Web Services API Developer Guide

Contents

Web Services API Developer Guide.. 3
Quick Start...4
Examples of the Web Service API Implementation... 13
Exporting Warehouse Data... 14
Exporting Stock Items...16
Simulating the Behavior of Add Buttons on the Purchase Receipts Form................ 19
Copying a Sales Order...24
Adding a New Cash Transaction Document... 27
Adding Records to the Business Accounts and Opportunities Forms....................... 29
Importing of Data With an Image Into the Journal Transactions Form..................... 32
Exporting of Data With an Image From the Journal Transactions Form...................35

Web Services API Developer Guide

The Acumatica Web Services Application Programming Interface (API) provides a fast, reliable, and convenient way of exposing business functionality and data managed by an Acumatica application for integration with any external business and operation support system. The Acumatica API is based on web service standards, such as SOAP and WSDL, and can be accessed with almost any current programming environment or integration tool. By using the development environment you are familiar with, you can easily create a client application that accesses the Acumatica Studio application through standard web services protocols to do any of the following:

- Authorize the programmer with the server running the Acumatica application
- Get query and access information from the Acumatica application
- Import information into the Acumatica application
- Create, update, and delete objects in the Acumatica application
- Execute some long-running processes and perform administrative tasks

Every operation that uses the Acumatica API is executed through the same business logic layer as the user interface.

Web Services API Overview

Acumatica introduces a simple, streamlined way of interacting with its web services. The system automatically generates a WSDL file describing the operations (services) and list of parameters and objects; you can access this file through the *Web Services* (SM.20.70.40) form.

You can implement advanced integration scenarios involving operations on one or more forms by using the new web services configuration form to generate custom WSDL files.

All the functionality of the application is available through the Web Services API; however, the functionality and information that will be exposed and available to the web services client depends on the access rights granted to the user logged in as a client to the Acumatica ERP instance.

Web Services Calls

To execute the API call, you need to prepare the SOAP message and send it to the remote server that provides web services by using the HTTP/HTTPS protocol.

To simplify this process, most development environments (such as Microsoft Visual Studio and NetBeans) support importing of the WSDL definition file and provide automation tools for the creation of proxy classes. This approach enables you to access the object model in a convenient and familiar way, while ensuring compile-time verification of the web services calls.

Web Services API Objects

Interaction with the API is made through an object called **Screen**. This object acts as a gateway between the web services client and Acumatica, so that you can log in and retrieve, insert, update, or delete data, as well as perform any action that may be exposed by the form.

The preparation and execution of web services calls is facilitated by the **Content** object, which you can retrieve by calling the *GetSchema()* API function. This function returns an object that closely matches the way the form is presented to the end user. Each area on the form is mapped to an object in the **Content** object. For example, the Account Settings area in the **General Info** tab of the *Customers* (AR.30.30.00) form is defined in the **GeneralInfoAccountSettings** object. This object exposes a public property for every field in this area. Actions that can be performed in the form are exposed in a property called **Actions**. The class diagram below illustrates the relationship between the **Screen** and **Content** objects and associated areas of the **Content** object.

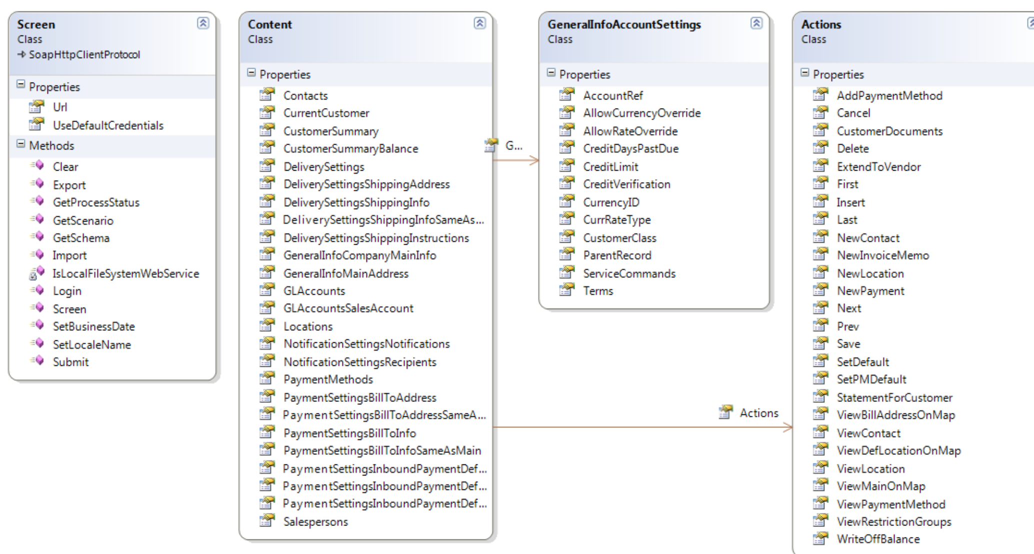


Figure: Sample web service class diagram

To execute an API call, you must build an array of commands and submit it to the form by calling the *Submit()* function. To process batch import and export operations, you define a scenario and use the *Import()* and *Export()* functions.

Quick Start

This mini-tutorial will help you get started with the Acumatica Web Services Application Programming Interface (API). To begin working with the Web Services API, perform the following steps:

- *Generate and Locate the WSDL File of the Web Services*
- *Import the WSDL File of the Web Services Into the Development Environment*
- *Review and Use the Code From the Sample Project*

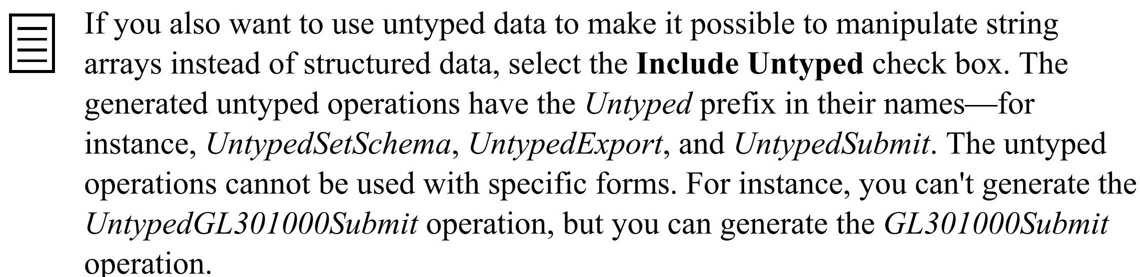
Step 1. Generate and Locate the WSDL File of the Web Services

Acumatica automatically generates a WSDL file describing the operations (services) and an XML description of parameters and objects for a form or multiple forms. You can access this file through the *Web Services* (SM.20.70.40) form of the Acumatica ERP application.

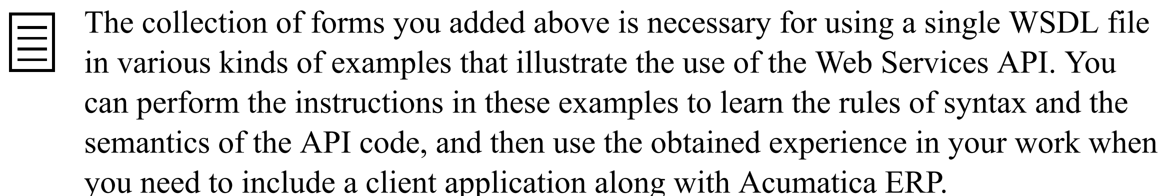
For more information about the WSDL standard, see *Web Services Description Language (WSDL) 1.1*.

To create a WSDL file for multiple forms, perform the following actions:

1. On the Web Services form, click **Add New Record** on the form toolbar, and type the **Service ID** name (for instance, *APITEST*, as shown in the figure below).
2. Keep the **Import**, **Export**, and **Submit** check boxes selected (as they are by default), and leave the **Include Untyped** check box cleared. Click **Save**.

   If you also want to use untyped data to make it possible to manipulate string arrays instead of structured data, select the **Include Untyped** check box. The generated untyped operations have the *Untyped* prefix in their names—for instance, *UntypedSetSchema*, *UntypedExport*, and *UntypedSubmit*. The untyped operations cannot be used with specific forms. For instance, you can't generate the *UntypedGL301000Submit* operation, but you can generate the *GL301000Submit* operation.
3. Click **Add Row** on the table toolbar, and then add the value for the **Screen ID** column by using the lookup window and finding the *Payments and Applications* (AR.30.20.00) form.
4. Repeat the previous step to add each of the following forms, as shown in the figure below: *Customers* (AR.30.30.00), *Transactions* (CA.30.40.00), *Leads* (CR.30.10.00), *Contacts* (CR.30.20.00), *Business Account* (CR.30.30.00), *Opportunities* (CR.30.40.00), *Journal Transactions* (GL.30.10.00), *Stock Items* (IN.20.25.00), *Warehouses* (IN.20.40.00), *Transfers* (IN.30.40.00), *Purchase Receipts* (PO.30.20.00), *Sales Orders* (SO.30.10.00), and *Shipments* (SO.30.20.00). Click **Save** again.

   The collection of forms you added above is necessary for using a single WSDL file in various kinds of examples that illustrate the use of the Web Services API. You can perform the instructions in these examples to learn the rules of syntax and the semantics of the API code, and then use the obtained experience in your work when you need to include a client application along with Acumatica ERP.

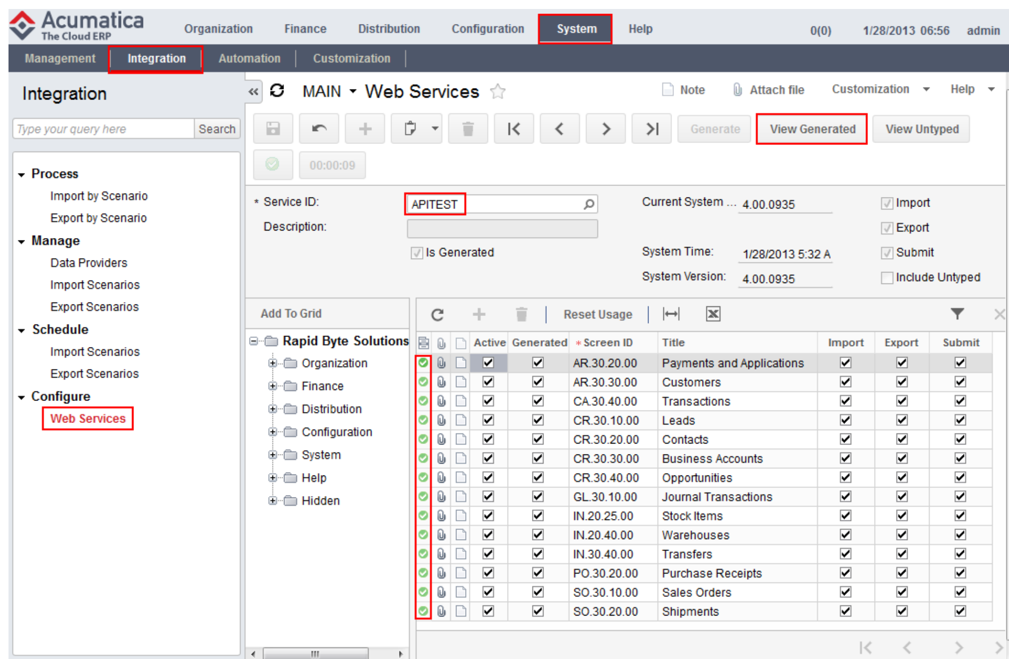


Figure: Creating the WSDL file

5. On the form toolbar, click **Generate** to start the process of generating the WSDL file. After the process is successfully completed, you can see the green flags in the leftmost column for each table row (that is, for each form).

6. Optional: Click **View Generated** to open the new window with the list of operations that are supported by the Acumatica Web Services API, as illustrated in the figure below. Note that some operations are bound with specific forms, because these operations support the particular structure of the appropriate form. To see the examples of SOAP client requests and HTTP server responses that can be implemented by using the appropriate operation, click any item.

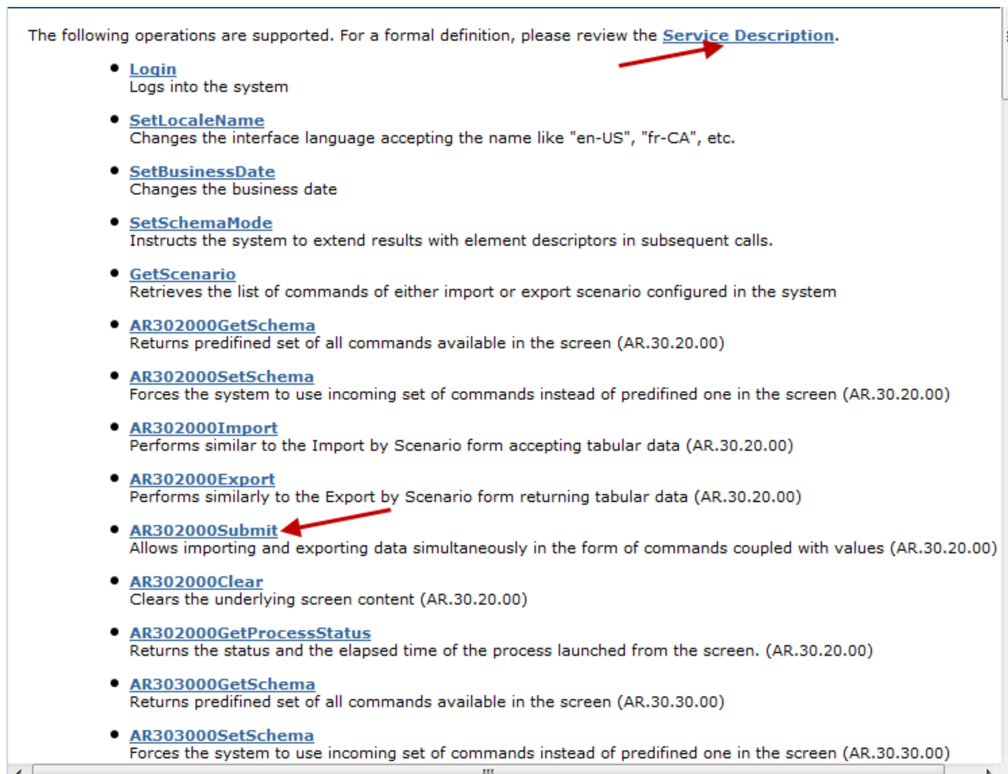


Figure: The list of available operations

7. Optional: Return to the previous screen, and click the **Service Description** reference to see the XML description of the generated WSDL file. A fragment of this file is shown in the figure below.
8. Close the window and return to the application.

Figure: The XML description of the generated WSDL file

To find the latest version of the WSDL file, use the following URL:

```
http://{domain}/Soap/{name}.asmx?WSDL
```

Replace *domain* with the actual URL path to your application and *name* with the ID of the web service. For example, the valid URL to access the *Customers* form could be either of the following, with the latter for the local Acumatica ERP instance:

```
http://www.acumatica.com/Demo/Soap/APITEST.asmx?WSDL
http://localhost/WebAPIVirtual/Soap/APITEST.asmx?WSDL
```

The WSDL file automatically generated by the system includes all the changes implemented to the application logic and its database structure through the customization. If you made any customization that affects the business logic or database structure that you use through the API support of the form, make sure that you have retrieved the latest version of the WSDL file after the customization is published. You may generate the WSDL file any number of times.

Step 2. Import the WSDL File of the Web Services Into the Development Environment

When the WSDL file is generated, you must import it into your development environment to generate proxy classes. If necessary, see the documentation of your development environment to find out the correct way of building the proxy classes based on the WSDL definition.

Programming languages supported by Microsoft Visual Studio.NET can access the Web Services API through the proxy classes created by using the WSDL description for corresponding server-side objects. Below you will find instructions on how to implement the proxy classes by using Visual Studio 2008 or later and NetBeans 6.9.

To generate proxy classes from the WSDL definition by using Visual Studio 2008 or later:

1. Start MS Visual Studio and select **File** > **New** > **Project**.
2. In the **New Project** window that appears, select the required template; most examples of Acumatica Web Service API implementation are based on the Visual C# *Console Application* template, although you can use any another template.
3. Define the name of the project and solution, as shown in the figure below, and click **OK**. (Although you can use any name for the project and solution, we recommend that you use a project name that is identical to the name of the solution that includes it.)

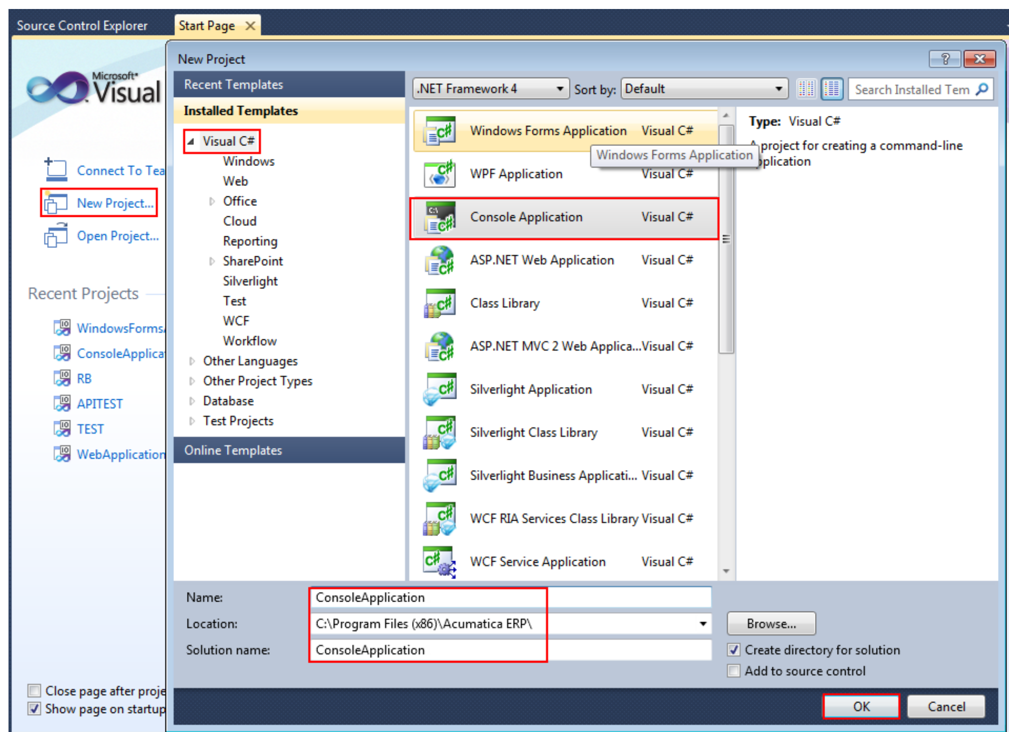


Figure: Creating the new project

4. Open the **Project** menu and select *Add Service Reference*.
5. In the dialog box that appears, click **Advanced**.
6. In the second dialog box that appears, click **Add Web Reference**.
7. In the third dialog box, type the path to Web Service WDSL descriptor file for the URL, as shown in the figure below. You can either use the local version of the WSDL file or provide the URL reference to the remote server.

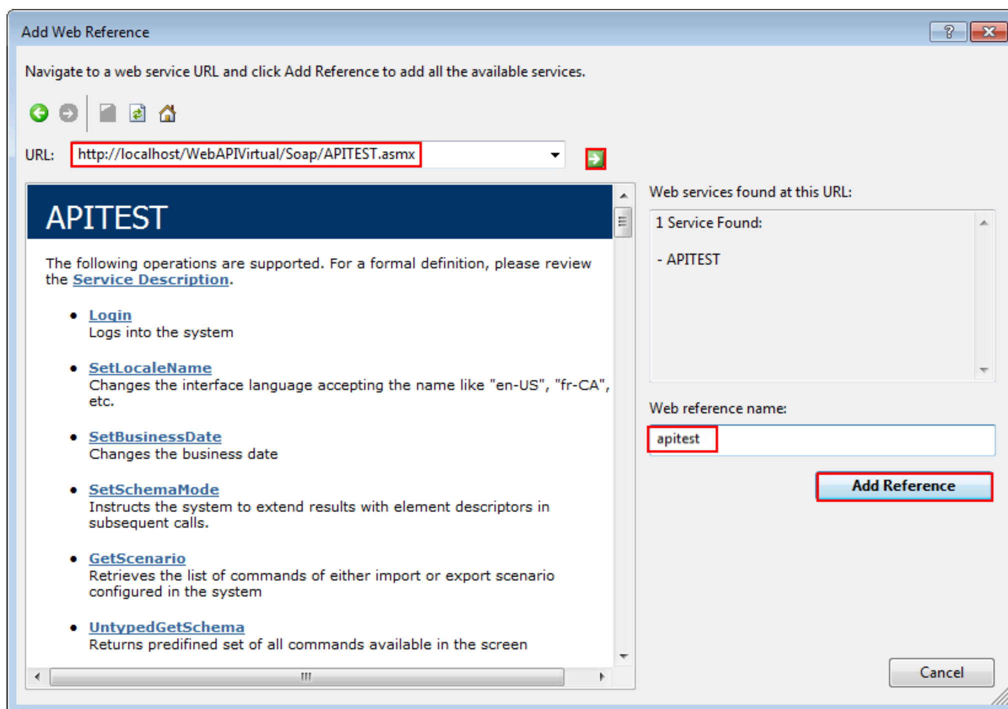


Figure: Specifying the URL of the WSDL file for the web reference

8. Click **GO** to continue.

9. Specify the **Web reference name**: *apitest*, for instance (see the figure above). This name will be used as a namespace for the generated web service proxy classes.

10. Click **Add Reference** to complete the creation process. As a result, in the Solution Explorer window, you can see the *Web References* folder with the reference to the WSDL file generated in Step 1, as shown in the figure below.

The new Visual Studio project now consists of the *Program* proxy class, which can be used for communication between the client application and Acumatica Web Services. The communication program code must be added within the body of the *Program* proxy class.

Because you may access multiple web services in the same Acumatica instance, we recommend that you name web references according to the original name of the WSDL file, but without capitalization: *apitest*.

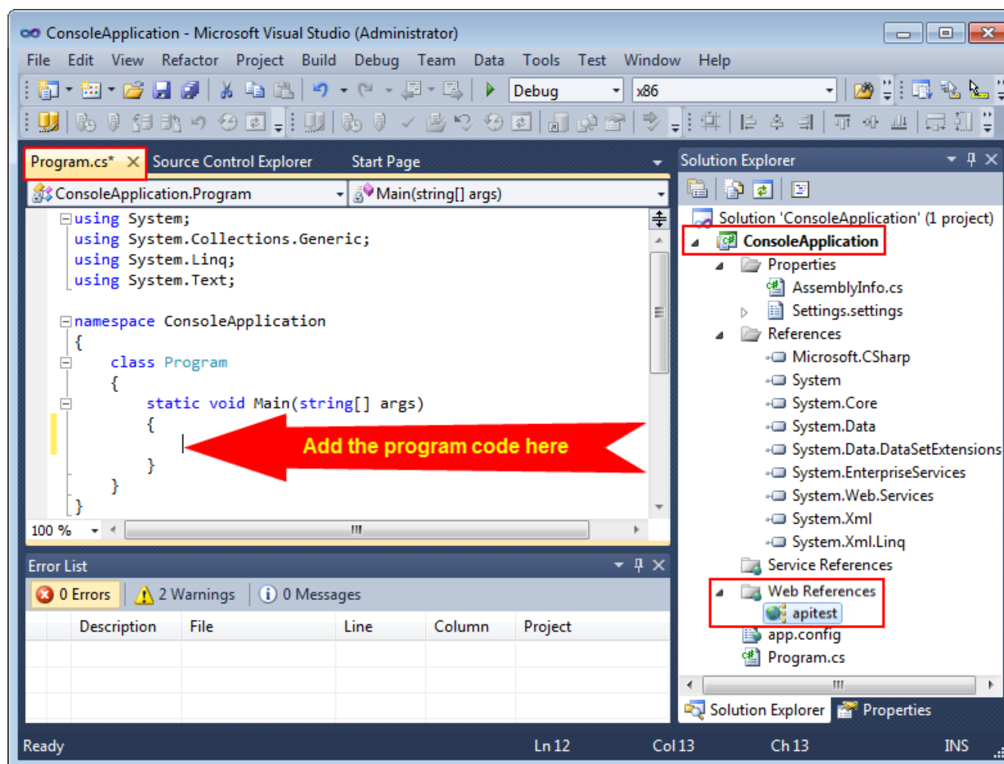


Figure: The *apitest* web reference and the *Program* proxy class

Java API for XML Web Services (JAX WS) supports the SOAP protocol and may be used with Acumatica Studio.

To generate proxy classes from the WSDL definition by using NetBeans 6.9 or later:

1. Right-click on your project, and select **New** > **Web Service Client**.
2. In the dialog box, for the URL input line, specify the path to the web service WDSL descriptor file.
3. Enter a package name.
4. Click **Finish** to complete the process.

NetBeans will process the specified WSDL definition and create a proxy class. This proxy class will be used for communication between the client application and the Acumatica Web Service.

Step 3. Review and Use the Code From the Sample Project

Once you have imported the WSDL file and created the proxy class, you can start development of your client application. The fastest way to learn how to develop a client application by using the Web Services API is to learn and use the client application code from the sample project. The first typical solution can be found in *Exporting Warehouse Data*.

To avoid possible errors, pay attention to the following points:

1. To avoid unexpected code conflicts, create each example of the client application code within the project of the new empty solution. Otherwise, you should replace all

previous code lines within the same project before starting to test the results of each code example.

2. Before adding the client application code, add to the proxy class code one line that contains the *using* command (as the figure below shows):

```
using ConsoleApplication.apitest;
```

Here *ConsoleApplication* is the name of your client application and *apitest* is the name of the bound web service.

3. Optional: Before you debug the client application, replace the URL of the WSDL file with the URL that corresponds to your file name and location. (In the figure below, you can see the example of the command line with the highlighted URL in the client application code that is to be replaced with the URL of your WSDL file.) This step is optional because if you don't specify the URL of the WSDL file, the system will use the URL set in the *App.Config* file.

4. Optional: Before debugging the client application, ensure that you have created the proper support of the authorization process; otherwise, you may need to make changes as follows (also shown in the figure below):

   - If your installation of Acumatica ERP includes the common company, use the simplest authorization code line:

     ```
     LoginResult result = context.Login("admin", "E618");
     ```

     Instead of *admin*, you may have another user name, but you should have enough rights to work with Web Services API services. Replace the password in the appropriate code line (*E618* by default) with the password that you had specified for the Acumatica ERP instance.

     In all the topics with examples, we use the common company and the simplest authorization code line.

   - If you work with more than one company but with the common branch, use the following modified authorization code line:

     ```
     LoginResult result = context.Login("user@CompanyCD", "E618");
     ```

     In the code line above, **Company CD** represents the required company short (CD) name.

   - If you work with more than one company and the company that you need has various branches, you should use the following modified authorization code line:

     ```
     LoginResult result = context.Login("user@CompanyCD:BranchCD", "E618");
     ```

     In the code line above, **CompanyCD** represents the required company short (CD) name, and **BranchCD** is the short branch name—that is, the CD name of the branch (for instance, *MAIN*, *NORTH*, or *SOUTH*) within the selected company.

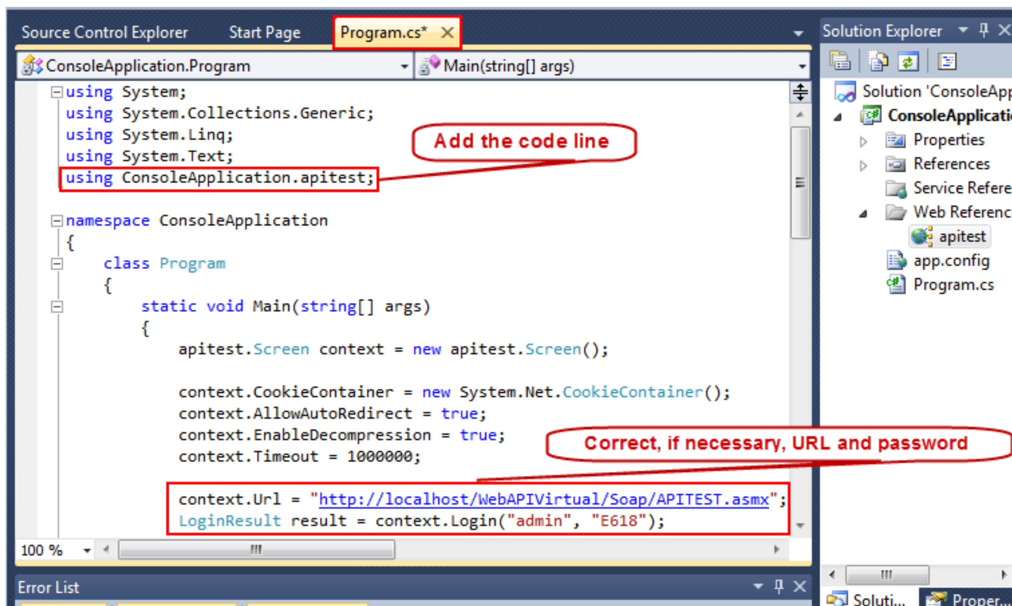


Figure: Correcting the code of the client application

Examples of the Web Service API Implementation

The examples in this section demonstrate how to use the following objects and properties of the Web Services API:

- **Screen**, an intermediary object that you will use for implementing the Web Services communication layer.
- The **CookieContainer** property, which preserves the session state between round trips. This property must be enabled in all client applications.
- **Content**, an object that defines the schema of the current form.

You can use the following links to directly access the examples of the Web Services API implementation:

- *Exporting Warehouse Data*
- *Exporting Stock Items*
- *Simulating the Behavior of Add Buttons on the Purchase Receipts Form*
- *Copying a Sales Order*
- *Adding a New Cash Transaction Document*
- *Adding Records to the Business Accounts and Opportunities Forms*
- *Importing of Data With an Image Into the Journal Transactions Form*
- *Exporting of Data With an Image From the Journal Transactions Form*

Exporting Warehouse Data

In this example, you create, run, and test a client application that exports to a string array required record fields from the *Warehouses* (IN.20.40.00) maintenance form of the Inventory module. The system filters exported data by the fixed **Warehouse ID** field value.

We make the following assumptions in this example:

1. You have installed the local client application instance (named *WEBAPIVirtual*) with the standard ERP demo application database. If you will use another application instance name, you should correct appropriate code lines in the code example shown in the next section.
2. You have created the Web Services WSDL definition file. (See *Quick Start, Step 1*.)
3. You have imported the Web Services WSDL definition file and generated the proxy class in the *ConsoleApplication.apitest* namespace. (See *Quick Start, Step 2*.) If you will use another WDSL file name, location, or namespace, you should correct appropriate code lines in the code example shown in the next section. You should also add your own password if it is different from the one used in the authorization code line in the code example. (See *Quick Start, Step 3*.)
4. You have primary information about the objects and properties of the Web Services API that the code lines of the example use. See the brief definitions in *Examples of the Web Service API Implementation*.

Create, Correct, and Run the Code Example

Add the code lines to the *Program* proxy class code and add the *using* operator, as shown in the code below. (The added code lines are preceded by +.)

```
 using System;
 using System.Collections.Generic;
 using System.Linq;
 using System.Text;
+using ConsoleApplication.apitest;

 namespace ConsoleApplication
 {
  class Program
  {
     static void Main(string[] args)
     {
+        apitest.Screen context = new apitest.Screen();
+        context.CookieContainer = new System.Net.CookieContainer();
+        context.AllowAutoRedirect = true;
+        context.EnableDecompression = true;
+        context.Timeout = 1000000;
+        context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
+        LoginResult result = context.Login("admin", "E618");

+        IN204000Content IN204000 = context.IN204000GetSchema();
+        context.IN204000Clear();
+        string[][] IN204000result = context.IN204000Export
+        (
+           new Command[]
+           {
+             IN204000.WarehouseSummary.WarehouseID,
+             IN204000.LocationTableLocationTable.LocationID,
+             new Field { FieldName = "LocationID", ObjectName =
+                   IN204000.LocationTableLocationTable.LocationID.ObjectName }
```

```
+               },
+                new Filter[]
+                {
+                    new Filter()
+                    {
+                      Field = new Field() { FieldName = IN204000.WarehouseSummary.
+                                    WarehouseID.FieldName, ObjectName = IN204000.
+                                        WarehouseSummary.WarehouseID.ObjectName },
+                      Condition = FilterCondition.Equals,
+                      Value = "GIT",
+                      Operator = FilterOperator.And
+                    }
+               },
+               0, false, false
+           );
      }
   }
}
```

This code implements the following process flow:

1. Using the *Export* method to export data from the form.
2. Using the *Filter* method to constrain the exported data by two fields of one record from the Warehouses form.

Figure: Exploring the Warehouses form

After you prepare the code, you should build the solution. Start the Acumatica ERP application instance with the WSDL file, navigate to **Distribution** > **Inventory**, select the **Configuration** submenu, and then select the **Manage** > **Warehouses** form. Select *GIT* as the **Warehouse ID**, and note the **Location ID** column values, as shown in the figure above. In Visual Studio, set appropriate breakpoints and then press F5 to run the client application in *Debug* mode. Use step-by-step debugging to ensure that the array contains exported data. (The figure below illustrates the test results.)

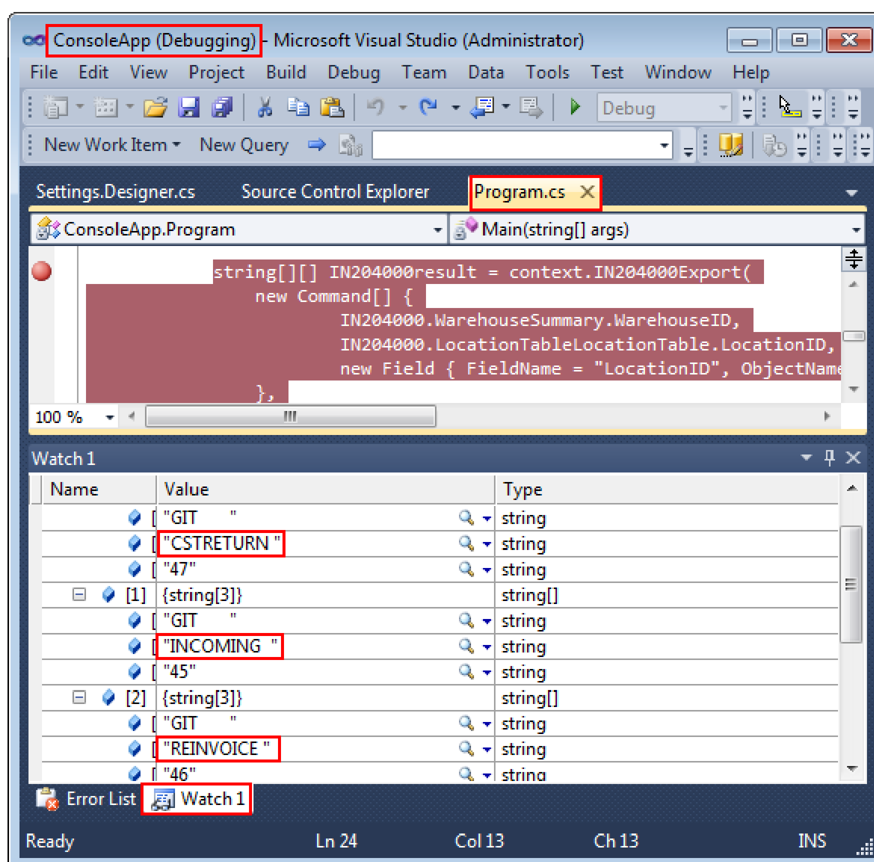


Figure: Checking the results in debug mode

Exporting Stock Items

In this example, you create, run, and test a client application that exports to a string array required record fields from the *Warehouses* (IN.20.25.00) maintenance form of the Inventory module. The system filter exports data by the hidden field **LastModifiedDateTime**. The date and time of the last modification of the Stock Items form must be fewer than 100 days before the current date.

We make the following assumptions in this example:

1. You have installed the local client application instance (named *WEBAPIVirtual*) with the standard ERP demo application database. If you will use another application instance name, you should correct appropriate code lines in the code example shown in the next section.
2. You have created the Web Services WSDL definition file. (See *Quick Start, Step 1*.)
3. You have imported the Web Services WSDL definition file and generated the proxy class in the *ConsoleApplication.apitest* namespace. (See *Quick Start, Step 2*.) If you will use another WDSL file name, location, or namespace, you should correct appropriate code lines in the code example shown in the next section. You should also add your own password if it is different from the one used in the authorization code line in the code example. (See *Quick Start, Step 3*.)
4. You have primary information about the objects and properties of the Web Services API that the code lines of the example use. See the brief definitions in *Examples of the Web Service API Implementation*.

Create, Correct, and Run the Code Example

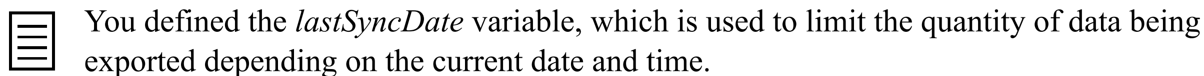
Add the code lines to the *Program* proxy class code and add two *using* operators, as shown in the code below. (The added code lines are preceded by +.)

```
 using System;
 using System.Collections.Generic;
 using System.Linq;
 using System.Text;
+using System.Globalization;
+using ConsoleApplication.apitest;

 namespace ConsoleApplication
 {
  class Program
  {
     static void Main(string[] args)
     {
+       apitest.Screen context = new apitest.Screen();
+       context.CookieContainer = new System.Net.CookieContainer();
+       context.AllowAutoRedirect = true;
+       context.EnableDecompression = true;
+       context.Timeout = 1000000;
+       context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
+       LoginResult result = context.Login("admin", "E618");

+       context.SetLocaleName(CultureInfo.CurrentCulture.Name);
+       DateTime lastSyncDate = DateTime.UtcNow;
+       lastSyncDate = lastSyncDate.AddDays(-100);
+       IN202500Content IN202500 = context.IN202500GetSchema();
+       context.IN202500Clear();
+       string[][] IN202500data = context.IN202500Export
+       (
+          new Command[]
+          {
+            IN202500.StockItemSummary.ServiceCommands.EveryInventoryID,
+            IN202500.StockItemSummary.InventoryID,
+            IN202500.WarehouseDetails.Warehouse,
+            IN202500.WarehouseDetails.QtyOnHand,
+            new Field
+             {
+               ObjectName = IN202500.StockItemSummary.InventoryID.ObjectName,
+                                            FieldName = "LastModifiedDateTime"
+             }
+          },
+            new Filter []
+             {
+              new Filter
+               {
+                Field = new Field { ObjectName =
+                         IN202500.StockItemSummary.InventoryID.ObjectName,
+                                        FieldName = "LastModifiedDateTime" },
+                Condition = FilterCondition.Greater,
+                Value = lastSyncDate.ToLongDateString()
+               }
+          },
+          0, false, false
+      );
     }
   }
 }
```

This code implements the following process flow:

1. Using the *Export* method to export data from the form.

2. Using the *Filter* method to limit the exported data by the date and time of the last modification of the Stock Items form. (The date and time of the last modification must be fewer than 100 days before the current date.)



You defined the *lastSyncDate* variable, which is used to limit the quantity of data being exported depending on the current date and time.

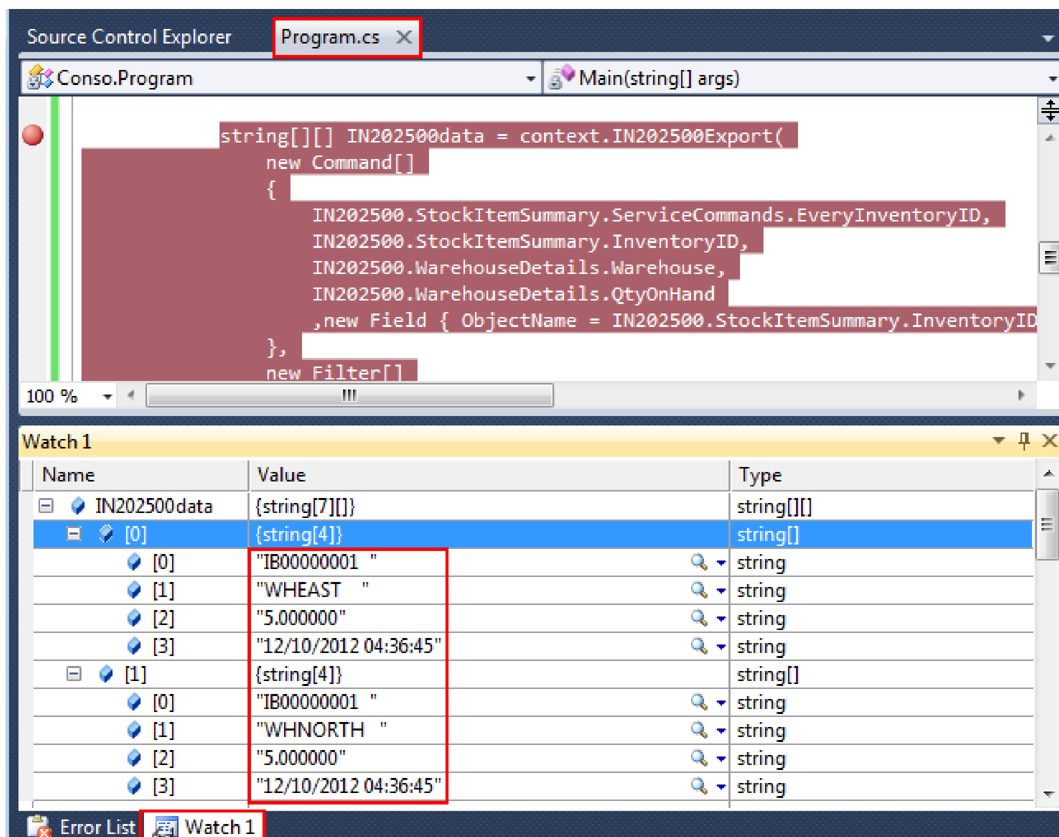
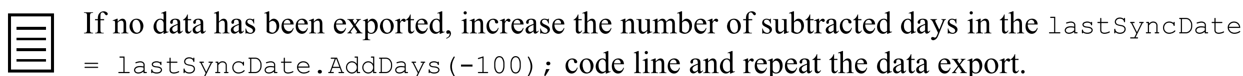


Figure: Checking the results in debug mode

After preparing the code, you should build the solution. Set appropriate breakpoints and then press F5 to run the client application in *Debug* mode. Use step-by-step debugging to ensure that the array contains exported data. (The figure above illustrates the test results.)

Optionally, you can start the Acumatica ERP application instance with the WSDL file and navigate to the **Distribution** > **Inventory** > **Manage** > **Stock Items** form. Select *IB00000001* as the **Inventory ID**, open the **Warehouse Detail** tab, and note the **Warehouse** and **Qty On Hand**column values, as shown in the figure below. Compare the column values with the values in the string array, displayed in the **Watch** window of Visual Studio in debug mode.



If no data has been exported, increase the number of subtracted days in the `lastSyncDate = lastSyncDate.AddDays(-100);` code line and repeat the data export.

Figure: Exploring the Warehouse Details tab of the Stock Items form

Simulating the Behavior of Add Buttons on the Purchase Receipts Form

In this example, you create, run, and test a command-line client application that adds lines to the details table of the *Purchase Receipts* (PO.30.20.00) form from the details table of the *Purchase Orders* (PO.30.10.00)form. (Both forms are located in the Purchase Orders module.) The client application will add lines from all purchase orders that have the same **VendorCD** field value. The application will imitate a user clicking the **Add PO** (the *AddPOOrder* action is called) button on the Purchase Receipts form and the user's next few steps.

We make the following assumptions in this example:

1. You have installed the local client application instance (named *WEBAPIVirtual*) with the standard ERP demo application database. If you will use another application instance name, you should correct appropriate code lines in the code example shown in the next section.
2. You have created the Web Services WSDL definition file. (See *Quick Start, Step 1*.)
3. You have imported the Web Services WSDL definition file and generated the proxy class in the *ConsoleApplication.apitest* namespace. (See *Quick Start, Step 2*.) If you will use another WDSL file name, location, or namespace, you should correct appropriate code lines in the code example shown in the next section. You should also add your own password if it is different from the one used in the authorization code line in the code example. (See *Quick Start, Step 3*.)
4. You have primary information about the objects and properties of the Web Services API that the code lines of the example use. See the brief definitions in *Examples of the Web Service API Implementation*.

Create, Correct, and Run the Code Example

In the steps below, before you create the code, you will add a purchase order and a purchase receipt. These tasks are necessary to test the result of running the client application when we know the values of key fields and use them in the code lines.

Do the following actions:

1. Start Acumatica ERP, and navigate to the **Distribution** > **Purchase Orders** > **Enter** > **Purchase Orders** form. Add a purchase order with the *Normal* type and the following values: *PORG000084* as the **Order Nbr**, *ACITAISYST* as the **Vendor**, and *MAIN* as the **Location**. Click **Save**.
2. Add to the **Document Details** tab three lines with any **Inventory ID**, **Order Qty**, and **Unit Cost** column values (as an example, see the figure below). Add *0* as the **Subitem** value (this column cannot be empty). Add the **Control Total** value (if this field appears in your system) so that it equals the **Order Total** value, and fill in the other mandatory fields (designated with asterisks); otherwise, the purchase order will not be saved. Click **Save**.
3. Clear the **Hold** check box and click **Save**.

Figure: Creating a new purchase order

4. Navigate to the **Distribution** > **Purchase Orders** > **Enter** > **Purchase Receipts** form, and add a receipt with the *Receipt* type and the following values: *PORE000079* as the **Receipt Nbr.**, *ACITAISYST* as the **Vendor**, and *MAIN* as the **Location**. Click **Save**.
5. Click **Add PO** on the table toolbar of the **Document Details** tab.
6. In the table of the **Add Purchase Order** dialog box that appears, notice one line with the field values of the purchase order added before. (In other cases, more than one line or no lines may be displayed.) Select the unlabeled check box and click **Add & Close**, as shown in the figure below. Notice that the **Add Purchase Order** window is closed, while on the **Document Details** tab, the three lines have been added. (You can see this in the figure in the end of this article.) You will implement this scenario in the C# client application code. Click **Cancel** to not save the added lines.

If you implement within one client application another scenario, based on the **Add PO Line** button, you should obtain the same result. You can prepare the code for the second scenario independently. This code is shown at the end of this topic; see *code of the second scenario*.

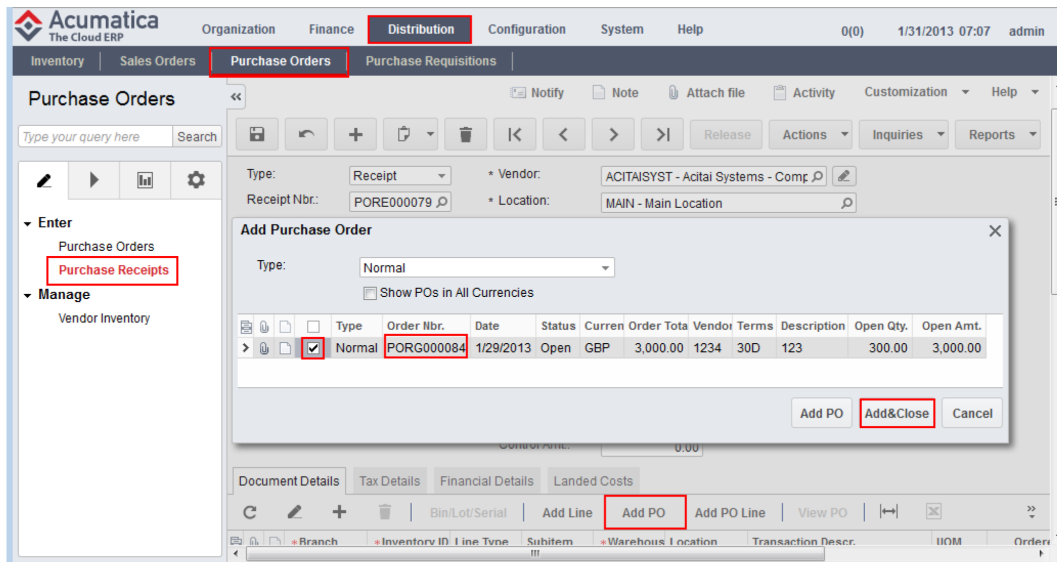
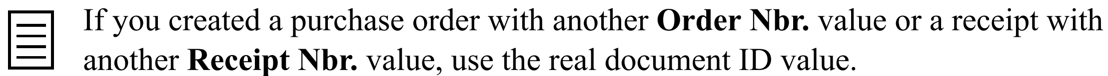


Figure: Adding new lines to the details table of the purchase receipt

7. Add code lines to the *Program* proxy class code but previously add the *using* operator, as shown in the code below. (The added code lines are preceded by +.)

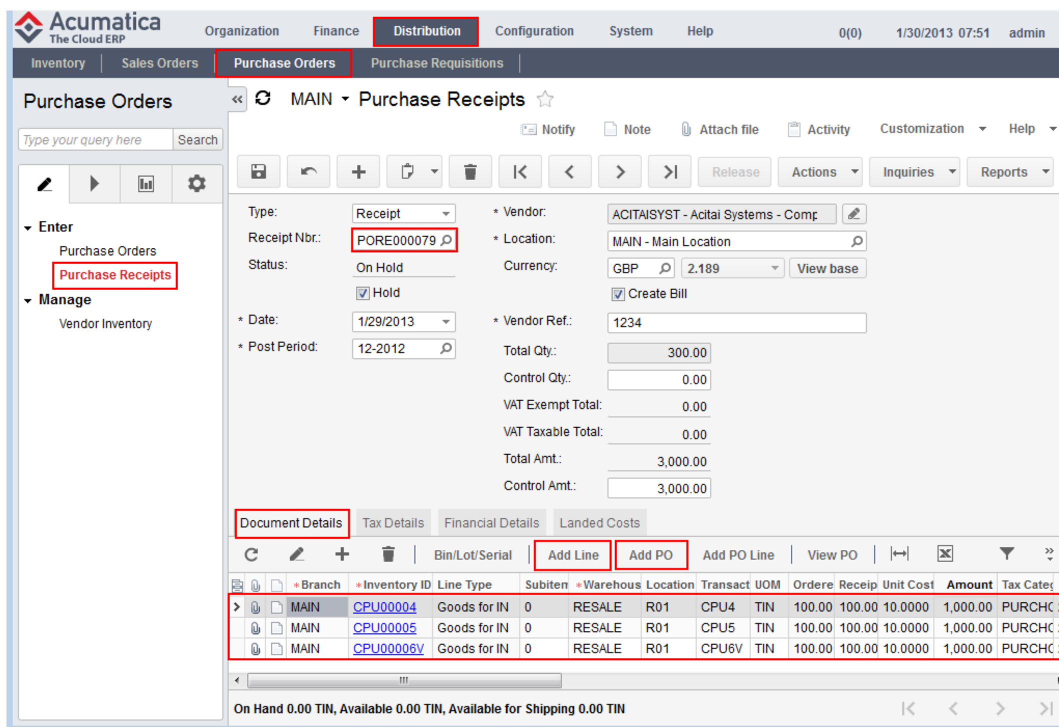
```
 using System;
 using System.Collections.Generic;
 using System.Linq;
 using System.Text;
+using ConsoleApplication.apitest;

 namespace ConsoleApplication
 {
  class Program
  {
     static void Main(string[] args)
     {
+       apitest.Screen context = new apitest.Screen();
+       context.CookieContainer = new System.Net.CookieContainer();
+       context.AllowAutoRedirect = true;
+       context.EnableDecompression = true;
+       context.Timeout = 1000000;
+       context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
+       LoginResult result = context.Login("admin", "E618");

+       PO302000Content PO302000 = context.PO302000GetSchema();
+       context.PO302000Clear();
+       PO302000.Actions.AddPOOrder.Commit = true;
+       PO302000.Actions.AddPOOrder2.Commit = true;
+       PO302000.AddPurchaseOrder.Selected.LinkedCommand = null;
+       PO302000.DocumentDetails.InventoryID.LinkedCommand = null;

+       PO302000Content[] PO302000result = context.PO302000Submit
+       (
+        new Command[]
+       {
+        new Value { Value = "PORE000079", LinkedCommand =
+                                  PO302000.DocumentSummary.ReceiptNbr },
+        new Value { Value = "OK", LinkedCommand =
+         PO302000.AddPurchaseOrder.ServiceCommands.DialogAnswer, Commit =
 true },
+         //uncomment the next two lines if you want to use multicurrency orders

+         //new Value { Value = "True", LinkedCommand =
```

```
+          //   PO302000.AddPurchaseOrderPOSelection.AnyCurrency, Commit = true },
+             PO302000.Actions.AddPOOrder,
+             new Key { Value = "='Normal'", FieldName =
+                                   PO302000.AddPurchaseOrder.OrderType.FieldName,
+                    ObjectName =
 PO302000.AddPurchaseOrder.OrderType.ObjectName },
+             new Key { Value = "='PORG000084'", FieldName =
+                                    PO302000.AddPurchaseOrder.OrderNbr.FieldName,
+                     ObjectName =
 PO302000.AddPurchaseOrder.OrderNbr.ObjectName },
+             new Value { Value = "True", LinkedCommand =
+                            PO302000.AddPurchaseOrder.Selected, Commit = true },
+             PO302000.Actions.AddPOOrder2
+        }
+        );
      }
   }
 }
```



If you created a purchase order with another **Order Nbr.** value or a receipt with another **Receipt Nbr.** value, use the real document ID value.

8. Build the solution, open the application, and press F5 to run the client application in *Debug* mode.
9. Again open Acumatica ERP, refresh the form, and ensure that the three lines have been added as a result of running the client application. (The figure below illustrates the test results.)



Branch	Inventory ID	Line Type	Subitem	Warehouse	Location	Transact	UOM	Ordered	Receipt	Unit Cost	Amount	Tax Categ
MAIN	CPU00004	Goods for IN	0	RESALE	R01	CPU4	TIN	100.00	100.00	10.0000	1,000.00	PURCHC 2
MAIN	CPU00005	Goods for IN	0	RESALE	R01	CPU5	TIN	100.00	100.00	10.0000	1,000.00	PURCHC 2
MAIN	CPU00006V	Goods for IN	0	RESALE	R01	CPU6V	TIN	100.00	100.00	10.0000	1,000.00	PURCHC 2

Figure: Three added lines as a result of running the client application code

This code implements the following process flow:

1. Activating the *AddPOOrder* and *AddPOOrder2* actions.
2. Invoking the *AddPOOrder* and *AddPOOrder2* actions to imitate adding lines to the details table by using the scenario that had been implemented for the **Add PO** button in Acumatica

ERP: selecting all the records in the table of the **Add Purchase Order** dialog box (after invoking the *Add PO Line* button, you can also specify through the code the required purchase order number), and clicking the **Add & Close** button.

The code for the second scenario follows.

```
apitest.Screen context = new apitest.Screen();
          context.CookieContainer = new System.Net.CookieContainer();
          context.AllowAutoRedirect = true;
          context.EnableDecompression = true;
          context.Timeout = 1000000;
          context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
          LoginResult result = context.Login("admin", "E618");

          PO302000.Actions.AddPOOrderLine.Commit = true;
          PO302000.Actions.AddPOOrderLine2.Commit = true;
          PO302000.AddPurchaseOrderLine.Selected.LinkedCommand = null;
          PO302000.DocumentDetails_.InventoryID.LinkedCommand = null;
          PO302000result = context.PO302000Submit(
          new Command[]
          {
            new Value { Value = "PORE000079", LinkedCommand =
                                           PO302000.DocumentSummary.ReceiptNbr},
            new Value { Value = "OK", LinkedCommand =
                   PO302000.AddPurchaseOrderLine.ServiceCommands.DialogAnswer,
                                                                Commit = true },
            PO302000.Actions.AddPOOrderLine,
            new Key { Value = "='PORG000084'", FieldName =
                  PO302000.AddPurchaseOrderLine.OrderNbr.FieldName, ObjectName =
                             PO302000.AddPurchaseOrderLine.OrderNbr.ObjectName },
            new Key { Value = "='CPU00004'", FieldName =
              PO302000.AddPurchaseOrderLine.InventoryID.FieldName, ObjectName =
                          PO302000.AddPurchaseOrderLine.InventoryID.ObjectName },
            new Value{ Value = "True", LinkedCommand =
                         PO302000.AddPurchaseOrderLine.Selected, Commit = true },
                                             PO302000.Actions.AddPOOrderLine2
            new Key{ Value = "='CPU00004'", FieldName =
                 PO302000.DocumentDetails_.InventoryID.FieldName, ObjectName =
                               PO302000.DocumentDetails_.InventoryID.ObjectName},
            new Value{ Value = "1.00", LinkedCommand =
                           PO302000.DocumentDetails_.ReceiptQty, Commit = true},
           // the next part of code is needed if you use Serial items
           PO302000.BinLotSerialNumbers.ServiceCommands.NewRow,
           new Value { Value = "R01", LinkedCommand =
                                     PO302000.BinLotSerialNumbers.Location },
           new Value { Value = "1.00", LinkedCommand =
                         PO302000.BinLotSerialNumbers.Quantity, Commit = true },
           new Value { Value = "25.00", LinkedCommand =
                            PO302000.DocumentDetails_.UnitCost, Commit = true },
           new Key { Value = "='CPU00004'", FieldName =
                  PO302000.DocumentDetails_.InventoryID.FieldName, ObjectName =
                              PO302000.DocumentDetails_.InventoryID.ObjectName },
           new Value { Value = "0.00", LinkedCommand =
                           PO302000.DocumentDetails_.ReceiptQty, Commit = true }
           PO302000.Actions.Save
                     }
                );
```

If you created a purchase order with another **Order Nbr.** value or a receipt with another **Receipt Nbr.** value, use the real document ID value.

Copying a Sales Order

In this example, you create, run, and test a simple command-line client application that copies key field and column values from an existing *Sales Orders* (SO.30.10.00) form of the Sales Orders module and pastes the values into an added sales order.

We make the following assumptions in this example:

1. You have installed the local client application instance (named *WEBAPIVirtual*) with the standard ERP demo application database. If you will use another application instance name, you should correct appropriate code lines in the code example shown in the next section.
2. You have created the Web Services WSDL definition file. (See *Quick Start, Step 1*.)
3. You have imported the Web Services WSDL definition file and generated the proxy class in the *ConsoleApplication.apitest* namespace. (See *Quick Start, Step 2*.) If you will use another WDSL file name, location, or namespace, you should correct appropriate code lines in the code example shown in the next section. You should also use your own password if it is different from the one used in the authorization code line in the code example. (See *Quick Start, Step 3*.)
4. You have primary information about the objects and properties of the Web Services API that the code lines of the example use. See the brief definitions in *Examples of the Web Service API Implementation*.

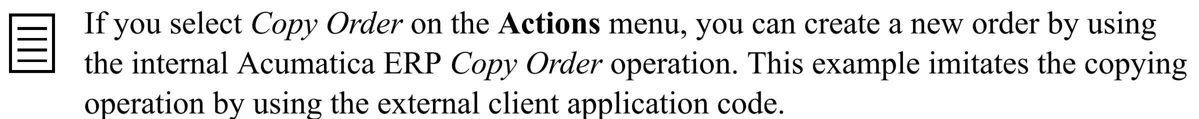
Create, Correct, and Run the Code Example

In the steps below, before you create the code example, you will ensure that a particular sales order with a specific **Order Nbr.** value exists—the order we plan to copy—and check some of its values. This step is necessary so you can later make sure the copying operation worked appropriately.

Do the following actions:

1. Start Acumatica ERP, and navigate to **Distribution** > **Sales Orders** > **Enter** > **Sales Orders**. In the **Order Type** field, select *SO*, and in the **Order Nbr.** field, select (by using the lookup window) *000097*. Note the values of the **Inventory ID** column in the details table on the **Document Details** tab (for the three rows) and the **Order Total** field in the main area of the form. (See the figure below).

Figure: The existing sales order



If you select *Copy Order* on the **Actions** menu, you can create a new order by using the internal Acumatica ERP *Copy Order* operation. This example imitates the copying operation by using the external client application code.

2. Add the code lines to the *Program* proxy class code and add the *using* operator, as shown in the code below. (The added code lines are preceded by +.)

```
 using System;
 using System.Collections.Generic;
 using System.Linq;
 using System.Text;
+using ConsoleApplication.apitest;

 namespace ConsoleApplication
 {
  class Program
  {
     static void Main(string[] args)
     {
+        apitest.Screen context = new apitest.Screen();
+        context.CookieContainer = new System.Net.CookieContainer();
+        context.AllowAutoRedirect = true;
+        context.EnableDecompression = true;
+        context.Timeout = 1000000;
+        context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
+        LoginResult result = context.Login("admin", "E618");

+        SO301000Content SO301000 = context.SO301000GetSchema();
+        context.SO301000Clear();
+        SO301000.Actions.CopyOrder.Commit = true;
+        SO301000Content[] SO301000Content = context.SO301000Submit
```

```
+          (
+          new Command[]
+            {
+               new Value { Value = "SO", LinkedCommand =
+                                         SO301000.OrderSummary.OrderType },
+               new Value { Value = "000097", LinkedCommand =
+                                         SO301000.OrderSummary.OrderNbr },
+               new Value { Value = "OK", LinkedCommand =
+                              SO301000.CopyTo.ServiceCommands.DialogAnswer },
+               new Value { Value = "QT", LinkedCommand =
+                                               SO301000.CopyTo.OrderType},
+            SO301000.Actions.CopyOrder,
+            SO301000.Actions.Save,
+            SO301000.OrderSummary.OrderNbr
+            }
+          );
     }
  }
 }
```

3. Build the solution, open the application, and press F5 to run the client application in *Debug* mode.
4. Again open Acumatica ERP and navigate to the Sales Order form. Select *QT* in the **Order Type** field, and select the new sales order (with the highest **Order Nbr.** value).Ensure that same three lines that existed in sales order *000097* have been added to the details table after you ran the client application, and make sure the **Order Total** field has the same value that you noted in sales order *000097*. (The figure below illustrates the test results.)

Figure: The added sales order as a result of running the client application code

As the introduction mentions, this code represents the simple example of a client application that is used for inserting a new sales order by copying many of its settings from an existing one. This code implements the following process flow:

1. Using the *Submit* method to provide the copying operation.

2. Invoking the *CopyOrder* action to imitate the selection of the *Copy Order* option on the **Actions** menu of the form.
3. Using the *SO301000.OrderSummary.OrderNbr* command to invoke the document autonumbering method implemented in Acumatica ERP.

Adding a New Cash Transaction Document

In this example, you create, run, and test a simple command-line client application that adds a new cash transaction document to the *Transactions* (CA.30.40.00) form of the Cash Management module.

We make the following assumptions in this example:

1. You have installed the local client application instance (named *WEBAPIVirtual*) with the standard ERP demo application database. If you will use another application instance name, you should correct appropriate code lines in the code example shown in the next section.
2. You have created the Web Services WSDL definition file. (See *Quick Start, Step 1*.)
3. You have imported the Web Services WSDL definition file and generated the proxy class in the *ConsoleApplication.apitest* namespace. (See *Quick Start, Step 2*.) If you will use another WDSL file name, location, or namespace, you should correct appropriate code lines in the code example shown in the next section. You should also add your own password if it is different from the one used in the authorization code line in the code example. (See *Quick Start, Step 3*.)
4. You have primary information about the objects and properties of the Web Services API that the code lines of the example use. See the brief definitions in *Examples of the Web Service API Implementation*.

Create, Correct, and Run the Code Example

Add the code lines to the *Program* proxy class code and add the *using* operator, as shown in the code below. (The added code lines are preceded by +.)

```
 using System;
 using System.Collections.Generic;
 using System.Linq;
 using System.Text;
+using ConsoleApplication.apitest;

 namespace ConsoleApplication
 {
  class Program
  {
     static void Main(string[] args)
     {
+       apitest.Screen context = new apitest.Screen();
+       context.CookieContainer = new System.Net.CookieContainer();
+       context.AllowAutoRedirect = true;
+       context.EnableDecompression = true;
+       context.Timeout = 1000000;
+       context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
+       LoginResult result = context.Login("admin", "E618");

+       try
```

```
+         {
+             CA304000Content CA304000 = context.CA304000GetSchema();
+             context.CA304000Clear();
+             CA304000Content[] CA304000result = context.CA304000Submit
+             (
+             new Command[]
+             {
+               new Value { Value = "100000", LinkedCommand =
+                                      CA304000.TransactionSummary.CashAccount },
+               new Value { Value = "PETTYEXP", LinkedCommand =
+                                            CA304000.TransactionSummary.EntryType },
+               new Value { Value = "111", LinkedCommand =
+                                         CA304000.TransactionSummary.DocumentRef },
+               new Value { Value = "true", LinkedCommand =
+                                            CA304000.TransactionSummary.Approved },
+               CA304000.TransactionDetails.ServiceCommands.NewRow,
+               new Value { Value = "408000", LinkedCommand =
+                                       CA304000.TransactionDetails.OffsetAccount },
+               new Value { Value = "00-00-00-00-000", LinkedCommand =
+                                    CA304000.TransactionDetails.OffsetSubaccount },
+               new Value { Value = "1", LinkedCommand =
+                                             CA304000.TransactionDetails.Quantity },
+               new Value { Value = "100", LinkedCommand =
+                                 CA304000.TransactionDetails.Price, Commit = true },
+               new Value { Value = "100", LinkedCommand =
+                         CA304000.TransactionSummary.ControlTotal, Commit = true },
+               CA304000.Actions.Save, CA304000.TransactionSummary.ReferenceNbr
+           }
+           );
+         }
+         catch (Exception ex)
+         {
+           Console.WriteLine(ex.Message);
+         }
      }
   }
  }
```

As the introduction mentions, this code represents the simple example of a client application that is used for inserting a new cash transaction document into the Transactions form of the Cash Management module. This code implements the following process flow:

1. Using the *Submit* method to add data to the form.
2. Invoking the *Save* action in the form.
3. Using the *CA304000.TransactionSummary.ReferenceNbr* command to invoke the document autonumbering method implemented in the Acumatica ERP.

After preparing the code, you can build the solution and then press F5 to run the client application in *Debug* mode. Start the Acumatica ERP application instance with the WSDL file, and navigate to **Finance** > **Cash Management** > **Enter** > **Transactions** to open the Transactions form. In the **Reference Nbr.** field, select the added transaction item (which has the highest reference number). Ensure that the item has been added with the needed values. (See the figure below.)

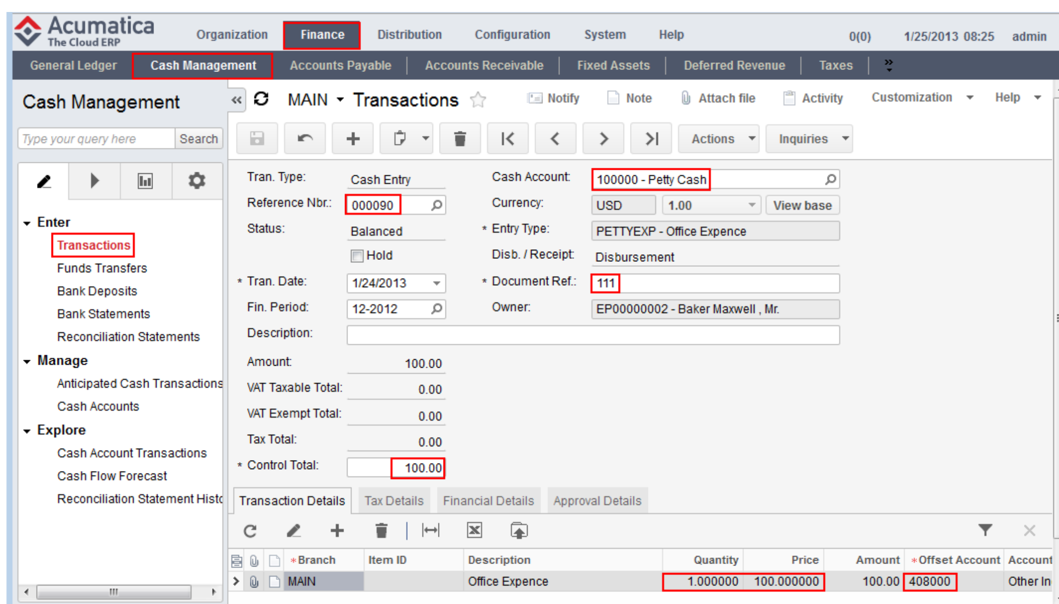


Figure: Testing the result of running the client application

Adding Records to the Business Accounts and Opportunities Forms

In this example, you create, run, and test a simple command-line client application that adds new records to the *Business Accounts* (CR.30.30.00) and *Opportunities* (CR.30.40.00) forms of the Customer Management module.

As with the previous example, we make the following assumptions in this example:

1. You have installed the local client application instance (named *WEBAPIVirtual*) with the standard ERP demo application database. If you will use another application instance name, you should correct appropriate code lines in the code example shown in the next section.
2. You have created the Web Services WSDL definition file. (See *Quick Start, Step 1*.)
3. You have imported the Web Services WSDL definition file and generated the proxy class in the *ConsoleApplication.apitest* namespace. (See *Quick Start, Step 2*.) If you will use another WDSL file name, location, or namespace, you should correct appropriate code lines in the code example shown in the next section. You should also add your own password if it is different from the one used in the authorization code line in the code example. (See *Quick Start, Step 3*.)
4. You have primary information about the objects and properties of the Web Services API that the code lines of the example use. See the brief definitions in *Examples of the Web Service API Implementation*.

Create, Correct, and Run the Code Example

Add the code lines to the *Program* proxy class code and add the *using* operator, as shown in the code below. (The added code lines are preceded by +.)

```
using System;
using System.Collections.Generic;
```

```
 using System.Linq;
 using System.Text;
+using ConsoleApplication.apitest;

 namespace ConsoleApplication
 {
  class Program
  {
     static void Main(string[] args)
     {
+       apitest.Screen context = new apitest.Screen();
+       context.CookieContainer = new System.Net.CookieContainer();
+       context.AllowAutoRedirect = true;
+       context.EnableDecompression = true;
+       context.Timeout = 1000000;
+       context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
+       LoginResult result = context.Login("admin", "E618");

+       CR303000Content CR303000 = context.CR303000GetSchema();
+       context.CR303000Clear();
+       CR303000Content[] CR303000Content = context.CR303000Submit
+      (
+       new Command[]
+           {
+             new Value { Value = "TEST123", LinkedCommand =
+                                   CR303000.AccountSummary.BusinessAccount },
+             new Value { Value = "TEST123", LinkedCommand =
+                                   CR303000.AccountSummary.BusinessAccountName },
+             new Value { Value = "US", LinkedCommand =
+                                          CR303000.DetailsMainAddress.Country },
+             new Value { Value = "Industry", LinkedCommand =
                                                 CR303000.Attributes.Attribute },
+             new Value { Value = "Banking", LinkedCommand =
+                                     CR303000.Attributes.Value, Commit = true },
+             CR303000.Actions.Save
+           }
+       );

+       CR304000Content CR304000 = context.CR304000GetSchema();
+       context.CR304000Clear();
+       CR304000Content[] CR304000Content = context.CR304000Submit
+       (
+         new Command[]
+         {
+           new Value { Value = "TEST123", LinkedCommand =
+                                 CR304000.OpportunitySummary.BusinessAccount },
+           new Value { Value = "MAIN", LinkedCommand =
+                                        CR304000.OpportunitySummary.NoteText },
+           new Value { Value = "INSIDE", LinkedCommand =
+                                                  CR304000.Details.ClassID },
+           new Value { Value = "DESCRIPTION", LinkedCommand =
+                                      CR304000.OpportunitySummary.Subject },
+           CR304000.Actions.Save
+        }
+      );
     }
   }
 }
```

As the introduction mentions, this code represents the simple example of a client application that adds new records to the Business Accounts and Opportunities forms of the Customer Management module. This code implements the following process flow:

1. Using the *Submit* method to add data to the forms.
2. Invoking the *Save* action in the form.

After preparing the code, you can build the solution and then press F5 to run the client application in *Debug* mode. Perform the following actions:

- Start the Acumatica ERP application instance with WSDL file. Navigate to **Organization** > **Cash Management** > **Manage** > **Business Account** to open the Business Account form, and in the **Business Account** field, find the added record by using a quick search and select it. (The new record has the number *TEST123*.) Ensure that the record has been added with the needed values. (See the figure below.)

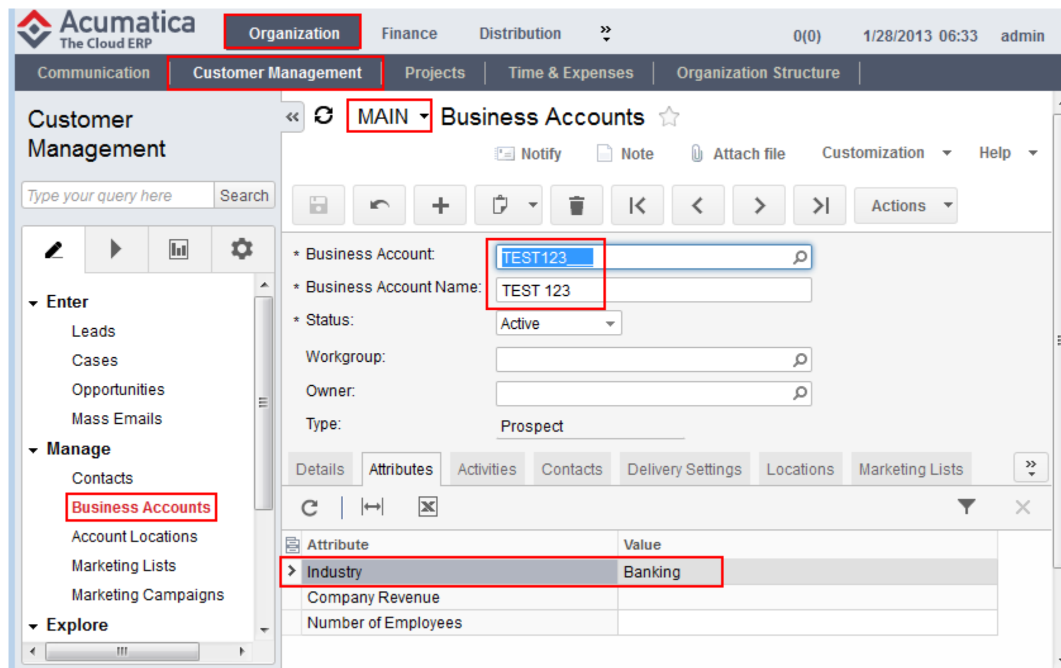


Figure: Testing the first result of running client application

- Navigate to **Organization** > **Cash Management** > **Manage** > **Opportunities** to open the Opportunities form, and in the **Opportunity ID** field, select the added record, which has the highest reference number. Ensure that the record has been added with the needed values. (See the figure below.)

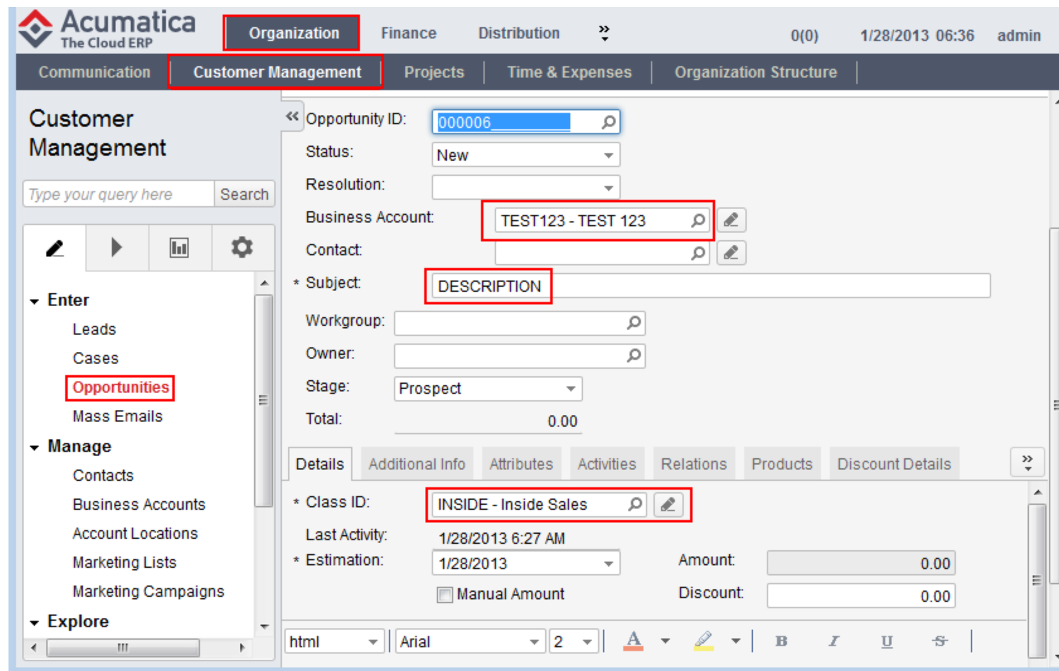


Figure: Testing the second result of running the client application

Importing of Data With an Image Into the Journal Transactions Form

In this example, you create, run, and test a client application that enables the import data with an image into the *Journal Transactions* (GL.30.10.00) form of the General Ledger module.

We make the following assumptions in this example:

1. You have installed the local client application instance (named *WEBAPIVirtual*) with the standard ERP demo application database. If you will use another application instance name, you should correct appropriate code lines in the code example shown in the next section.
2. You have created the Web Services WSDL definition file. (See *Quick Start, Step 1*.)
3. You have imported the Web Services WSDL definition file and generated the proxy class in the *ConsoleApplication.apitest* namespace. (See *Quick Start, Step 2*.) If you will use another WDSL file name, location, or namespace, you should correct appropriate code lines in the code example shown in the next section. You should also add your own password if it is different from the one used in the authorization code line in the code example. (See *Quick Start, Step 3*.)
4. You have primary information about the objects and properties of the Web Services API that the code lines of the example use. See the brief definitions in *Examples of the Web Service API Implementation*.

Create, Correct, and Run the Code Example

Add the code lines to the *Program* proxy class code and add the *using* operator, as shown in the code below. (The added code lines are preceded by +.)

```
 using System;
 using System.Collections.Generic;
 using System.Linq;
 using System.Text;
+using ConsoleApplication.apitest;

 namespace ConsoleApplication
 {
  class Program
  {
     static void Main(string[] args)
     {
+       apitest.Screen context = new apitest.Screen();
+       context.CookieContainer = new System.Net.CookieContainer();
+       context.AllowAutoRedirect = true;
+       context.EnableDecompression = true;
+       context.Timeout = 1000000;
+       context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
+       LoginResult result = context.Login("admin", "E618");

+       byte[] filedata;
+       using (System.IO.FileStream file =
+               System.IO.File.Open(@"D:\01.jpg", System.IO.FileMode.Open))
+         {
+           filedata = new byte[file.Length];
+           file.Read(filedata, 0, filedata.Length);
+         }
+       GL301000Content GL301000 = context.GL301000GetSchema();
+       context.GL301000Clear();
+       GL301000ImportResult[] GL301000ImportResult = context.GL301000Import
+       (
+          new Command[]
+           {
+           new Value
+            {
+              Value = "GL", LinkedCommand = GL301000.BatchSummary.Module },
+              GL301000.BatchSummary.BatchNumber,
+              GL301000.BatchSummary.ControlTotal,
+              new Value
+               {
+                   FieldName = "01.jpg", LinkedCommand =
-                         GL301000.BatchSummary.ServiceCommands.Attachment
+               }
+             GL301000.TransactionDetails.Account,
+             GL301000.TransactionDetails.Subaccount,
+             GL301000.TransactionDetails.RefNumber,
+             GL301000.TransactionDetails.CreditAmount,
+             GL301000.TransactionDetails.DebitAmount,
+             GL301000.Actions.Save
+           },
+           null,
+           new string [][]
+            { new string[] { "00003849", "10", Convert.ToBase64String(filedata),
+                              "100000", "US-00-00-00-000", "REF", "10,0", "0,0" },
+             new string[] { "00003849", "10", Convert.ToBase64String(filedata),
+                              "101000", "US-00-00-00-000", "REF", "0,0", "10,0" },
+            },
+            false, false, true);
+      );
     }
   }
 }
```

This code implements the following process flow:

1. Using the *Import* method to import data into the form.
2. Using the standard .Net classes (*FileStream*, *File*, and *FileMode*) to open and read the byte content of the external file.

3. Using the *Attachment* service command to attach the external file to the form.

Figure: Exploring the Journal Transactions form

Test the results of data importing as follows:

- After preparing the code, build the solution and then press F5 to run the application in debug mode. Start the Acumatica ERP application instance with the WSDL file, and navigate to the **Finance** > **General Ledger** > **Enter** > **Journal Transactions** form. In the **Batch Number** lookup field, find and select the largest batch number, and note the transaction values of the two transactions in the details table (these values must equal those used in the code lines), as shown in the figure above.
- To see the attached file, click **Attach file** on the title bar and select the attached file name. (The figure below illustrates the process of opening the attached file.)

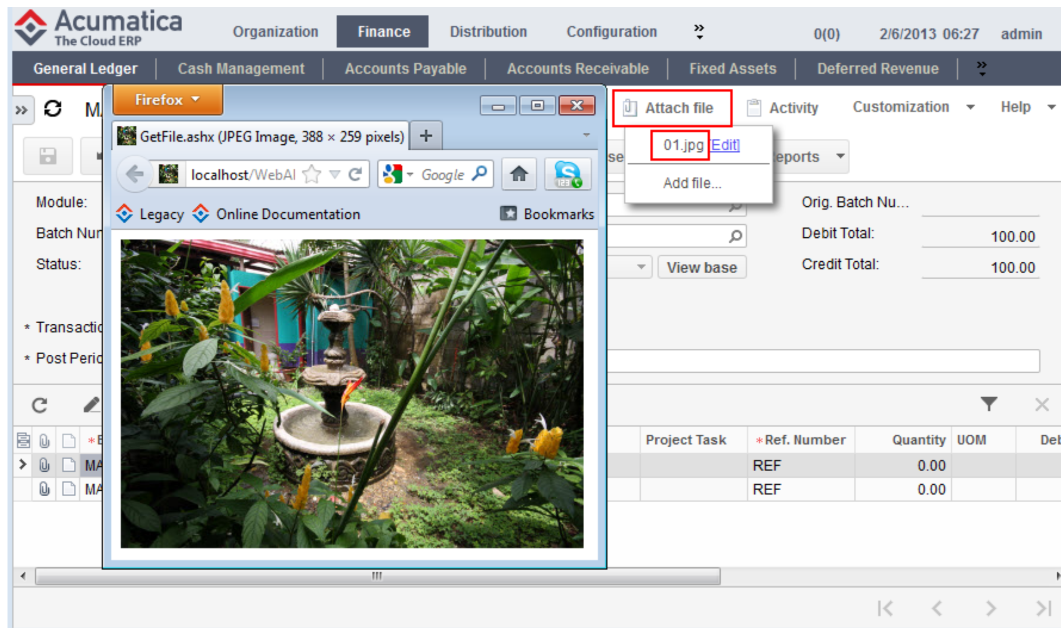


Figure: Opening the attached file

Exporting of Data With an Image From the Journal Transactions Form

In this example, you create, run, and test a client application that exports data with an image from the *Journal Transactions* (GL.30.10.00) form of the General Ledger module to a string array and limits exported data with filter conditions.

Before performing the actions of this example, import data with an image, as described in the previous example (see *Importing of Data With an Image Into the Journal Transactions Form*).

We make the following assumptions in this example:

1. You have installed the local client application instance (named *WEBAPIVirtual*) with the standard ERP demo application database. If you will use another application instance name, you should correct appropriate code lines in the code example shown in the next section.
2. You have created the Web Services WSDL definition file. (See *Quick Start, Step 1*.)
3. You have imported the Web Services WSDL definition file and generated the proxy class in the *ConsoleApplication.apitest* namespace. (See *Quick Start, Step 2*.) If you will use another WDSL file name, location, or namespace, you should correct appropriate code lines in the code example shown in the next section. You should also add your own password if it is different from the one used in the authorization code line in the code example. (See *Quick Start, Step 3*.)
4. You have primary information about the objects and properties of the Web Services API that the code lines of the example use. See the brief definitions in *Examples of the Web Service API Implementation*.

Create, Correct, and Run the Code Example

Add the code lines to the *Program* proxy class code and add the *using* operator, as shown in the code below. (The added code lines are preceded by +.)

```
 using System;
 using System.Collections.Generic;
 using System.Linq;
 using System.Text;
+using ConsoleApplication.apitest;

 namespace ConsoleApplication
 {
  class Program
  {
     static void Main(string[] args)
     {
+       apitest.Screen context = new apitest.Screen();
+       context.CookieContainer = new System.Net.CookieContainer();
+       context.AllowAutoRedirect = true;
+       context.EnableDecompression = true;
+       context.Timeout = 1000000;
+       context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
+       LoginResult result = context.Login("admin", "E618");

+       GL301000Content GL301000 = context.GL301000GetSchema();
+       context.GL301000Clear();
+       string[][] export = context.GL301000Export
+       (
+          new Command[]
+           {
+           new Value
+            {
+              Value = "GL", LinkedCommand = GL301000.BatchSummary.Module },
+              GL301000.BatchSummary.ServiceCommands.EveryBatchNumber,
+              new Field
+              {
+                 ObjectName = GL301000.BatchSummary.BatchNumber.ObjectName,
-                       FieldName = "LastModifiedDateTime", Value = "TS"
+              }
+             GL301000.BatchSummary.BatchNumber,
+             GL301000.BatchSummary.ControlTotal,
+             new Value {
+                        FieldName = "01.jpg", LinkedCommand =
+                        GL301000.BatchSummary.ServiceCommands.Attachment
+                       },
+             GL301000.TransactionDetails.Account,
+             GL301000.TransactionDetails.Subaccount,
+             GL301000.TransactionDetails.RefNumber,
+             GL301000.Transactionetails.CreditAmount,
+             GL301000.TransactionDetails.DebitAmount
+           },
+           new Filter[]
+           { new Filter { Field = GL301000.BatchSummary.TransactionDate,
+             Condition = FilterCondition.GreaterOrEqual, Value = DateTime.Today }
+            },
+            0, true, true);
+      );
     }
   }
 }
```

This code implements the following process flow:

1. Using the *Export* method to export data from the form to the string array.

2. Using the **Filter** object with the *FilterCondition* property to filter exported data to the string array. (This exports only transactions from the current day.)
3. Using the *Attachment* service command to identify and download the attached file from the form.

Test the results of data exporting as follows:

- After preparing the code, build the solution and then press F5 to run the application in debug mode. Start the Acumatica ERP application instance with the WSDL file, navigate to the **Finance** > **General Ledger** > **Enter** > **Journal Transactions** form. In the **Batch Number** lookup field, find and select the largest batch number, change the **Transaction Date** field value to the current date value (if necessary), and note the transaction values of the two transactions in the details table (which must equal the values that will be obtained in the watch window of Visual Studio), as shown in the figure above. Compare the transaction values with the debugging results, as shown in the figure below.
- In Visual Studio, set appropriate breakpoints and then press F5 to run the client application in *Debug* mode. Use step-by-step debugging to ensure that the array contains exported data with the attached image file code. (The figure below illustrates the test results.)

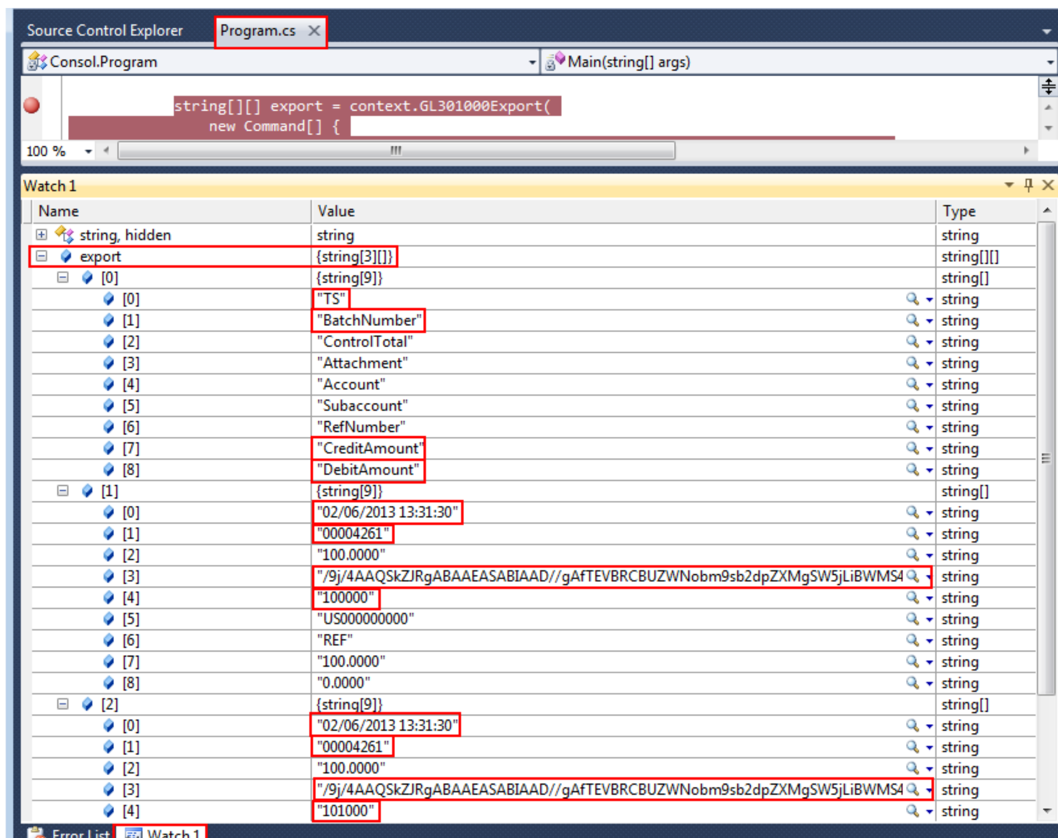


Figure: Checking the results in debug mode