XE5 - Fun with external Java libraries

Now, I have to assume a few things here - you are reasonably familiar with delphi AND you can create a library APK yourself. It's not hard, honest. You DO need to be able to create an APK or JAR with a classes.dex AND I am only (for my purposes) interested in talking to a java class which I have code for - its possible to do this without the actual java code but you will need the classes.dex file. You also need to be brave to delve into the world of JNI.

Android API

To do any of this you have to hit the Android API support functions that are provided, specifically you need;

- Androidapi.Jni
- Androidapi.JNI.JavaTypes
- Androidapi.JNI.Dalvik
- Androidapi.JNIBridge
- Androidapi.JNI.GraphicsContentViewText
- FMX.Helpers.Android

It took me a bit of grepping and source spelunking to work this lot out, I would advise you do the same – it is (interestingly) informative.

Loading a class dynamically

Java provides a set of mechanisms to allow you split code across JAR files, these involve classloaders and generally add some reflection to interact with the contained classes.

Android extends this to provide support for compiled and compressed java classes contained in DEX files - DEXClassLoader. This android class will allow you to load and get class references of a contained class – which you can then use to create objects and interact with.

I found a Delphi import for DEXClassLoader in Androidapi.Jni.Dalvik – add this to your project and you can use it. But hold on a moment – go read the Android developer page on the DexClassLoader and you will see that there are a couple of things we need to do first.

1. We need a full path to the APK/JAR file that is correct for the device you are targeting.
2. We need a application writable folder to put the optimized DEX file in (don't ask).
3. We need a "parent" for the class loader.

These things stumpted me for a while – but help is at hand, well a helper class anyway. Delphi helpfully supplies a FMX.Helpers.Android unit – this has a few useful things in it, but most importantly it provides access to the Android level context and native activity objects. If you don't understand what these are then I would suggest a bit of research and learning about android before you get in over your head.

Pete
2013-09-18 21:32:41

Marked set by Pete

Setting up

Presuming you have created a java class project using eclipse or android studio AND presuming that it compiles then you need to copy that to your device. Logically it should be done as part of the application deployment, but at the time of writing I hadn't figured that part out (or the deploying of extra files is broken somehow, now that XE5 is released I must try this part again) – I put mine into the Downloads folder of the internal (non-system) disk of my Nexus 7.

Using a handy SManager app I had already installed (which contains a workable file manager) I managed to get a file path for my APK that was (ready for this?)

"/storage/emulated/0/Download/<APK Filename>"

There is almost certainly a way of getting this folder at run time using the Android API – I'll leave you to figure that out. Unless you are also targeting a Nexus 7 this path is probably going to be useless to you – put your APK into a location that makes sense to you and using a file manager app on the device to get the one that's correct for you. OR go work out how to get the external storage from the API – I'll give you a hint in a mo!

If you have a rooted device and you can get access to the deployment folder in /data/data then you can deploy the APK into the libs folder (as you are supposed to); however I found that first of you have to root your device to do it which is probably not a good idea unless you are targeting rooted devices specifically and secondly that you will have to do it *each time you hit the run or debug button in Delphi*. I quickly got bored of that game and placed my APK elsewhere. I will have to test actually installing the library APK and see where it goes and if you can load it still in the same way. Something for another day perhaps.

Get the directories

FMX.Helpers.Android contains a global var "SharedActivityContext" – we can use this to get some bits of information back about the local Android environment – notably the "getDir" function to query where you can put stuff – this in really useful in android apps – you can use this to work out where to store state information for your app, or where to put databases you are using. The Android Dev help page for this is here.

You'll need to use the helper function STRINGTOJSTRING to convert any Delphi strings to JSTRINGS to pass them to a java call – including the GETDIR function.

Code sample

```
const
  test_apk_fn='/storage/emulated/0/Download/test.apk';
var
  context:JContext;
  dexpath_jfile,optimizedpath_jfile:JFile;
  dexpath_jstring,optimizedpath_jstring:JString;
begin
  context:=SharedActivityContext;

  dexpath_jstring:=StringToJString(test_apk_fn);
  optimizedpath_jfile:=context.getDir(StringToJString('outdex'),
                            TJContext.javaclass.mode_private);
```

```
  optimizedpath_jstring:=optimizedpath_jfile.getAbsolutePath;
```

Note that we have to jump through a couple of hoops – notably getDir returns a JFILE type which we call the GETABSOLUTEPATH member of to return something useful , also used the TJContext.JavaClass to access the MODE_PRIVATE constant – this is true of all the imported android java classes, the <classname>.javaclass member provides convenient access to class level constants and class procedures and functions that are not immediately visible in delphi.

Load the library, then load the class

Next step – the title says it all really, we are going to load the class. The important part here is the call to the correct constructor – calling the TDelphiClass.create constructor never worked for me, I had to find the constructor (typically called INIT) and hit it directly. This allows you to specify where the APK/JAR is and where to optimise the classes to – this second folder has to be application writeable so it is suggested that you make use of the *context.getDir* function to ensure that you have full access at run time.

Using the above code example, the call to create the class loader looks like this:-

```
cl:=TJDexClassLoader.JavaClass.init(dexpath_jstring,optimizedpath_js
tring,nil,TJDexClassLoader.JavaClass.getSystemClassLoader);
```

Where *cl* is a *JDexClassLoader* variable – *not a TJDexClassLoader*. IF this works you get a *JDexClassLoader* back – if the result is NIL then you have gotten something wrong with your paths or the APK/JAR library.

This done, you make a call to the *loadclass* function – this (if it works) will return a JLang_Class object which represents the class in java. For my test I had a simple, parameter-less constructor, if you have to pass parameters to the constructor then you will need to jump through more hoops I'm afraid – you'll have to hit the JNI to find the correct constructor (using the method signature) – I'll be covering this in a mo.

For my testing I could use a simple call to *newInstance*.

Code sample

```
const
  test_apk_fn='/storage/emulated/0/Download/test.apk';
var
  context:JContext;
  dexpath_jfile,optimizedpath_jfile:JFile;
  dexpath_jstring,optimizedpath_jstring:JString;
  cl:JDexClassLoader;
  jLoadedClass:Jlang_Class;
  jLoadedObject:JObject;
begin
  context:=SharedActivityContext;

  dexpath_jstring:=StringToJString(test_apk_fn);
  optimizedpath_jfile:=context.getDir(StringToJString('outdex'),
                              TJContext.javaclass.mode_private);
  optimizedpath_jstring:=optimizedpath_jfile.getAbsolutePath;
```

```
  cl:=TJDexClassLoader.JavaClass.init(dexpath_jstring,
optimizedpath_jstring,nil,
TJDexClassLoader.JavaClass.getSystemClassLoader);

  {you should test for a nil return here and implement error
handling}

  jLoadedClass:=cl.loadclass(
                    stringtoJString('com/example/test/testclass'));
  jLoadedObject=jLoadedClass.newInstance;
```

Note the class path, you are not using the expected package naming *com.domain.package* that you would use in Java – if you take the time to examine some of the RTL Android imports you will see that the java signatures take the *com/domain/package/classname* format – you can list the classes and the class methods defined in a APK/JAR by extracting the *classes.dex* file and running DEXDUMP on it (which is included in the Android SDK, helpfully enough).

Finally, also note that class, package and indeed everything is Java is case sensitive.

Using JNI to find and call methods

When using java classes that are defined by the RTL, JNI is taken care of behind the scenes, the classes and class methods are mapped to interfaces according to name and signature. This process is seemless and very easy to use should you find an Android class which is not already imported for you by the RTL.

There are a couple of things you need to know before you start, however. Firstly you need to know the *exact* case of the java method you are calling, its best to either have the source code to check OR to run DEXDUMP on the *classes.dex* file.

You gain access to the JNI base calls by using the PJNIENV structure from Android.JNI. You get a reference to one valid to the current run time environment by using the JNIRESOLVER class in Android.JNIBridge.

Code sample

```
Uses
  Android.JNIBridge;
var
  JavaEnv:PJNIEnv;
Begin
  JavaEnv:=TJNIResolver.GetJNIEnv;
End;
```

This done, you can now make base JNI calls using this structure – rather than being a class, PJBIEnv is in fact a record.

There is a final bit of trickery needed to use JNI – you need to get JNIObject references for the loaded class and the java object to pass the JNI methods. Fortunately, you can get to these objects by using interfaces.

Code sample

```
var
  jLoadedObject:JObject;
  jLoadedClass:JLang_Class;
  jLoadedClassID,jLoadedObjectID:JNIObject;
begin
  {...do the class load and get the references to the java class and
java object first!}

  jLoadedClassID:=(jLoadedClass as ILocalObject).GetObjectID;
  jLoadedObjectID:= (jLoadedObject as ILocalObject).GetObjectID;
end;
```

ALL java imports implement the *ILocalObject* interface – however you can wrap this in error checking and trapping easily enough by doing a *QueryInterface* on the java objects. Note that the java class returned by *loadclass* is, in fact, just another object to the JNI.

Finally, you have everything you need to query and call members of the java classes you import – the simplest way of querying a java class is to use the *TJNIResolver* class (although you can do it with direct JNI calls using the *PJNIEnv* structure).

For my internal testing I implemented two simple methods in my java class – on to set a internal integer value and one to return it. This is designed to be a proof of concept and nothing more, as a result I have not attempted to cover calling java methods that take multiple parameters, although I don't think its much more difficult than this.

Code sample

```
var
  jLoadedClassID,jLoadedObjectID:JNIObject;
  jGetMethod,jSetMethod:JNIMethodID;
begin
  {...don't forget the other bits first!}

  jGetMethod:=TJNIResolver.GetJavaMethodID(jLoadedClassID
,'getIntValue','()I');
  jSetMethod:=TJNIResolver.GetJavaMethodID(jLoadedClassID
,'setIntValue','(I)V');
end;
```

Note that you need to know the method signature before you call it – doing a *DEXDUMP* on *classes.dex* will tell you this – although its not hard to translate from the source. My test class had two methods – function "getIntValue" returning a 32bit integer and procedure "setIntValue" taking 1 integer parameter. This gives a method signature of "*()I*" and "*(I)V*", respectively.

Calling these methods requires you to use two JNI *CallXXXXMethod* members, there are many defined for passing parameters and returning results in three ways. I am using the A methods which take the arguments as a JNIValue (well, a pointer to one or a pointer to an array of them).

Code sample

```
var
  jLoadedClassID,jLoadedObjectID:JNIObject;
  jGetMethod,jSetMethod:JNIMethodID;
  JavaEnv:PJNIEnv;
  jiReturnedValue:JNIInt;
  jvValuetoSet:JNIValue;
begin
  {...don't forget the other bits first!}

  jiReturnedValue:=JavaEnv^.CallIntMethodA(JavaEnv,
jLoadedObjectID,jGetMethod);

  jvValuetoSet.i:=1234567890;
  JavaEnv^.CallVoidMethodA(JavaEnv, jLoadedObjectID, jSetMethod,
@jvValuetoSet);
end;
```

There you have it – I believe you have to manage the objects you create in this way yourself, so don't forget to properly dispose of them (either cast the objects to TJavaImport and free them or call *DeleteGlobalRef* of *TJNIResolver*). Easy when you know how, eh?