Remobjects for Webservices

Michaël Van Canneyt

December 6, 2008

Abstract

The RemObjects SDK is a versatile tool which is likely to provide a solution for all your remoting needs: it implements a complete RPC (Remote Procedure Call) mechanism in pure Object Pascal. Using this toolkit, it is easy to write publicly available webservices, or to write high-performance RPC mechanism for use in all kinds of applications.

1 Introduction

Remobjects has been around for quite a while now, and has meanwhile - currently at version 5 named 'Vinci' - grown to a robust toolkit for implementation of client/server communications and implementation or use of web-services. It is available for pretty much any platform out there: Windows, Linux, Mac and the web. It can use native code or .NET assemblies, and objective C on Mac OS X.

It allows to create new services, i.e., server applications that provide an API to any client that needs them, or it can be used to consume existing services, i.e. it translates the API of a third-party service to something the client can work with (Object Pascal, for Delphi).

The website where this product can be downloaded is:

`http://www.remobjects.com/`

The Delphi version of the Remobjects SDK installs itself in the Delphi IDE - any version as of Delphi 6 will work. It creates a menu item 'Remobjects SDK', registers some new project types in the 'File|New' menu, and adds a lot of components to the 'Remobjects SDK' tab of the component palette.

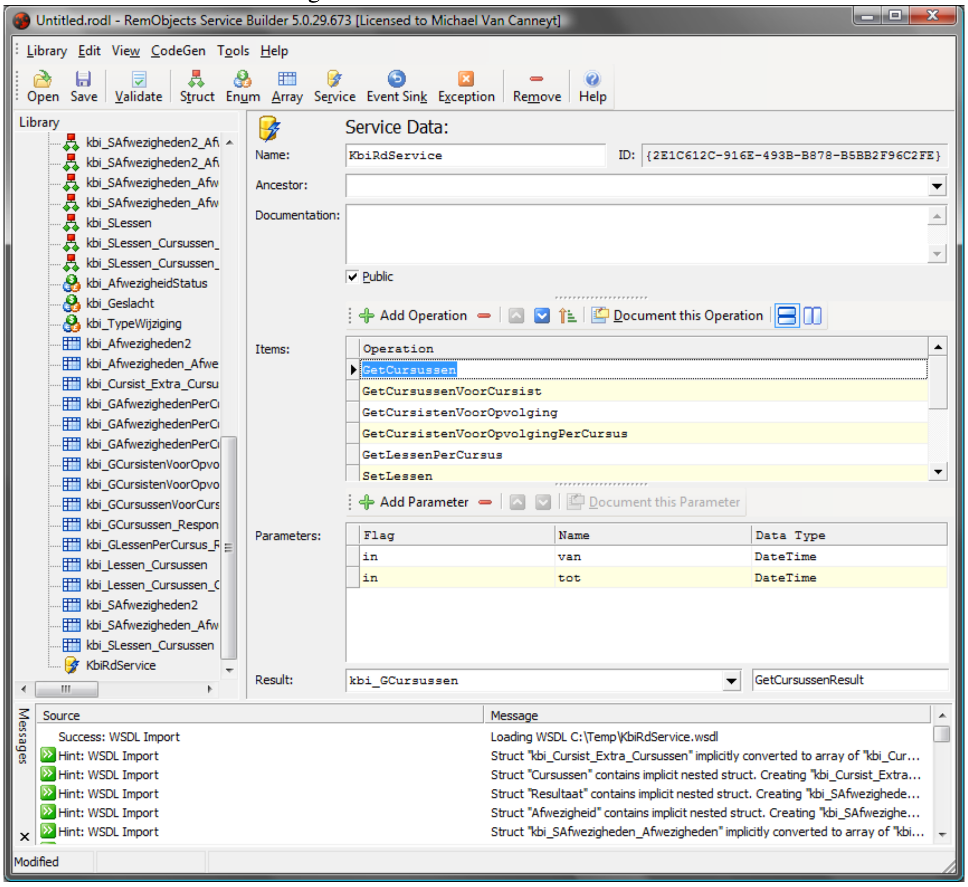
2 Getting started

To get started with Remobjects, the first thing one needs to get acquainted with is the Remobjects Service Builder. This stand-alone application is used to define the service one wishes to implement or access: it is the starting point for any RemObjects-based project. The Remobjects service builder can be started from the 'Remobjects SDK' menu in the Delphi IDE.

There are 2 options:

- One imports an existing service description: this can be a RemObjects service, or a service created by a third party: a webservice or an ActiveX server. In the first case, a RODL file is needed. For webservices, a WSDL file describing the webservice is needed. For the ActiveX server the type library or OCX is needed.

Figure 1: The service builder at work


Untitled.rodl - RemObjects Service Builder 5.0.29.673 [Licensed to Michael Van Canneyt]
Library Edit View CodeGen Tools Help
Open Save Validate Struct Enum Array Service Event Sink Exception Remove Help
Library
Add Operation
Document this Operation
Add Parameter
Document this Parameter
Messages
Modified

- kbi_SAfwezigheden2_Afi
- kbi_SAfwezigheden2_Afi
- kbi_SAfwezigheden_Afw
- kbi_SAfwezigheden_Afw
- kbi_SLessen
- kbi_SLessen_Cursussen_
- kbi_SLessen_Cursussen_
- kbi_AfwezigheidStatus
- kbi_Geslacht
- kbi_TypeWijziging
- kbi_Afwezigheden2
- kbi_Afwezigheden_Afwe
- kbi_Cursist_Extra_Cursu
- kbi_GAfwezighedenPerCi
- kbi_GAfwezighedenPerCi
- kbi_GAfwezighedenPerCi
- kbi_GCursistenVoorOpvo
- kbi_GCursistenVoorOpvo
- kbi_GCursussenVoorCurs
- kbi_GCursussen_Respon
- kbi_GLessenPerCursus_F
- kbi_Lessen_Cursussen
- kbi_Lessen_Cursussen_C
- kbi_SAfwezigheden2
- kbi_SAfwezigheden_Afw
- kbi_SLessen_Cursussen
- KbiRdService

Service Data:

Name: KbiRdService ID: {2E1C612C-916E-493B-B878-B5BB2F96C2FE}

Ancestor:

Documentation:

✓ Public

Items:

Operation
GetCursussen
GetCursussenVoorCursist
GetCursistenVoorOpvolging
GetCursistenVoorOpvolgingPerCursus
GetLessenPerCursus
SetLessen

Parameters:

Flag	Name	Data Type
in	van	DateTime
in	tot	DateTime

Result: kbi_GCursussen GetCursussenResult

Source	Message
Success: WSDL Import	Loading WSDL C:\Temp\KbiRdService.wsdl
Hint: WSDL Import	Struct "kbi_Cursist_Extra_Cursussen" implicitly converted to array of "kbi_Cur...
Hint: WSDL Import	Struct "Cursussen" contains implicit nested struct. Creating "kbi_Cursist_Extra...
Hint: WSDL Import	Struct "Resultaat" contains implicit nested struct. Creating "kbi_SAfwezighede...
Hint: WSDL Import	Struct "Afwezigheid" contains implicit nested struct. Creating "kbi_SAfwezighe...
Hint: WSDL Import	Struct "kbi_SAfwezigheden_Afwezigheden" implicitly converted to array of "kbi...

- It is also possible to define a complete new service.

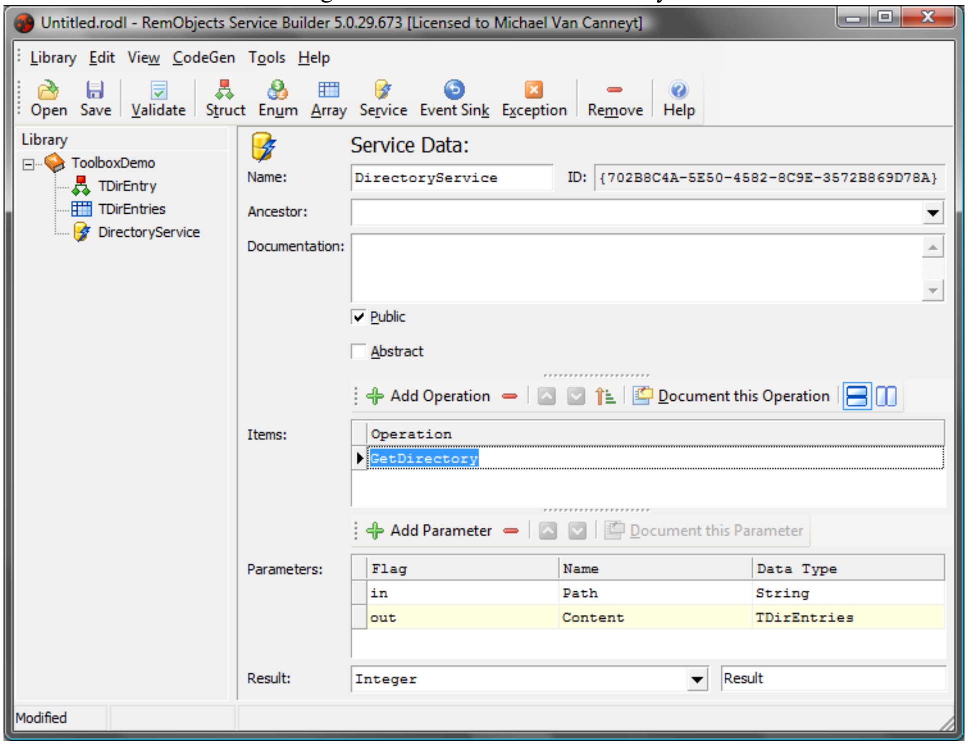
In figure 1 on page 2, the service builder is shown after it has succesfully imported a third-party WSDL file.

The end product of the service builder - regardless of the fact if it is a new or imported service - is a series of files:

- A RODL (RemObjects Data Language) file that describes the service.
- If the new service will be accessible as a web service, the service builder can create a WSDL file, describing the service.
- It can create a skeleton implementation in Object Pascal for the new service: this is needed if the new service will be implemented with Remobjects. The skeleton must then be filled with code.
- It creates an invoker: the invoker code translates the remoting messages to calls to the implementation generated in the previous point.
- It creates a pascal source containing all needed data structures and a series of interface definitions. This file also contains the proxy implementation which can be used to call the service. It is used both on the client and server side.

To demonstrate this, a small service will be implemented which returns the contents of a directory.

Figure 2: The toolboxdemo library


Untitled.rodl - RemObjects Service Builder 5.0.29.673 [Licensed to Michael Van Canneyt]
Library Edit View CodeGen Tools Help
Open Save Validate Struct Enum Array Service Event Sink Exception Remove Help
Library
ToolboxDemo
TDirEntry
TDirEntries
DirectoryService
Add Operation
Document this Operation
Add Parameter
Document this Parameter
Modified

Service Data:

Name: DirectoryService ID: {702B8C4A-5E50-4582-8C9E-3572B869D78A}

Ancestor:

Documentation:

✓ Public

☐ Abstract

Items:

Operation
GetDirectory

Parameters:

Flag	Name	Data Type
in	Path	String
out	Content	TDirEntries

Result: Integer Result

4 things are needed:

TDirEntry A record (called a Struct in the service builder) describing an item in a directory: It has 4 fields: `FileName`, `Hidden`, `Directory`, `TimeStamp`.

TDirEntries An array of `TDirEntry` records: The service will return the directory contents in a `TDirEntries` array.

DirectoryService This is the actual service definition. It can be compared with an interface specification. It contains the various public calls of the service.

GetDirectory This is the only operation in the service: it is the only actual call in the service. It has 2 parameters: `Path`, a string with the name of the directory whose contents must be returned. The second - out - parameter is `Content`, of type `TDirEntries`. Finally the function result is an integer, which will return the number of entries or a negative number in case of an error.

All this is collected in a library which will be named 'ToolboxDemo'.

Defining all this in the service builder is straightforward: the GUI speaks for itself and is quite intuitive: for each of the elements that can be added to a service, a button is available in the toolbar. The finished service description can be seen in figure 2 on page 3.

The library must now be saved as a .RODL file (name it toolboxdemo), and then the 'Codegen' menu can be used to create the Pascal definition (and possibly the implementation) of the service. 3 files must be produced:

1. The library interface definition. The service builder proposes toolboxdemo_intf.pas. It contains the interface and all data structures defined in the service.

2. The service implementation: DirectoryService_impl.pas is proposed. In case there are more than 1 service in the library, multiple implementation files must be created. These files must be filled with code in the Delphi IDE.
3. The service invoked: toolboxdemo_invk.pas. This file is needed on the server and will call the implementation based on the incoming request: the skeleton implementation file uses this unit.

Normally, only the implementation file must be edited: the two other must be added to the program, but will normally never be edited.

3 The server

When this is done, the files can be used in a Delphi server application. There are various ways of doing this: either one starts a new RemObjects service application (the 'New' menu contains a 'Remobjects SDK' section where many kinds of applications can be chosen from) and defines the service starting from there, or an existing application can be turned into a RemObjects server. This can be done using the 'Remobjects SDK' menu in the IDE. The toolboxdemo.rodl file can then be imported into the server application, using the same menu.

For the demo, the latter approach is used, and the 3 server-side files are added to a new VCL forms project. The `DirectoryService_impl.pas` must be opened and the implementation of the `IDirectoryService` server must be implemented in this file. The generated file is very simple, and looks like this:

```
TDirectoryService=class(TRORemotable,IDirectoryService)
protected
  function GetDirectory(const Path: String;
                        out Content: TDirEntries): Integer;
end;
```

The implementation of `GetDirectory` – an empty procedure was already generated – can be filled as follows:

```
function TDirectoryService.GetDirectory(const Path: String;
                        out Content: TDirEntries): Integer;

Var
  Info : TSearchRec;
  E : TDirEntry;
  P : String;

begin
  P:=IncludeTrailingPathDelimiter(Path)+'*.*';
  Result:=FindFirst(P,faAnyFile,Info);
  If (Result=0) then
    try
      Content:=TDirEntries.Create;
      Repeat
        E:=Content.Add;
        E.FileName:=Info.Name;
        E.Hidden:=(info.Attr and faHidden)<>0;
        E.Directory:=(Info.Attr and faDirectory)<>0;
        E.TimeStamp:=FileDateToDateTime(Info.Time);
        inc(Result);
      Until (FindNext(Info)<>0);
    finally
      FindClose(Info);
    end
  else if (Result>0) then
    Result:=-Result;
end;
```

Which is a very simple routine. Note that the `TDirEntries` array from the service description is implemented as a class, which must be instantiated. The RemObjects framework code will free this array, after the result was sent back to the client. The `Add` function to add a new element to the array is generated by the RemObjects service builder.

Note the initialization code of the implementation unit. It has been filled by the service builder with the following code:

```
Initialization
  TROClassFactory.Create('DirectoryService',
                         Create_DirectoryService,
                         TDirectoryService_Invoker);
```

This line tells the RemObjects message dispatcher mechanism that it should use a `TDirectoryService_Invoker` class to process messages for the `DirectoryService` service, and that the `Create_DirectoryService` function (which is also generated by the service builder) must be used to create a new instance of the `TDirectoryServce` implementation.

To finish the server application, it must be given 2 more components, which will be dropped on the main form:

1. A Server component.
2. A message component.

The server component represents the transport channel by which the server receives messages from the client and sends messages back to the client. This can be HTTP, SMTP, plain TCP/IP, named pipes, windows messages or plain DLL calls. For each of these transport mechanisms, a server component is available, each with it's own properties that describe how the server listens for messages from the client.

It is possible to make a server that listens to many transport channels at once. For instance, it is possible to create a server application that listens on a HTTP port and on a windows message channel at the same time.

For the purpose of the demo, a windows message server is used (named `ROServer`); This channel uses a window handle and the (very fast) Windows message copying mechanism. The only property that must be set is the `ServerID` property: this should be a unique string that can be used to reach the window handle used by the server (It is usually a good idea to use a GUID for this property). For the demo 'DirectoryServer' is used as a ServerID.

The message component determines how the client message must be formatted and how the serves formats its response to a request. Messages can be encoded in binary form (very fast) or as SOAP messages (required for webservices), or XMLRPC formatted messages. For each of these formats, a component must be dropped and optionally configured, and the `channel` component must be told to use this message in it's `Dispatchers` property. For the purpose of the demo, a binary message is dropped on the main form (`ROMessage`) and the `DisPatchers` property editor of the `ROServer` component is used to tell the server that it should use binary messages in its communication with the client.

With this, the server part is nearly ready. All that needs to be done is to activate the server component. This is done in the 'OnCreate' handler of the server's main form:

```
procedure TServerForm.FormCreate(Sender: TObject);
begin
  ROServer.Active := true;
end;
```

And with this, the server is ready to be used: As soon as the application is started, the `ROServer` component will start listening for client requests. It will decode the requests using the `ROMessage` component, and feed it to the dispatching mechanism of RemObjects: that will use the invoker class `TDirectoryService_Invoker` to transform the message to a call to the `TDirectoryService` class.

4 The client

Now that the server is ready, a client can be created. The client is also easily created using the RemObjects toolkit: Any project can be used as a RemObjects client.

3 components are needed to make a project a RemObjects client:

1. A transport channel component
2. A message component
3. A `TRORemoteService` component.

Only one instance of the two first components is needed in any client, so it may be a good idea to drop them on a TDataModule that is shared by all forms that need RemObjects functionality.

The meaning of the first 2 components should be clear: the transport channel component is used to transport the message from the client to the server: it should obviously match the transport mechanism that the server uses to listen for messages. Similarly, the message component should be set up identically as the one on the server. There are many transport channels available: some channels (e.g. HTTP) are even implemented in several ways, using different TCP/IP toolkits such as Indy, Synapse or native windows using the wininet DLL.

For the demo application, a binary message is used, and a `TWinMessageChannel` instance for the windows message transport channel. It's `ServerID` property must be set to `DirectoryServer`, so it knows where the server is listening.

The third component (`TRORemoteService`) represents the server service. It needs to be pointed to the used channel and message components by it's `Channel` and `Message` properties: it then knows how to connect to the server and send a message. The `ServiceName` property tells the service which service on the server it should contact, because a server can implement more than one service at once. From this it follows that one `TRORemoteService` component per accessed service on the server is needed.

With all this, everything is ready to show the directory structure on the server in a treeview. To call the server, it is sufficient to typecast the `TRORemoteService` component to the service interface that one wishes to call, for instance:

```
  Res:=(RORemoteService
          as IDirectoryService).GetDirectory(Apath,E);
```

`IDirectoryService` is the service that was defined and implemented on the server.

This can now be used to display the directory structure on the server in a treeview: So, an edit control (`EPath`) and a button `BExecute` are dropped on the form, as well as a treeview (`TVDirs`). In the `OnClick` event of the button, the following code is executed:

```
procedure TClientForm.BExecuteClick(Sender: TObject);
begin
  TVDir.Clear;
  FillTreeFromPath(Nil,Epath.Text);
end;
```

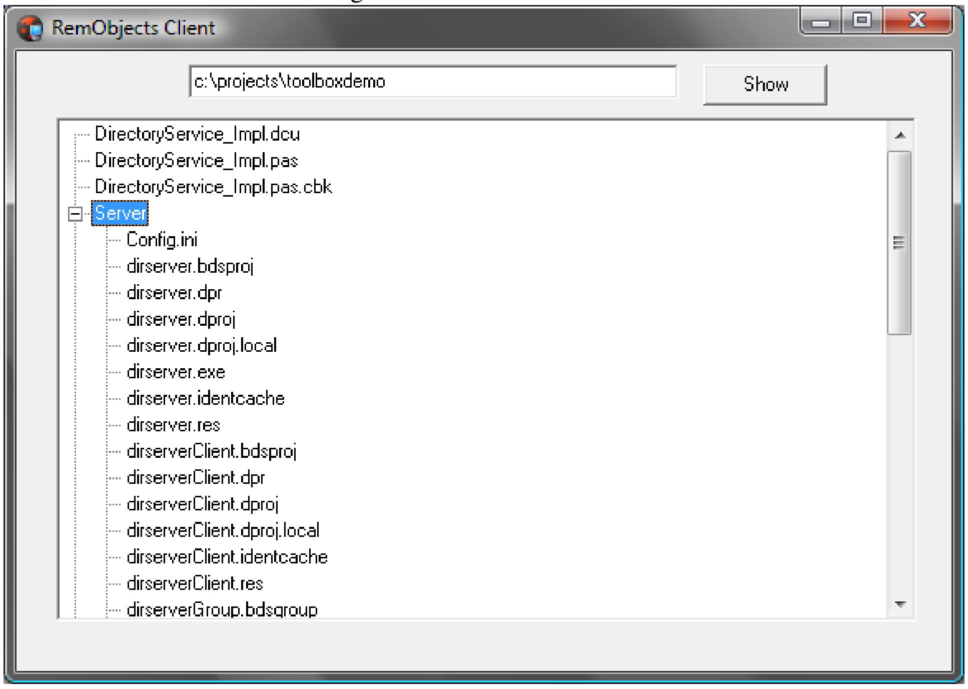
The `FillTreeFromPath` method is a recursive method:

```
procedure TClientForm.FillTreeFromPath(AParent : TTreeNode;
                                       APath : String);

Var
  E : TDirEntries;
  Res : integer;
  N : TTreeNode;
  S : String;
  I : integer;

begin
  Apath:=IncludeTrailingPathDelimiter(Apath);
  Res:=(RORemoteService as
          IDirectoryService).GetDirectory(Apath,E);
  if Res>0 then
    begin
    for I:=0 to E.Count - 1 do
      begin
      S:=E[i].FileName;
      if (S<>'.') and (s<>'..') then
        begin
        N:=TVDir.Items.AddChild(AParent,S);
        if E[i].Directory then
          FillTreeFromPath(N,APath+E[i].FileName);
        end;
      end;
    FreeAndNil(E);
    end;
end;
```

The first line executes the actual server call; it returns a list of filenames in the specified path: Then a loop is executed in which all filenames are added to the treenode which was passed as the parent node. If a directory entry is encountered, the `FillTreeFromPath` is called again with the newly added directory node as the parent.

That is all. As can be seen, no extra code is needed to use the service on the server from a remobjects client: everything is handled transparently by remobjects: the server interface can be called as if the server object was present in the client.

Figure 3: The client in action



```
c:\projects\toolboxdemo

DirectoryService_Impl.dcu
DirectoryService_Impl.pas
DirectoryService_Impl.pas.cbk
Server
    Config.ini
    dirserver.bdsproj
    dirserver.dpr
    dirserver.dproj
    dirserver.dproj.local
    dirserver.exe
    dirserver.identcache
    dirserver.res
    dirserverClient.bdsproj
    dirserverClient.dpr
    dirserverClient.dproj
    dirserverClient.dproj.local
    dirserverClient.identcache
    dirserverClient.res
    dirserverGroup.bdsgroup
```

To test all this, the server must be run first and then the client can be started. A push on the 'Show' button should then result in something like in figure 3 on page 8.

5 Conclusion

Remobjects makes client/server communication via webservices or using a private protocol quite easy: it completely hides the communication details from the application programmer, presenting him with an way of operation as if he was programming using standard Delphi interfaces. This article has barely scratched the surface of what is possible: there is more to remobjects than was presented here. Maybe a future contribution can explore the possibilities more in depth than the first meeting in this article.