webMethods Package for Microsoft .NET Client API Programmer's Guide

Version 9.0

March 2013

WEBMETHODS

This document applies to webMethods Microsoft Package 9.0 and to all subsequent releases.

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

Document ID: ADAPTER-MIP-CLIENTAPI-PG-90-20210422

Table of Contents

About this Guide

This guide describes the installation and use of the webMethods Microsoft Package. This package facilitates use of the Microsoft .NET application platform in conjunction with webMethods products. You can administer and manage the webMethods Microsoft Package from any standard browser. Using the webMethods for Microsoft Plug-in, you can create services from existing .NET assemblies. Using the webMethods Add-In for Microsoft Visual Studio, you can browse services on Integration Server and generate the code to invoke those services.

To use this guide effectively, you must:

- Understand the basic concepts of Microsoft .NET and Microsoft .NET Framework.
- Be familiar with the setup and operation of the webMethods Integration Server.
- Have a general idea about how to perform tasks with Software AG Designer.

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Narrowfont	Identifies service names and locations in the format *folder.subfolder.service*, APIs, Java classes, methods, properties.
Italic	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at http://documentation.softwareag.com.

Software AG Empower Product Support Website

If you do not yet have an account for Empower, send an email to empower@softwareag.com with your name, company, and company email address and request an account.

Once you have an account, you can open Support Incidents online via the eService section of Empower at https://empower.softwareag.com/.

You can find product information on the Software AG Empower Product Support website at https://empower.softwareag.com.

To submit feature/enhancement requests, get information about product availability, and download products, go to Products.

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the Knowledge Center.

If you have any questions, you can find a local or toll-free number for your country in our Global Support Contact Directory at https://empower.softwareag.com/public_directory.aspx and give us a call.

Software AG TECHcommunity

You can find documentation and other technical information on the Software AG TECHcommunity website at http://techcommunity.softwareag.com. You can:

- Access product documentation, if you have TECHcommunity credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

1 Generating Microsoft .NET Clients

Overview

The code samples in this chapter show two different methods of invoking a selected webMethods service from the context of the Microsoft .NET environment. The invocation uses the webMethods .NET API. The webMethods .NET Client API functionality is supplied in a set of DLLs. You need to reference these DLLs by means of a `using` statement. The examples in this chapter do not create whole executables; rather, they show how to generate objects you can use in your own code.

For the first example, the webMethods add-in for Microsoft Visual Studio allows you to select a service from a running Integration Server and generate a C# or Visual Basic class. This class encapsulates the integration service, allowing for interaction with the service in an object oriented way. This example is described in "Invoking a Service using Visual Studio" on page 11.

If you are familiar with the Java Client API for Integration Server, you may prefer to use the standard IData pipeline interface for invoking Integration Services. For more on using IData, see "Invoking a Service using Java Client API from Designer" on page 17.

Note:
You can generate only C# code using the Java Client API for Integration Server 9.0 and higher. Visual Basic is not supported with Integration Server 9.0 and higher.

The Structure of Client Code

The .NET sample code is generated using the same structure as is used for generation of other language samples. This structure breaks essential elements out into separate routines. For purposes of simplification, we condense the essential elements of those samples into a more concise form. The essential elements of a .NET webMethods client are:

1. Import library references by means of the `using` statement.
2. Create a server context object.
3. Connect the server context to a running Integration Server.
4. Populate the service inputs.
5. Invoke one or more services.
6. Retrieve the service outputs.
7. Disconnect from Integration Server.

This same set of steps applies to both examples described in this chapter.

2 Invoking a Service using Visual Studio

Using webMethods Add-In for Microsoft Visual Studio

Using the webMethods add-in for Microsoft Visual Studio, you can select a service from a running Integration Server and generate a C# or Visual Basic class, as shown in "Generating a C# Client Code in Visual Studio" on page 12. This sample shows how you would go about creating a console application, but does not attempt to depict the end-to-end actions needed to create an executable.

This sample requires that you add two references to the .NET project, which you can find in the webMethods Add-In for Microsoft Visual Studio installation directory.

The files are CGUTIL.dll and wmClientAPI.dll.

This sample is based on the WmPublic.pub:concat service delivered with Integration Server.

Generating a C# Client Code in Visual Studio

Using the webMethods add-in for Microsoft Visual Studio, you can connect to a running Integration Server and generate an object from an integration service. This sample uses the concat service.

To create a C# file from the concat service

1. Open Visual Studio .NET.

2. Open a new .NET project using the **Console Application** template.

3. On the **Tools** menu, click **webMethods add-in for Microsoft Visual Studio**.

   Note:
   If this menu item is not visible, make sure you have installed the add-in. For information on installing the add-in, see the *webMethods Package for Microsoft .NET Installation and User's Guide* .

4. In the **webMethods add-in for Microsoft Visual Studio** dialog box, provide the following information about the instance of Integration Server to which to connect:

In this field...	Type this...
webMethods Integration Server	Integration Server host name and port in the format *host*:*port*.
UserID	Name of a valid user account on this Integration Server.
Password	Password for the user account. Passwords are case-sensitive.

5. Click **Connect**. The webMethods add-in for Microsoft Visual Studio window is displayed. This window contains a tree view of packages and services on the Integration Server to which you are connected.

Tip:
The add-in opens as a floating window. You can dock the window by dragging it to the Visual Studio .NET toolbar.

6. In the Solution Explorer, select the **Project** under which you want to create the C# client code. If the Solution Explorer is not already open, in the **View** menu, click **Solution Explorer**.

7. In the tree view of packages, go to the WmPublic.pub.string:concat service.

8. Right-click the concat service and, click **Generate Code**, and then click **C# Code**.

9. If the Solution Explorer is not already open, in the **View** menu, click **Solution Explorer**. The concat.cs file should be visible in the Solution Explorer panel.

10. Add the references by doing the following:

    a. In the Solution Explorer panel, right-click the **References** node, and then click **Add Reference**.

    b. Click **Browse** tab and go to the default add-in directory for Visual Studio, where the webMethods Add-In for Microsoft Visual Studio is installed.

       Typically, the webMethods Add-In for Microsoft Visual Studio is installed in the directory `C:\Documents and Settings\All Users\Application Data\Microsoft\MSEnvShared\Addins\`

    c. Select CGUTIL.dll and wmClientAPI.dll.

    d. Click **OK**. The two DLLs should now appear under the **References** node in the Solution Explorer panel.

The following sections provide brief descriptions of the sample code as they relate to "Generating Microsoft .NET Clients" on page 9:

You can find the complete code sample in "Complete Generated Code Sample" on page 15.

List Packages to be Used (Generated Code)

The first stage of the code sample contains `using` statements that specify the .NET packages to use:

```
using System;
```

```
using webMethods.ClientAPI;
```

Class Declaration (Generated Code)

The second stage of the code creates the context in which the client operates:

```
namespace Pub.String
{
  /// <summary>
  /// Sample client demonstrating invocation of the 'concat' Integration   ///
Service using an object generated by the webMethods Visual
  /// Studio Add-in.
  /// </summary>
  class ClientSample2
```

Connect to Integration Server (Generated Code)

The third stage creates the connection to Integration Server:

```
{
      String returnString = null;
      // create our connection context with the Integration Server
      Context serverContext = new Context();
      // connect to the server
      try
      {
        serverContext.connect( "localhost:5555", "Administrator", "manage" );
      }
      catch( Exception ex )
      {
        Console.WriteLine("Connection to server failed, reason=" + ex );
        return null;
      }
```

Populate the Service Inputs (Generated Code)

The fourth stage of the code creates input data variables:

```
// create the Concat service object
      //(created by the Add-in for the pub.string:concat service)
      Concat concatService = new Concat();
      // populate the inputs
      concatService.in_inString1 = string1;
      concatService.in_inString2 = string2;
```

Invoke the Service and Retrieve the Output (Generated Code)

The fifth and sixth stages are to invoke the service and retrieve the output:

```
// invoke the service
      concatService.invoke( serverContext );

      // extract the output
      returnString = concatService.out_value;
```

Disconnect from Integration Server (Generated Code)

The seventh stage is to disconnect from Integration Server:

```
// disconnect from the server
         serverContext.disconnect();
```

Complete Generated Code Sample

The following sample shows the all of the C# code used in "Generating a C# Client Code in Visual Studio" on page 12 but does not attempt to depict the end-to-end actions needed to create an executable:

```
using System;
using webMethods.ClientAPI;

namespace Pub.String
{
  /// <summary>
  /// Sample client demonstrating invocation of the 'concat' Integration   ///
Service using an object generated by the webMethods Visual
  /// Studio Add-in.
  /// </summary>
  class ClientSample2
  {
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
      // execute the generated service object
      String result = concatStrings( "testString1", "testString2" );
      Console.WriteLine( "concat=" + result );
    }

    /**
     * Concatentate strings using the Add-in generated class for the
     * pub.strings:concat service
     */
    public static String concatStrings( String string1, String string2 )
    {
      String returnString = null;

      // create our connection context with the Integration Server
      Context serverContext = new Context();

      // connect to the server
      try
      {
        serverContext.connect( "localhost:5555", "Administrator", "manage" );
      }
      catch( Exception ex )
      {
        Console.WriteLine("Connection to server failed, reason=" + ex );
        return null;
      }
```

```
      // create the Concat service object
      //(created by the Add-in for the pub.string:concat service)
      Concat concatService = new Concat();

      // populate the inputs
      concatService.in_inString1 = string1;
      concatService.in_inString2 = string2;

      // invoke the service
      concatService.invoke( serverContext );

      // extract the output
      returnString = concatService.out_value;

      // disconnect from the server
      serverContext.disconnect();

      return returnString;

    }

  }
}
```

3 Invoking a Service using Java Client API from Designer

Using the Java Client API for Integration Server

Using the Java Client API for Integration Server, you can use the standard IData pipeline interface for invoking Integration Services. Using Software AG Designer, you can generate a C# class to invoke the service from a running Integration Server. This sample does not attempt to depict the end-to-end actions needed to create an executable.

This sample requires that you add one reference to the .NET project, which you can find in the webMethods Add-In for Microsoft Visual Studio installation directory or the *Integration Server_directory* \lib directory.

The file is wmClientAPI.dll.

This sample is based on the Integration Server built-in service WmPublic.pub.math:addInts.

Generating a C# Client Code from Designer

Using Designer, you can generate C# client code for invoking the integration service. This sample uses the addInts service.

Important:
Visual Basic is not supported in Integration Server 9.0 and higher.

To create a C# file from the addInts service

1. In Designer, select the addInts service and right-click.
2. Select **Generate Code...** from the menu.
3. In the **Code Generation** wizard, select the option **For calling this service from a client** and click **Next**.
4. Select **C# for .NET** from the programming languages, and specify the folder to save the C# file.

The following sections provide brief descriptions of the sample code as they relate to "Generating Microsoft .NET Clients" on page 9:

You can find the complete code sample in "Complete IData Code Sample" on page 20.

List Packages to be Used (IData)

The first stage of the code sample contains `using` statements that specify the .NET packages to be used:

```
using System;
using webMethods.ClientAPI;
using webMethods.ClientAPI.Data;
```

Class Declaration (IData)

The second stage of the code creates the context in which the client operates:

```
namespace IDataClientSample
{
  /// <summary>
  /// Sample client demonstrating invocation of an Integration Service
  /// using the standard IData pipeline mechanism
  /// </summary>
```

Connect to Integration Server (IData)

The third stage creates the connection to Integration Server:

```
{
      String returnString = null;
      // create our connection context with the Integration Server
      Context serverContext = new Context();
      // connect to the server
      try
      {
        serverContext.connect( "localhost:5555", "Administrator", "manage" );
      }
      catch( Exception ex )
      {
        Console.WriteLine("Connection to server failed, reason=" + ex );
        return null;
      }
```

Populate the Service Inputs (IData)

The fourth stage of the code creates input data variables. Service inputs are passed to Integration Server as an IData object. This code fragment demonstrates construction of the input IData pipeline:

```
// create the service inputs as an IData object
      IData inputIData = IDataFactory.create();
      // get a data cursor for our input IData
      IDataCursor inputCursor = inputIData.getCursor();
      // the addInts service takes 2 inputs: num1 and num2 (as strings)
      IDataUtil.put( inputCursor, "num1" , num1.ToString() );
      IDataUtil.put( inputCursor, "num2" , num2.ToString() );
```

```
      // done with the input cursor, destroying it only cleans up the
      // cursor resources, the IData object is left unaffected
      inputCursor.destroy();
```

Invoke the Service and Retrieve the Output (IData)

The fifth and sixth stages are combined in the serverContext.invoke method, which invokes the service with the IData input object and receives the reply as an Idata output object:

```
{
        // Invoke the service
        IData outputIData = serverContext.invoke( serviceFolder,
          serviceName, inputIData );
        // get a data cursor for out output IData
        IDataCursor outputCursor = outputIData.getCursor();
        // the addInts service has one return variable, "value" as string
        // however we can retrieve the value as an int using the IDataUtil
        // helper method getInteger();
        sum = IDataUtil.getInt( outputCursor, "value", 0 );
      }
      catch( Exception ex )
      {
        Console.WriteLine("The service invoke failed, reason=" + ex );
      }
```

Disconnect from Integration Server (IData)

The seventh stage is to disconnect from Integration Server, as shown in this fragment:

```
// disconnect from the server
      serverContext.disconnect();
```

Complete IData Code Sample

The following sample shows the all of the C# code used in "Generating a C# Client Code from Designer " on page 18 but does not attempt to depict the end-to-end actions needed to create an executable:

```
using System;
using webMethods.ClientAPI;
using webMethods.ClientAPI.Data;


namespace IDataClientSample
{
  /// <summary>
  /// Sample client demonstrating invocation of an Integration Service
  /// using the standard IData pipeline mechanism
  /// </summary>
  class ClientSample1
  {
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
```

```
static void Main(string[] args)
{
  // invoke the sample service using an IData pipeline
  int sum = addInts( 11, 22 );
  Console.WriteLine( "sum=" + sum );
}

/**
 * Add two integers together by invoking the pub.math:addInts service
 * using an IData pipeline
 */
public static int addInts( int num1, int num2 )
{
  // the resulting sum of the 2 integers
  int sum = 0;

  // create our connection context with the Integration Server
  Context serverContext = new Context();

  // connect to the server
  try
  {
    serverContext.connect( "localhost:5555", "Administrator", "manage" );
  }
  catch( Exception ex )
  {
    Console.WriteLine("Connection to server failed, reason=" + ex );
    return 0;
  }

  // create the service inputs as an IData object
  IData inputIData = IDataFactory.create();
  // get a data cursor for our input IData
  IDataCursor inputCursor = inputIData.getCursor();

  // the addInts service takes 2 inputs: num1 and num2 (as strings)
  IDataUtil.put( inputCursor, "num1" , num1.ToString() );
  IDataUtil.put( inputCursor, "num2" , num2.ToString() );

  // done with the input cursor, destroying it only cleans up the
  // cursor resources, the IData object is left unaffected
  inputCursor.destroy();

  try
  {
    // Invoke the service
    IData outputIData = serverContext.invoke( pub.math,
      addInts, inputIData );
    // get a data cursor for out output IData
    IDataCursor outputCursor = outputIData.getCursor();

    // the addInts service has one return variable, "value" as string
    // however we can retrieve the value as an int using the IDataUtil
    // helper method getInteger();
    sum = IDataUtil.getInt( outputCursor, "value", 0 );

  }
  catch( Exception ex )
  {
    Console.WriteLine("The service invoke failed, reason=" + ex );
```

```
      }

      // disconnect from the server
      serverContext.disconnect();


      return sum;
    }

  }
}
```

Notes About IData and IDataUtil

IData can store any data object that is an instance of the Common Data Types, arrays with two or less dimensions, or instances of any class derived from System.Object. However, instances of custom classes may cause errors when used to invoke an integration service. Integration Server runs as a Java process; any custom objects passed to an integration service are meaningless because Integration Server does not have a definition of that class. The same holds true for services that attempt to return an instance of a custom Java class. The .NET client does not have a class definition with which to create the Java class.

The Client API includes the helper class IDataUtil for extracting variables from IData with common data types. For example, getString() extracts an object from an IData object and casts it as a String. Similar methods exist for integers, arrays, and so forth. For further details, see the IDataUtil class documentation as explained in "Microsoft .NET Client API Documentation" on page 24.

Output variables that are native arrays are retrieved from the output IData as System.Array instances, not as their native array type. For example, input data can be entered as a native array type:

```
IDataUtil.put( cursor, "testArray", new int[2] {1 ,2 } );
```

However, when retrieved from IData, the array is of type System.Array.

```
System.array testArray = IDataUtil.getArray( cursor, "testArray");
```

4 Accessing Microsoft .NET Client API Documentation

Microsoft .NET Client API Documentation

The Client API documentation is installed as part of the package. These documents are available in a format similar to Javadoc and can be found in the Resources section of the webMethods Microsoft Package administration page.

To access the webMethods Microsoft Package from a connection to Integration Server

1. Start your browser.

2. Point your browser to the host and port for an instance of Integration Server where the webMethods Microsoft Package is installed. Use the syntax http://*host:port*. For example, if Integration Server is running on the default port on a machine named myhost, you would type http://myhost:5555.

   If Integration Server is running, you are prompted for a user name and password.

3. Log on with a user name that has administrator privileges. For information on maintaining administrator privileges, see the *webMethods Integration Server Administrator's Guide* for your release.

4. In the **Solutions** menu of the navigation frame, click **Microsoft Package**.

5. Expand the **Resources** node and click **Client API Doc**.