Dave's Development Blog
Software Development using Borland / Codegear / Embarcadero RAD Studio

Browsing Package Information

By David | May 14, 2016 4 Comment

Overview

After creating the OTA Interface Index to both cross reference the blog posts I've done against the `ToolsAPI.pas` file and understand how much of that file I've covered its apparent that I've only just scratched the surface. So I spent half an hour wandering through the `ToolsAPI.pas` file from the Berlin release looking for topics to write about. There are currently four in the mix of which this is the first but one of the things I want to do is make sure that I provide practical examples that you can use and extend rather than a desktop review of an interface.

So in this blog I'm going to cover the interfaces that allow you to know what packages are loaded into the IDE and what information about these packages you can obtain. It will take the form of a modal package viewer a bit like my Delphi IDE Explorer.

About Boxes and Splash Screens Revisited

After reading a post on the Delphi Developer G+ groups it set the cogs turning on something I read and made me go back and examine things. In the post a part of the comments from the splash screen interface was quoted in which it talked about bitmaps that are 24×24. I thought, hold on, no they're not they're 48×48. So grumpy here went off to investigate. I turns out that in RAD Studio 2005/6, the first to support splash screens had bitmaps that were 48×48 however with RAD Studio 2007 they were changed to 24×24. It explains why my splash screen icons never looked quite right in later IDEs. I my defence (your honour) I did code the original splash screen code in 2006 as I didn't have 2007. The next IDE I got was 2009 and I didn't think to check that this had changed. I suppose the moral of the story is simply check all your existing interfaces for changes.

Anyway I now create 2 icons for my splash screens: one 24×24 for the splash screen for 2007 and above and a 48×48 for the about dialogue and the 2005/6 splash screen (see the code below).

```
Constructor TDGHPackageViewerWizard.Create;

Begin
  {$IFDEF D2005}
  FAboutPluginIndex := -1;
  BuildNumber(FVersionInfo);
  FSplashScreen48 := LoadBitmap(hInstance, 'SplashScreen48');
  With FVersionInfo Do
    FAboutPluginIndex := (BorlandIDEServices As IOTAAboutBoxServices).AddPluginInfo(
      Format(strSplashScreenName, [iMajor, iMinor, Copy(strRevision, iBugFix + 1, 1),
        Application.Title]),
      'An IDE Expert to allow you to browse the loaled packages in the IDE.',
      FSplashScreen48,
      False,
      Format(strSplashScreenBuild, [iMajor, iMinor, iBugfix, iBuild]),
      Format('SKU Build %d.%d.%d.%d', [iMajor, iMinor, iBugfix, iBuild]));
  FSplashScreen24 := LoadBitmap(hInstance, 'SplashScreen24');
  With FVersionInfo Do
    (SplashScreenServices As IOTASplashScreenServices).AddPluginBitmap(
      Format(strSplashScreenName, [iMajor, iMinor, Copy(strRevision, iBugFix + 1, 1),
        Application.Title]),
      {$IFDEF D2007}
```

```
      FSplashScreen24, // 2007 and above
      {$ELSE}
      FSplashScreen48, // 2006 ONLY
      {$ENDIF}
      False,
      Format(strSplashScreenBuild, [iMajor, iMinor, iBugfix, iBuild]));
  {$ENDIF}
End;
```

You will also notice in the code above that I've changes the code that generates the splash screen / about box title to use the `Application.Title` `$IFDEFs` as I've shown before. This works well for the splash and about box at startup as there are no projects loaded into the IDE however if you install a package while the IDE is open then the about box title will contain the project name as well.

For more information on splash screen icons you might like to read David Millington's blog article Antialiased images on the Delphi splash screen.

Package Interfaces

There are a number of interfaces for handling packages as below (these are sourced from the RAD Studio XE10.1 Berlin release). The number in brackets after the interface refers to the package version that was appended when the interface was extended in later version of the IDE.

IOTAPackageServices[140]

This interface has been around in RAD Studio since Delphi 5 at least (I don't have any earlier versions).

```
IOTAPackageServices140 = interface(IUnknown)
  ['{26EB0E4D-F97B-11D1-AB27-00C04FB16FB3}']
  function GetPackageCount: Integer;
  function GetPackageName(Index: Integer): string;
  function GetComponentCount(PkgIndex: Integer): Integer;
  function GetComponentName(PkgIndex, CompIndex: Integer): string;
  property PackageCount: Integer read GetPackageCount;
  property PackageNames[Index: Integer]: string read GetPackageName;
  property ComponentCount[PkgIndex: Integer]: Integer read GetComponentCount;
  property ComponentNames[PkgIndex, CompIndex: Integer]: string read GetComponentName;
end;
```

In general it allows you to iterate the packages and get their names and iterate the components in a package and also get their names. I'll describe the properties below as the methods are simply getter methods for the properties.

PackageCount

This method returns the number of packages that are currently loaded in the IDE.

PackageName

This property takes a zero based index and returns the name (filename and extension, no path) for the indexed package.

ComponentCount

This property returns the number of components in the package for a given package index (zero based).

ComponentNames

This property returns the type name (`TButton` for example) of the indexed package and component (both zero based).

IOTAPackageInfo

This interface was introduced in RAD Studio XE and provides access to a number of properties for the package.

```
IOTAPackageInfo = interface(IUnknown)
  ['{F41DB233-500B-4B0D-93A0-9072E10EE069}']
  function GetDescription: string;
  function GetFileName: string;
  function GetName: string;
  function GetSymbolFileName: string;
```

```
  procedure GetContainsList(List: TStrings);
  procedure GetRequiresList(List: TStrings);
  procedure GetImplicitList(List: TStrings);
  procedure GetRequiredByList(List: TStrings);
  function GetRuntimeOnly: Boolean;
  function GetDesigntimeOnly: Boolean;
  function GetIDEPackage: Boolean;
  function GetProducer: TOTAPackageProducer;
  function GetConsumer: TOTAPackageConsumer;
  function GetLoaded: Boolean;
  procedure SetLoaded(Value: Boolean);
  property FileName: string read GetFileName;
  property Name: string read GetName;
  property RuntimeOnly: Boolean read GetRuntimeOnly;
  property DesigntimeOnly: Boolean read GetDesigntimeOnly;
  property IDEPackage: Boolean read GetIDEPackage;
  property Loaded: Boolean read GetLoaded write SetLoaded;
  property Description: string read GetDescription;
  property Producer: TOTAPackageProducer read GetProducer;
  property Consumer: TOTAPackageConsumer read GetConsumer;
  property SymbolFileName: string read GetSymbolFileName;
end;
```

Below are explanations for each property and methods not associated with the properties.

FileName

This property returns the full filename including path, filename and extension for the given package.

Name

This property returns the name of the given package.

RuntimeOnly

This property returns whether the given package is a run-time only package.

DesigntimeOnly

This property returns whether the given package is a design-time only package.

IDEPackage

This property returns whether the given package is a built-in IDE package.

Loaded

This property gets or set whether the given package is loaded. The comment that accompanies this property states that setting the loaded state to `True` only has an effect for a package that have been cached or one that has been previously unloaded via a call to `Loaded := False`. Setting the loaded state to `False` will cause the package to be unloaded from memory but will not remove it from the package list (it will be loaded the next time the IDE loads). To unload a package and remove it from the list you should use `IOTAPackageServices.UninstallPackage`.

Description

This is the description that is embedded in the package using the `{$DESCRIPTION 'Xxxx'}` directive

Producer

This property returns an enumerate to describe which tool created the package, Delphi, C++ Builder or Unknown.

Consumer

This property returns an enumerate to describe which tool can use this package: Delphi, C++ Builder, Both or Unknown.

SymbolFileName

This property seems to return the name of the package. The comment in the code states that it returns the base name of the symbol file associated with the package. For Delphi-built packages this is typically the .dcp file name. For C++-built packages, this is typically

the .bpi file name. This may differ from the package name if the package has a LIBPREFIX, LIBSUFFIX or LIBVERSION defined.

GetContainsList, GetRequiresList, GetImplicitList and GetRequiredByList

These method all work in the same way. You need to pass to the method a valid `TStringList` or descendant and the method will fill the string list with the units the package contains, requires, implicitly loads or is required by respectively.

IOTAPackageServices[210]

This interface was introduced in RAD Studio XE.

```
IOTAPackageServices210 = interface(IOTAPackageServices140)
  ['{2C96711A-267A-4024-9C54-B11FCC596A6F}']
  function InstallPackage(const PackageName: string): Boolean;
  function UninstallPackage(const PackageName: string): Boolean;
  function GetPackage(Index: Integer): IOTAPackageInfo;
  property Package[Index: Integer]: IOTAPackageInfo read GetPackage;
end;
```

It provides the ability Install and Uninstall packages as well as get detailed information about each package.

InstallPackage

Although I haven't used this method in my example here I assume that the `PackageName` parameter is the fully qualified path, filename and extension for the package to be installed otherwise I can't see how the IDE could know what to install.

UninstallPackage

Although I haven't used this method in my example where I assume that the `PackageName` parameter is the fully quanlitied path, filename and extension for the package to be uninstalled as above.

Package

This property returns an `IOTAPackageInfo` interface for the zero indexed package which as described above and provides access to more detailed information about the package (see `IOTAPackageInfo`).

IOTAPAckageServices

This interface was introduced in RAD Studio XE10 Seattle.

```
IOTAPAckageServices = interface(IOTAPackageServices210)
  ['{1E8AB2DA-CC56-4FA5-851A-9CDC957D1D65}']
  procedure RegisterPackageNotifier(Proc: TPackageNotifier);
  procedure UnregisterPackageNotifier(Proc: TPackageNotifier);
end;
```

It provides the ablity to install package notifiers to get notification of when they are Installing, Installed and Uninstalled.

Both methods take a procedure with the following declaraton.

```
TPackageOp = (poInstalled, poUninstalling, poUninstalled);
TPackageNotifier = procedure (const PackageName: string; PackageOp: TPackageOp) of object;
```

RegisterPackageNotifier

This method installs your call back procedure which will be called when packages are installing, installed and uninstalled.

UnregisterPackageNotifier

This method uninstalls your call back procedure.

I will cover these methods and the call back procedure in more detail in another post (i.e. one of the four mentioned above) where I'll create a dockable window to log ALL notifications from the IDE.

If you have an earlier version of RAD Studio you may find the interface name slightly different (without the package numbers) that is due to the latest interface always being without a number at the end (99% of the time) and with previous interfaces containing the package number for the version of RAD Studio in which the extensions were introduced.

Implementation

So, now for the implementation. This is relatively straight forward in that I use a `For` loop to iterate over the loaded packages and add their components and properties to a tree view structure for ease of viewing.

I've broken down the single procedure into 4 parts to help explain the code. In this first part I iterate through the packages using `PackageCount` and `PackageName` and create nodes in the tree for each package using the package name. Then for each package, if there are components (`ComponentCount > 0`) I add another child node called `Components` and then add a sub-sub-node for each named component.

```
Procedure TfrmDGHPackageViewer.IteratePackages;

Var
  PS: IOTAPAckageServices;
  iPackage: Integer;
  P, N: TTreeNode;
  iComponent: Integer;
  sl: TStringList;
  frm : TfrmDGHPackageViewerProgress;

Begin
  tvPackages.Items.BeginUpdate;
  Try
    PS := (BorlandIDEServices As IOTAPAckageServices);
    frm := TfrmDGHPackageViewerProgress.Create(Application.MainForm);
    Try
      frm.ShowProgress(PS.PackageCount);
      For iPackage := 0 To PS.PackageCount - 1 Do
        Begin
          P := tvPackages.Items.AddChild(Nil, PS.PackageNames[iPackage]);
          if PS.ComponentCount[iPackage] > 0 then
            Begin
              N := tvPackages.Items.AddChild(P, 'Components');
              for iComponent := 0 to PS.ComponentCount[iPackage] - 1 do
                tvPackages.Items.AddChild(N, Ps.ComponentNames[iPackage, iComponent]);
            End;
          ...
          frm.UpdateProgress(Succ(iPackage));
        End;
        frm.HideProgress;
      Finally
        frm.Free;
      End;
  Finally
    tvPackages.Items.EndUpdate;
  End;
End;
```

This second portion of code (which sits in the above code where the ellipsis is) is only avaiable for RAD Studio XE and above, hence the `$IFDEF`s. It obtains an `IOTAPackageInfo` interface for the current iterated package and adds various information to a `Properies` node under the package. Please refer to the above information on the `IOTAPackageInfo` interface for more details on the information provided.

```
{$IFDEF DXE00}
N := tvPackages.Items.AddChild(P, 'Properties');
tvPackages.Items.AddChild(N, Format('FileName: %s', [PS.Package[iPackage].FileName]));
tvPackages.Items.AddChild(N, Format('Name: %s', [PS.Package[iPackage].Name]));
tvPackages.Items.AddChild(N, Format('Run-Time Only: %s', [strBoolean[PS.Package[iPackage].RuntimeOnly]]));
tvPackages.Items.AddChild(N, Format('Design-Time Only: %s',
[strBoolean[PS.Package[iPackage].DesigntimeOnly]]));
tvPackages.Items.AddChild(N, Format('IDE Package: %s', [strBoolean[PS.Package[iPackage].IDEPackage]]));
tvPackages.Items.AddChild(N, Format('Loaded: %s', [strBoolean[PS.Package[iPackage].Loaded]]));
tvPackages.Items.AddChild(N, Format('Description: %s', [PS.Package[iPackage].Description]));
tvPackages.Items.AddChild(N, Format('SymbolFileName: %s', [PS.Package[iPackage].SymbolFileName]));
```

```
tvPackages.Items.AddChild(N, Format('Producer : %s', [strProducer[PS.Package[iPackage].Producer]]));
tvPackages.Items.AddChild(N, Format('Consumer : %s', [strConsumer[PS.Package[iPackage].Consumer]]));
sl := TstringList.Create;
Try
  PS.Package[iPackage].GetContainsList(sl);
  AddStringList(N, sl, 'Contains');
  PS.Package[iPackage].GetRequiresList(sl);
  AddStringList(N, sl, 'Requires');
  PS.Package[iPackage].GetImplicitList(sl);
  AddStringList(N, sl, 'Implicit');
  PS.Package[iPackage].GetRequiredByList(sl);
  AddStringList(N, sl, 'Required By');
Finally
  sl.Free;
End;
{$ENDIF}
```

This third portion of code is a local support procedure to the main method which adds the string list of references to various sub nodes to the properties.

```
{$IFDEF DXE00}
Procedure AddStringList(N : TTreeNode; sl : TStringList; strListName : String);

Var
  i: Integer;
  M: TTreeNode;

Begin
  M := tvPackages.Items.AddChild(N, strListName);
  for i := 0 to sl.Count - 1 do
  tvPackages.Items.AddChild(M, sl[i]);
End;
{$ENDIF}
```

Finally the code below provides a couple of constant string arrays for converting enumerates to strings for display in the tree view.

```
{$IFDEF DXE00}
Const
strBoolean : Array[False..True] Of String = ('False', 'True');
strProducer : Array[Low(TOTAPackageProducer)..High(TOTAPackageProducer)] Of String = (
'ppOTAUnknown', 'ppOTADelphi', 'ppOTABCB');
strConsumer : Array[Low(TOTAPackageConsumer)..High(TOTAPackageConsumer)] Of String = (
'pcOTAUnknown', 'pcOTADelphi', 'pcOTABCB', 'pcOTABoth');
{$ENDIF}
```

While testing the final code and trying to get a screen shot of the dialogue I found myself searching for packages with components in that should have had components and wondered why (I have a package of my own components for my applications). It took me a while to work it out (it was late at night) but the components will only be available when the package is loaded, i.e a package that is registered but not loaded (cached) will not provide access to its components. So the code you download will be a little different from above as I've added a button at the bottom of the dialogue to toggle the `Loaded` property of the package (only available in RAD Studio XE and above). When the package is loaded the components will be searched for and added to the tree. I also thought that greying out the unloaded packages would be a useful hint to their state.

Once installed the package viewer can be access from the Help menu of the IDE (i.e. its a `IOTAMenuWizard`). For more information on compiling and loading this package please refer to the article Compiling and Installing my experts and wizards….

The code for this blog can be downloaded from the link below or from the separate page for this IDE add-in.

- DGH Package Viewer 1.0 2006 to XE10.1 or Package Viewer.

I hope this has been useful and there will be more soon.

No related posts.

Category: Open Tools API Tags: IOTAPackageInfo, IOTAPackageServices, IOTAPackageServices140, IOTAPackageServices210

4 thoughts on “Browsing Package Information”

dummzeuch
May 14, 2016

You've got a typo “eariler” when mentioning where the package info toolsapi started.

David Post author
May 14, 2016

Drat ☹ Editing now…

Nicholas Ring
May 15, 2016

I can see another “drat” coming on: Just scrolling through, I noticed the following typo:

UnregisterPackageNotifier
This method uninstalls yor call back procedure.?

Having said that, thanks for all your articles!

David Post author
May 15, 2016

Thanks Nicolas, now corrected.

Comments are closed.