Writing Applications with the .NET Wrapper

This chapter covers the following topics:

- Writing a Client Application
- Writing a Server DLL
- Deploying Wrapped .NET Servers
- Creating ASP.NET Web Services
- Using Internationalization 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

The .NET Wrapper generates C# sources from an IDL file. If there is a related client-side mapping file (Natural | COBOL), this is also used (internally).

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

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* 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
```