, with the following params:
  - `"getWindow"` is the method name
  - `"()Landroid/view/Window;"` is the method signature
    - the method has no params
    - returns `Window` object

How to keep the screen on from Qt

```
1 //...
2     auto window = QtAndroid::androidActivity().callObjectMethod("getWindow",...
3     if (on)
4       window.callMethod<void>("addFlags", "(I)V", FLAG_KEEP_SCREEN_ON);
5     else
6       window.callMethod<void>("clearFlags", "(I)V", FLAG_KEEP_SCREEN_ON);
7   }
8 }
```

Use `T QAndroidJniObject::callMethod(const char *methodName, const char *signature, ...) const` to call `void addFlags/clearFlags(int flags)` methods with the following params:

- `"addFlags"/"clearFlags"` is the method name
- `"(I)V"` is the method signature
  - has a single param, an int
  - returns nothing
- `FLAG_KEEP_SCREEN_ON` is the parameter that will be passed to `addFlags/clearFlags` method

Use case 1: access Java code from C/C++

- How to keep the screen on
- **How to use Android Toast**

Android Toasts are small popups which are used to show the user some feedback. There are two ways to create **Toast** popups

- by creating a `Toast` object and setting all properties manually
- by using `Toast Toast.makeText(Context context, CharSequence text, int duration)` static method which creates a `Toast` object and sets the needed properties. This method needs 3 params:
  - the context (we'll use the activity)
  - the text to show
  - and the duration: one of `Toast.LENGTH_SHORT` (0) and `Toast.LENGTH_LONG` (1)

After we have the Toast object, we just need to call `Toast.show()` method to show it

we're going to use the second way in our example

How to use Android Toast

```
# Changes to your .pro file
# ....
QT += androidextras
# ....
```

```
enum Duration {
  SHORT = 0,
  LONG = 1
};

void showToast(const QString &message, Duration duration = LONG) {
  QtAndroid::runOnAndroidThread([message, duration] {
    auto javaString = QAndroidJniObject::fromString(message);
    auto toast = QAndroidJniObject::callStaticObjectMethod("android/widget/Toast",
       "makeText",
       "(Landroid/content/Context;Ljava/lang/CharSequence;I)Landroid/widget/Toast;",
       QtAndroid::androidActivity().object(),
       javaString.object(),
       jint(duration));

    toast.callMethod<void>("show");
  });
}
```

Yep, is that simple !

How to use Android Toast

```
1  enum Duration {
2    SHORT = 0,
3    LONG = 1
4  };
5
6  void showToast(const QString &message, Duration duration = LONG) {
7    QtAndroid::runOnAndroidThread([message, duration] {
8      //....
```

- define `SHORT` & `LONG`, though we could use `QAndroidJniObject::getStaticField` to get them
- use `QtAndroid::runOnAndroidThread` to run the C++ code on Android UI thread
  - capture `message` and `duration` params by value not by reference!

```
1 //...
2 auto javaString = QAndroidJniObject::fromString(message);
3 //...
```

- use `QAndroidJniObject QAndroidJniObject::fromString(const QString &string)` to convert a `QString` to a `QAndroidJniObject` string object

How to use Android Toast

```
1 //...
2 auto toast = QAndroidJniObject::callStaticObjectMethod("android/widget/Toast",
3    "makeText",
4    "(Landroid/content/Context;Ljava/lang/CharSequence;I)Landroid/widget/Toast;",
5    QtAndroid::androidActivity().object(),
6    javaString.object(),
7    jint(duration));
8 //...
```

Use `QAndroidJniObject QAndroidJniObject::callStaticObjectMethod(const char *className, const char *methodName, const char *signature, ...)` to call `Toast Toast.makeText(Context context, CharSequence text, int duration)` Java static method, with the following params:

- `"android/widget/Toast"` is the fully qualified class name
- `"makeText"` is the static method name
- `"(Landroid/content/Context;Ljava/lang/CharSequence;I)Landroid/widget/Toast;"` is the method signature
- `QtAndroid::androidActivity().object(), javaString.object(), jint(duration)` are the arguments for `makeText` method

```
1 //...
2     toast.callMethod<void>("show");
3   });
4 }
```

Use `T QAndroidJniObject::callMethod(const char *methodName) const` to call `Toast.show()` method

Create the Java function

Let's see how to call our own Java functions

```
1  // java file android/src/com/kdab/training/MyJavaClass.java
2  package com.kdab.training;
3
4  public class MyJavaClass
5  {
6      // this method will be called from C/C++
7      public static int fibonacci(int n)
8      {
9          if (n < 2)
10             return n;
11         return fibonacci(n-1) + fibonacci(n-2);
12     }
13 }
```

Demo android/JNIIntro

Call a static Java method from C/C++

Let's see what **fibonacci** function call looks like using the Qt androidextras module.

```
# Changes to your .pro file
# ....
QT += androidextras
# ....
```

```
// C++ code
#include <QAndroidJniObject>
int fibonacci(int n)
{
    return QAndroidJniObject::callStaticMethod<jint>
                        ("com/kdab/training/MyJavaClass" // java class name
                        , "fibonacci" // method name
                        , "(I)I" // signature
                        , n);
}
```

Yes, that's all folks !

In order to access Java world from C/C++, using the old fashioned way, you'll need to get access to two objects pointers:

- JavaVM
  - needed to get a JNIEnv pointer
  - can be shared between threads
- JNIEnv
  - provides most of the JNI functions
  - cannot be shared between threads

#NOTES

Call a Java method from C/C++, the old way

```
1  static JavaVM *s_javaVM = 0;
2  static jclass s_myJavaClass = 0;
3  static jmethodID s_methodID = 0;
4
5  static bool findClass(JavaVM *vm, JNIEnv *env)
6  {
7      s_javaVM = vm;
8
9      s_myJavaClass = env->FindClass("com/kdab/training/MyJavaClass");
10     if (!s_myJavaClass)
11         return false;
12
13     s_methodID = env->GetStaticMethodID(s_myJavaClass, "fibonacci", "(I)I");
14     if (!s_methodID)
15         return false;
16
17     return true;
18 }
19
20 // call findClass inside JNI_OnLoad
```

#NOTES

Call a Java method from C/C++, the old way

```
1  jint fibonacci(jint n)
2  {
3      JNIEnv* env = 0;
4
5      // Qt loop runs in a different thread than Android one.
6      // We must first attach the VM to that thread before we call anything.
7      if (s_javaVM->AttachCurrentThread(&env, NULL) < 0)
8          exit(0);
9
10     jint res = env->CallStaticIntMethod(s_myJavaClass, s_methodID, n);
11
12     s_javaVM->DetachCurrentThread();
13
14     return res;
15 }
```

#NOTES

Call a Java method from C/C++, the old way

As you can see it not that easy to call (just) a static Java function using the old fashioned plain JNI APIs, and things will become even nastier when you have to instantiate Java classes, and use Java objects from C/C++.

#NOTES

- Use case 1: access Java code from C/C++
- **Use case 2: access C/C++ code from Java**

What can we do from Java ?

- call static C/C++ functions

Your job

- declare a native method in Java using `native` keyword (see slide 43)
- register native method in C/C++ (see slide 44, slide 47)
- do the actual call (see slide 43)

Declare and invoke the Java native methods

```java
// java file android/src/com/kdab/training/MyJavaClass.java
package com.kdab.training;

class MyJavaNatives
{
    // declare the native method
    public static native void sendFibonaciResult(int n);
}

public class MyJavaClass
{
    // this method will be called from C/C++
    public static int fibonacci(int n)
    {
        if (n < 2)
            return n;
        return fibonacci(n-1) + fibonacci(n-2);
    }

    public static void compute_fibonacci(int n)
    {
        // callback the native method with the computed result.
        MyJavaNatives.sendFibonaciResult(fibonacci(n));
    }
}
```

C/C++ register native methods

Registering functions using
Java_Fully_Qualified_Class_Name_MethodName template.

```
1  // C++ code
2  #include <jni.h>
3
4  #ifdef __cplusplus
5  extern "C" {
6  #endif
7
8  JNIEXPORT void JNICALL
9      Java_com_kdab_training_MyJavaNatives_sendFibonaciResult(JNIEnv */*env*/,
10                                                             jobject /*obj*/,
11                                                             jint n)
12 {
13     qDebug() << "Computed fibonacci is:" << n;
14 }
15
16 #ifdef __cplusplus
17 }
18 #endif
```

Use **Java_Fully_Qualified_Class_Name_MethodName**

Pro:

- **easier to declare**, you don't need to specify the function signature
- **easier to register**, you don't need to explicitly register it

Con:

- the function names are **huge**:
  `Java_com_kdab_training_MyJavaNatives_sendFibonaciResult`
- the library will export lots of functions
- **unsafer**, there is no way for the VM to check the function signature

Use **JNIEnv::RegisterNatives** to register native functions

Step 4 call JNIEnv::RegisterNatives(java_class_ID, methods_vector, n_methods)

Step 3 find the ID of java class that declares these methods using JniEnv::FindClass

Step 2 create a vector with all C/C++ methods that you want to register

Step 1 get JNIEnv pointer by defining
JNIEXPORT jint JNI_OnLoad(JavaVM* vm, void* /*reserved*/).
You can define it **(once per .so file)** in any .cpp file you like

Use JNIEnv::RegisterNatives

```
1  // C++ code
2  #include <jni.h>
3
4  // define our native method
5  static void sendFibonaciResult(JNIEnv */*env*/, jobject /*obj*/, jint n)
6  {
7      qDebug() << "Computed fibonacci is:" << n;
8  }
9
10 // step 2
11 // create a vector with all our JNINativeMethod(s)
12 static JNINativeMethod methods[] = {
13       { "sendFibonaciResult", // const char* function name;
14         "(I)V", // const char* function signature
15         (void *)sendFibonaciResult // function pointer }
16 };
```

Use JNIEnv::RegisterNatives

```
1  // step 1
2  // this method is called automatically by Java after the .so file is loaded
3  JNIEXPORT jint JNI_OnLoad(JavaVM* vm, void* /*reserved*/)
4  {
5      JNIEnv* env;
6      // get the JNIEnv pointer.
7      if (vm->GetEnv(reinterpret_cast<void**>(&env), JNI_VERSION_1_6) != JNI_OK)
8          return JNI_ERR;
9
10     // step 3
11     // search for Java class which declares the native methods
12     jclass javaClass = env->FindClass("com/kdab/training/MyJavaNatives");
13     if (!javaClass)
14         return JNI_ERR;
15
16     // step 4
17     // register our native methods
18     if (env->RegisterNatives(javaClass, methods,
19                              sizeof(methods) / sizeof(methods[0])) < 0) {
20         return JNI_ERR;
21     }
22     return JNI_VERSION_1_6;
23 }
```

Use JNIEnv::RegisterNatives

Pro:

- the native function can have any name you want
- the library needs to export only one function **(JNI_OnLoad)**.
- **safer**, because the VM checks the declared signature

Con:

- is slightly harder to use

C/C++ register native methods

It's just a matter of taste which method you decide to use to register your native functions.
We do recommend you to use **JNIEnv::RegisterNatives** as it offers you extra protection because the VM checks the functions signature

- What's new in Qt 5.7 for Android
- JNI Crash Course
- **Android Services with Qt**
  - Getting started
  - Use QtRemoteObject for communication

- **Getting started**
- Use QtRemoteObject for communication

Extend QtService for every service

Every single Qt Android Service must have its own Service java class which extends QtService

```
1 // java file android/src/com/kdab/training/MyService.java
2 package com.kdab.training;
3 import org.qtproject.qt5.android.bindings.QtService;
4
5 public class MyService extends QtService
6 {
7 }
```

Add the service section(s) to your AndroidManifest.xml file

- copy & paste the template from https://wiki.qt.io/AndroidServices
- set `android:name` attribute with your service class name

```
<application ... >
  <!-- .... -->
  <service android:process=":qt" android:name=".MyService">
  <!-- android:process=":qt" is needed to force the service to run on a separate
process than the Activity -->

    <!-- .... -->

    <!-- Background running -->
    <meta-data android:name="android.app.background_running" android:value="true"/>
    <!-- Background running -->
  </service>
  <!-- .... -->
</application>
```

How to start the service ?

There are two ways to start a service:

- at boot time
- on demand

Start service at boot time

There are three steps:

- add RECEIVE_BOOT_COMPLETED permission

```
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
```

- add a receiver element

```
<application ... >
    <!-- .... -->
    <receiver android:name=".MyBroadcastReceiver">
        <intent-filter>
            <action android:name="android.intent.action.BOOT_COMPLETED" />
        </intent-filter>
    </receiver>
    <!-- .... -->
</application>
```

- implement MyBroadcastReceiver class

```
public class MyBroadcastReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Intent startServiceIntent = new Intent(context, MyService.class);
        context.startService(startServiceIntent);
    }
}
```

Start service on demand

Use `Context.startService(Intent intent)` method:

- add a static method to your MyService

```
1  // java file android/src/com/kdab/training/MyService.java
2  package com.kdab.training;
3
4  import android.content.Context;
5  import android.content.Intent;
6  import org.qtproject.qt5.android.bindings.QtService;
7
8  public class MyService extends QtService
9  {
10     public static void startMyService(Context ctx) {
11         ctx.startService(new Intent(ctx, MyService.class));
12     }
13 }
```

- call it from Qt when you want to start the service

```
1  QAndroidJniObject::callStaticMethod<void>("com/kdab/training/MyService",
2                                            "startMyService",
3                                            "(Landroid/content/Context;)V",
4                                            QtAndroid::androidActivity().object());
```

Where to put your Qt Service ?

- in the same **.so** file with the application
- in a separate **.so** file

Same .so for app & service(s)

- pass some arguments to know if and what kind of service to start.

```
<service ... >
    <!-- ... -->
    <!-- Application arguments -->
    <meta-data android:name="android.app.arguments" android:value="-service"/>
    <!-- Application arguments -->
    <!-- ... -->
</service>
```

- set the same `android.app.lib_name` metadata for both service(s) & activity elements

```
<service ... >
    <!-- ... -->
    <meta-data android:name="android.app.lib_name"
android:value="-- %%INSERT_APP_LIB_NAME%% --"/>
    <!-- ... -->
</service>
```

Separate .so files for app & service(s)

- create the server .pro file

```
1 TEMPLATE = lib
2 TARGET = server
3 CONFIG += dll
4 QT += core
5 SOURCES += server.cpp
```

- the server .so main entry is the `main` function

```
1 #include <QDebug>
2
3 int main(int argc, char *argv[])
4 {
5     qDebug() << "Hello from service";
6     return 0
7 }
```

- set the service `android.app.lib_name`

```
1 <service ... >
2     <!-- ... -->
3     <meta-data android:name="android.app.lib_name" android:value="server"/>
4     <!-- ... -->
5 </service>
```

- Getting started
- **Use QtRemoteObject for communication**

What is QtRemoteObjects

QtRemoteObjects is a playground Qt module led by Ford, for object remoting between processes/devices:

- exports QObjects remotely (properties, singals & slots)
- exports QAbstractItemModels remotely
- creates a replicant on the client side you can interface with
- repc generates source & replica (server & client) source files from .rep files
  - .rep file is the QtRemoteObjects IDL (interface description language)

Get QtRemoteObjects

QtRemoteObjects is located at http://code.qt.io/cgit/playground/qtremoteobjects.git/ :

```
1 $ git clone git://code.qt.io/playground/qtremoteobjects.git
2 $ cd qtremoteobjects
3 $ qmake -r && make && make install
```

Use QtRemoteObjects

- add QtRemoteObjects to your .pro files

```
1 # ...
2 QT += remoteobjects
3 # ...
```

- create .rep file(s)

```
1 class PingPong {
2     SLOT(void ping(const QString &msg));
3     SIGNAL(pong(const QString &msg));
4 }
```

- add it to the server .pro file

```
1 # ...
2 REPC_SOURCE += pingpong.rep
3 # ...
```

- add it to the client .pro file

```
1 # ...
2 REPC_REPLICA += pingpong.rep
3 # ...
```

QtRemoteObjects source(server) side implementation

- implement .rep interfaces (`PingPongSource`)
- export `PingPong` object using `enableRemoting`

```
 1 #include <QCoreApplication>
 2 #include "rep_pingpong_source.h"
 3
 4 class PingPong : public PingPongSource {
 5 public slots:
 6     // PingPongSource interface
 7     void ping(const QString &msg) override {
 8         emit pong(msg + " from server");
 9     }
10 };
11
12 int main(int argc, char *argv[])
13 {
14     QCoreApplication app(argc, argv);
15
16     QRemoteObjectHost srcNode(QUrl(QStringLiteral("local:replica")));
17     PingPong pingPongServer;
18     srcNode.enableRemoting(&pingPongServer);
19
20     return app.exec();
21 }
```

Demo android/Service

QtRemoteObjects replica(client) side implementation

- use `QRemoteObjectNode` to connect to `QRemoteObjectHost`
- use `QRemoteObjectNode:acquire` to link the local object to the remote one

```
1  #include "rep_pingpong_replica.h"
2
3  // ....
4      QRemoteObjectNode repNode;
5      repNode.connectToNode(QUrl(QStringLiteral("local:replica")));
6      QSharedPointer<PingPongReplica> rep(repNode.acquire<PingPongReplica>());
7      bool res = rep->waitForSource();
8      Q_ASSERT(res);
9      QObject::connect(rep.data(), &PingPongReplica::pong, [](const QString &msg){
10         qDebug() << msg;
11     });
12     rep->ping("Hello");
```

Questions ?

Thank you for your time!

Contact us:

- **http://www.kdab.com**
- **qtonandroid@kdab.com**
- **info@kdab.com**
- **training@kdab.com**
- **bogdan@kdab.com**