webMethods EntireX

EntireX .NET Wrapper

Version 9.5 SP1

November 2013

This document applies to webMethods EntireX Version 9.5 SP1.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: EXX-EEXXDOTNETWRAPPER-95SP1-20140628

Table of Contents

EntireX .NET Wrapper

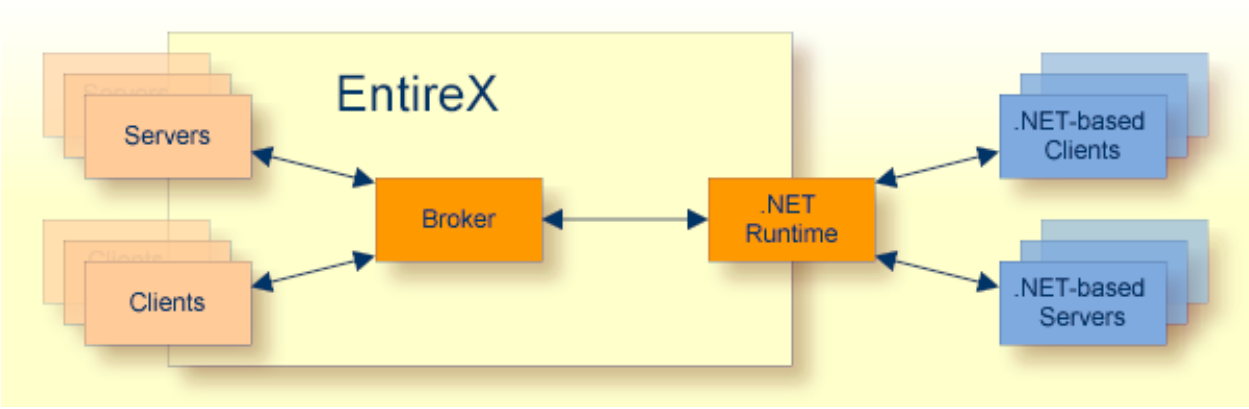
The EntireX .NET Wrapper provides access to RPC servers for .NET client applications and access to .NET servers for any RPC client. The .NET Wrapper generation tools of the Workbench take as input a Software AG IDL file, which describes the interface of the RPC, and generates C# classes that implement the methods and data types of the interface.

Introduction	Describes the functionality of the EntireX .NET Wrapper.
Using	How to use the .NET Wrapper: the generation process; using the .NET Wrapper interactively
Visual Studio Wizard for .NET Wrapper	Using .NET Wrapper with Microsoft Visual Studio Add-in.
Using the .NET Wrapper in IDL Compiler Command-line Mode	Using the .NET Wrapper in IDL Compiler command-line mode
IDL to .NET Mapping	Mapping Software AG IDL data types to .NET data types.
Writing Applications	Writing a client application with the EntireX .NET Wrapper.
EntireX .NET Wrapper Application Configuration	Configuring a .NET Wrapper application.
Reliable RPC	Introduction to reliable RPC; writing a client and a server for Reliable RPC; Broker configuration.
Reference	Reference material (attributes and classes).

1 Introduction to the .NET Wrapper

Description

The EntireX .NET Wrapper provides access to RPC servers for .NET client applications and access to .NET servers for any RPC client. The .NET Wrapper generation tools of the Workbench take as input a Software AG IDL file, which describes the interface of the RPC, and generates C# classes that implement the methods and data types of the interface.



The generated classes can be compiled with the C# compiler into a .NET assembly which can then be called from any .NET language.

The .NET Wrapper works as follows:

- C# code is generated from the Software AG IDL file. Using C# is a natural choice when full-fledged .NET programming is required, since C# was designed for the .NET platform.
- The .NET Wrapper runtime implements functionality that is not specific to a given IDL file (e.g., marshalling and unmarshalling of data). The generated C# code makes use of the .NET Wrapper runtime functionality. The customer interface and the .NET Wrapper runtime is “managed”.NET code (C#) and makes use of advanced .NET features such as Attributes, VersionInfo, etc.
- The .NET Wrapper runtime makes use of the functionality of the “unmanaged” RPC C runtime (dllimport in C#). “Managed”.NET code and “unmanaged” DLL code can be combined safely.
- The Software AG IDL Compiler and an appropriate template are used for the C# code generation.

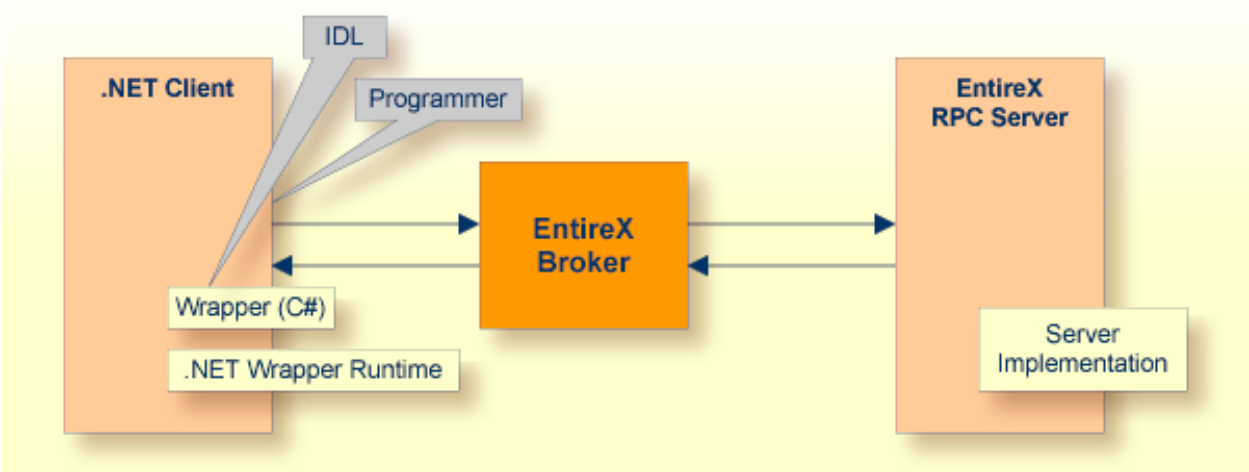
Generic .NET Wrapper Runtime

In order to minimize the amount of code generated for a specific IDL, all service-type functionality required by the client stub or the server DLL is implemented in a generic .NET Wrapper runtime *SoftwareAG.EntireX.NETWrapper.Runtime.dll*. The generic .NET Wrapper runtime implements service classes, i.e.:

- Marshalling .NET data types to Software AG IDL data types
- Unmarshalling Software AG IDL data types to .NET data types
- Connecting to RPC servers via Broker
- Connecting .NET servers via Broker with any RPC client.

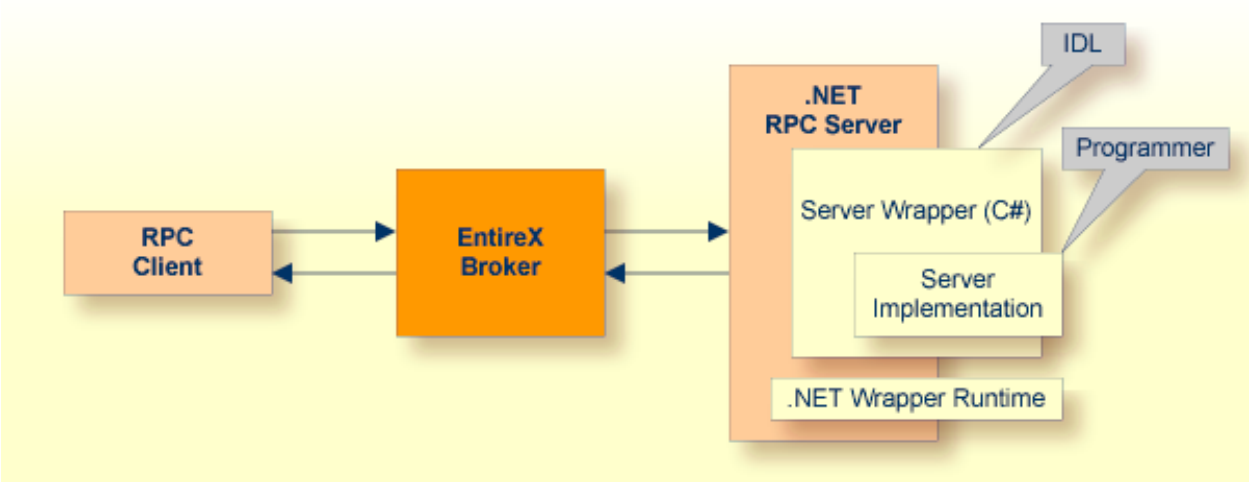
.NET Client Applications

For a given IDL file, the Software AG IDL Compiler and a C# code generation template for clients are used to generate a client stub. The source code generated by the .NET Wrapper can be compiled into a .NET assembly with the C# compiler. Application developers can use the generated client stub assembly to write .NET applications that access RPC servers. They are not limited to C# as programming language. Any .NET programming language based on the Common Language Runtime (CLR) can make use of the client stub assembly. Choices are C#, VisualBasic.NET or managed C++.



.NET Server DLL

The Software AG IDL Compiler and a C# code generation template for servers are used to generate a C# code frame for a specific IDL. Application developers can use the generated frame to write their own server code for each program in the IDL. The source code can be compiled into a .NET assembly (DLL) with the C# compiler. The assembly name needs to match the library name as specified in the IDL file.



2 Using the .NET Wrapper

Generation Process

To generate the C# client or server code, use the *EntireX Workbench*. This can be done interactively with the graphical user interface or in *Command-line Mode*.

Using .NET Wrapper Interactively

To use the .NET Wrapper functions, open your Eclipse Workspace.

Setting Wrapper Options

Before you start the generation of C# code for the first time, adjust the global options for the .NET Wrapper in the Eclipse preferences under **Software AG > EntireX > .NET Wrapper**.

On the **General** tab, set the paths to the Microsoft .NET Framework directory and the EntireX .NET Wrapper runtime (*SoftwareAG.EntireX.NETWrapper.Runtime.dll*). The preferences on the **Generate Client** and **Generate Server** tabs are identical. Choose your default settings for the client/server generation.

Option	Description
C# compiler options	Used to define additional options for the C# compiler (cse.exe).
Project relative output directory	A folder (structure) for C# code generation and compilation relative to the Eclipse project where the IDL file is located.
String handling	Default/String/StringBuilder: in the default case, "string" is used for IN and "StringBuilder" is used for OUT/INOUT parameters. In the case of "String", the C# type "string" is used for IN/INOUT/OUT. In the case of "StringBuilder", the C# class "StringBuilder" is used.
Class name prefix for inner classes	A string is used to prefix the name of the inner classes when the Sanitize option is selected (only necessary for Visual Basic clients).
Use IDL file base name for output	Use this flag only if you have large environments built with previous versions of the .NET Wrapper. If this flag is set and you have more than one library in your IDL file, a C# file is generated with the file base name of the IDL file (base name=file name without extension). If this flag is not set, the library name is used as file base name for the generated C# file (one file for every library in the IDL file).
Sanitize	If this flag is set, the IDL names are sanitized according to the programming conventions for C#. See *Mapping IDL Data Types to .NET Data Types*.
Generate "char" for A1 instead of String	The C# data type "char" is used for IDL parameters of type A1.
Generate "byte" for B1 instead of byte	The C# data type "byte" is used for IDL parameters of type B1.

Option	Description
Remove trailing blanks	Remove trailing blanks after unmarshalling the data. This flag is useful on the client side to remove trailing blanks before the data returned from the server is put into the C# classes `string`	`StringBuilder`.

These options are then used as default for the properties of your individual IDL files. You can change these options (except those on the **General** tab) for every individual IDL file.

3 Microsoft Visual Studio Wizard for EntireX .NET Wrapper

The Visual Studio Wizard for .NET Wrapper is a Software AG add-in for Microsoft Visual Studio 2010 that makes the client functionality of the EntireX .NET Wrapper available to Microsoft Visual Studio 2010.

Prerequisites for all EntireX components are described centrally. See *Windows Prerequisites* in the EntireX Release Notes.

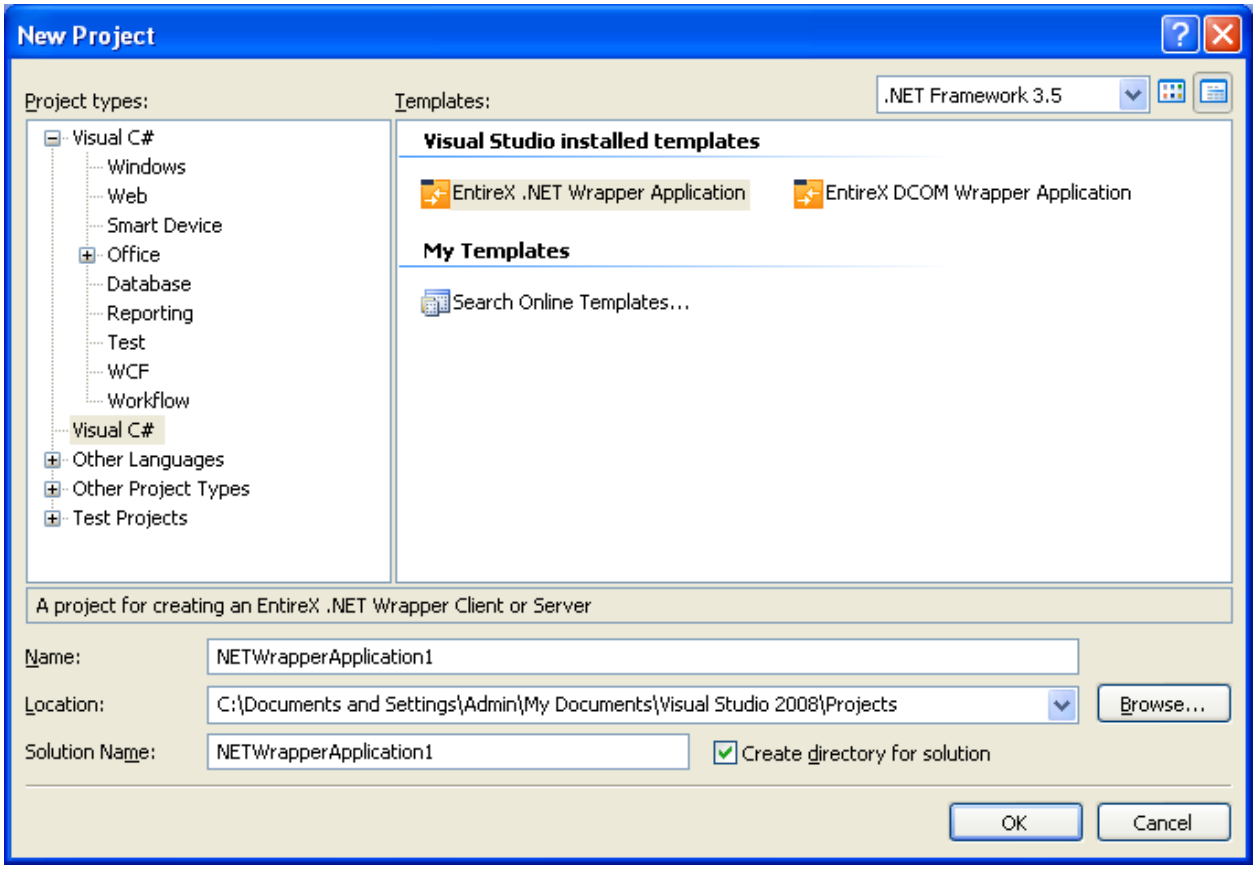
Installing the Add-in

The EntireX .NET Wrapper Wizard Add-in for Visual Studio .NET is part of the EntireX installation. After you have installed EntireX, you can find the installer under *etc* in your EntireX installation path. To install EntireX .NET Wrapper Add-in, start *NetVSAddIn90.msi* and follow the instructions.

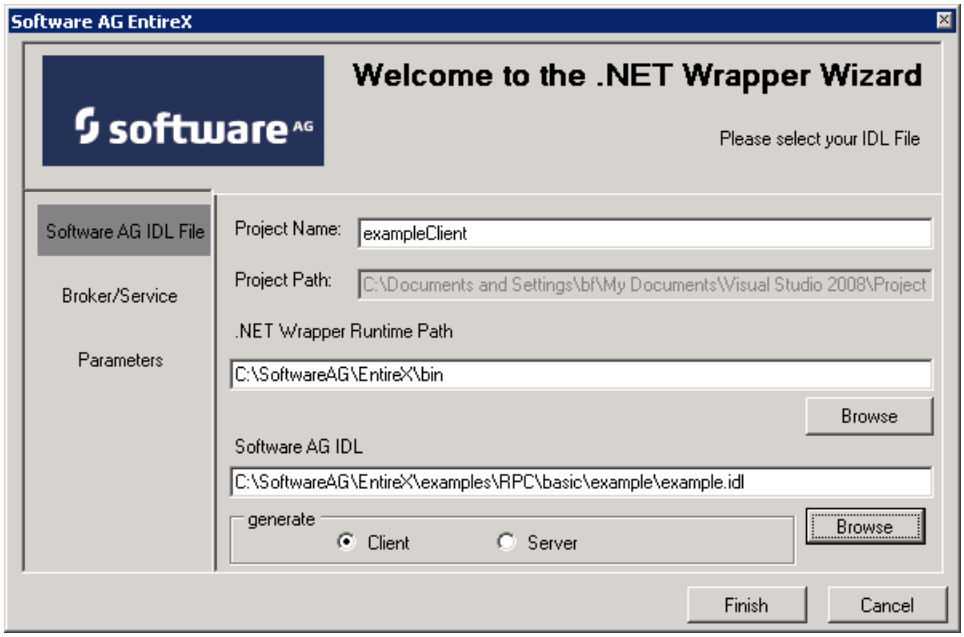
Caution: The installation path must include the *bin* directory (e.g. *C:\SoftwareAG\EntireX\bin*) of the corresponding EntireX installation, otherwise the add-in will not work properly!

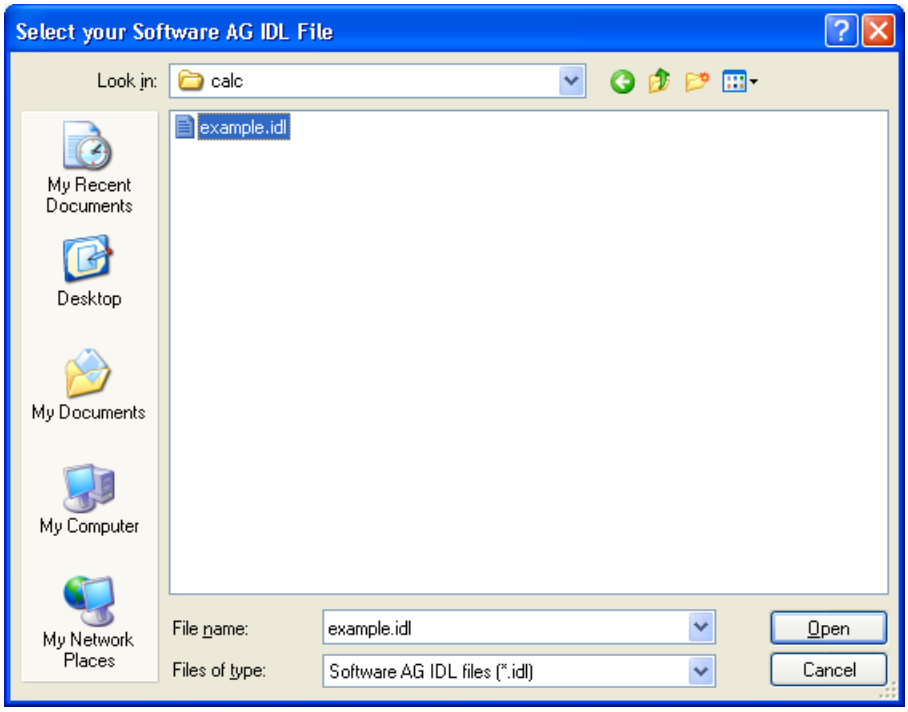
Using the Add-in

Once the wizard has been installed, start Microsoft Visual Studio 2010. Under **Project Types / Visual C#; Projects**, you will find a new template called EntireX .NET Wrapper Application.

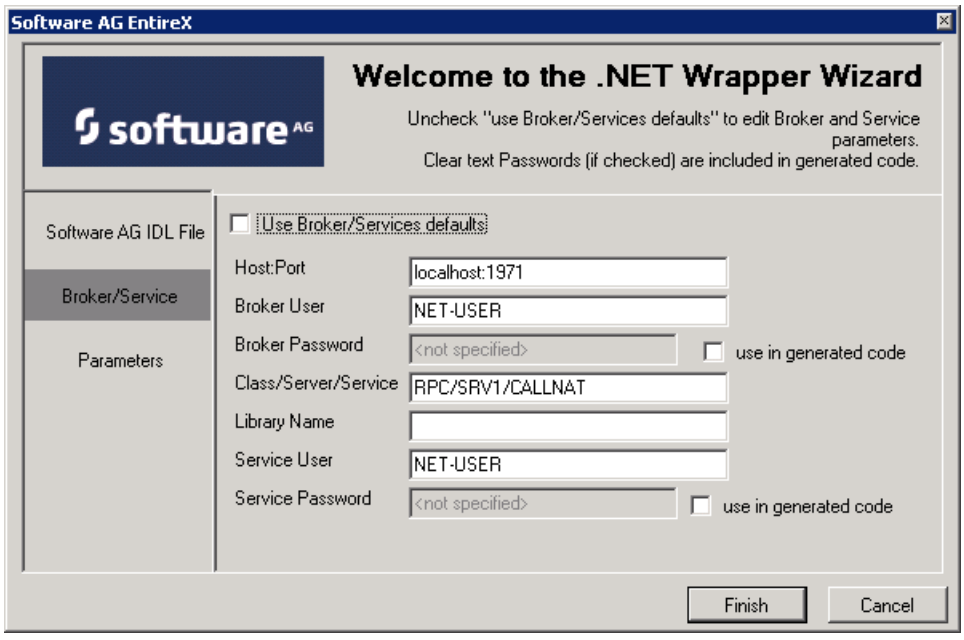


Select this template to start the .NET Wrapper Wizard.

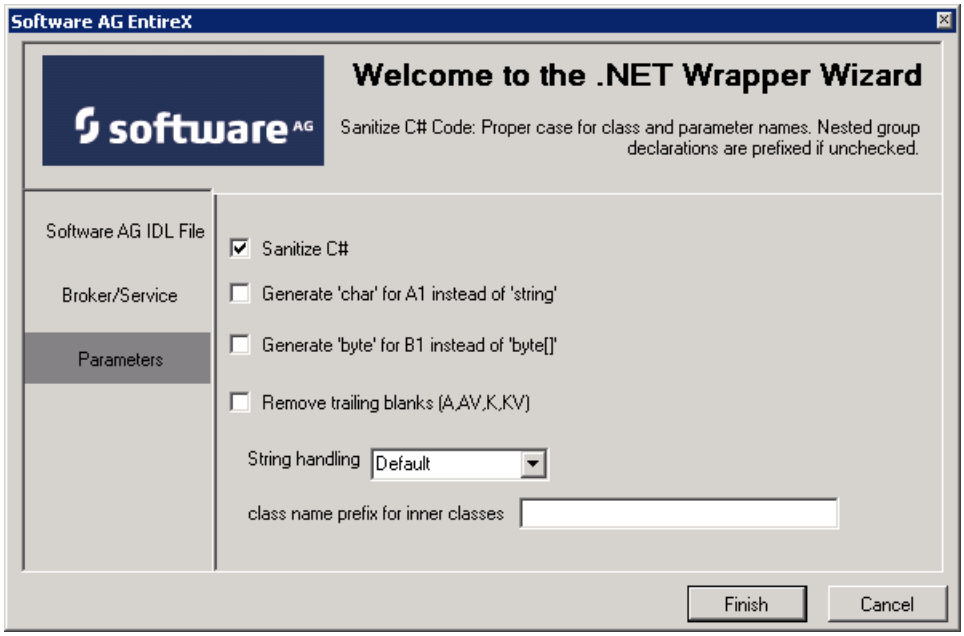




First enter the name of a Software AG IDL file in the opening window. You can select whatever you want to generate client or server code. The project name will be set to *IDLNameClient* or *IDLNameServer* automatically. You can enter the name of the path of the .NET Wrapper Runtime DLL if it is not located in the default path.



On the page **Broker/Service** you can change the default settings for Broker and Service.



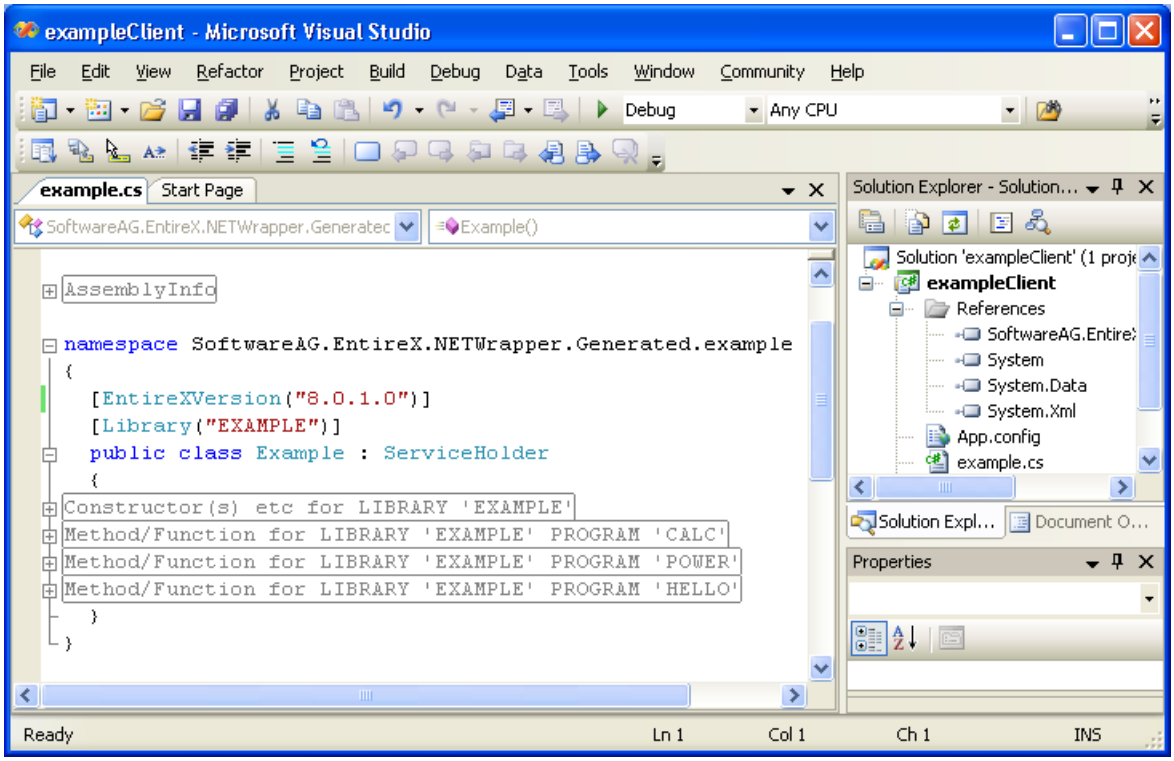
On the parameters page you can select the options **Sanitize** and char/string support.

For more information on Broker/Service and parameters, see the EntireX .NET Wrapper documentation.

When all data has been entered, click the button **Finish**. A new Visual Studio .NET solution will be generated which includes a project with the name *IDLNameClient* or *IDLNameServer*. This project contains the Software AG IDL file, the generated *.cs* file C# file) and references to the *System.dll* and the *EntireX.NetWrapper.Runtime.dll*.

The project will generate a class library (DLL), which can be used in any other .NET project C# or VB.NET). For this purpose an additional *App.config* file is generated which can be used in a project where an *.exe* file is generated. The *App.config* file contains information about Broker, Services etc.

Caution: Any changes to the Software AG IDL file will trigger the EntireX AddIn after saving. The *.cs* file will be re-generated and all specifications you made during the implementation will be lost.



Uninstalling the Add-in

You can uninstall the EntireX .NET Wrapper Wizard Add-in for Visual Studio 2010 by using the Windows **Control Panel** > **Add or Remove Programs**. Select **Software AG EntireX .NET Wrapper Wizard** and choose **Remove**.

The EntireX .NET Wrapper Wizard Add-in for Visual Studio 2010 will be removed from your computer.

Note: The add-in must be uninstalled before you uninstall EntireX, otherwise the uninstall of EntireX will fail.

4 Using the .NET Wrapper in IDL Compiler Command-line Mode

The table below shows the command-line options for the .NET Wrapper if the IDL Compiler is used. Options can be valid for client and server side.

Option	Req/ Opt	Description
`-D BROKER=nnn`	R	The EntireX Broker.
`-D SERVICE=nnn`	R	The EntireX Service.
`-t nnn`	R	Template for client (csharp_client.tpl) or server generation (csharp_server.tpl).
`-o nnn`	R	Project relative output directory or absolute Path
`-D ATOSTRING=String|StringBuilder`	O	String handling (Default if omitted).
`-D CLASSNAMEPREFIX=nnn`	O	Class name prefix for inner classes.
`-D A1TOCHAR=n`	O	Generate "char" for A1 instead of String (1 if required).
`-D B1TOBYTE=n`	O	Generate "byte" for B1 instead of byte (1 if required).
`-D TRIM=n`	O	Remove trailing blanks (1 if required).
`-PSANITIZE`	O	Sanitize.
`-F nnn`	O	File base name for output.

See also *Starting the IDL Compiler* in the IDL Editor documentation and *IDL Compiler Usage Examples* in the IDL Editor documentation.

Example

To start the IDL Compiler with the parameters for the stub generation, enter, for example the following in a single command line:

```
java -classpath "%ProgramFiles%\Software
AG\EntireX\classes\saglic.jar";"%ProgramFiles%\Software
AG\EntireX\Classes\exxidlcompiler.jar" -Dsagcommon="%CommonProgramFiles%\Software
AG" com/softwareag/entirex/idlcompiler/TplParser -PSANITIZE -D BROKER="localhost:1971"
-D SERVICE="RPC/SRV1/CALLNAT" -t "%ProgramFiles%\Software
AG\EntireX\Template\csharp_client.tpl" -F example -o NET example.idl
```

The client stub is generated in the subdirectory NET.

Status and processing messages are written to standard output (stdout), which is normally set to the executing shell window.

5 Software AG IDL to .NET Mapping

Mapping IDL Data Types to .NET Data Types

The table below lists the metasymbols and informal terms that are used for the Software AG IDL.

- The metasymbols [and] surround optional lexical entities.
- The informal term “number” is a sequence of numeric characters, for example 123.

Software AG IDL	Description	.NET Data Types	Note
A1	Alphanumeric	char or String/StringBuilder	1, 5
A*number*	Alphanumeric	String/StringBuilder	1
AV	Alphanumeric variable length	String/StringBuilder	1
AV[*number*]	Alphanumeric variable length with maximum length	String/StringBuilder	1
B1	Binary	byte or byte[]	6
B*number*	Binary	byte[]	
BV	Binary variable length	byte[]	2
BV[*number*]	Binary variable length with maximum length	byte[]	
D	Date	DateTime	3, 7
F4	Floating point (small)	float	
F8	Floating point (large)	double	
I1	Integer (small)	sbyte	
I2	Integer (medium)	short	
I4	Integer (large)	int	
K*number*	Kanji	String/StringBuilder	1
KV	Kanji variable length	String/StringBuilder	1
KV[*number*]	Kanji variable length with maximum length	String/StringBuilder	1
L	Logical	bool	
N*number* [.*number*]	Unpacked decimal	decimal	
NU*number* [.*number*]	Unpacked decimal unsigned	decimal	
P*number* [.*number*]	Packed decimal	decimal	
PU*number*[.*number*]	Packed decimal unsigned	decimal	
T	Time	DateTime	4, 7

Notes:

1. `System.String` for direction `in`, otherwise `System.Text.StringBuilder` if `Default` is used for parameter `ATOSTRING`. If `String` is used for `ATOSTRING`, `System.String` is used everywhere, and if `StringBuilder` is used for `ATOSTRING`, `System.Text.StringBuilder` is used everywhere. See ***Using the .NET Wrapper***.
2. Unsigned integer ranging from 0 to 255.
3. Count of days AD (anno domini, after the birth of Christ). The valid range is from 1.1.0001 up to 28.11.2737 (only the date part of `DateTime` is used).
4. Count of tenths of a second AD (Anno Domini, after the birth of Christ). The valid range is from 1.1.0001 00:00:00.0 up to 16.11.3168 09:46:39 plus 0.9 seconds.
5. If `-D A1TOCHAR=1` is defined in the `erxidl` call, `A1` is mapped to `char`, otherwise to `String/StringBuilder`.
6. If `-D B1TOBYTE=1` is defined in the `erxidl` call, `B1` is mapped to `byte`, otherwise to `byte[]`.
7. The Natural `DATE` type allows for the value 01.01.0000 to denote an undefined date. In order to avoid the .NET runtime throwing an exception when attempting to assign the invalid date value 01.01.0000 to a .NET `DateTime` variable, the .NET runtime converts an incoming neutral date/time value 01.01.0000 00:00:00.0 into the special .NET `DateTime` value `DateTime.MaxValue` - 1 tick (that is 31.12.9999:23:59:59.9999998). When this value is passed to the EntireX runtime to be sent to an EntireX RPC service, it is converted back into the neutral RPC date/time value 01.01.0000 00:00:00.0.

Please also note the hints and restrictions on the IDL data types valid for all programming language bindings as described under *IDL Data Types* under *Software AG IDL File* in the IDL Editor documentation.

Mapping Library Name and Alias

The library name as specified in the IDL file is sent from a client to the server. Special characters are not replaced. The library alias is not sent to the server.

In the RPC server, the IDL library name sent may be used to locate the target server. See *Locating and Calling the Target Server* in the platform-specific administration or RPC server documentation.

The name of the .NET server assembly must match the library name.

The library name as given in the IDL file is used to compose the names of the generated output files. See `library-definition` under *Software AG IDL Grammar* in the *IDL Editor* documentation. Therefore the allowed characters are restricted by the underlying file system. The name is composed from `<library-name>.idl` to `<library-name>.cs` as default. The name of the client stub file can be changed by using the `-F` option of the `erxidl` command. See ***Using the .NET Wrapper in IDL Compiler Command-line Mode***.

In accordance with the C# conventions, the class name is built as follows with the default setting `-PSANITIZE`:

- The initial character and characters following one of the special characters '#', '$', '&', '+', '-', '_', '.', '/' and '@' are converted to uppercase.
- All other characters are converted to lowercase.
- The special characters '#', '$', '&', '+', '-', '_', '.', '/' and '@' are removed.

Other special characters used in the library name are not changed and may lead to problems with your underlying file system and to compile errors.

If there is an alias for the library name in the `library-definition`, this alias is used “as is” to form the class name. Therefore, this alias must be a valid C# class name. To fully control the output, use alias names and do not use `SANITIZE`.

Examples:

`MY-CLASS` to `MyClass (class)`

`MY-CLASS alias YOUR_CLASS` to `YOUR_CLASS(class)`

Mapping Program Name and Alias

The program name is sent from a client to the server. Special characters are not replaced. The program alias is not sent to the server.

In the RPC server, the IDL program name sent is used to locate the target server. See *Locating and Calling the Target Server* in the platform-specific administration or RPC server documentation.

The program names as given in the IDL file are mapped to methods within the generated C# sources. See `program-definition` under *Software AG IDL Grammar* in the *IDL Editor* documentation.

In accordance with the C# conventions method names are built as follows with the default setting `-PSANITIZE`:

- Characters are converted to lowercase with the following exceptions
  - The special characters '#', '$', '&', '+', '-', '_', '.', '/' and '@' are removed
  - The character following one of the special characters is converted to uppercase.

Other special characters used in the program name are not changed and may lead to compile errors.

If there is an alias for the program name in the `program-definition` under *Software AG IDL Grammar* in the *IDL Editor* documentation, this alias is used "as is" for the method name. Therefore, this alias must be a valid C# method name. To fully control the output, use alias names and do not use `SANITIZE`.

Examples:

`MY-PROGRAM` to `MyProgram (method).`

`MY-PROGRAM alias YOUR_PROGRAM` to `YOUR_PROGRAM(method).`

Mapping Parameter Names

The parameter names as given in the `parameter-data-definition` of the IDL file are mapped to parameters of the generated C# methods.

In accordance with the C# conventions the parameter names are built as follows with the default setting `-PSANITIZE`:

- Characters are converted to lowercase except
  - The special characters '#', '$', '&', '+', '-', '_', '.', '/' and '@' are removed
  - The character following one of those special characters is converted to uppercase.

IDL files that use C# keywords (e.g. `string` or `float`) as parameter names are not supported. Do not use C# keywords such as `string` or `float` as parameter names. Modify your IDL file accordingly.

To fully control the output do not use `SANITIZE`.

Example:

`MY-PARAM` to `myParam (parameter)`

Mapping Fixed and Unbounded Arrays

Arrays in the IDL file are mapped to C# arrays. If an array value does not have the correct number of dimensions or elements, this will result in an exception. If the value `null` (null pointer) is used as an input parameter (for `IN` and `INOUT` parameters), an array will be instantiated by the runtime.

Mapping Groups and Periodic Groups

Groups in the IDL file are mapped to C# classes.

The namespace for group classes is `SoftwareAG.EntireX.NETWrapper.Generated.`*`filename`*`.Groups` on the client side, and `SoftwareAG.EntireX.NETWrapper.Server.`*`libraryname`*`.Groups` on the server side.

Mapping Structures

Structures in the IDL file are mapped to C# classes.

The namespace for structure classes is `SoftwareAG.EntireX.NETWrapper.Generated.`*`filename`*`.Structs` on the client side, and `SoftwareAG.EntireX.NETWrapper.Server.`*`libraryname`*`.Structs` on the server side.

See *Mapping Groups and Periodic Groups*.

Mapping the Direction Attributes IN, OUT and INOUT

- `IN` parameters are implemented as normal parameters of the generated C# class method.
- `OUT` parameters are implemented as out parameters of the generated C# class method.
- `INOUT` parameters are implemented as ref parameters of the generated method.

Note that only the direction information of the top-level fields (level 1) is relevant. Group fields always inherit the specification from their parent. A different specification is ignored.

See `attribute-list` under *Software AG IDL Grammar* in the *IDL Editor* documentation for the syntax on how to describe attributes within the IDL file and refer to the `direction` attribute.

Mapping the ALIGNED Attribute

Not supported.

Calling Servers as Procedures or Functions

The IDL syntax allows definitions of procedures only. It does not have the concept of a function. A function is a procedure which, in addition to the parameters, returns a value. Procedures and functions are transparent between clients and server, i.e. a client using a function can call a server implemented as a procedure and vice versa.

In C# a procedure corresponds to a method with result type void, a function returns a value of some type.

It is possible to treat an `OUT` parameter of a procedure as the return value of a function. The .NET Wrapper generates a method with a non-void result type when the following two conditions are met:

- The last parameter of the procedure definition is of type `OUT`;
- This last parameter of the procedure definition has the name `Function_Result`.

In this case no function parameter is generated for this `OUT` parameter.

See the .NET Wrapper example that comes with EntireX.

6 Writing Applications with the .NET Wrapper

Writing a Client Application

Required Steps

Writing a client application with the EntireX .NET Wrapper typically requires the following steps:

- Starting from an IDL file, generate a C# client stub using either the *EntireX Workbench* .NET Wrapper GUI or the Software AG IDL Compiler (*erxidl*) and the *csharp_client.tpl* template from the command line.
- Build a .NET assembly from the generated C# client stub.
- Create an application that uses the generated client stub assembly and the .NET Wrapper runtime *SoftwareAG.EntireX.NETWrapper.Runtime.dll*.

The following description outlines as an example the steps required to build a .NET Wrapper client application (solution) with the Microsoft Visual Studio.

Generating the .NET Wrapper Client Stub from a Software AG IDL File

We assume the IDL source file has the name *example.idl* and there is an EntireX RPC service available that implements the interface described in the IDL file.

If the IDL file was generated from a source containing Natural REDEFINEs, a CVM file is required. See *Redesigning the Extracted Interface* in the IDL Extractor for Natural documentation, and also *CVM File*.

1. Open the *EntireX Workbench*, select the *example.idl* file.
2. From the .NET menu, choose Generate client. This will generate a C# source file *example.cs*.

See also ***Using the .NET Wrapper***.

Creating a Microsoft Visual Studio Solution

1. Start Microsoft Visual Studio.
2. From the **File** menu, choose **New > Blank Solution**.... and choose an appropriate name for the solution.

Creating the .NET Wrapper Client Stub Library (Assembly)

1. Select the solution and choose **Add**, choose **New Project**.
2. In the New Project dialog, choose **Visual C# Projects** and **Class Library**. Choose an appropriate name for the class library, e.g. "exampleClientStub".
3. Delete the default class file *Class1.cs*.
4. Select the new project and choose **Add > Add Existing Item** and add the *example.cs* file generated previously.
5. Select References, choose **Add Reference** and add the .NET Wrapper runtime *SoftwareAG.EntireX.NETWrapper.Runtime.dll*.
6. Build the class library.

Creating the .NET Wrapper Client Application

1. Add a new project to the solution: Choose the solution, **Add**, **New Project**..., **Visual C# Projects**, **Console Application**. Choose an appropriate name for the project, for example, "exampleClient".
2. Rename the default class file *Class1.cs* as appropriate.
3. Choose **References > Add Reference** and add the .NET Wrapper runtime *SoftwareAG.EntireX.NETWrapper.Runtime.dll*.
4. Choose **References > Add Reference > Projects** and add the .NET Wrapper client stub *exampleClientStub*.
5. Now implement your client application. Add the following lines to the top of the class file:

```
using SoftwareAG.EntireX.NETWrapper.Runtime;
using SoftwareAG.EntireX.NETWrapper.Generated.example;
```

6. In a method of the application class implement the connection to an EntireX Broker, for example,

```
Broker broker = new Broker("localhost:1971", "ERX-USER");
broker.Logon("ERX-PASS");
```

   and an EntireX RPC service, for example,

```
Service service = new Service(broker, "RPC/SRV1/CALLNAT", "EXAMPLE");
service.UserIDAndPassword("RPC-USER", "RPC-PASSWORD");
```

7. The example class can now be instantiated, for example,

```
Example e = new Example( service );
```

and the example methods called, for example,

```
int result = ex.Calculator( "+", 10, 15);
```

Writing a Server DLL

Required Steps

Writing a server DLL with the EntireX .NET Wrapper typically requires the following steps:

- Starting from a Software AG IDL file, generate a C# file using either the *EntireX Workbench* .NET Wrapper GUI or the Software AG IDL Compiler (*erxidl*) and the *csharp_server.tpl* template from the command line.
- Insert your server-specific code at the required position for the programs (methods).
- Build a .NET assembly (server DLL) from the generated C# file.

Building a .NET Wrapper server DLL with the Microsoft Visual Studio follows the rules for building a client stub library.

Note: The file name of the server DLL and the name of the library/class in the generated C# file must be identical.

Deploying Wrapped .NET Servers

The easiest way to deploy and run a .NET server is the so-called XCOPY-deployment. This means that all relevant files of the server are installed in one folder. No additional registration and configuration is required. The only prerequisite is that the EntireX runtime is installed. The following files are typically required:

- the server wrapper and implementation assembly (or assemblies)
- the .NET Wrapper runtime (*SoftwareAG.EntireX.NETWrapper.Runtime.dll*)
- the .NET server user exit DLL (*dotNetServer.dll*)
- the RPC server executable (*rpcserver.exe*)
- a configuration file (.cfg) for the RPC server according to the rules described under ***Configuring the EntireX RPC Server for use with the .NET Wrapper*** under ***EntireX .NET Wrapper Application Configuration***.

To make the .NET server available to EntireX clients, the .NET RPC server must be up and running and able to locate the server implementation.

The described XCOPY deployment method has the drawback that copies of the .NET Wrapper runtime and the .NET RPC server have to be deployed with the application. It is possible to avoid this by making use of the .NET Framework's application configuration capabilities. Various parameters of a .NET application, say *myapp.exe*, can be configured in a configuration file *myapp.exe.config* that must be located in the executable's folder. The configuration file defines in XML format several parameters of the application, such as the dependent assemblies, version and location and others. Using this method, neither the .NET Wrapper runtime nor the .NET RPC server needs to be deployed. However, the configuration file for the .NET RPC server must be located in the same folder as the RPC server itself, which by default is the *bin* folder of the EntireX installation. As a consequence, if there are multiple .NET servers deployed on the system, they all need to be configured in the .NET RPC server's configuration file.

Creating ASP.NET Web Services

The generated C# client stub can be used in an ASP.NET Web service to publish EntireX RPC services as Web services. With Visual Studio you can easily create an ASP.NET Web service that publishes methods of the EntireX RPC service (or your own methods that just use the EntireX RPC service).

Note: The .NET Wrapper Runtime uses unmanaged DLLs. For this reason, ASP.NET applications have to run in full-trust mode.

Example

You have built the .NET Wrapper example *EntireX\examples\RPC\basic\example\dotNetClient* as described in the README file.

Then create a new “ASP.NET Web service” project with references to the generated client stub and the .NET Wrapper runtime.

You can use the following example code (in the .asmx file) to implement a Web method `add` that exposes the `calc` method of the example.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using System.Text;
using SoftwareAG.EntireX.NETWrapper.Runtime;
```

```
using SoftwareAG.EntireX.NETWrapper.Generated.example;

namespace WebService1
{
/// <summary>
/// Summary description for Service1.
/// </summary>
public class Service1 : System.Web.Services.WebService
{
public Service1()
{
//CODEGEN: This call is required by the ASP.NET Web Services Designer
InitializeComponent();
}

#region Component Designer generated code

//Required by the Web Services Designer
private IContainer components = null;

/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
}

/// <summary>
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing )
{
if(disposing && components != null)
{
components.Dispose();
}
base.Dispose(disposing);
}

#endregion

// WEB SERVICE EXAMPLE
 [WebMethod]
public int add(int sum1, int sum2)
{
Example e = new Example();

int result = e.calc("+", sum1,sum2);
return result;
}
```

```
}
}
```

Using Internationalization with the .NET Wrapper

It is assumed that you have read the document *Internationalization with EntireX* and are familiar with the various internationalization approaches described there.

The .NET Wrapper uses by default the "current locale" encoding set up on the Windows system for converting UNICODE (UTF-16) representations of strings to single-byte or multibyte representations that are sent to the Broker, and vice versa.

If you want to adapt the locale settings of your Windows system, use the Regional and Language Options in the Windows Control Panel.

The *Broker* class of the .NET Wrapper Runtime makes use of the .NET Framework class *System.Text.Encoding* for character conversion.

Refer also to the .NET Framework class library documentation for *System.Text.Encoding*.

The *CharacterEncoding* property of the *Broker* class, that guides the character conversion, is initialized with `System.Text.Encoding.GetEncoding(0)` (current locale). The codepage that corresponds to this encoding is automatically transferred to the Broker as part of the locale string, specifying the encoding of the data, when communicating with a Broker version 7.2 and above.

The application programmer can also assign a custom encoding object to the Broker class' character encoding property for custom character conversions. If an encoding object is provided, the corresponding codepage is transferred as part of the locale string to the Broker for all Broker versions.

If communicating with a Broker version 7.1 and below and if no encoding is provided by the .NET Wrapper programmer, an EntireX administrator can force a codepage string to be sent to the Broker by setting the environment variable `ERX_CODEPAGE` to the name of the respective codepage. See *ERX_CODEPAGE*.

When setting the codepage with the environment variable `ERX_CODEPAGE`:

- The `ERX_CODEPAGE` environment variable is ignored if the application programmer has already provided a codepage.
- The value of the `ERX_CODEPAGE` environment variable must be the name of the system's default codepage. Under Windows, simply apply the value `"LOCAL"` to specify the default Windows ANSI codepage.
- The codepage specified must be one that is supported by the Broker, depending on the Broker's internationalization approach. See *Locale String Mapping* in the internationalization documentation for information on how the broker derives the codepage from the locale string.

- Before starting the application, set the locale string with the environment variable `ERX_CODEPAGE`.

Example:

```
ERX_CODEPAGE=LOCAL
```

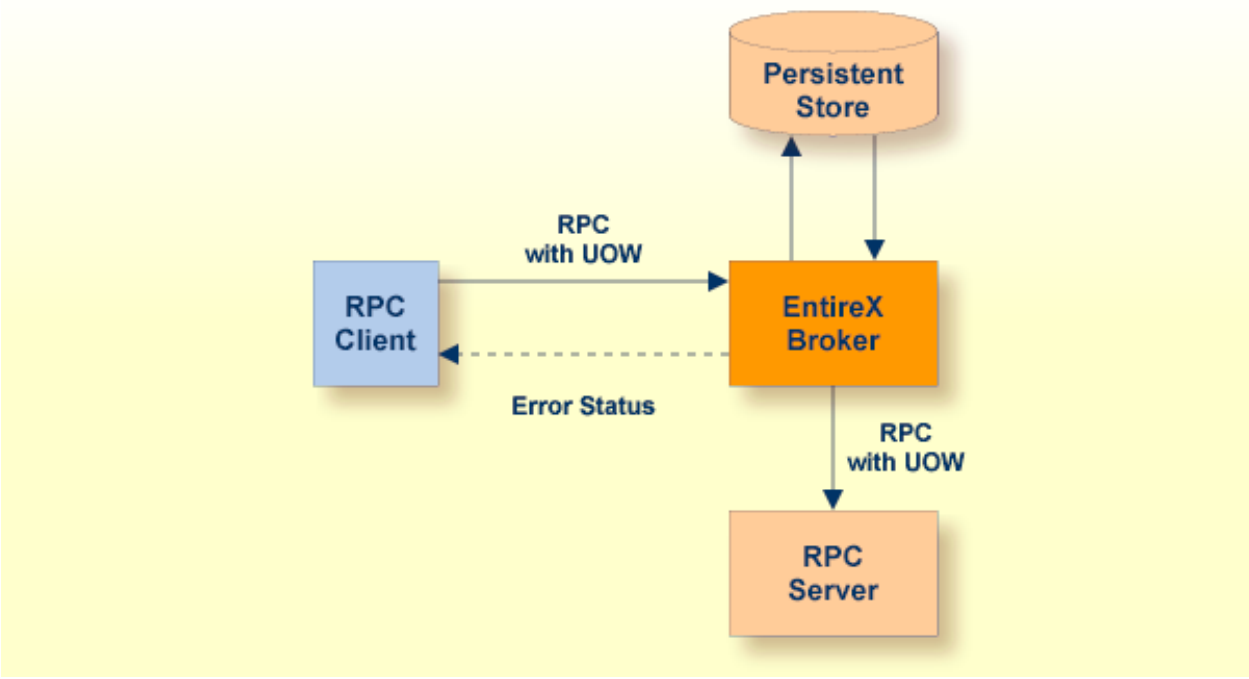
7 Reliable RPC for .NET Wrapper

Introduction to Reliable RPC

In the architecture of modern e-business applications (such as SOA), loosely coupled systems are becoming more and more important. Reliable messaging is one important technology for this type of system.

Reliable RPC is the EntireX implementation of a reliable messaging system. It combines EntireX RPC technology and persistence, which is implemented with units of work (UOWs).

- Reliable RPC allows asynchronous calls ("fire and forget")
- Reliable RPC is supported by most EntireX wrappers
- Reliable RPC messages are stored in the Broker's persistent store until a server is available
- Reliable RPC clients are able to request the status of the messages they have sent



Reliable RPC is used to send messages to a persisted Broker service. The messages are described by an IDL program that contains only `IN` parameters. The client interface object and the server interface object are generated from this IDL file, using the EntireX .NET Wrapper.

Reliable RPC is enabled at runtime. The client has to set one of two different modes before issuing a reliable RPC request:

- `AUTO_COMMIT`
- `CLIENT_COMMIT`

While AUTO_COMMIT commits each RPC message implicitly after sending it, a series of RPC messages sent in a unit of work (UOW) can be committed or rolled back explicitly using CLIENT_COMMIT mode.

The server is implemented and configured in the same way as for normal RPC.

Writing a Client

All methods for reliable RPC are available on the service class object. See description of class *Service* for details. The methods are:

- Service.SetReliableState
- Service.getReliableState
- Service.ReliableCommit
- Service.ReliableRollback
- Service.GetReliableId
- Service.GetReliableStatus

Example (this example is included as source in folder *examples\ReliableRPC\NetClient*)

Create Broker object and interface object.

```
Mail mail = new Mail();
mail.service.broker.logon();
```

Enable reliable RPC with CLIENT_COMMIT:

```
mail.SetReliableState(Service.ReliableState.RELIABLE_AUTO_COMMIT);
```

The first RPC message.

```
mail.Sendmail("mail receiver", "subject 1", "Text 1");
```

Check the status: get the message ID first and use it to retrieve the status.

```
StringBuilder reliableID = new StringBuilder();
StringBuilder reliableStatus = new StringBuilder();

mail.service.GetReliableID(ref reliableID);
mail.service.GetReliableStatus(reliableID, ref reliableStatus);
Console.Out.WriteLine("Reliable ID = " + reliableID.ToString());
Console.Out.WriteLine("Reliable Status = " + reliableStatus.ToString());
```

The second RPC message.

```
mail.Sendmail("mail receiver", "subject 2", "Text 2");
```

Commit the two messages.

```
mail.service.ReliableCommit();
```

Check the status again for the same message ID.

```
mail.service.GetReliableStatus(reliableID, ref reliableStatus);
Console.Out.WriteLine("Reliable ID = " + reliableID.ToString());
Console.Out.WriteLine("Reliable Status = " + reliableStatus.ToString());
```

The third RPC message.

```
mail.Sendmail("mail receiver", "subject 3", "Text 3");
```

Check the status: get the new message ID and use it to retrieve the status.

```
mail.service.GetReliableID(ref reliableID);
mail.service.GetReliableStatus(reliableID, ref reliableStatus);
Console.Out.WriteLine("Reliable ID = " + reliableID.ToString());
Console.Out.WriteLine("Reliable Status = " + reliableStatus.ToString());
```

Roll back the third message and check status.

```
mail.service.ReliableRollback();
mail.service.GetReliableStatus(reliableID, ref reliableStatus);

Console.Out.WriteLine("Reliable ID = " + reliableID.ToString());
Console.Out.WriteLine("Reliable Status = " + reliableStatus.ToString());

mail.service.broker.logoff();
```

Limitations

1. All program calls that are called in the same transaction (CLIENT_COMMIT) must be in the same IDL library.
2. It is not allowed to switch from CLIENT_COMMIT to AUTO_COMMIT in a transaction.
3. Messages (IDL programs) must have IN parameters only.

Writing a Server

There are no server-side methods for reliable RPC. The server does not send back a message to the client. The server can run deferred, thus client and server do not necessarily run at the same time. If the server fails, it throws an exception. This causes the transaction (unit of work inside the broker) to be cancelled, and the error code is written to the user status field of the unit of work.

Broker Configuration

A Broker configuration with `PSTORE` is recommended. This enables the Broker to store the messages for more than one Broker session. These messages are still available after Broker restart. The attributes `STORE`, `PSTORE`, and `PSTORE-TYPE` in the Broker attribute file can be used to configure this feature. The lifetime of the messages and the status information can be configured with the attributes `UWTIME` and `UWSTAT-LIFETIME`. Other attributes such as `MAX-MESSAGES-IN-UOW`, `MAX-UOWS` and `MAX-UOW-MESSAGE-LENGTH` may be used in addition to configure the units of work. See *Broker Attributes* in the administration documentation.

The result of the function `Service.GetReliableStatus` depends on the configuration of the unit of work status lifetime in the EntireX Broker configuration. If the status is not stored longer than the message, the function returns the error code 00780305 (`no matching UOW found`).

8 .NET Wrapper Reference

Attributes

Attribute classes are defined and implemented in the .NET Wrapper runtime and used in the C# client stub code to hold information extracted from the IDL file.

EntireXVersionAttribute

This attribute contains version information.

Example

```
[EntireXVersion("9.5.1.n")]
public class Example
```

LibraryAttribute

This attribute contains the library name.

Example

```
[Library("EXAMPLE")]
public class Example
```

BrokerAttribute

This attribute contains the Broker ID.

Example

```
[Broker("localhost:1971")]
public class Example
```

ServiceAttribute

This attribute contains the service name.

Example

```
[Service("RPC/SRV1/CALLNAT")]
public class Example
```

ProgramAttribute

This attribute contains the program name.

Example

```
[Program("CALC")]
public int Calculator(
  [SendAs(IdlType.A, Length=1f)][In] string operation,
  [SendAs(IdlType.I4)][In] int operand1,
  [SendAs(IdlType.I4)][In] int operand2
)
```

SendAsAttribute

This attribute contains type, length (fixed or dynamic) and dimension (fixed or dynamic) information.

Direction Attributes (In, Out)

These attributes contain direction information. They are supported natively by C#.

Example

```
[Program("HELLO")]
public void Hello(
  [SendAs(IdlType.A, Length=80f)][In] string client,
  [SendAs(IdlType.A, Length=80f)][In, Out] ref StringBuilder mail
)
```

Classes

The .NET Wrapper runtime defines and implements several generic service classes that are used in the generated C# client stub and by .NET client applications.

Broker

This class represents an EntireX Broker session and handles the connections to the Broker.

Constructors

```
public Broker()
```

Default Broker for default user.

The values for the default Broker and user are taken in the following order

- from the application's configuration file or
- from the [Broker] attribute of the client stub (Broker values only) or
- from hard-coded constants `localhost:1971` and `ERX-USER`.

```
public Broker(string hostName)
```

Broker on `hostName` for default user (`ERX-USER`).

```
public Broker(string hostName, string userName)
```

Broker on `hostName` for `userName`.

```
public Broker(string hostName, string userName, string token)
```

Broker on `hostName` for `userName` with `token`.

Methods

```
public void Logon()
```

Performs a logon to the Broker with the default user ID and password (that were set, for example, with the `UserID` and `Password` property).

```
public void Logon(string password)
```

Performs a logon to the Broker with the given password.

```
public void Logon(string password, string newPassword)
```

Performs a logon to the Broker with the given password and changes the `password` to `newPassword`

```
public void Logon(string userID, string token, string password)
```

Performs a logon to the Broker with the given user ID, token and password

```
public void Logoff()
```

Performs a logoff from the Broker.

Properties

```
public bool ForceLogon
```

Specifies whether force logon is performed. The default is false.

```
public char BrokerSecurity
```

Sets or retrieves the level and type of Broker security to be used.

'N' : no security
'Y' : default EntireX Security
'C' : user-specific security

```
public int CompressionLevel
```

Specifies what compression level should be used. Possible values are in the range 0 to 9.
The following values have a dedicated purpose.

0: do not compress
1: use compression method with best speed
6: use default compression
8: deflated
9: use best compression
The default value is 0 (no compression)

```
public int EncryptionLevel
```

Specifies what encryption level should be used. Possible values are:

0: no encryption
1: encrypt communication with the Broker
2: encrypt communication with the RPC server
The default value is 0.

```
public string BrokerID
```

Retrieves the Broker ID of the given Broker class instance. This property is read only.

```
public byte[] IAFToken
```

Sets or retrieves the IAF token of a given Broker instance.

```
public string Password
```

Sets the password of a given Broker class instance for subsequent authentication. This property is write only.

```
public byte[] SecurityToken
```

Sets or retrieves the security token of a given Broker class instance. The default value is null.

```
public string Token
```

Sets or retrieves the token of the given Broker class instance. The default value is null.

```
public string UserID
```

Sets or retrieves the user ID of the given Broker class instance for subsequent authentication.

Deprecated Properties

```
public Compress Compression
```

Please use the `CompressionLevel` property instead.

```
public bool Encryption
```

Please use the `EncryptionLevel` property instead.

Example

```
Broker broker = new Broker("ibm2:3762", "ERX-USER");
broker.Logon("ERX-PASS");
```

Service

Constructors

```
public Service()
```

Default service with default Broker.

```
public Service(string libraryName)
```

Service for given library with default Broker.

```
public Service(Broker broker)
```

Service for given Broker.

```
public Service(Broker broker, string trinity)
```

Service for given Broker and service name: `class/server/service` (for example `RPC/SRV1/CALLNAT`).

```
public Service(string Broker broker, string trinity, libraryName)
```

Service for given Broker, service name: `class/server/service` and library.

Methods

```
public int SetReliableState(int uReliableState)
```

Set the Reliable State. Possible values:

`RELIABLE_OFF (0)` - default value
`RELIABLE_AUTO_COMMIT (1)`
`RELIABLE_CLIENT_COMMIT (2)`

See *Reliable RPC for .NET Wrapper*.

```
public int ReliableCommit()
```

Do a commit in Reliable State `RELIABLE_CLIENT_COMMIT`.

```
public int ReliableRollback()
```

Do a rollback in Reliable State `RELIABLE_CLIENT_COMMIT`.

```
public int GetReliableID(ref StringBuilder ReliableID)
```

Get the `ReliableID`.

```
public int GetReliableStatus(StringBuilder ReliableID, ref StringBuilder ↩
ReliableStatus)
```

Get the Reliable Status. Possible values:

```
RECEIVED
ACCEPTED
DELIVERED
BACKEDOUT
PROCESSED
CANCELLED
TIMEOUT
DISCARDED
```

See *Broker ACI Fields* in the ACI Programming documentation for more information.

```
public void CloseConversation()
```

Close an RPC conversation.

```
public void CloseConversationCommit()
```

Close an RPC conversation and commit.

```
public void UserIDAndPassword(string user, string password)
```

Specify user ID and password for a service.

```
public void OpenConversation()
```

Open an RPC conversation.

```
public unsafe object Invoke ( string library , string method , params object[] ↩
objArray )
```

where	library	is the name of the class in the generated client stub
	method	the name of the method to be invoked
	objArray	the methods parameters as an array of objects - the array size must fit the parameter count of the method .

`Invoke` returns the result (if any) of the invoked method.

The initialisation of the parameter array follows the rules:

1. Parameters of type groups, structs and arrays have to be assigned as follows

```
int[] numbers = new int[10] ;
...
objArray[i] = numbers ;
```

2. [in,out] and [out] parameters of the simple data types bool, char, byte, sbyte, decimal, float, double, short, int and DateTime have to be assigned as follows:

```
int number = 4711 ;
...
objArray[i] = new Ref ( ref number )  ;
```

   where `Ref` is the class `SoftwareAG.EntireX.NETWrapper.Runtime.Ref`.

Note: The name of the class and the assembly name (file name) have to be identical. For each class, a separate assembly is required. All these assemblies have to be placed in the folder of the client executable or have to be configured according to the rules described in *Configuring the EntireX RPC Server for use with the .NET Framework*.

Properties

```
public Encoding CharacterEncoding
```

Define an encoding for character translation. Default is `System.Text.Encoding.GetEncoding(0)` `(current locale)`. See also the .NET Framework class library documentation for `System.Text.Encoding`.

```
public bool Encryption
```

Specify whether encryption is used. The default is false.

```
public bool NaturalLogon
```

Specify whether Natural logon should be performed. The default is false. If `NaturalLogon` is set to true but no `RPCUserID` and `RPCPassword` have been defined, the runtime uses the Broker user ID and password (provided the Broker password has been set with the `Password` property).

```
public Broker Broker
```

Sets or retrieves the Broker instance associated with the given Service instance.

```
public string RPCUserID
```

Sets or retrieves the RPC user ID of a given Service instance.

```
public string RPCPassword
```

Sets the RPC user password of a given Service instance.

```
public string ServerAddress
```

Retrieves the server address (class/server/service triplet) of a given Service instance.

```
public string Library
```

Sets or retrieves the library name of a given Service instance.

```
public Uint Timeout
```

Sets or retrieves the timeout value for a given Service instance. `Timeout = 0` is invalid. If 0 is set, a default of 50 seconds will be used.

Example

```
Service service = new Service( broker, "RPC/SRV1/CALLNAT", "EXAMPLE");
service.UserIDAndPassword("RPC-USER", "RPC-Password");
```

XException

Properties

```
public int errorCode
```

If an `XException` is thrown, `errorCode` contains the specific error code.

```
public string Message
```

If an `XException` is thrown, `Message` contains the specific error message. See *Message Class 2002 - .NET Wrapper* under *Error Messages and Codes*.

Example

```
try {
      ...
    } catch (EntireX.XException e) {
      Console.WriteLine( e.Message ) ;
    };
```

```
    Output: "02150148: EntireX Broker not active.
```

9 EntireX .NET Wrapper Application Configuration

Most applications require some configuration parameters that represent durable applications or user preferences.

The .NET framework includes configuration functionality that loads an application's configuration automatically at runtime without programmer intervention. For a standalone application, named, for example, *myapp.exe* you must name the configuration file (containing configuration settings in a given XML format) *myapp.exe.config*. The framework will then be able to load and parse the configuration file automatically when *myapp.exe* is run. For an ASP.NET Web application the configuration file is named *web.config*.

Assembly Versioning

.NET Framework assemblies support a strong versioning concept. The specific version of an assembly and the versions of dependent assemblies are recorded in the assembly's manifest. The versions of the dependent assemblies to be loaded at runtime are determined depending on the version policy in effect.

The default version policy is that applications run only with the exact versions of dependent assemblies they were built with. Thus applications that are deployed together with their dependent assemblies are not affected by newer or older versions of some of these assemblies. However, it is sometimes desirable to update an assembly with a newer version. In order to make this possible, the default version policy can be overridden by explicit version policies specified in configuration files, for example, the application configuration file (*<appname>.exe.config* or *web.config* for Web applications), the computer's machine configuration file (machine.config) or a publisher's policy file.

The following example shows a configuration file fragment that, when placed in a standalone application's *<appname>.exe.config* file or a Web application's web.config file or in the machine.config file, directs the .NET runtime loader to load version 9.5.1.n of the .NET Wrapper runtime whenever earlier versions in the range 7.1.1.0-7.2.1.73 are required.

```
<runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    <dependentAssembly>
      <assemblyIdentity name="SoftwareAG.EntireX.NETWrapper.Runtime"
publicKeyToken="645917c53ee5c617" />
      <bindingRedirect oldVersion="7.1.1.0-7.2.1.73" newVersion="9.5.1.n" />
      <codeBase version="9.5.1.n"
      ↩
href="file:///C:\SoftwareAG\EntireX\bin\SoftwareAG.EntireX.NETWrapper.Runtime.dll"/>
    </dependentAssembly>
  </assemblyBinding>
</runtime>
```

Note: The *<runtime>* configuration fragment must come after the *<configSections>* and *<appSettings>* sections of the configuration file, otherwise the .NET runtime will report errors.

See also the Microsoft .NET Framework documentation on assembly versioning.

Client Configuration

The .NET Wrapper Runtime supports the .NET framework configuration mechanism for several EntireX Broker and (RPC) Service class properties. By making use of this configuration mechanism, .NET Wrapper client applications can avoid constructing Broker and Service objects explicitly and leave this task to the .NET Wrapper Runtime.

There is one section group named EntireX with the two sections Broker and Service where you can specify the settings for EntireX .NET Wrapper Broker and Service class instances respectively.

Example

```
    <sectionGroup name="EntireX">  <!-- EntireX Configuration Section Group ↩
Definition -->
     <section name="Broker"  type="System.Configuration.NameValueSectionHandler" />
     <section name="Service" type="System.Configuration.NameValueSectionHandler" />
    </sectionGroup>
```

For an ASP.NET web.config configuration file, the parameters of the NameValueSectionHandler that processes the configuration must be specified in more detail.

```
<sectionGroup name="EntireX">
  <section name="Broker"  type="System.Configuration.NameValueSectionHandler,
    ↩
System,Version=2.0.0.0,Culture=neutral,PublicKeyToken=b77a5c561934e089,Custom=null" />
  <section name="Service" type="System.Configuration.NameValueSectionHandler,
    ↩
System,Version=2.0.0.0,Culture=neutral,PublicKeyToken=b77a5c561934e089,Custom=null" />
</sectionGroup>
```

Broker Configuration Section

If the default constructor Broker() is used to construct a Broker object, i.e. if there is no Broker name (or Broker ID) supplied, then the application's configuration file is examined for configuration settings to be taken as values. If no entry is found for a given setting name, the default values listed in the table below will apply.

The following can be configured for Broker instances:

Key	Description
name	Specifies the Broker name (or Broker ID). The default value is "localhost:1971". Only the URL-style broker ID is supported. See *URL-style Broker ID*.
userID	Specifies the user ID to be used to connect to the Broker; The default value is "NET-USER".
password	Specifies the password to be used to connect to the Broker. This setting is only considered when userID is also specified. The default value is "NET-PASS".
compression	Specifies whether the data sent to the Broker should be compressed. Possible values are: ■ NO_COMPRESSION (CompressionLevel=0) ■ BEST_COMPRESSION (CompressionLevel=9) ■ DEFAULT_COMPRESSION (CompressionLevel=6) this.compression=Compression.DEFAULT_COMPRESSION; ■ BEST_SPEED (CompressionLevel=1) ■ DEFLATED (CompressionLevel=8) The default value is NO_COMPRESSION. Use either Compression or CompressionLevel.
compressionLevel	Specifies what compression level should be used. Possible values are in the range 0 to 9 (see CompressionLevel property in the Broker class). Use either Compression or CompressionLevel.
encryptionLevel	Specifies the encryption level used for the Broker. Possible values are 0,1,2. See `ENCRYPTION-LEVEL`.
forceLogon	Specifies whether a forceLogon should be performed. Possible values are "true" and "false". The default value is "false".
token	Specifies a token value to be used in conjunction with the user ID. The default value is "".

Broker Configuration Example

```
<Broker>
  <!-- EntireX Broker Configuration -->
  <add key="name"                    value="localhost:1971" />
  <add key="locationTransparency"    value="false" />
  <add key="locationTransparencySet" value="DefaultSet" />
  <add key="userID"                  value="NET-USER" />
  <add key="password"                value="NET-PASS" />
  <add key="compression"             value="NO_COMPRESSION" />
  <add key="encryptionLevel"         value="0" />
  <add key="forceLogon"              value="false" />
  <add key="token"                   value="top secret" />
 </Broker>
```

Service Configuration Section

If the default constructor Service() is used to construct a Service object, i.e. there is no Service name (class/server/service) supplied, then the application's configuration file is examined for configuration settings to be taken as values. If no entry is found for a given setting name, then the default values apply as listed below.

The following can be configured for Service instances.

Key	Description
name	Specifies the name of the service. Default value is "RPC/SRV1/CALLNAT".
encryption	Specifies whether the data sent to the RPC Server should be encrypted. Possible values are "true" and "false". The default value is "false". **Note:** This setting has been deprecated!
naturalLogon	Specifies whether a Natural logon should be performed. Possible values are "true" and "false". The default value is "false".
userID	Specifies the user ID to be used to connect to the RPC Server.
password	Specifies a password to be used to connect to the RPC Server. This setting is only considered when userID is also specified.

Service Configuration Example

```
<Service>
  <!-- EntireX Service Configuration -->
  <add key="name"                    value="RPC/SRV1/CALLNAT" />
  <add key="locationTransparency"    value="false" />
  <add key="locationTransparencySet" value="DefaultSet" />
  <add key="libraryName"             value="" />
  <add key="naturalLogon"            value="false" />
</Service>
```

An Example Configuration File

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <sectionGroup name="EntireX">  <!-- EntireX Configuration Section Group ↩
Definition -->
     <section name="Broker"  type="System.Configuration.NameValueSectionHandler" />
     <section name="Service" type="System.Configuration.NameValueSectionHandler" />
    </sectionGroup>
  </configSections>
  <EntireX>
    <!-- EntireX Configuration Section -->
    <Broker>
      <!-- EntireX Broker Configuration -->
      <add key="name"                    value="localhost:1971" />
      <add key="locationTransparency"    value="false" />
      <add key="locationTransparencySet" value="DefaultSet" />
      <add key="userID"                  value="NET-USER" />
      <add key="password"                value="NET-PASS" />
      <add key="compression"             value="NO_COMPRESSION" />
      <add key="encryptionLevel"         value="0" />
      <add key="forceLogon"              value="false" />
      <add key="token"                   value="top secret" />
    </Broker>
    <Service>
      <!-- EntireX Service Configuration -->
      <add key="name"                    value="RPC/SRV1/CALLNAT" />
      <add key="locationTransparency"    value="false" />
      <add key="locationTransparencySet" value="DefaultSet" />
      <add key="libraryName"             value="" />
      <add key="naturalLogon"            value="false" />
    </Service>
  </EntireX>
  <appSettings>
    <!-- other app settings go here -->
  </appSettings>
</configuration>
```

Server Configuration

The requirements for the .NET RPC Server are: the EntireX RPC Server (*rpcserver.exe*), *dotNetServer.dll* and *dotNetServer.cfg* from the EntireX bin directory, Software AG's .NET Wrapper (*SoftwareAG.EntireX.NETWrapper.Runtime.dll*) and a .NET Server DLL (Assembly).

See *Administering the EntireX RPC Server* in the Windows administration documentation.

Starting the RPC Server

Before starting the EntireX RPC server, ensure that all dynamically loaded objects (server stubs and server) can be accessed using the search path.

To start the EntireX RPC server manually

- Use the format

```
RPCserver CFG=<name> [-option] [Brokerid] [Class] [ServerName] [Service]
```

  where *<name>* determines the configuration file in use.

  Options:

  - -smhport number
    Sets the RPC server parameter smhport to number. Typically used by SMH Facility.
  - -serverlog *<file>*
    Defines an alternative log file for Window services. Typically used by Windows Services. See *Running an EntireX RPC Server as a Windows Service* in the Windows administration documentation.
  - -s[ilent]: Run server in silent mode, that is: no terminal input will be required (e.g. acknowledge error messages). The job will terminate automatically. Recommended for background jobs.
  - -TraceDestination *<file>*
    Set the trace destination parameter.
  - -TraceLevel None.Standard.Advanced. Set the trace level parameter.

  Note: The server input arguments will be resolved from left to right. Thus parameters that can be applied on the command line as well in the configuration file may be overriden.

To start the EntireX RPC server using Windows services

- See *Running an EntireX RPC Server as a Windows Service* in the Windows administration documentation.

Note: For reasons of compatibility with versions before 5.1.1, the old command to start the server

```
RPCserver <Brokerid> <Class> <ServerName> <Service>
```

will continue to be supported. However, a server started with this call will use the default parameters. Parameters other than `Broker ID`, `Class`, `ServerName`, `Service` require the CFG= form of the server start command.

To start the EntireX RPC server using System Mangagement Hub

1. See *Administering the EntireX RPC Servers using System Management Hub* in the UNIX and Windows administration documentation for information on adding an EntireX RPC server to the System Management Hub.
2. The System Management Hub facility “Adding a Local RPC Server” will use the batch script *startcserver.bat* of the EntireX Installation to apply the server parameters. Change the batch script according to your system installation or add parameters to the System Management Hub “Start Command” input property.

Stopping the Server

To stop the EntireX RPC server

- Use one of the System Management Hub functions **Deregister a Service** or **Deregister a Server**. This method ensures that the deregistration from the Broker will be complete and correct.

  See also *EntireX RPC Server Return Codes* under *Error Messages and Codes*.

A file that corresponds to dotNetServer.cfg must be used as configuration file (configuration of the EntireX RPC Server for use with the .NET Wrapper). The .NET Server Assembly (containing one IDL library) will then be loaded when a program from this library is first accessed. With the .NET Framework there are two distinct ways to locate the .NET Server Assembly:

1. If the *rpcserver.exe*, our .NET Wrapper and the .NET Server Assembly are all in the same directory, the .NET Server Assembly will also be loaded from this directory.
2. If *rpcserver.exe*, our .NET Wrapper and the .NET Server Assembly are in different libraries, the *rpcserver.exe* must be configured in the context of the .NET Framework (Configuring the EntireX RPC Server for the .NET Framework). The Server Assembly must also have a strong name. This

is described in the Microsoft documentation for the .NET Framework (in our example server in *EntireXDir**Examples**NET Wrapper**server**calc* the Server Assembly has a strong name and in ***Configuring the EntireX RPC Server for use with the .NET Framework***).

Whereas method 1 is very well suited for test and development purposes, method 2 is to be recommended for more complex production environments. The user can decide which method to use.

Configuring the EntireX RPC Server for use with the .NET Wrapper

For the EntireX RPC Server to function with the .NET Wrapper properly, the file *dotNetServer.cfg* from the installation or a similar file, should be used. The entries

- Class=RPC
- ServerName=dotNetServer
- Service=CALLNAT
- CallExit=dotNetServer
- RunOption=Reset

should not be changed or deleted. No other run options should be added. All of the other settings for the configuration of an EntireX RPC Server are available. See *Administering the EntireX RPC Server* in the Windows administration documentation.

Note: Any server name can be used. However, we recommend using the name *dotNetServer* to distinguish this one from any other RPC servers.

Configuring the EntireX RPC Server for use with the .NET Framework

Due to an incompatibility of the .NET Framework 2.0 to the earlier versions, we had to add an additional section in the configSections part of rpcserver.exe.config:

```
<configSections>
  <!-- EntireX Configuration Section Group Definition -->
  <sectionGroup name="EntireX">
    <section name="Assemblies" type="System.Configuration.NameValueSectionHandler, ↩
System, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089, ↩
Custom=null" />
  </sectionGroup>
</configSections>
<EntireX>
  <!-- EntireX Assembly Configuration -->
  <Assemblies>
    <add key="SoftwareAG.EntireX.NETWrapper.Runtime" ↩
value="C:\SoftwareAG\EntireX\bin\SoftwareAG.EntireX.NETWrapper.Runtime.dll" />
  </Assemblies>
</EntireX>
```

where the location of our .NET Runtime is replaced by the location used in your EntireX installation. Add an entry in the Assemblies section for each of your server assemblies:

```
<add key="MyAssembly", value="MyLocation"/>
```

where *MyAssembly* and *MyLocation* represent the name and location of your server assembly. If versioning is required for your assemblies, follow the rules under ***Assembly Versioning***.