Launching activities and handling results in Delphi XE6 Android apps

Brian Long Consultancy & Training Services Ltd.
April 2014

This is an update to an article based on Delphi XE5 available here.

Accompanying source files available through this download link.

Contents

Introduction

Delphi Android apps can use Android APIs, both those already present in the RTL and others we can translate manually, in order to interact with parts of the Android system. This article will look at how we can launch a variety of functionality using standard Android intent objects to instigate execution of available activities. Some of these may be part of the OS, but others will not be. Doing so will enable our own applications to integrate with the Android infrastructure and take advantage of convenient functionality installed on the device.

Through this article you should gain an insight into the techniques involved in communicating with Android activities, both launching them with appropriate parameters, but also receiving responses back from them when they are done.

What sort of activities are we talking about?

As mentioned above we'll use the Android `Intent` object, which is an object that represents a message of sorts. We'll set the object up and use it to launch a variety of different activities in order to bring functionality present in the Android eco-system 'into' our application. We'll look at displaying battery usage, viewing URLs, looking at maps, sending SMS and email messages, and scanning barcodes.

A toast before we start

Since we're looking at Android-specific things in this article it seems appropriate to make use of Android-specific features. Rather than showing messages with message boxes, we'll use Android toast messages instead. Toast messages are messages that just pop up onto the screen, rather like toast does out of a toaster. The toast API is not wrapped up by Delphi, but it's a simple API so I include my toast import unit below:

```
unit Androidapi.JNI.Toast;

//Java bridge class imported by hand by Brian Long (http://blong.com)

interface

uses
  Androidapi.JNIBridge,
  Androidapi.JNI.JavaTypes,
  Androidapi.JNI.GraphicsContentViewText;

type
  TToastLength = (LongToast, ShortToast);

  JToast = interface;

  JToastClass = interface(JObjectClass)
```

```
  ['{69E2D233-B9D3-4F3E-B882-474C8E1D50E9}']
    {Property methods}
    function _GetLENGTH_LONG: Integer; cdecl;
    function _GetLENGTH_SHORT: Integer; cdecl;
    {Methods}
    function init(context: JContext): JToast; cdecl; overload;
    function makeText(context: JContext; text: JCharSequence; duration: Integer): JToast; cdecl;
    {Properties}
    property LENGTH_LONG: Integer read _GetLENGTH_LONG;
    property LENGTH_SHORT: Integer read _GetLENGTH_SHORT;
  end;

  [JavaSignature('android/widget/Toast')]
  JToast = interface(JObject)
  ['{FD81CC32-BFBC-4838-8893-9DD01DE47B00}']
    {Methods}
    procedure cancel; cdecl;
    function getDuration: Integer; cdecl;
    function getGravity: Integer; cdecl;
    function getHorizontalMargin: Single; cdecl;
    function getVerticalMargin: Single; cdecl;
    function getView: JView; cdecl;
    function getXOffset: Integer; cdecl;
    function getYOffset: Integer; cdecl;
    procedure setDuration(value: Integer); cdecl;
    procedure setGravity(gravity, xOffset, yOffset: Integer); cdecl;
    procedure setMargin(horizontalMargin, verticalMargin: Single); cdecl;
    procedure setText(s: JCharSequence); cdecl;
    procedure setView(view: JView); cdecl;
    procedure show; cdecl;
  end;
  TJToast = class(TJavaGenericImport<JToastClass, JToast>) end;

procedure Toast(const Msg: string; Duration: TToastLength = ShortToast);

implementation

uses
  FMX.Helpers.Android,
  Androidapi.Helpers;

procedure Toast(const Msg: string; Duration: TToastLength);
var
  ToastLength: Integer;
begin
  if Duration = ShortToast then
    ToastLength := TJToast.JavaClass.LENGTH_SHORT
  else
    ToastLength := TJToast.JavaClass.LENGTH_LONG;
  CallInUiThread(procedure
  begin
    TJToast.JavaClass.makeText(SharedActivityContext, StrToJCharSequence(msg),
      ToastLength).show
  end);
end;

end.
```

The ins and outs of using the Java Bridge interface to represent Java classes using Delphi interfaces (one for the class methods and one for the interface methods, bound together by the `TJavaGenericImport`-based generic bridge class) are a bit beyond the scope of this particular article, but I did go into it in my CodeRage 8 session on using Android APIs from Delphi XE5:

Even if we ignore the details of the listing you'll see that the specifics of the toast message, including invoking the toast in the Java UI thread as opposed to the Delphi FireMonkey thread, have been tucked away and hidden behind a simple `Toast` procedure, which we'll use in the sample code. The toast import unit has been named similar to how the Delphi RTL's Android import units are named with an Androidapi.JNI. prefix.

Looking at battery usage

Ok, let's take a simple example to start with. Let's look at how your application can launch the standard system battery usage activity. You'll see over in the Android API documentation that `ACTION_POWER_USAGE_SUMMARY` is a Java string constant defined within the Android `Intent` class, and is designed specifically for this purpose. This string defines the action that the intent represents, the launching of the battery usage summary activity. So we need to create an Android `Intent` object, tell it that its action is `ACTION_POWER_USAGE_SUMMARY` and then start the activity thus represented.

This can be done with this code:

```
unit ShowBatteryUsageActivity;

interface

procedure ShowBatteryUsage;

implementation

uses
  FMX.Helpers.Android, Androidapi.JNI.GraphicsContentViewText, Androidapi.JNI.Toast;

procedure ShowBatteryUsage;
var
  Intent: JIntent;
  ResolveInfo: JResolveInfo;
begin
  Intent := TJIntent.JavaClass.init(TJIntent.JavaClass.ACTION_POWER_USAGE_SUMMARY);
  ResolveInfo := SharedActivity.getPackageManager.resolveActivity(Intent, 0);
  if ResolveInfo = nil then
    Toast('Cannot display battery usage', ShortToast)
  else
    SharedActivity.startActivity(Intent);
end;

end.
```

An Android `Intent` is represented in Delphi by the `JIntent` interface, which is returned after constructing an instance of the Android `Intent` class through the `TJIntent` bridge class's `init` method. `init` is a class method that represents a Java constructor, and this particular overload takes a Java string as an action parameter.

So the bridge class offers up the class methods (including constructors) of the class it represents (`Intent` in this case) through its `JavaClass` property. This constructor call returns us a `JIntent` interface, through which we can access the instance methods of our constructed `Intent` instance.

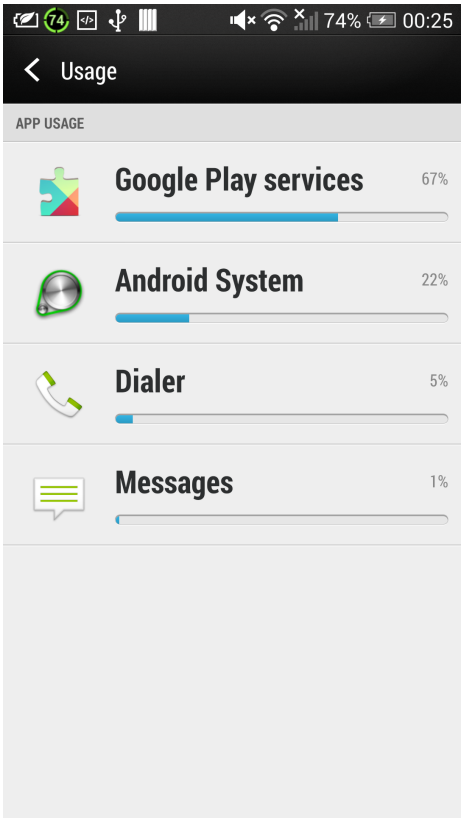
If you spend any time browsing the Android SDK documentation then do bear in mind that what we call class methods Java programmers call static methods.

The second statement calls onto an Android helper method to give us an interface reference that represents the FMX activity that sits behind the UI of our Android app. `SharedActivity` gives us a `JActivity` reference to represent the app's activity (although in truth the Delphi app uses an activity descendant called `NativeActivity` as the base class that it inherits from, which is common practice in a native code application). Through the activity reference we gain access to the device's `PackageManager`, which can help us understand if there is an installed package that will be able to process the intended action.

Assuming it looks like something will understand the request we ask our activity to start the activity represented by the `Intent` object.

It's quite straightforward stuff really and represents standard Android API programming, just represented in Delphi using the relevant bridge classes and interfaces that are necessary, most of which are provided in various RTL units, but some of which have been added in (for the toast API).

Here's the battery usage display on a HTC One:



Launching a URL

It is a common requirement to show a web page from an app. You could embed a browser component to do this or alternatively just ask the system browser to run the URL by requesting the system start an activity that can view the URL in question.

In this case we need to specify an action to view some data and pass the URL (or URI) in as the data to view. So this time we need to initialise the intent with two pieces of information: an action and some data to act upon.

Since the code we need is not very different from what we wrote above, we'll pull that code apart into more reusable routines, along with a variation that solves this new requirement:

```
unit LaunchActivities;

interface

procedure ShowBatteryUsage;
procedure LaunchURL(const URL: string);

implementation

uses
  FMX.Helpers.Android,
  Androidapi.Helpers,
  Androidapi.JNI.GraphicsContentViewText,
  Androidapi.JNI.Toast,
  Androidapi.JNI.JavaTypes,
  Androidapi.JNI.Net;

function LaunchActivity(const Intent: JIntent): Boolean; overload;
var
  ResolveInfo: JResolveInfo;
begin
  ResolveInfo := SharedActivity.getPackageManager.resolveActivity(Intent, 0);
  Result := ResolveInfo <> nil;
  if Result then
    SharedActivity.startActivity(Intent);
end;
```

```
function LaunchActivity(const Action: JString): Boolean; overload;
var
  Intent: JIntent;
begin
  Intent := TJIntent.JavaClass.init(Action);
  Result := LaunchActivity(Intent);
end;

function LaunchActivity(const Action: JString; const URI: Jnet_Uri): Boolean; overload;
var
  Intent: JIntent;
begin
  Intent := TJIntent.JavaClass.init(Action, URI);
  Result := LaunchActivity(Intent);
end;

procedure ShowBatteryUsage;
begin
  if not LaunchActivity(TJIntent.JavaClass.ACTION_POWER_USAGE_SUMMARY) then
    Toast('Cannot display battery usage', ShortToast)
end;

procedure LaunchURL(const URL: string);
begin
  LaunchActivity(TJIntent.JavaClass.ACTION_VIEW, TJnet_Uri.JavaClass.parse(StringToJString(URL)))
end;

end.
```

So now we have 3 overloaded versions of a new `LaunchActivity` function. One takes an action string, one takes an action string and a URI, and one takes an intent object (and is called by both the other overloads once they have built up an intent object).

The battery usage routine calls the overload that just takes an action string.

The new `LaunchURL` routine takes the Delphi URL string and translates it into a Java string before passing this off to the `parse` class method of the Android `URI` class, in order to get a `URI` object returned. An intent object is then created this time with two constructor parameters: a view action and the URI.

The `ACTION_VIEW` data display action is designed to launch the sensible viewing action for many data types. There's bound to be one suitable or displaying URL, either the system browser, or maybe an additional browser you have installed, so the code doesn't bother checking the return value from `LaunchActivity` . Indeed if there is more than one available choice and the user has not previously specified one option to always use then Android will display a chooser dialog showing all the options.

It's useful to note that Android understands a variety of URI formats and will display them in appropriate applications. For example:

- Regular URLs such as http://flickr.com are handled by web browser apps
- Map URIs such as geo:37.422,-122.084?z=17 are displayed by map apps
- StreetView URIs such as google.streetview:cbll=37.422044,-122.083849&cbp=12,7.59,,0,-34.22&mz=21 are displayed by Google Maps or Google Street View
- Contact URIs like content://contacts/people or content://contacts/people/1 are displayed by the device address book
- Telephone number URIs such as tel:123456789 are displayed by the phone dialler

Sending an SMS

You can see above how easy it is to prepare a phone call for the user. You need no special permissions to ask the system dialler to show up with a number plugged in. If you wanted to actually make the call, then you'd need to use the telephony APIs and require the appropriate `CALL_PHONE` permission to use it, which is perhaps not as desirable.

It's a similar story with SMSs. There are APIs that allow you to craft an SMS message and then send it, and optionally identify if it was received and so forth. However use of these APIs requires the `SEND_SMS` permission, and it is advisable to keep permission requirements to a minimum to avoid arousing suspicion in the user who may bypass/reject your app because of them.

That said, if you are happy to require the permission (set in your Android manifest file as documented by Google here but achieved in Delphi through the project options dialog as documented by Embarcadero here) then that is just fine. Here's a unit that does what is required:

```
unit AndroidStuff;

interface

function HasPermission(const Permission: string): Boolean;
procedure SendSMS(const Number, Msg: string);

implementation
```

```
uses
  System.UITypes,
  FMX.Dialogs,
  FMX.Helpers.Android,
  Androidapi.Helpers,
  Androidapi.JNI.JavaTypes,
  Androidapi.JNI.GraphicsContentViewText,
  Androidapi.JNI.Telephony;

function HasPermission(const Permission: string): Boolean;
begin
  //Permissions listed at http://d.android.com/reference/android/Manifest.permission.html
  Result := SharedActivity.checkCallingOrSelfPermission(
    StringToJString(Permission)) =
    TJPackageManager.JavaClass.PERMISSION_GRANTED
end;

procedure SendSMS(const Number, Msg: string);
var
  SmsManager: JSmsManager;
begin
  if not HasPermission('android.permission.SEND_SMS') then
    MessageDlg('App does not have the SEND_SMS permission',
      TMsgDlgType.mtError, [TMsgDlgBtn.mbCancel], 0)
  else
  begin
    SmsManager := TJSmsManager.JavaClass.getDefault;
    SmsManager.sendTextMessage(
      StringToJString(Number),
      nil,
      StringToJString(Msg),
      nil,
      nil);
  end;
end;

end.
```

Now, back to the point about avoiding permissions. If you want to keep your permission requirements right down then the smart thinking involves launching the system SMS activity and having the user press the Send button. Launching activities requires no extra permissions primarily because the activity already exists on the device. In order to be there it is part of the system or the user has already accepted the permission requirements when installing it.

To launch the SMS app we have another routine, `CreateSMS`, to add to the LaunchActivities unit:

```
uses
  System.SysUtils, ...

...

procedure CreateSMS(const Number, Msg: string);
var
  Intent: JIntent;
  URI: Jnet_Uri;
begin
  URI := TJnet_Uri.JavaClass.parse(StringToJString(Format('smsto:%s', [Number])));
  Intent := TJIntent.JavaClass.init(TJIntent.JavaClass.ACTION_VIEW, URI);
  Intent.putExtra(StringToJString('sms_body'), StringToJString(Msg));
  LaunchActivity(Intent);
end;
```

A couple of things to note here:

1. Launching the SMS activity is done through yet another form of URI launch
2. As well as the view action and the URI data, this code also inserts an extra data item named `sms_body` into the intent to represent the content of the SMS message

Sending an email

With SMSs above we saw the intent starting to be loaded up with additional parameters required by the target activity. For launching an email we set even more parameters. Here's another routine for the LaunchActivities unit:

```
uses
  Androidapi.JNIBridge, ...

...

procedure CreateEmail(const Recipient, Subject, Content: string); overload;
var
  Intent: JIntent;
```

```
begin
  Intent := TJIntent.JavaClass.init(TJIntent.JavaClass.ACTION_SEND);
  Intent.putExtra(TJIntent.JavaClass.EXTRA_EMAIL, StringToJString(Recipient));
  Intent.putExtra(TJIntent.JavaClass.EXTRA_SUBJECT, StringToJString(Subject));
  Intent.putExtra(TJIntent.JavaClass.EXTRA_TEXT, StringToJString(Content));
  // Intent.setType(StringToJString('plain/text'));
  Intent.setType(StringToJString('message/rfc822'));
  // LaunchActivity(Intent);
  LaunchActivity(TJIntent.JavaClass.createChooser(Intent, StrToJCharSequence('Which email app?')));
end;
```

This sends an email to a single recipient with a specified subject and content, adding these data items in as extra string data. You can see that you could choose to send a plain text email but an RFC 822 format email is specified to try to ensure that only email apps will pick up the intent.

As well as EXTRA_TEXT you can also send HTML format content by also specifying EXTRA_HTML_TEXT.

The code doesn't wait to see if the system will pop up a chooser dialog in the case that there are multiple apps that can handle the intent. Instead it directly invokes the chooser dialog for our intent using a custom prompt string.

If you want to support multiple email recipients you have to get a bit clever with setting up Java arrays in Delphi. In this case you need to put the recipients in a Java string array. The following additional overload has some code to show you how this looks:

```
procedure CreateEmail(const Recipients: array of string; const Subject, Content: string); overload;
var
  Intent: JIntent;
  I: Integer;
  JRecipients: TJavaObjectArray<JString>;
begin
  Intent := TJIntent.JavaClass.init(TJIntent.JavaClass.ACTION_SEND);
  JRecipients := TJavaObjectArray<JString>.Create(Length(Recipients));
  for I := 0 to Pred(Length(Recipients)) do
    JRecipients.Items[I] := StringToJString(Recipients[I]);
  Intent.putExtra(TJIntent.JavaClass.EXTRA_EMAIL, JRecipients);
  Intent.putExtra(TJIntent.JavaClass.EXTRA_SUBJECT, StringToJString(Subject));
  Intent.putExtra(TJIntent.JavaClass.EXTRA_TEXT, StringToJString(Content));
  Intent.setType(StringToJString('message/rfc822'));
  LaunchActivity(TJIntent.JavaClass.createChooser(Intent, StrToJCharSequence('Which email app?')));
end;
```

As well as EXTRA_EMAIL for the recipient(s) you can also use EXTRA_CC and EXTRA_BCC.

If you want to take it a step further and send an attachment this is also possible but I'm guessing it's a little less common. There's a post on StackOverflow that shows some code to do this by getting a URI for the file location and adding that URI into an Android Parcelable object and then adding that to the intent as an EXTRA_STREAM item (though other Android code I've seen puts the URI directly into the intent, as per the documentation).

Scanning a bar code

Something that crops up regularly in discussions around mobile app building is how to incorporate barcode scanning behaviour.

Many Android users have installed the open source ZXing (Zebra Crossing) barcode scanner app, which is designed to make a scan activity available. It is quite straightforward to invoke the activity if you've checked the ZXing "documentation" and therefore know the action string and optionally some data strings:

```
//For more info see https://github.com/zxing/zxing/wiki/Scanning-Via-Intent
procedure LaunchQRScanner;
var
  Intent: JIntent;
begin
  Intent := TJIntent.JavaClass.init(StringToJString('com.google.zxing.client.android.SCAN'));
  Intent.setPackage(StringToJString('com.google.zxing.client.android'));
  // If you want to target QR codes
  //Intent.putExtra(StringToJString('SCAN_MODE'), StringToJString('QR_CODE_MODE'));
  if not LaunchActivity(Intent) then
    Toast('Cannot display QR scanner', ShortToast);
end;
```

Communication from the launched activity

However we now have something of a problem on our hands. Having launched the activity, how do we get information back from it to identify what barcode, if any, was scanned?

Before looking at the standard Android approach to this problem, let's have a small digression to a quite popular solution that ZXing makes possible. ZXing scanner puts the scanned barcode data on the clipboard when scanned, so you simply need to check the device's clipboard and if a new value appears there then that's likely to be what was scanned. This approach was documented for Delphi XE5 by John Whitham and for documented for Delphi XE6 by Steffen Nyeland.

And now back to the standard Android approach.... Earlier code snippets that launch an activity do so using the current activity's `startActivity()` method, which launches an activity and forgets all about it. More interesting for our current purpose is `startActivityForResult()`, which is an activity method that launches an activity with a specified request code and then waits for the activity to exit and return a result code. When the launched activity does exit your activity's `onActivityResult()` method is called with your request code and the activity's result code.

To prepare for this new API scheme here is some modified activity launching code:

```
function LaunchActivityForResult(const Intent: JIntent; RequestCode: Integer): Boolean;
var
  ResolveInfo: JResolveInfo;
begin
  ResolveInfo := SharedActivity.getPackageManager.resolveActivity(Intent, 0);
  Result := ResolveInfo <> nil;
  if Result then
    SharedActivity.startActivityForResult(Intent, RequestCode);
end;

//For more info see https://github.com/zxing/zxing/wiki/Scanning-Via-Intent
procedure LaunchQRScanner(RequestCode: Integer);
var
  Intent: JIntent;
begin
  Intent := TJIntent.JavaClass.init(StringToJString('com.google.zxing.client.android.SCAN'));
  Intent.setPackage(StringToJString('com.google.zxing.client.android'));
  // If you want to target QR codes
  //Intent.putExtra(StringToJString('SCAN_MODE'), StringToJString('QR_CODE_MODE'));
  if not LaunchActivityForResult(Intent, RequestCode) then
    Toast('Cannot display QR scanner', ShortToast);
end;
```

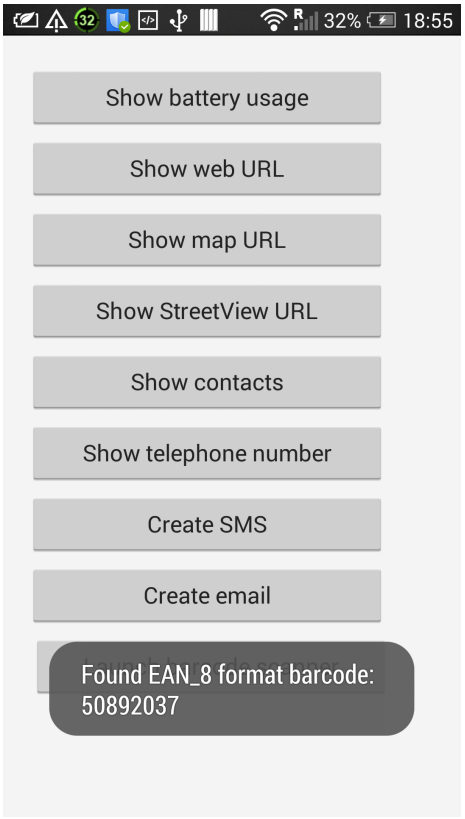
In order to launch an activity and pick up its result we *could* potentially implement `onActivityResult()` for Delphi's underlying activity, which would seem the obvious way forward, given that's how Android expects us to operate.

In the previous version of this article I showed how to do exactly that with Delphi XE5's Android support (it is quite long-winded and hairy) but as of XE6 there is no need to get our hands anywhere near as dirty. In fact `onActivityResult()` is already hooked by Delphi XE6 and activity results are now sent into the RTL cross-platorm messaging system. We use message-receiving code like this to pick up launched activity results.

```
uses
  System.Messaging,
  ...

type
  TMainForm = class(TForm)
    ...
  private
    const ScanRequestCode = 0;
    var FMessageSubscriptionID: Integer;
    procedure HandleActivityMessage(const Sender: TObject; const M: TMessage);
    function OnActivityResult(RequestCode, ResultCode: Integer; Data: JIntent): Boolean;
    ...
  end;
...
uses
  FMX.Platform.Android,
  Androidapi.Helpers,
  Androidapi.JNI.App,
  Androidapi.JNI.Toast,
  LaunchActivities,
  ...
procedure TMainForm.BarcodeScannerButtonClick(Sender: TObject);
begin
  FMessageSubscriptionID := TMessageManager.DefaultManager.SubscribeToMessage(TMessageResultNotification,
    HandleActivityMessage);
  LaunchQRScanner(ScanRequestCode);
end;

procedure TMainForm.HandleActivityMessage(const Sender: TObject; const M: TMessage);
begin
  if M is TMessageResultNotification then
    OnActivityResult(TMessageResultNotification(M).RequestCode, TMessageResultNotification(M).ResultCode,
      TMessageResultNotification(M).Value);
end;

function TMainForm.OnActivityResult(RequestCode, ResultCode: Integer; Data: JIntent): Boolean;
var
  ScanContent, ScanFormat: string;
begin
  Result := False;
```

```
  TMessageManager.DefaultManager.Unsubscribe(TMessageResultNotification, FMessageSubscriptionID);
  FMessageSubscriptionID := 0;

  // For more info see https://github.com/zxing/zxing/wiki/Scanning-Via-Intent
  if RequestCode = ScanRequestCode then
  begin
    if ResultCode = TJActivity.JavaClass.RESULT_OK then
    begin
      if Assigned(Data) then
      begin
        ScanContent := JStringToString(Data.getStringExtra(StringToJString('SCAN_RESULT')));
        ScanFormat := JStringToString(Data.getStringExtra(StringToJString('SCAN_RESULT_FORMAT')));
        Toast(Format('Found %s format barcode:'#10'%s', [ScanFormat, ScanContent]), LongToast);
      end;
    end
    else if ResultCode = TJActivity.JavaClass.RESULT_CANCELED then
    begin
      Toast('You cancelled the scan', ShortToast);
    end;
    Result := True;
  end;
end;
```

You can see that we use `TMessageManager.DefaultManager` to subscribe to `TMessageResultNotification` notifications passing in `HandleActivityMessage` as the handler before launching the activity, where `TMessageResultNotification` is actually a class inheriting from a common `TMessage` base notification message class. When the activity exits and returns a result the message notification passes all the information to the `HandleActivityMessage` method as a `TMessage` and we cast to type `TMessageResultNotification` to access the information we need. The parameters are then passed along to `OnActivityResult`, which unsubscribes and processes the information returned from the activity.



Practical limitations of Delphi XE6's Android support

Delphi apps on Android are native code libraries starting at a few megabytes in size in size, packaged into an .apk file along with some compiled Java, a few resources and an Android manifest. The native ARMv7 code can communicate with the Java world using the *Java Bridge* (or JNI Bridge as it is sometimes called), but this relies on Delphi representations of the Java classes being present.

This is quite like Delphi for Win32 talking to Windows APIs - many APIs are pre-declared for you and some you need to create the declarations for yourself if they aren't catered for by the Delphi RTL. Similarly Delphi XE5 has many Android classes represented, but also many more that are not, so there is good scope for being required to get on top of Delphi's Java Bridge to call into the Android API.

FireMonkey provides much functionality necessary for business applications needing to access data and display it to the user, but some common aspects of the Android OS are not 'wrapped up' in FireMonkey classes.

At this point you can call upon the various Android APIs that have been provided, or craft new Java Bridge APIs if need be (as we did for the toast API earlier).

In cases where this isn't simply a case of calling into more APIs whose declarations need to declared with Java Bridge interfaces this poses a problem. Some examples of areas of the Android OS that are not wrapped up in XE5 and are difficult to incorporate are:

- App splash screens
- Android services
- Broadcast receivers
- Bluetooth support (typically requires a broadcast receiver)
- USB device support (typically requires a broadcast receiver)

In Delphi XE6 these things and more require additional Java code to implement. Then some light hacking is needed to make use of this Java code in a Delphi package, which tends to get in the IDE's way, so IDE building/installing of the application becomes interesting, and debugging becomes next to impossible. These consequences tend to force you into managing a pair of synchronised projects:

- one without the things that require the hacking, which can be built and run from the IDE
- one with the extra items that require the hacking, which can be built and installed in whatever way works, such as with a command-line script, and then launched manually from the Android device

Fortunately for launched activity result access we no longer need to resort to such low level jiggery-pokery in Delphi XE6, unlike the predecessor, XE5.

However, it's not all problem-free and rose-scented coding in Delphi XE6 either. There are still some issues with the code you put in an application that is invoked from an Android callback, as we have done here. You have seen that we can be successfully notified of an activity finishing and returning a result code, and we've emitted a toast message and all was well.

Or so it seemed.

Actually I deliberately chose to use a toast rather than a message box (as in `ShowMessage`, `MessageDlg` et al) for the simple reason that I didn't want my demo app to hang.

The thing is that for some odd reason, which has been consistent through the release of both Delphi XE5 and Delphi XE6, if FMX code is called from some Android callback routine and that FMX code invokes code that has a nested message loop (as in the sort of thing that `ShowMessage` and `MessageDlg` do) then you won't see the message box. Instead the app will turn into a black screen. Eventually after some delay the OS will step in with an ANR message (Application is Not Responding), offering to close the offending app.

This is not good.

You *can* work around the issue using a custom message box routine written in a Java source file as a method in a class inherited from the native activity class that underlies each Delphi application, surfacing it via JNI calls. However now that Delphi XE6 permits activity result acquisition without going to such painful lengths, I'm not going to do so just in order to display a message box. A toast message will do fine. If you really want to display a message box, please see the process as illustrated at length in the Delphi XE5 article.

I gather the issue is still on the R&D radar, so hopefully will get fixed at some point.

Conclusion

You can easily extend the functionality of your application by taking advantage of pre-installed system activities, and you can also make use of custom activities if you check whether they are available. This is a convenient model and goes some way to bring us the component-building model of bolting together large lumps of code into an application that was promised in the 90s.

You can also receive feedback from launched activities in a Delphi Android application, as long as you are careful what sort of code you run at that point.

So there are some wrinkles in how this all comes together, but Delphi XE6 improves upon XE5, Embarcadero's first stab at Android development support. Hopefully some of these wrinkles will smooth out over time and things will continue to improve in future product updates and releases.

Go back to the top of this page
