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

Chapter 1: Starting an Open Tools API Project

By David | August 7, 2009 4 Comments

Before You Start

Think about your audience

Before you start, its a good idea to think a little about the management of the project files. Although you may think that you'll build this add-in for you current version of the IDE, if you up grade and require backward compatibility (or choose to distribute your add-in) you may come to regret some of your previous decisions. So below is some suggestions about the organisation of information and management of files based around the way I've found most flexible.

Structure

I like to structure my project directory to keep different types of files in different places. The directories I create for an IDE Add-in project are as follows:

- DCUs
- DLL
- Package
- Source

The DCUs and Source directories should be self explanatory. The DLL and package directories are for the DLL and Package version of the project we're going to build. This just keeps the project root directory clearer.

Name of Projects

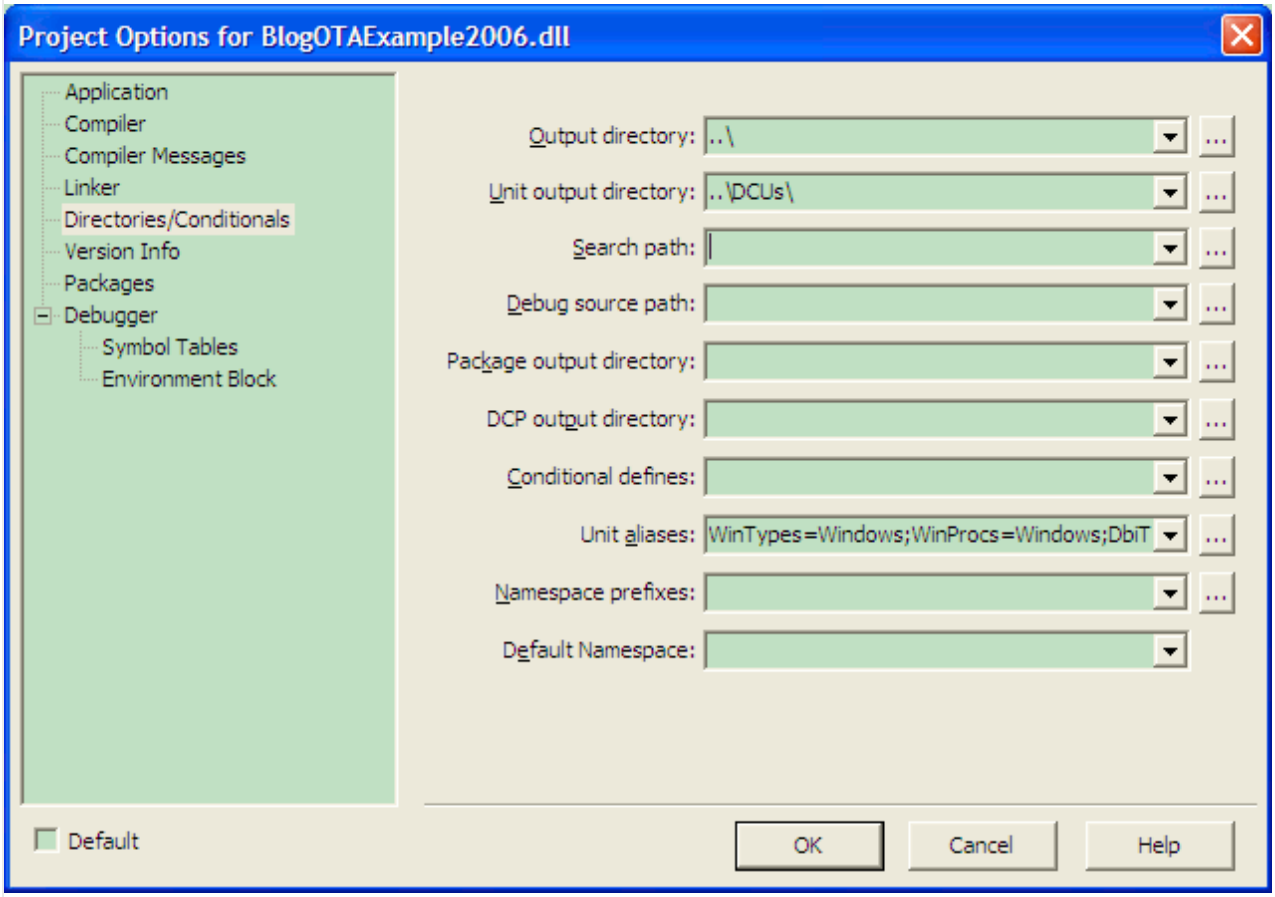
Append to the end of the project name the IDE version, i.e. 50 for Delphi 5 or 2006 for BDS 2006. Always create a new set of project files for each new compiler, you can not maintain different versions though the same project files. I've noticed that RAD Studio 2009 (don't have 2007) doesn't allow 2 project with the same name like BDS 2006 does, so you may have to append an extra P to the end of the package version.

Bear Bones

Open your IDE, I'm going to be using BDS 2006, but all of what I will do can be done with Delphi 5 through to RAD Studio 2009. Do the following:

- Add a new project group;
- Right click on the project group and add a new DLL project;
- Right click again on the project group and add a new Package project.
- Next select "Save All" from the File menu and save the DLL project to the DLL directory, the Package project to the Package directory and save the project group to the project root directory.

Next we need to configure the project options for each of the two projects. Fill in the options as you see fit, but there are 2 that need specific attention due to the directory structure we're using. These are Output Directory and Unit Output Directory and should be configured as shown the in image below.



There is one further change that needs to be made to the both projects in order that they can access the IDEs Open Tools API interfaces and they are handled slightly different for DLLs and Pacakages. For the DLL you need to add DesignIDE to the list of packages but for the Package you need to add DesignIDE to the packages Requires clause.

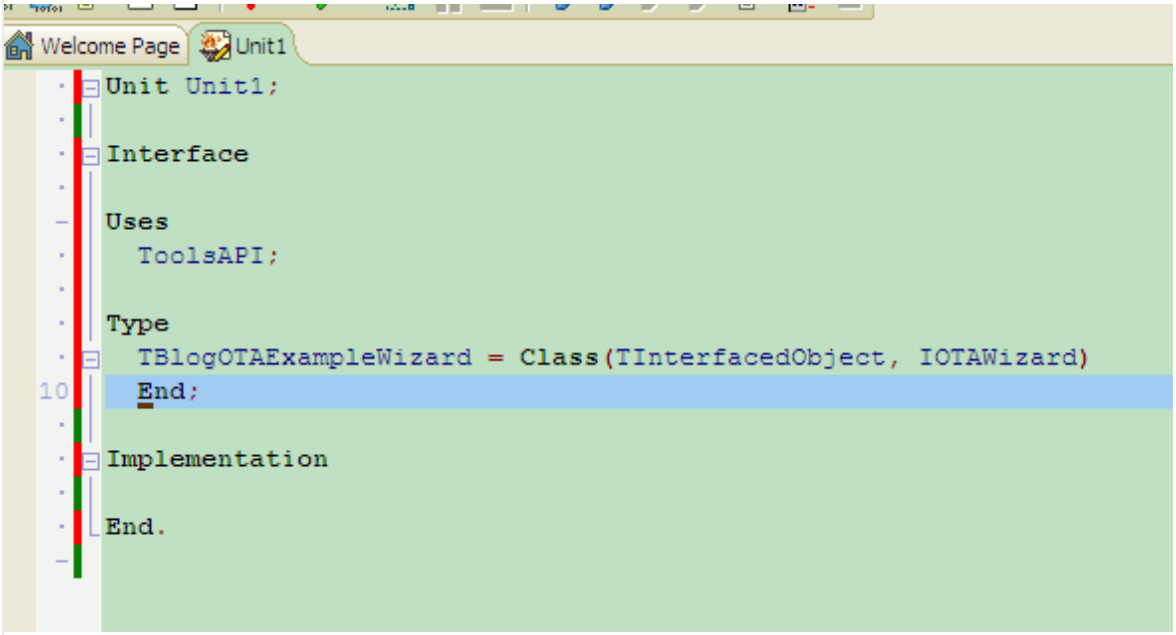
At this point the projects will compile but there will not actually do anything if loaded by the IDE. So the next step is to create a simple IDE expert / wizard.

Creating a Simple Wizard

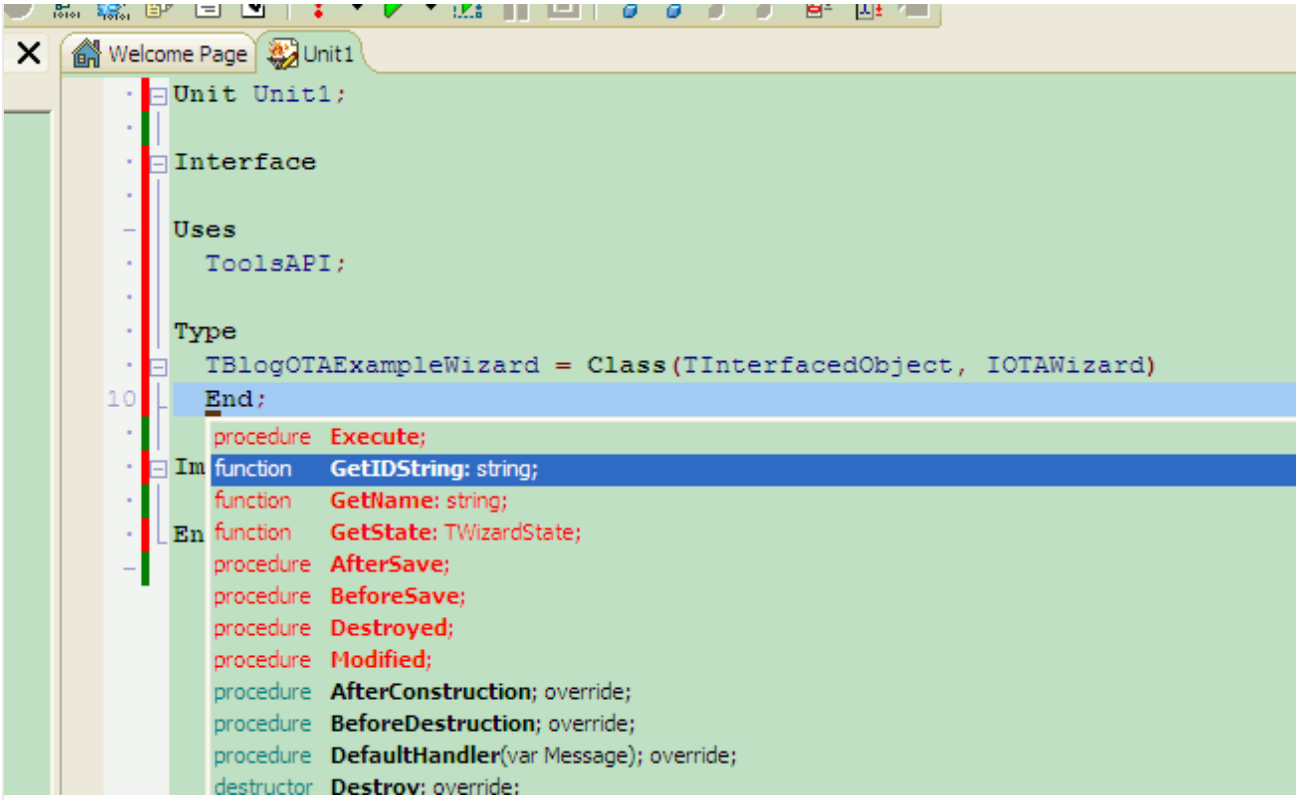
The code which follows (and has come before) can be used as a template to all new wizards. The differences between the wizards depend on the code you write inside the wizard and the interfaces that your wizard implements.

We need to start with a new unit, so right click on the DLL project and select a new unit. Save this to the Source directory with a meaningful name like BlogOTAExampleWizard.pas.

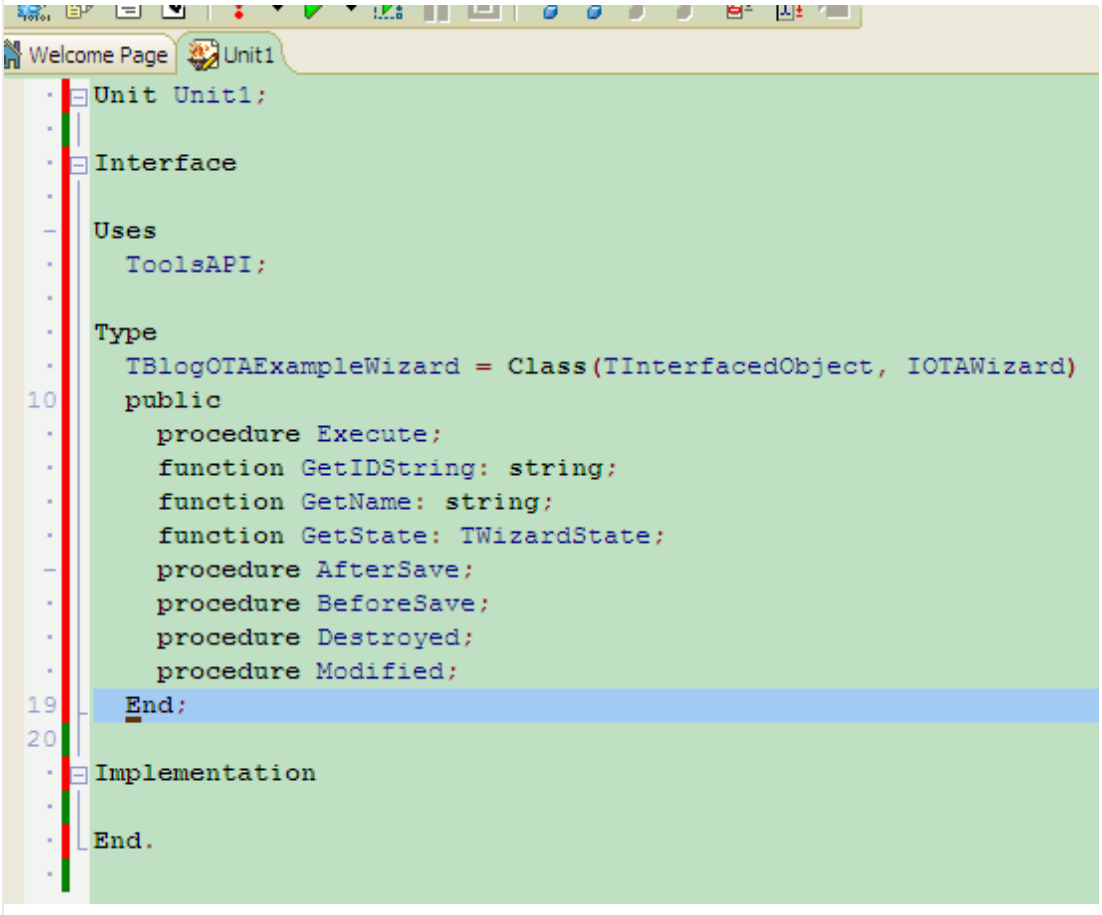
Next we need to add the Wizard class definition as follows:



At this point we need to implement the methods of IOTAWizard which aren't already implemented by our ancestor class TInterfacedObject. In BDS 2006 and above there's an easy way – place the cursor at the start of the End keyword in the class definition and press Ctrl+Space.



The methods that need to be implemented are the one in red. If you select all the red methods and press <Enter> then those methods will be added to the class declaration.



Use class completion (Ctrl+C) to write the implementations of these methods and save the unit as

BlogOTAExampleWizard.pas in the Source directory. To ensure that the Package also uses this same unit, drag and drop the unit from the DLL to the Package.

Implementing the Interface Methods

Now we need to implement the interface method sthat weren't handled by TInterfacedObject, i.e. the methods created by Class Compltion above. These method should be:

- procedure Execute;
- function GetIDString: string;
- function GetName: string;
- function GetState: TWizardState;
- procedure AfterSave; // Not called for IOTAWizards
- procedure BeforeSave; // Not called for IOTAWizards
- procedure Destroyed;
- procedure Modified; // Not called for IOTAWizards

So from above we only need to implement 5 out of the 8 methods as the other 3 are not called in wizards.

We only need to implement the Destroyed method IF we need to know when the wizard is being destroyed so that we can free memory used by the wizard. In this example we have no need to implement Destroyed.

GetIDString

This method returns to the IDE a unique identification string to distinguish your add-in from all others. Combine your name / company with the name of the add-in.

```
function TBlogOTAExampleWizard.GetIDString: string;
begin
  Result := 'David Hoyle.BlogIOTAExample';
end;
```

GetName

This method returns to the IDE a name for your add-in.

```
function TBlogOTAExampleWizard.GetName: string;
begin
  Result := 'Open Tools API Example';
end;
```

GetState

This method returns the state of the add-in this is a set of enumerates which only contains 2 items – wsEnabled and wsChecked. For our add-in we'll return just wsEnabled.

```
function TBlogOTAExampleWizard.GetState: TWizardState;
begin
  Result := [wsEnabled];
end;
```

Execute

This method for the moment will be left empty.

Making the IDE See the Wizard

At the moment our add-in will compile and could be loaded into the IDE (see below) but the IDE will complain that it can find the wizard. DLLs and Packages handle the informing of the IDE differently BUT you can still use the same code.

DLLs

DLLs require a function named InitWizard with the following signature:

```
Function InitWizard(Const BorlandIDEServices : IBorlandIDEServices;
  RegisterProc : TWizardRegisterProc;
  var Terminate: TWizardTerminateProc) : Boolean; StdCall;

Begin
  InitialiseWizard;
End;
```

Don't worry about the InitialiseWiard call just yet, we'll come to that in a minute. This function also needs to be exported by the DLL so that the IDE can know that the function exists in the DLL and initialise the DLL. To export the function you need to add the following declaration to your wizard unit including the interface definition of the InitWizard function.

```
  Function InitWizard(Const BorlandIDEServices : IBorlandIDEServices;
    RegisterProc : TWizardRegisterProc;
    var Terminate: TWizardTerminateProc) : Boolean; StdCall;

Exports
  InitWizard Name WizardEntryPoint;
```

Packages

Packages do thing slightly differently by requiring a procedure call Register. Note this is case sensitive.

```
Procedure Register;

Begin
  InitialiseWizard;
End;
```

Again, ignore the InitialiseWizard call. As with the DLL above we need to declare the interface declaration of the Register procedure.

InitialiseWizard

I've always started in the past by building my add-in as packages and then later on making them DLL compatible. This has lead to duplication of code in the 2 different initialisation procedures so I'm going to try a different approach and use a single initialisation procedure to do all the work – this is work in progress so bear with me here 🙂

DLLs and Packages load their wizards differently through different mechanism but we would like to minimise the duplication of code as much as possible. DLLs load the wizard by passing an instance of the wizard to the procedure RegisterProc in the signature of InitWizard. Packages on the other hand use the IOTAWizardServices to add the wizard to the IDE. The reason for the differences are to do with the fact that Packages can be loaded and unloaded dynamically throughout the life time of the IDE but DLLs are loaded only once during the IDE start-up process. Below is the code needed to initialise the wizard in the different mechanisms.

```
Var
  iWizard : Integer = 0;

Function InitialiseWizard(BIDES : IBorlandIDEServices) : TBlogOTAExampleWizard;

Begin
  Result := TBlogOTAExampleWizard.Create;
  Application.Handle := (BIDES As IOTAServices).GetParentHandle;
End;

Function InitWizard(Const BorlandIDEServices : IBorlandIDEServices;
  RegisterProc : TWizardRegisterProc;
  var Terminate: TWizardTerminateProc) : Boolean; StdCall;

Begin
  Result := BorlandIDEServices <> Nil;
  RegisterProc(InitialiseWizard(BorlandIDEServices));
End;

Procedure Register;

Begin
  iWizard := (BorlandIDEServices As IOTAWizardServices).AddWizard(
    InitialiseWizard(BorlandIDEServices));
End;
```

You will notice that InitialiseWizard is a function which returns the instance of our wizard to the 2 different method for loading the wizard in DLLs and Packages. With the DLL, once the wizard is created it does not need freeing but the Package version does. You will notice that AddWizard returns an integer reference for the wizard. This is used in a call to RemoveWizard to remove the wizard from memory. The best location for this is in the units Finalization section as below.

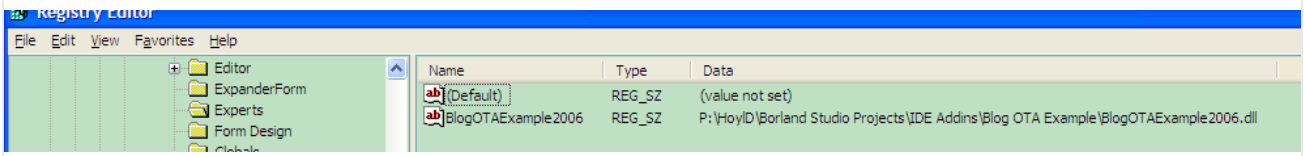
```
Initialization
Finalization
  If iWizard > 0 Then
117   (BorlandIDEServices As IOTAWizardServices).RemoveWizard(iWizard);
End.
```

You will notice a couple of things about these 2 piece of code. Firstly, the iWizard integer in initialised to zero – this is done so that the call to RemoveWizard is only used if the iWizard variable is greater than zero. iWizard will remain zero for a DLL and therefore the RemoveWizard call will not be make for DLLs only Packages.

Another thing you might have noticed is that there are no calls to free the wizard. This is because this is done by the IDE for you. If you are not convinced simply add a constructor and destructor to the wizard and add OutputDebugString calls to each and debug the application (see below).

Telling the IDE about the Wizard

To get the IDE to load the wizard you need to do 2 different things for DLLs and Packages. For packages simple load the IDE, open the package project, right click on the package and select "Install" and the package will be compiled and installed into the IDE. For a DLL its a little more complicated. We need to add a new register key to the IDE register entries. Since I'm using BDS 2006 I'll use its registry as an example but you will see its easy to do the same for other versions of the IDE. BDS 2006 stores its information under the registry location "My Computer\HKEY_CURRENT_USER\Software\Borland\BDS\4.0\". We need to add a new key (if not already there) called "Experts". Inside this key we need to add a new string entry named after the add-in with its value being the drive:\path\filename.ext of the DLL as below.

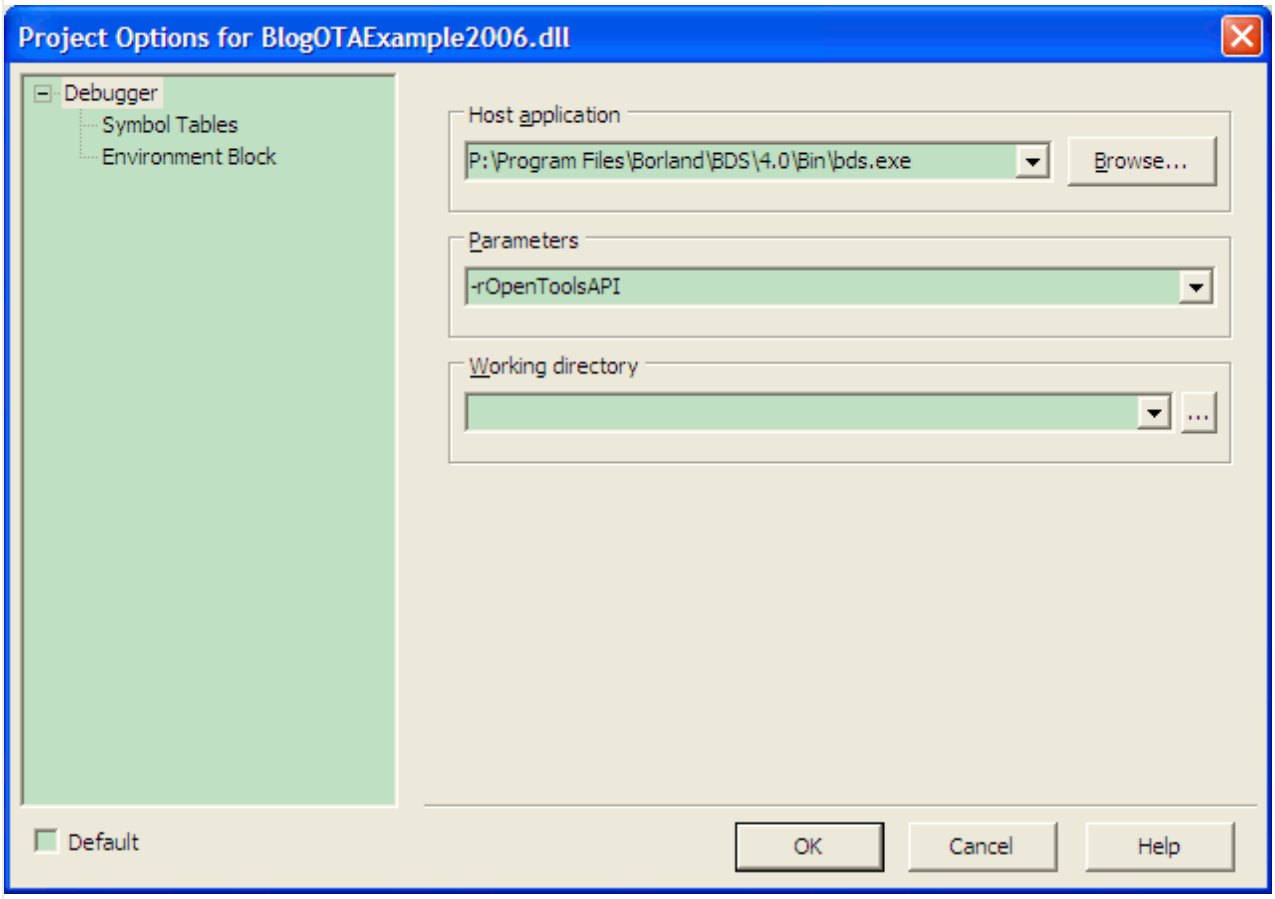


Now we can run the add-in wizard as either a package or DLL – DON'T DO BOTH!

Testing and Debugging

You could get instant gratification from your new add-in wizard by installing the package but if you've got things wrong then you could crash the IDE so its better to test and debug your add-in in a second instance of the IDE first. According to the Delphi Wiki entry there has been a command line switch for the IDE since Delphi 7 that tells it to use a different register key. I've never tried it with anything earlier than BDS 2006

but this works well. You need to setup the parameters for the debugging as below (note, use a different key for DLLs and Packages so that you don't get them both loading). Additionally, for these secondary IDEs the above Experts keys will need to be made in the alternate registry location (instead of \BDS\4.0\ it will be \OpenToolsAPI\4.0\).



Now if you run the DLL and debug it you should find that its does exactly nothing BUT doesn't crash the IDE on loading or unloading. Now we need to make the wizard do something.

Making the Wizard Actual Do Something

To make the add-in do something we need to add another interface to our class IOTAMenuWizard and implement 1 new method GetMenuText. Do not changes the current IOTAWizard reference to IOTAMenuWizard as your code will not compile. The IOTAWizard interface is required by RegisterProc. In the new method just return a string representing the text of your menu item. This menu item will appear under the Help menu. There is no more thing to do and than is do something in the Execute method. Add a call to ShowMessage with a simple string like "Hello World!" – you will have to add the Dialogs unit to the uses clause for the add-in to compile.

If you run the add-in now and select the menu item you will get you message displayed.

Conclusion

I hope this has been straight forward. Next time we'll provide a proper menu interface, one that appears

within the IDEs menus 🙂 Here are the source files for this example (Chapter1.zip).

Category: Open Tools API Tags: Borland, CodeGear, Delphi, Experts, IOTAWizard, IOTAWizardMenu, OTA, RAD Studio

4 thoughts on "Chapter 1: Starting an Open Tools API Project"

Mahmood_N
November 2, 2010

So Useful ! , thanks a lot …

David Post author
July 20, 2011

There is an error in the above code that stops the package version from being loaded properly by the IDE. All you need to do is type

Procedure Register;

in the interface section of the wizard module so the IDE can find the procedure to load the wizard. NOTE: Register IS case sensitive.

regards
Dave.

Piper Muskus
April 19, 2012

Does your site have a contact page? I'm having a tough time locating it but, I'd like to send you an email. I've got some ideas for your blog you might be interested in hearing. Either way, great website and I look forward to seeing it expand over time.

David Post author
April 19, 2012

Hi,

I do not have a contacts page to avoid spam. If you have ideas for the blog, please place them in a comment and I'll review them.

Dave.

Comments are closed.