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

# Chapter 17: Options Page(s) inside the IDE's Options Dlg

By David | April 3, 2016 3 Comments

## Overview

The topic for this blogs is putting your Options/Configuration pages for your experts/wizards inside the IDE's options dialogue. I actually didn't know you could do this until I saw David Millington's IDE add-ins and thought, "I wonder how he did that?". As it turns out this functionality has been around since RAD Studio XE, so I will in this blog go through how to do this with code that will work before RAD Studio XE and after, all using the same interface code.

I'll also cover a number of issues that arise from creating an IDE expert/wizard in a later version of RAD Studio when trying to back-port them to earlier versions.

## Things to trip you up

### To Namespace or NOT to Namespace…

All but one of my IDE experts/wizards have been build using old versions of RAD Studio and have been migrated to the newer IDEs with very few changes required and only 1 re-implementation due to a depreciated interface. As a consequence of this I've not had issues with namespaces before as the code didn't contain them however with my latest add-in for providing F1 context help in the IDE when the IDE doesn't have any help context I've actually been back-porting the application to older IDE versions. It was originally built with XE7 and moved to XE10 without any issues (BTW I call RAD Studio 10 Seattle XE10). I then started to move it backwards to XE2 and that went without a hitch but RAD Studio XE was a show stopper for the compiler (BTW I'm using a special batch files to build all the different versions with the command line compiler DCC32). At first I thought it was a problem with the installation as I had removed C++ from the RAD Studio XE installation to make some space for XE10 Enterprise but after fixing the registration problem I got the same error in the IDE. When I looked at the `Lib\Win32\Release` directory it became apparent that `WinAPI.windows.pas` was now `windows.pas`, i.e. the namespace `WinAPI` had disappeared.

So that solution is to remove ALL namespaces from all the code files and make sure the IDEs have the namespace search criteria instead.

### Properties

One of the other issues you will get if you design a form or frame in a more modern IDE and then try to compile them in an older IDE is that they will compile BUT when they are run the IDE will crash when loading the DFM file due to the presence of properties in the DFM file that do not exist in that version of the IDE.

Below are a list of properties that I usually look for and remove from forms when designing an IDE expert/wizard for backward compatibility. Unfortunately I have not done this exercise for this expert/wizard and I'll have to decide how far back I go with the compatibility of this project.

- `ExplicitLeft`: Introduced in 2005/2006;
- `ExplicitTop`: Introduced in 2005/2006;
- `ExplicitWidth`: Introduced in 2005/2006;
- `ExplicitHeight`: Introduced in 2005/2006;
- `DoubleBuffered`: Introduced in 2005/2006;
- `Margins.*`: Introduced in 2005/2006;
- `Padding.*`: Introduced in 2005/2006.

Note: Therefore this generally affects going back to Delphi 7 and before.

By extension it should also be obvious that newer components like `TGridPanel` are only supported by a number of the newer IDEs and again this is something I have used and will need to decide whether to replace the implementation with panels and code or curtail the compatibility. Note: these will be picked up by the compiler as their definitions will be in the Published section of the form classes.

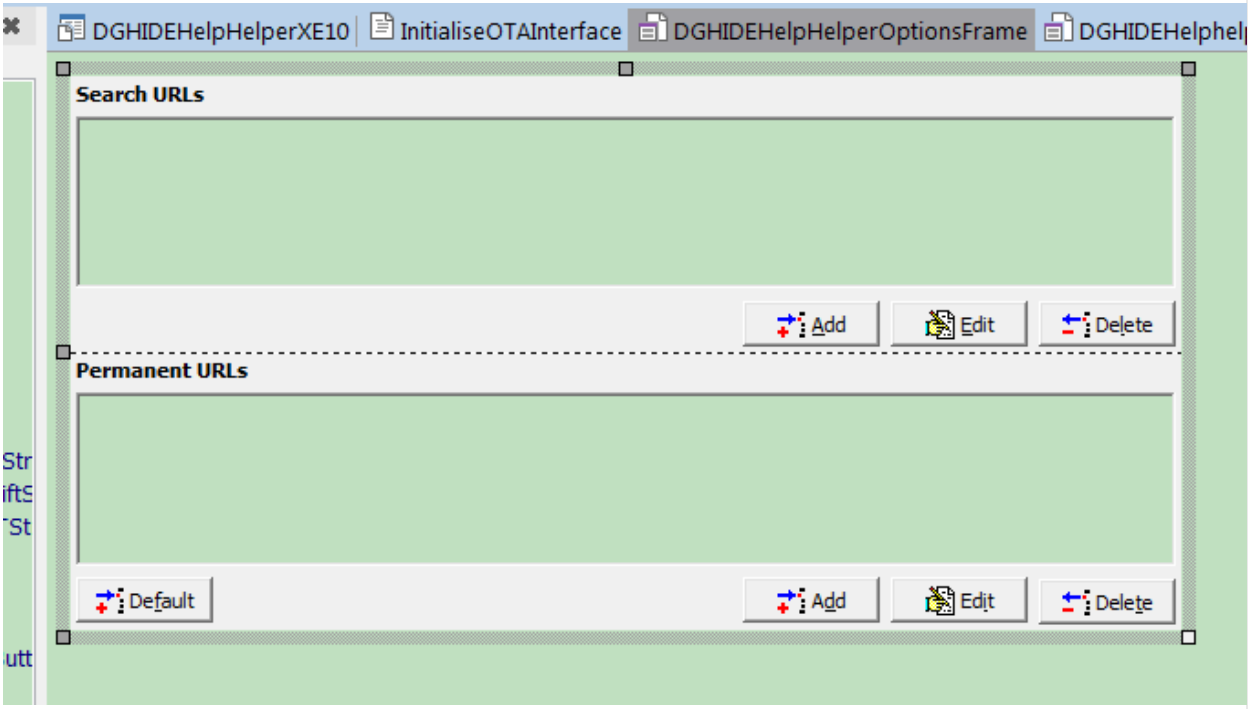
## Creating your options page(s)

The following sections walk you through how to design your Options page(s) for the IDEs such that it can be used in IDEs that support the functionality to host your options page and those that don't with a single set of code.

### Framing your options

In order for the IDE to host your options page in the IDE's options dialogue you must define a frame (not a form) which contains your Options interface and provide a reference to the class to the IDE. The IDE will then create this for you and you are given a number of interface methods in which to set the frame's information and retrieve this information along with a methods to check the validity of the data.

Below is a screen shot of my options frame in the IDE. It contains all the controls for manipulating the options BUT NOT the OK, Cancel or Help buttons.




I'm not going to go through the code in the form for managing the data, you can have a look at that in the current version of the expert/wizard however I will go through how I handle settings and retrieving the data from the frame. A very long time ago when I first read about object oriented programme in Turbo Pascal 5.5 one thing suck in my mind – you should not directly access the attributes of the object (fields) from outside the object, you should always provide a method to do this. Back in TP 5.5 there were no scope keywords like `Private` or `Public`, everything was public therefore you could actually access anything. In the modern Object Pascal language I always use `Strict Private` and `Strict Protected` to enforce this idiom.

So what I'm trying to get across is that I don't believe you should access the controls of the frame from outside the frame but rather provide one or more methods to do this. Therefore I've implemented 2 public methods `InitialiseFrame` and `FinaliseFrame` to set the frame interface and retrieve the information from the frame respectively. Below are the methods with a quick explanation of their use.

```
Procedure TfmIDEHelpHelperOptions.InitialiseFrame(slSearchURLs,
  slPermanentURLs: TStringList; iSearchURL: Integer);

Begin
  FClickIndex := -1;
  lbxSearchURLsClick(Nil);
  lbxPermanentURLsClick(Nil);
  lbxSearchURLs.Items.Assign(slSearchURLs);
  lbxPermanentURLs.Items.Assign(slPermanentURLs);
  If (iSearchURL > -1) And (iSearchURL <= lbxSearchURLs.Items.Count - 1) Then
    lbxSearchURLs.Checked[iSearchURL] := True;
```

```
End;
```

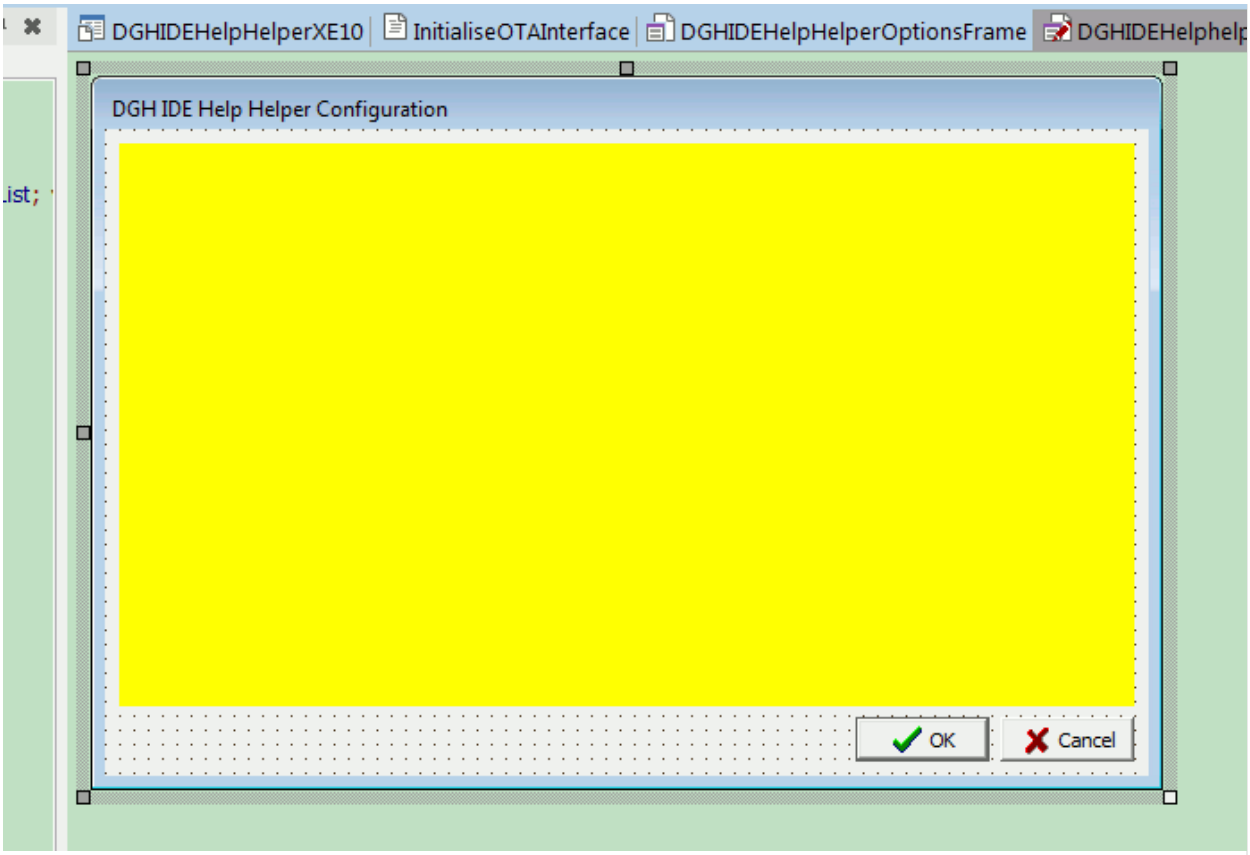
This method runs some event handlers to initialise the various buttons to their initial state (disabled) and assigns the various passed string lists to the listboxes and finally checks the currently in use search URL.

```
Procedure TfmIDEHelpHelperOptions.FinaliseFrame(slSearchURLs,
  slPermanentURLs: TStringList; Var iSearchURL: Integer);

Var
  i: Integer;

Begin
  slSearchURLs.Assign(lbxSearchURLs.Items);
  slPermanentURLs.Assign(lbxPermanentURLs.Items);
  iSearchURL := -1;
  For i := 0 To lbxSearchURLs.Items.Count - 1 Do
    If lbxSearchURLs.Checked[i] Then
      Begin
        iSearchURL := i;
        Break;
      End;
End;
```

This is a little more straight forward in that it gets the string lists from the listboxes and the selected search URL.

## Maintaining an existing interface dialogue

So now we have a frame for our options logic, for a traditional dialogue interface all I've done is create a form with an OK button, a Cancel button and a panel area (temporarily highlighted yellow below) to host the frame.




One of the things I use very heavily when creating my forms and dialogues in my applications is to use Anchors so that when the form / frame interface changes size the controls size / move with the form / frame. The IDE when it creates your frame, will make the frame fill the whole right hand side of the options dialogue therefore a dynamic interface that sizes will better suit the IDE's options dialogue.

Anyway, back to a standard dialogue. First the FormCreate event handler creates an instance of the frame and inserts it into the panel control as below:

```
Procedure TfrmDGHIDEHelphelperConfig.FormCreate(Sender: TObject);

Begin
  FFrame := TfmIDEHelpHelperOptions.Create(Self);
  FFrame.Parent := pnlFrame;
  FFrame.Align := alClient;
End;
```

Then the second part of this is the implementation of an `Execute` method to initialise the form, wait for the user to confirm or dismiss the dialogue and then if confirmed extract the information from the dialogue as below:

```
Class Function TfrmDGHIDEHelphelperConfig.Execute(slSearchURLs,
  slPermanentURLs : TStringList; var iSearchURL : Integer): Boolean;
```

```
Begin
  Result := False;
  With TfrmDGHIDEHelphelperConfig.Create(Nil) Do
    Try
      FFrame.InitialiseFrame(slSearchURLs, slPermanentURLs, iSearchURL);
      If ShowModal = mrOk Then
        Begin
          FFrame.FinaliseFrame(slSearchURLs, slPermanentURLs, iSearchURL);
          Result := True;
        End;
    Finally
      Free;
    End;
End;
```

This is a class method which creates the form and calls the `InitialiseFrame` method of the frame before displaying the form and if confirmed calls the frame's `FinaliseFrame` method.

## Adding your Frame to the IDE

Below is the interface definition of `INTAAddInOptions`. You will notice that this is a native interface rather than a standard OTA interface.

```
INTAAddInOptions = interface(IUnknown)
  ['{4B348F3E-6D01-4D88-A565-4C8C0EBF4335}']
  function GetArea: string;
  function GetCaption: string;
  function GetFrameClass: TCustomFrameClass;
  procedure FrameCreated(AFrame: TCustomFrame);
  procedure DialogClosed(Accepted: Boolean);
  function ValidateContents: Boolean;
  function GetHelpContext: Integer;
  function IncludeInIDEInsight: Boolean;
  property Area: string read GetArea;
  property Caption: string read GetCaption;
  property FrameClass: TCustomFrameClass read GetFrameClass;
  property HelpContext: Integer read GetHelpContext;
end;
```

So to get your options frame into the IDE you need to implement the above interface in a class as below:

```
TIDEHelpHelperIDEOptionsInterface = Class(TInterfacedObject, INTAAddInOptions)
Strict Private
```

```
    FFrame : TfmIDEHelpHelperOptions;
  Strict Protected
  Public
    Procedure DialogClosed(Accepted: Boolean);
    Procedure FrameCreated(AFrame: TCustomFrame);
    Function GetArea: String;
    Function GetCaption: String;
    Function GetFrameClass: TCustomFrameClass;
    Function GetHelpContext: Integer;
    Function IncludeInIDEInsight: Boolean;
    Function ValidateContents: Boolean;
  End;
```

Below are the implementations of each of the above interface methods with an explanation of what they do.

### DialogClosed

This method is called by the IDE when the IDE's options dialogue is being closed. The `Accepted` parameter is `True` if the dialogue is confirmed or `False` if it is dismissed.

```
Procedure TIDEHelpHelperIDEOptionsInterface.DialogClosed(Accepted: Boolean);

Var
  iSearchURL: Integer;

Begin
  If Accepted Then
    Begin
      FFrame.FinaliseFrame(AppOptions.SearchURLs, AppOptions.PermanentURLs,
iSearchURL);
      AppOptions.SearchURLIndex := iSearchURL;
    End;
End;
```

In this method, if the dialogue is confirmed I use a previously assigned reference to my IDE frame to collect the options data and save it back to the applications options.

### FrameCreated

This method is called immediately after the IDE has created your Options frame for you in the options dialogue but before the dialogue is displayed. Here is where you should initialise your options frame.

```
Procedure TIDEHelpHelperIDEOptionsInterface.FrameCreated(AFrame: TCustomFrame);
```

```
Begin
  If AFrame Is TfmIDEHelpHelperOptions Then
    Begin
      FFrame := AFrame As TfmIDEHelpHelperOptions;
      FFrame.InitialiseFrame(AppOptions.SearchURLs, AppOptions.PermanentURLs,
        AppOptions.SearchURLIndex);
    End;
End;
```

Here I make sure that the frame is the correct type before calling the frames initialisation method after storing a temporary reference to the frame for use in the `DialogClose` method.

### GetArea

The string you return here will be the name of the main tree element of the IDEs options dialogue where your options page will appear. The OTA documentation suggests for third party IDE experts/wizards that this should returns an empty string to place your options dialogue under a dedicated Third Party node in the options treeview.

```
Function TIDEHelpHelperIDEOptionsInterface.GetArea: String;

Begin
  Result := '';
End;
```

Here I return an empty string to place the options under the third party node of the IDE's dialogue.

### GetCaption

This method should return a string presenting the name of the node under the area main node where you options frame is to be displayed. Sub pages can be created using a period as the separator.

```
Function TIDEHelpHelperIDEOptionsInterface.GetCaption: String;

Begin
  Result := 'IDE Help Helper.Options';
End;
```

Here I return the name of the expert/wizard concatenated with the word Options to create a sub node to Third Party named after my expert/wizard and then a further sub sub node called Options

for the actual page.

### GetFrameClass

This method expects you to returns a class reference to your frame class so that the IDE can create the frame for you when it opens the IDE's options page.

```
Function TIDEHelpHelperIDEOptionsInterface.GetFrameClass: TCustomFrameClass;

Begin
  Result := TfmIDEHelpHelperOptions;
End;
```

Here I return a class reference to my implementation of the `INTAAddinOptions` interface.

### GetHelpContext

This method should return an integer for the help context.

```
Function TIDEHelpHelperIDEOptionsInterface.GetHelpContext: Integer;

Begin
  Result := 0;
End;
```

Here I return `0` to signify no help context.

### IncludeInIDEInsight

This method should return a boolean value which determines whether the frames controls can be searched for in the IDE's Insight search. It is recommended that you return `True`.

```
Function TIDEHelpHelperIDEOptionsInterface.IncludeInIDEInsight: Boolean;

Begin
  Result := True;
End;
```

Here I return `True`.

### ValidateContents

This method is called when the OK button of the IDE's Options dialogue is pressed in order for you to validate the contexts of your options frame. If there are any issues with the data you are recommended to display an error message and return `false` from this method or return `True` if all is okay.

```
Function TIDEHelpHelperIDEOptionsInterface.ValidateContents: Boolean;

Begin
  Result := True;
End;
```

I return `True` as there is nothing to validate.

### Registering you Options Frame

The final part of the puzzle is to tell the IDE about your interface by registering it. Unlike other interfaces the IDE does not return an integer value you can use to unregister the interface. Rather the IDE expects the interface's implementation instance for both here I chose to implement this registration / unregistration in the main expert/wizard interface as follows:

```
Constructor TWizardTemplate.Create;

Begin
  FOpFrame := TIDEHelpHelperIDEOptionsInterface.Create;
  (BorlandIDEServices As
INTAEnvironmentOptionsServices).RegisterAddInOptions(FOpFrame);
End;
```

```
Destructor TWizardTemplate.Destroy;

Begin
  (BorlandIDEServices As
INTAEnvironmentOptionsServices).UnregisterAddInOptions(FOpFrame);
  FOpFrame := Nil;
  Inherited Destroy;
End;
```

You will notice that these interfaces are registered using the `INTAEnvironmentOptionsServices` interface service which is implemented by `BorlandIDEServices` interface.

Make sure you only set your frame interface reference to `Nil` when destroying your expert/wizard and not `Free` it as you will have a catastrophic failure of the IDE as its a reference counted object.

Obviously you can open the IDE's options dialogue and scroll to the third party element to see your options page however you can implement a button that options the IDE's options page on your expert's/wizard's options page as follows:

```
(BorlandIDEServices As IOTAServices).GetEnvironmentOptions.EditOptions('', 'IDE
Help Helper.Options');
```

I hope this proves useful to all of you creating you own add-ins.

This article will be included in a revision of the OTA Book at some time in the future. All the code associated with this article can be found here.

Category: Open Tools API Tags: Borland, BorlandIDEServices, CodeGear, Embarcadero, Experts, INTAAddInOptions, INTAEnvironmentOptionsServices, IOTAServices, OTA, RAD Studio

## 3 thoughts on "Chapter 17: Options Page(s) inside the IDE's Options Dlg"

Uli Gerhardt
April 5, 2016

Nice article! In FrameCreated I found one of my pet peeves – if-is-then-as. 🙂
You have checked the type of AFrame in the if clause with is, so you don't need to check it again with as later on – a hard cast TfmIDEHelpHelperOptions(AFrame) would suffice.

David Post author
April 5, 2016

Hi Uli,

I acknowledge your comment. Its the way I've always done this which I think I've picked up from how Borland suggested typecast of this sort should be done (it be safe typecasting if I remember correctly). I think it makes it more understandable from the point of view of someone who's not so familiar with Object Pascal. There probably is a small overhead in the "As" typecast verse TXxx() but in this context I don't think it matters.

I will try in future not to implement any pet-peeves 🙂 [What about a big With statement :-)]

Per

April 6, 2016

I also use it – you could argue that if you (as in this case) only use it once, then an explicit typecast may be okay,.
But as soon as you reference the frame more than once, this pattern removes any clutter (arising from the typecast) from the code, making it much cleaner to read IMHO.

Per