Web Services

I320 Data Manipulation

Training Guide

Acumatica
The Cloud ERP

Last Revision: 11/20/2019

Acumatica ERP 2019 R2

Contents

Copyright

Software Version - 2019 R2

Introduction

External systems can use Acumatica ERP integration interfaces to access the business functionality and data of Acumatica ERP. Acumatica ERP provides the following integration interfaces:

- The Open Data (OData) interface
- The contract-based REST API
- The contract-based SOAP API
- The screen-based SOAP API

This course shows how you can submit data to Acumatica ERP and process data in Acumatica ERP with the contract-based REST and SOAP APIs. The data submission and data processing with the screen-based SOAP API is outside of the scope of this course. The OData interface cannot be used for this type of integration.

Information about data retrieval through all these integration interfaces is available in the *I300 Web Services: Basic | Data Retrieval* and *I310 Web Services: Advanced | Data Retrieval* training courses, which are prerequisites for this course.

This course is intended for developers who need to create applications that interact with Acumatica ERP.

The course is based on a set of examples of web integration scenarios that demonstrate the process of developing a client application that uses the Acumatica ERP integration interfaces. The course gives you ideas about how to develop your own applications by using the web services APIs. As you go through the course, you can complete the examples for a particular integration interface or for multiple integration interfaces.

After you complete all the lessons of the course, you will be familiar with the advanced techniques of data submission and data processing through the Acumatica ERP web services APIs.

How to Use This Course

To complete the course, you will complete the lessons from each part of the course in the order in which they are presented and pass the assessment test. More specifically, you will do the following:

1. Complete *Course Prerequisites*, and carefully read *Company Story and MyStoreIntegration Application*.
2. Complete the lessons in all parts of the training guide. In each lesson, you should review the description of the lesson, at least one of the examples for the integration interface that you are interested in, and the lesson summary. You can skip the other examples of the lesson, or you may prefer to review multiple examples in the lesson if you are looking for the best solution for your integration scenario.
3. In Partner University, take *I320 Certification Test: Data Manipulation*. This test does not include any questions that are specific to only one integration interface. That is, you can pass this test if in each part of the guide, you have read the descriptions of all lessons, at least one example in each lesson, and the *Lesson Summary* topics.

After you pass the certification test, you will be given the Partner University certificate of course completion.

What Is in a Part?

All parts of the course are dedicated to the implementation of particular web integration scenarios that you may need to implement in a third-party application that integrates an external system with Acumatica ERP.

Each part of the course consists of lessons you should complete.

What Is in a Lesson?

Each lesson is dedicated to a particular web integration scenario that you can implement by using the web services APIs. Each lesson consists of a brief description of the web integration scenario and examples of the implementation of this scenario.

The lesson may also include *Additional Information* topics, which are outside of the scope of this course but may be useful to some readers.

Each lesson ends with a *Lesson Summary* topic, which summarizes the possible options for the implementation of the web integration scenario with different integration interfaces.

What Are the Documentation Resources?

All the links listed in the *Related Links* sections refer to the documentation available on the *https://help.acumatica.com/* website. These and other topics are also included in the Acumatica ERP instance, and you can find them under the **Help** menu.

Course Prerequisites

To complete this course, you should be familiar with the financial and distribution functionality of Acumatica ERP, the basic principles of the system, and the principles of data retrieval with the Acumatica ERP integration interfaces. We recommend that you complete the *I300 Web Services: Basic | Data Retrieval* and *I310 Web Services: Advanced | Data Retrieval* training courses before you go through this course.

You need to perform the following prerequisite actions before you start to complete the course:

1. Make sure the environment that you are going to use for the training course conforms to the *System Requirements*.
2. Make sure that the Web Server (IIS) features that are listed in *Configuring Web Server (IIS) Features* are turned on.
3. Deploy an instance of Acumatica ERP2019 R2 with the name *MyStoreInstance* and a tenant that contains the *I100* data. If you have completed the *I300 Web Services: Basic | Data Retrieval* or *I310 Web Services: Advanced | Data Retrieval* training course, you can use the instance that you have deployed for this course. For information on how to deploy the instance for the training course, see *Deploying an Acumatica ERP Instance for the Training Course*.
4. If you are going to complete the examples that illustrate the use of the REST API, make sure the following conditions are met:
    - The Postman application should be installed on your computer. To download and install Postman, follow the instructions on *https://www.getpostman.com/apps*.
    - A Postman collection should be configured to use the OAuth 2.0 authorization, or the requests for signing in to and signing our from Acumatica ERP should be included in the collection. You can use the Postman collection that you have created in the *I310 Web Services: Advanced | Data Retrieval* training course. For details about how to configure OAuth 2.0 authorization, see Part 1 in the *I310 Web Services: Advanced | Data Retrieval* training course. For details about the sign-in and sign-out methods, see *Login to the Service* and *Logout from the Service* in the documentation.
5. If you are going to complete the examples that illustrate the use of the SOAP API, make sure the following conditions are met:
    - Microsoft Visual Studio 2015 or later should be installed on your computer.

      The instructions in this guide are designed for Microsoft Visual Studio 2019. If you use a different version of Visual Studio, the menu paths and the user interface may differ.
    - A Visual Studio project should be configured to use the *Default/18.200.001* endpoint of the *MyStoreInstance* Acumatica ERP instance and to use either the OAuth 2.0 authorization or the sign-in and sign-out methods of the contract-based SOAP API. You can use the Visual Studio project that you have created in the *I310 Web Services: Advanced | Data Retrieval* training course. For details about how to configure OAuth 2.0 authorization, see Part 1 in the *I310 Web Services: Advanced | Data Retrieval* training course. For details about the sign-in and sign-out methods, see *Login() Method* and *Logout() Method* in the documentation.
6. Make sure you have HTTP access from the computer where you work with the examples to the Acumatica ERP instance so you can work with the integration interfaces.
7. If you want to work with the examples that describe how to process credit card payments through the Acumatica ERP website or if you use the OAuth 2.0 authorization, configure HTTPS on the Acumatica ERP website, as described in *Configuring a Website for HTTPS*. If you have

configured HTTPS on the Acumatica ERP website for the *I310 Web Services: Advanced | Data Retrieval* training course and you use the same site for the current course, you do not need to configure HTTPS. If you are not going to process credit card payments and you do not use the OAuth 2.0 authorization, you can call API methods via HTTP.

Deploying an Acumatica ERP Instance for the Training Course

Instead of deploying a new instance, you can use the Acumatica ERP instance that you have deployed for the *I300 Web Services: Basic | Data Retrieval* or *I310 Web Services: Advanced | Data Retrieval* training course.

You deploy an Acumatica ERP instance and configure it as follows:

1. Open the Acumatica ERP Configuration Wizard, and deploy a new application instance as follows:
   a. On the **Database Configuration** page of the Acumatica ERP Configuration Wizard, type the name of the database: `MyStoreInstance`.
   b. On the **Tenant Setup** page, set up one tenant with the *I100* data inserted by specifying the following settings:
      - **Login Tenant Name**: `MyStore`
      - **New**: Selected
      - **Insert Data**: *I100*
      - **Parent Tenant ID**: 1
      - **Visible**: Selected

   The system creates a new Acumatica ERP instance, adds a new tenant, and loads the selected data.
2. Sign in to the new tenant by using the following credentials:
   - Login: `admin`
   - Password: `setup`

   Change the password when the system prompts you to do so.
3. Click the user name in the top right corner of the Acumatica ERP window, and click **My Profile**. On the **General Info** tab of the User Profile (SM203010) form, which opens, select *MYSTORE* in the **Default Branch** box; then click **Save** on the form toolbar. In subsequent sign-ins to this account, you will be signed in to this branch.

Configuring a Website for HTTPS

In the examples of this guide, you will use the secure connection between the API client application, Acumatica ERP, and the Authorize.Net payment gateway for making transactions to the Authorize.Net payment gateway through the Acumatica ERP website, as shown in the following diagram. (Acumatica ERP uses the Authorize.Net payment gateway for processing credit card payments.)

You can make API calls to Acumatica ERP (item 1 in the diagram) through HTTP; however, we recommend that you use a secure connection between the API client application and Acumatica ERP for credit card processing.

A secure connection between the Authorize.Net payment gateway and the Acumatica ERP website (item 2 in the diagram) with a Secure Socket Layer (SSL) certificate is required for making transactions. Therefore, you have to set up the Acumatica ERP website for HTTPS to be able to process credit card payments, as described in this topic.

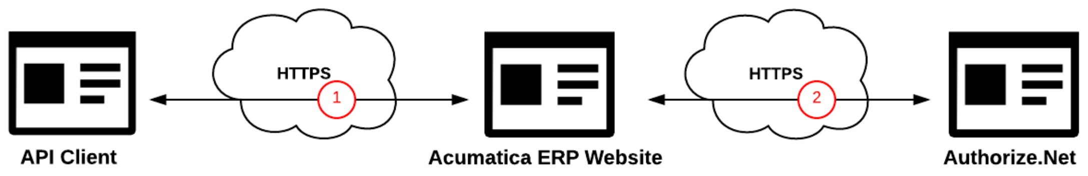


Figure: API interaction with Acumatica ERP and Authorize.Net

A secure connection between the client application and the Acumatica ERP website with an SSL certificate is also required for the authorization of the client application through OAuth 2.0.

As the Microsoft IIS documentation states, the steps for configuring SSL for a site include the following:

1. You obtain an appropriate certificate. (For the purposes of completing the course, you can create a self-signed server certificate.)
2. You create an SSL binding on a site.
3. You test the website by making a request to the site.
4. Optional: You configure the SSL options.

To complete the examples of this guide, you should create a self-signed certificate and configure SSL binding as follows:

1. Create a self-signed certificate by doing the following:
    a. In the Control Panel, open **Administrative Tools** > **Internet Information Services (IIS) Manager**.
    b. In the **Features View**, double-click **Server Certificates**.
    c. Click **Create Self-Signed Certificate** in the **Actions** pane.
    d. Enter a name for the new certificate, and click **OK**.
2. Do the following to create an SSL binding:
    a. Select a site in the tree view, and click **Bindings** in the **Actions** pane.
    b. In the **Site Bindings** dialog box, click **Add** to add your new SSL binding to the site.
    c. In the **Type** drop-down list, select *https*.
    d. Select the self-signed certificate you created, and click **OK** to close the dialog box.

3. In the **Actions** pane, under **Browse Web Site**, click the link associated with the binding you just created (*Browse*:443(https)*). The browser will display an error page because the self-signed certificate was issued by your computer rather than by a trusted certificate authority.
4. Click the link to proceed with this website and disregard the error. The HTTPS website opens.

Company Story and MyStoreIntegration Application

In this course, you will simulate the integration of Acumatica ERP with the online store of a small retail company, MyStore. This company is a single business entity that has no branches or subsidiaries. MyStore uses Acumatica ERP for customer management, sales order processing, and payment collection.

MyStore has been acting upon plans to extend its business and start selling goods online. To do this, MyStore has been investigating the options available in Acumatica ERP for integration with eCommerce applications.

In the first stage of the implementation, which is covered in the *I310 Web Services: Advanced | Data Retrieval* training course, MyStore has developed the integration application, MyStoreIntegration. Due to the development during this stage, MyStoreIntegration retrieves information about stock items, sales orders, and payments from Acumatica ERP.

In the second stage of the implementation, which this course covers, MyStoreIntegration will be expanded to submit information about customers, sales orders, and payments from the online store to Acumatica ERP. For the implementation of the MyStoreIntegration application, the MyStore company can use one of the following interfaces:

- Contract-based REST API
- Contract-based SOAP API
- Screen-based SOAP API

The OData interface can be used only for the implementation of the data retrieval part of the MyStoreIntegration application; data submission should be performed by other integration interfaces because data submission is not possible through OData. The examples of this course show the implementation of the MyStoreIntegration application with the contract-based REST and SOAP APIs. The implementation of the scenarios with the screen-based SOAP API is outside of the scope of this course.

The following diagram shows how the MyStoreIntegration integration application fits in the integration of the MyStore online store with Acumatica ERP.

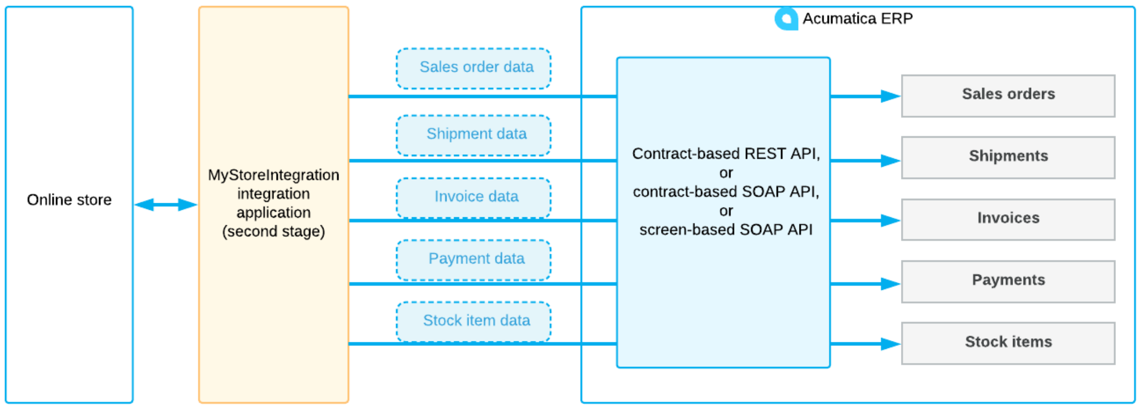


Figure: Integration of the MyStore online store and Acumatica ERP

Integration Requirements

Two types of users work with the online store application of the MyStore company: the customers who purchase goods, and the administrators who manage the online store.

In the second stage of implementation, the MyStoreIntegration application should implement integration with Acumatica ERP to support the following usage scenarios in the online store:

- A registered customer should be able to do the following:
    - Purchase goods
    - Update the purchase before it is shipped
    - Have the goods shipped
    - Register a credit card in the online store
- An administrator should be able to do the following:
    - Add new stock items to the catalog of the online store
    - Add notes and attachments to stock items

In this course, you do not implement the online store application itself; instead, you implement the integration part between the online store and Acumatica ERP, which provides the support for the listed scenarios in the online store application.

Part 1: Creation of Records

In this part of the course, you will create records through the Acumatica ERP web services APIs. You will create a shipment for particular sales orders. You will also create a stock item with attributes.

As a result of completing the lessons of this part, you will know the best practices associated with the creation of a record with detail lines through the contract-based APIs. You will also learn how to identify the attributes whose values you want to assign.

Lesson 1.1: Creating a Shipment for Sales Orders

A customer of the MyStore online store should be able to request a shipment of the goods that the customer has ordered. The customer can add multiple orders to the shipment. Information about the requested shipment should be passed to Acumatica ERP, where each shipment can be displayed on the Shipments (SO302000) form.

In the examples of lesson, you will add to the MyStoreIntegration application a request that creates a shipment for a customer from two sales orders, both of which have the *Open* status; you will do this by using the contract-based REST API and by using the contract-based SOAP API. You will add all items from the sales orders to the shipment. To add all items of a sales order to a shipment, you need to specify only the order type and order number in the details of the shipment. You can use a similar request to add any number of sales orders to the shipment.

If you need to include in a shipment particular items (rather than all items) from a sales order, you need to retrieve the sales orders with the included items from Acumatica ERP by the key fields, and then include the needed items in the shipment (by specifying the inventory ID, the warehouse ID, and the type and number of the sales order in which the item is included in the details of the shipment).

Shipments are auto-numbered in the system; therefore, during the creation of the shipment, you will not specify the shipment number, which is the key field of the document. As a result of each example in this lesson being executed, the system will create a shipment with the *On Hold* status and assign the next shipment number to it.

For more information on the workflow of sales order processing, see *Sales of Stock Items: General Information* in the documentation.

You will use the `Shipment` entity of the *Default/18.200.001* endpoint; this entity is mapped to the Shipments form.

Although you are creating a shipment with multiple detail lines, you will use one request for the creation of the shipment. (That is, you do not need to submit each detail of the shipment in a separate request.) For optimal performance of the application, it is important to use the minimum number of requests.

The implementation of this scenario with the screen-based SOAP API is outside of the scope of this course. For a summary of this implementation, see *Additional Information: Creation of a Shipment with the Screen-Based SOAP API*.

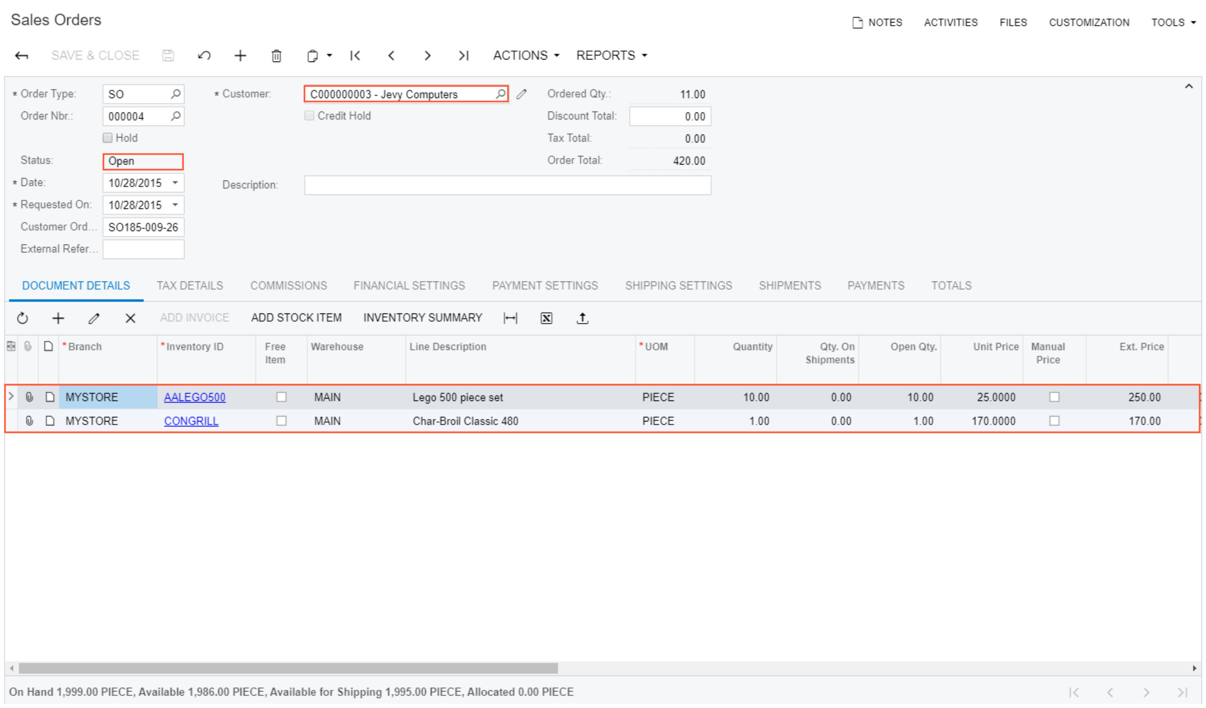
In this lesson, you will not specify the values of any custom fields (that is, fields that have been added to Acumatica ERP forms as part of a customization project) or user-defined fields, which is outside of the scope of this course. For basic information about how to specify the values of custom and user-defined fields, see *Additional Information: Setting of the Values of Custom and User-Defined Fields*. You will not specify the values of multi-language fields (that is, text boxes on Acumatica ERP forms in which users can type values in multiple languages if multiple locales have been configured in the system) either. For basic information about multi-language fields, see *Additional Information: Setting of the Values of Multi-Language Fields*.

Lesson Objective

In this lesson, you will learn how to create a new record in Acumatica ERP by using the contract-based APIs.

Prerequisites

Before you complete the examples of this lesson, on the *Sales Orders* (SO301000) form, find the sales orders with the order numbers *000004* and *000006*. These are the preconfigured sales orders of the customer with the customer ID *C000000003*. Make sure that these sales orders have the *Open* status. The sales order *000004*, shown in the following screenshot, includes two stock items (that is, two detail lines). The sales order *000006* includes one stock item.



* Order Type:	SO
Order Nbr.:	000004
Hold	☐
Status:	Open
* Date:	10/28/2015
* Requested On:	10/28/2015
Customer Ord...	SO185-009-26
External Refer...	
* Customer:	C000000003 - Jevy Computers
Credit Hold	☐
Description:	
Ordered Qty.:	11.00
Discount Total:	0.00
Tax Total:	0.00
Order Total:	420.00

* Branch	* Inventory ID	Free Item	Warehouse	Line Description	* UOM	Quantity	Qty. On Shipments	Open Qty.	Unit Price	Manual Price	Ext. Price
MYSTORE	AALEGO500	☐	MAIN	Lego 500 piece set	PIECE	10.00	0.00	10.00	25.0000	☐	250.00
MYSTORE	CONGRILL	☐	MAIN	Char-Broil Classic 480	PIECE	1.00	0.00	1.00	170.0000	☐	170.00

On Hand 1,999.00 PIECE, Available 1,986.00 PIECE, Available for Shipping 1,995.00 PIECE, Allocated 0.00 PIECE

Figure: One of two sales order to create a shipment for

If you want to perform an example of this lesson multiple times with the same data or complete both the REST example and the SOAP example of this lesson, you need to remove the shipment that is created as a result of the completion of the example before you try to create the shipment once again. To remove the shipment, on the *Shipments* (SO302000) form, select the shipment, and click **Delete** on the form toolbar.

When you complete the examples in this lesson, you can also use the data of any other two sales orders with the *Open* status created for the same customer. If you do this, you need to update the data in the examples accordingly.

Example 1.1.1: Using One PUT Request (REST)

In this example, by using the contract-based REST API, you will create a shipment for multiple sales orders.

To retrieve sales orders with the included items for the shipment, you will use the `GET` HTTP method with the `$expand` and `$select` parameters. You will specify the key fields of the sales order in the URL of the request.

To submit a shipment with all items of the particular sales orders to Acumatica ERP, you will use one `PUT` HTTP request. You will specify the data of the new shipment in the body of the request. In the `PUT` request, you will use the `$expand` and `$select` parameters to limit the list of fields whose values are returned in the response. The response body will contain only the fields specified in the `$select` parameter and the fields specified in the body of the request. (In the `$expand` parameter, you have to specify all the nested entities whose fields you specify in the `$select` parameter.) If you do not specify any parameters, the response will contain all fields of the new record.

Creating a Shipment for Sales Orders

To create a shipment for two sales orders by using the contract-based REST API, do the following:

Before you proceed with these instructions, you need to obtain the access token (if your application uses OAuth 2.0 authorization) or perform the sign-in request.

1. Create the shipment for the *000004* and *000006* sales orders as follows:

a. In the Postman collection, add a contract-based REST API request with the following settings:

Instead of creating a collection, you can import to Postman the collection provided with this course (`REST.postman_collection.json`). This collection has been configured for this course and already contains all the requests that are used in the course. You can use this collection for testing the requests. (You can find the file in *https://github.com/Acumatica/Help-and-Training-Examples/tree/2019R2/IntegrationDevelopment/I320*.)

- HTTP method: `PUT`
- URL: *https://localhost/MyStoreInstance/entity/Default/18.200.001/Shipment*
- Parameters of the request:

Parameter	**Value**
`$expand`	`Details`
`$select`	`Type,ShipmentNbr,Status,Details/InventoryID`

- Headers:

Key	**Value**
`Accept`	`application/json`
`Content-Type`	`application/json`

- Body:

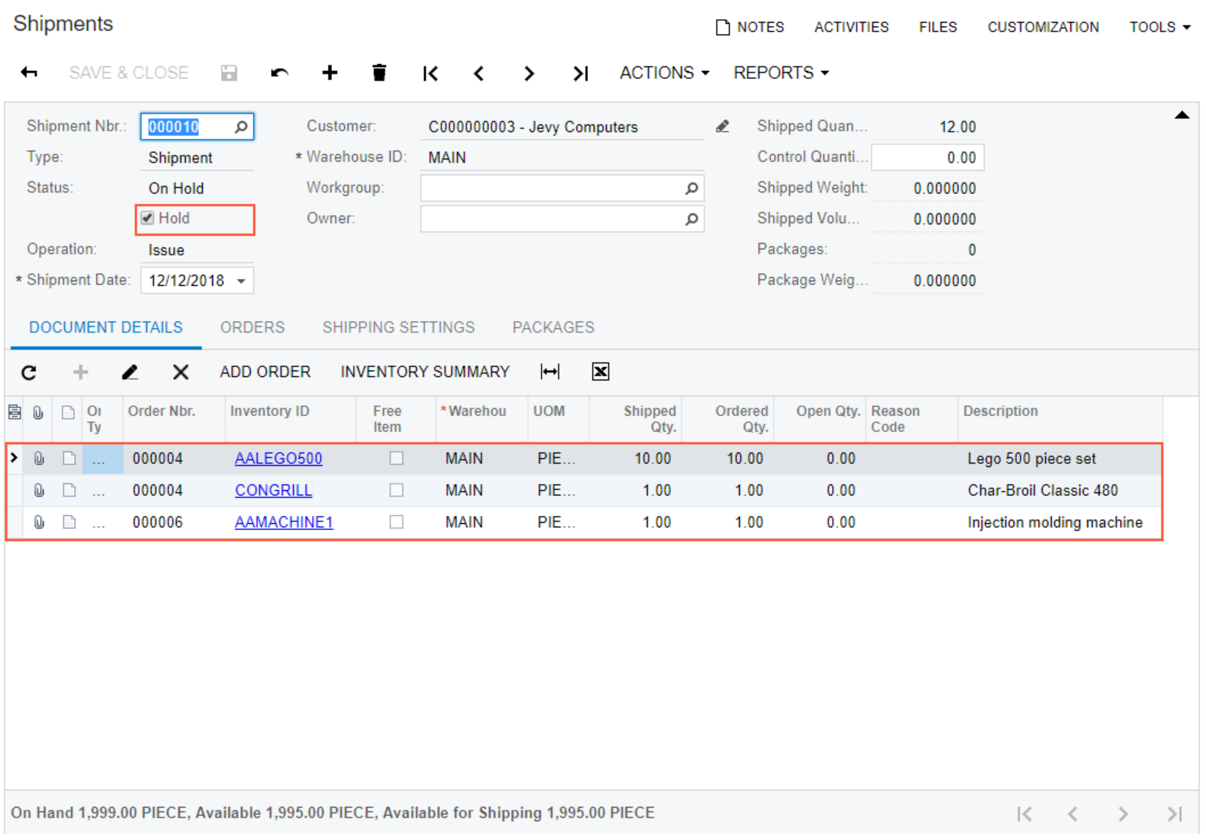
```
{
 "Type":{"value":"Shipment"},
 "CustomerID":{"value":"C000000003"},
 "WarehouseID":{"value":"MAIN"},
 "Details":[
  {
   "OrderType":{"value":"SO"},
   "OrderNbr":{"value":"000004"}
  },
  {
   "OrderType":{"value":"SO"},
   "OrderNbr":{"value":"000006"}
  }
 ]
}
```

b. Send the request. If the request is successful, the response contains the `200 OK` status code and includes the list of the requested fields of the new shipment record. The following code example shows the response body.

```
{
    "id": "10b20afa-a4d8-46d7-8195-60e759399cb4",
    "rowNumber": 1,
    "note": "",
    "CustomerID": {
        "value": "C000000003"
    },
    "Details": [
        {
            "id": "624acfa9-e539-4250-9f85-bd69f0654b27",
            "rowNumber": 1,
            "note": "",
            "InventoryID": {
                "value": "AALEGO500"
            },
            "OrderNbr": {
                "value": "000004"
            },
            "OrderType": {
                "value": "SO"
            },
            "WarehouseID": {
                "value": "MAIN"
            },
            "custom": {},
            "files": []
        },
        {
            "id": "c56da0d9-11d8-47f9-8744-253f6c200394",
            "rowNumber": 2,
            "note": "",
            "InventoryID": {
                "value": "CONGRILL"
            },
            "OrderNbr": {
                "value": "000004"
            },
            "OrderType": {
                "value": "SO"
            },
            "WarehouseID": {
                "value": "MAIN"
            },
            "custom": {},
            "files": []
        },
        {
            "id": "ed58d9ad-c75d-4472-9382-503258c6152b",
            "rowNumber": 3,
            "note": "",
            "InventoryID": {
                "value": "AAMACHINE1"
            },
            "OrderNbr": {
                "value": "000006"
            },
            "OrderType": {
                "value": "SO"
            },
            "WarehouseID": {
                "value": "MAIN"
            },
            "custom": {},
            "files": []
        }
    ],
    "ShipmentNbr": {
        "value": "000010"
    },
    "Status": {
        "value": "On Hold"
    },
    "Type": {
        "value": "Shipment"
    },
    "WarehouseID": {
        "value": "MAIN"
    },
    "custom": {},
    "files": []
}
```

c. Save the request.

On the *Shipments* (SO302000) form, make sure the shipment with the number returned in the `ShipmentNbr` field exists, has the *On Hold* status, and contains three detail lines, as shown in the following screenshot.



Shipment Nbr.:	000010	Customer:	C000000003 - Jevy Computers	Shipped Quan...	12.00
Type:	Shipment	* Warehouse ID:	MAIN	Control Quanti...	0.00
Status:	On Hold	Workgroup:		Shipped Weight:	0.000000
	☑ Hold	Owner:		Shipped Volu...	0.000000
Operation:	Issue			Packages:	0
* Shipment Date:	12/12/2018			Package Weig...	0.000000

Or Ty	Order Nbr.	Inventory ID	Free Item	*Warehou	UOM	Shipped Qty.	Ordered Qty.	Open Qty.	Reason Code	Description
...	000004	AALEGO500	☐	MAIN	PIE...	10.00	10.00	0.00		Lego 500 piece set
...	000004	CONGRILL	☐	MAIN	PIE...	1.00	1.00	0.00		Char-Broil Classic 480
...	000006	AAMACHINE1	☐	MAIN	PIE...	1.00	1.00	0.00		Injection molding machine

On Hand 1,999.00 PIECE, Available 1,995.00 PIECE, Available for Shipping 1,995.00 PIECE

Figure: The shipment with detail lines

Related Links

Creation of a Record
Representation of a Record in JSON Format
Parameters for Retrieving Records

Example 1.1.2: Using One Call of the Put() Method (SOAP)

This example shows how you can create a shipment for multiple sales orders by using the contract-based SOAP API.

To submit a shipment with all items of the particular sales orders to Acumatica ERP, you will use one call of the `Put()` method of the instance of the `DefaultSoapClient` class. You will specify the data of the new shipment by using the `StringValue` classes. You will specify `ReturnBehavior = ReturnBehavior.OnlySpecified` and use the `StringReturn` classes to limit the number of fields returned after the new shipment is created.

Creating a Shipment for Sales Orders

To create a shipment for two sales orders by using the contract-based SOAP API, do the following:

Before you proceed with these instructions, you need to obtain the access token (if your application uses OAuth 2.0 authorization) or perform the sign-in request.

1. In the `Integration` folder of the *MyStoreIntegration* project, create the `CreationOfRecords` class, and add the `MyStoreIntegration.Default using` directive in the created file.

   You can use the solution provided with this course (`MyStoreIntegrationStage2\CBAPI\MyStoreIntegration.sln`). This solution has been configured for this course and already contains all the methods that are used in this course. You can use this solution for testing.

2. In the `CreationOfRecords` class, define the `CreateShipment()` method as follows.

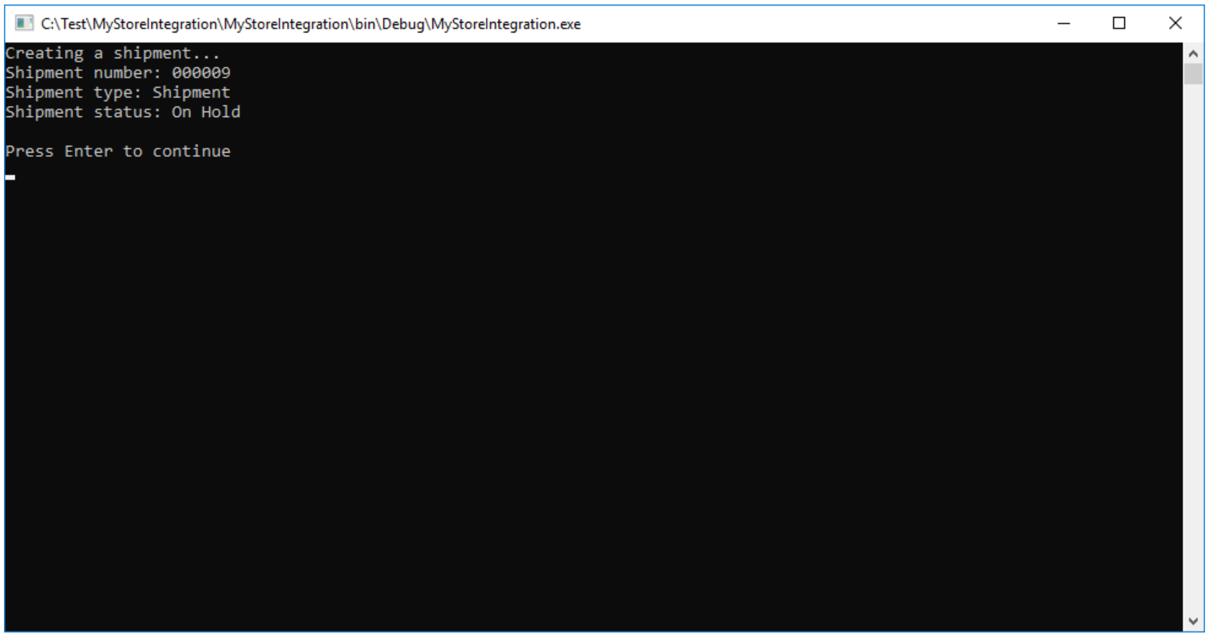
```
//Creating a shipment
public static void CreateShipment(DefaultSoapClient soapClient)
{
    Console.WriteLine("Creating a shipment...");

    //Shipment data
    string shipmentType = "Shipment";
    string customerID = "C000000003";
    string warehouse = "MAIN";
    //Sales order with the Open status for the specified customer
    string firstOrderNbr = "000004";
    string firstOrderType = "SO";
    //Sales order with the Open status for the specified customer
    string secondOrderNbr = "000006";
    string secondOrderType = "SO";

    //Specify the values of a new shipment
    Shipment shipment = new Shipment
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        Type = new StringValue { Value = shipmentType },
        ShipmentNbr = new StringReturn(),
        Status = new StringReturn(),
        CustomerID = new StringValue { Value = customerID },
        WarehouseID = new StringValue { Value = warehouse },
        Details = new[]
        {
            new ShipmentDetail
            {
                OrderType = new StringValue {Value = firstOrderType },
                OrderNbr = new StringValue {Value = firstOrderNbr},
            },
            new ShipmentDetail
            {
                OrderType = new StringValue {Value = secondOrderType },
                OrderNbr = new StringValue {Value = secondOrderNbr},
            }
        }
    };
```

```
        //Create a shipment with the specified values
        Shipment newShipment = (Shipment)soapClient.Put(shipment);

        //Display the summary of the created record
        Console.WriteLine("Shipment number: " + newShipment.ShipmentNbr.Value);
        Console.WriteLine("Shipment type: " + newShipment.Type.Value);
        Console.WriteLine("Shipment status: " + newShipment.Status.Value);
        Console.WriteLine();
        Console.WriteLine("Press Enter to continue");
        Console.ReadLine();
    }
```

3. In the `try` block of the `Main()` method of the `Program` class, call the `CreateShipment()` method of the `CreationOfRecords` class, as the following code shows.

```
//Create a shipment
CreationOfRecords.CreateShipment(soapClient);
```

4. Rebuild the project, and run the application. The type, number, and status of the created shipment are displayed in the console application window, as shown in the following screenshot. The system assigns to the shipment the next number according to the numbering sequence.


C:\Test\MyStoreIntegration\MyStoreIntegration\bin\Debug\MyStoreIntegration.exe

```
Creating a shipment...
Shipment number: 000009
Shipment type: Shipment
Shipment status: On Hold

Press Enter to continue
```

Figure: Console application window

On the *Shipments* (SO302000) form, open the shipment with the number returned by the console application. Notice that it includes three stock items from two sales orders (*000004* and *000006*). The created shipment has the *On Hold* status.

Related Links

Put() Method
ReturnBehavior Property (Contract Version 3)

Additional Information: Creation of a Shipment with the Screen-Based SOAP API

This scenario is outside of the scope of this course but may be useful to some readers.

To create a record through the screen-based SOAP API, you use the `Submit()` method of a `Screen` object that corresponds to the needed Acumatica ERP form. For details about this method, see *Submit() Method*.

Before you start specifying the sequence of commands for the `Submit()` method, you should review the form you will use for submitting data and understand the sequence of actions that occur when a user manually enters data on the form. To review this process, you can enter one record manually and pay attention to the sequence of actions. This sequence of commands is similar to the sequence of commands you configure when creating import and export scenarios.

To add items to the shipment, you need to imitate (by using the screen-based SOAP API) the opening of the **Add Sales Order** dialog box on the *Shipments* (SO302000) form. In this dialog box, you need to select for shipment the stock items from the open sales orders. For the configuration of the commands for this sequence of actions, you may find useful the following information:

- *Commands for Setting the Values of Elements*
- *Commands That Require a Commit*
- *Commands for Pop-Up Panels*
- *Commands for Clicking Buttons on a Form*
- *Commands for Retrieving the Values of Elements*

Additional Information: Setting of the Values of Custom and User-Defined Fields

In a customization project, you can add custom fields to Acumatica ERP forms. You can also add user-defined fields to Acumatica ERP forms and include them in a customization project. (For details about user-defined fields, see *User-Defined Fields*.)

You can specify the values of custom fields through the web services APIs, while the values of user-defined fields can be specified only through the contract-based APIs. This scenario is outside of the scope of this course but may be useful to some readers.

Using the Contract-Based REST API

To specify the values of custom and user-defined fields through the contract-based REST API, you specify the values of these fields in the body of the `PUT` request. You specify the values of custom and user-defined fields in JSON format, as described in *Representation of a Record in JSON Format*. You can use the same approach to specify the values of any fields that are not included in the entity definition. For details about working with fields that are not included in the entity definition, see *Custom Fields* in the documentation.

You can also add custom fields to a custom endpoint or endpoint extension and work with them by using the same approach as was described in this lesson for the fields of the system endpoint. User-defined fields cannot be added to custom endpoints and endpoint extensions. For details about custom endpoints and endpoint extensions, see *Custom Endpoints and Endpoint Extensions* in the documentation.

Using the Contract-Based SOAP API

To specify the values of custom and user-defined fields through the contract-based SOAP API, you should use the `CustomFields` property of an entity of the endpoint. You can use the same approach to specify the values of any fields that are not included in the entity definition. For details about working with fields that are not included in the entity definition, see *Custom Fields* in the documentation. For details about the `CustomFields` property, see *CustomFields Property*.

You can also add custom fields to a custom endpoint or endpoint extension and work with them by using the same approach as was described in this lesson for the fields of the system endpoint. User-defined fields cannot be added to custom endpoints and endpoint extensions. For details about custom endpoints and endpoint extensions, see *Custom Endpoints and Endpoint Extensions* in the documentation.

Using the Screen-Based SOAP API

To specify the values of custom fields through the screen-based SOAP API, you need to generate a WSDL description of the service for the Acumatica ERP form that contains custom fields after the publication of the customization project that has added these fields. After you have added the service reference to the project of your application, you can work with the custom fields in the same way as you work with the fields available on the Acumatica ERP forms out of the box.

The values of user-defined fields cannot be specified through the screen-based SOAP API.

Additional Information: Setting of the Values of Multi-Language Fields

For some text boxes on Acumatica ERP forms, users can type values in multiple languages if multiple locales have been configured in Acumatica ERP. You can specify the values of these fields through the web services APIs. (This scenario is outside of the scope of this course but may be useful to some readers.)

For details about how to specify the values of multi-language fields, see the following topics in the documentation:

- For the REST API, *Multi-Language Fields*
- For the contract-based SOAP API, *Multi-Language Fields*
- For the screen-based SOAP API, *Commands for Working with Multi-Language Fields*

Lesson Summary

In this lesson, you have learned how to create a record in Acumatica ERP by using the contract-based APIs. During record creation, you have submitted the fully configured record (with all detail lines) in one request.

You have also reviewed how the creation of a record can be implemented with the screen-based SOAP API, and how the values of custom fields and multi-language fields can be submitted to Acumatica ERP.

Lesson 1.2: Creating a Stock Item with Attributes

The administrator of the MyStore company's online store can add inventory items to the catalog by using the administrator interface of the online store. The online store passes the settings of the added stock items to Acumatica ERP by using the contract-based API.

In this lesson, you will add to the MyStoreIntegration REST or SOAP application a method that creates a stock item record with attributes in Acumatica ERP. An attribute is a special property of an object in the system that specifies additional information that is not defined by the standard properties of the object (that is, those supported by the standard UI elements).

For this lesson, in the Acumatica ERP instance that you are using for the training course, two attributes have been preconfigured to provide the ability to add and track additional information to stock items. The attributes with the *OPERATSYST* and *SOFTVER* identifiers have been configured on the *Attributes* (CS205000) form. These attributes have been assigned to the *STOCKITEM* item class on the *Item Classes* (IN201000) form. Thus, if you select this item class in the **Item Class** box on the **General Settings** tab of the *Stock Items* (IN202500) form for a stock item, the **Attributes** tab is available on the form and you can specify the values of the attributes. In this lesson, you will specify the values of these attributes for a new stock item through the contract-based APIs.

To create a stock item, you will use the `StockItem` entity of the *Default/18.200.001* endpoint. The `StockItem` entity is mapped to the Stock Items form. In the stock item data, you will specify the value of the inventory ID, which is the key field of a stock item. Because the key field value is passed in the stock item data, the system searches for a stock item record with the specified key and does one of the following:

- If the record has been found, updates this record
- If the record has not been found, adds a new stock item record

To specify the values of attributes, you will use the `Attributes` field of the `StockItem` entity. To identify the attribute whose value you want to specify, in the `AttributeID` field of the `AttributeValue` entity, you will specify the attribute name, as specified in the **Description** box on the Attributes form).

The creation of records with attributes through the screen-based SOAP API is outside of the scope of this course. For summary information about this scenario, see *Additional Information: Creation of Records with Attributes with the Screen-Based SOAP API*.

Lesson Objective

In this lesson, you will learn how to create records with attributes.

Example 1.2.1: Using the Attributes Field (REST)

In this example, by using the contract-based REST API, you will create in Acumatica ERP the *BASESERV1* stock item, which has the **Operation System** and **Version of Software** attributes specified.

To create a stock item, you will use the `PUT` HTTP method. You will specify the data of the new stock item (including the values of the attributes) in the body of the request. In the `PUT` request, you will use the `$expand` and `$select` parameters to limit the list of fields whose values are returned in the response. The response will contain the values of the following fields:

- The fields that you will specify in the `$select` parameter
- All the fields of the entity that you will specify in the `$expand` parameter (because you will not specify particular fields of this entity in the `$select` parameter)
- The fields that you will specify in the request body

Creating a Stock Item with Attributes

To create a stock item by using the contract-based REST API, do the following:

1. In Postman, add a request with the following settings:
    - HTTP method: `PUT`
    - URL: *https://localhost/MyStoreInstance/entity/Default/18.200.001/StockItem*
    - Parameters of the request:

Parameter	**Value**
`$expand`	`Attributes`
`$select`	`InventoryID`

    - Headers:

Key	**Value**
`Accept`	`application/json`
`Content-Type`	`application/json`

    - Body:

```
{
 "InventoryID":{"value":"BASESERV1"},
 "Description":{"value":"Baseline level of performance"},
 "ItemClass":{"value":"STOCKITEM"},
 "Attributes":[
  {
   "AttributeID":{"value":"Operation System"},
   "Value":{"value":"Windows"}
  },
  {
   "AttributeID":{"value":"Version of Software"},
   "Value":{"value":"Server 2012 R2"}
  }
 ]
}
```

2. Send the request. If the request is successful, the response contains the `200 OK` status and includes the list of requested fields of the new stock item record in JSON format. The following code example shows the response body.

```
{
    "id": "50e6754f-b383-4745-88b8-8f7bc127f8d8",
    "rowNumber": 1,
    "note": null,
    "Attributes": [
        {
            "id": "2322ad17-9335-4a15-80bf-ffc82e7cb718",
            "rowNumber": 1,
            "note": null,
            "AttributeID": {
                "value": "Operation System"
            },
            "Required": {
                "value": false
            },
            "Value": {
                "value": "Windows"
            },
            "custom": {},
            "files": []
        },
        {
            "id": "cc843a41-f935-496e-9538-fbe7f6c69cdf",
            "rowNumber": 2,
            "note": null,
            "AttributeID": {
                "value": "Version of Software"
            },
            "Required": {
                "value": false
            },
            "Value": {
                "value": "Server 2012 R2"
            },
            "custom": {},
            "files": []
        }
    ],
    "Description": {
        "value": "Baseline level of performance"
    },
    "InventoryID": {
        "value": "BASESERV1"
    },
    "ItemClass": {
        "value": "STOCKITEM"
    },
    "custom": {},
    "files": []
}
```

3. Save the request.

Related Links

Creation of a Record

Example 1.2.2: Using the Attributes Field (SOAP)

In this example, by using the contract-based SOAP API, you will create in Acumatica ERP the *BASESERV2* stock item, which has the **Operation System** and **Version of Software** attributes specified.

To create a stock item, you will use the `Put()` method of an instance of the `DefaultSoapClient` class. You will specify the data of the new stock item (including the values of the attributes) by using the `StringValue` classes.

You will specify `ReturnBehavior = ReturnBehavior.OnlySpecified` to limit the number of fields returned after the new stock item is created.

Creating a Stock Item with Attributes

To create a stock item with attributes by using the contract-based SOAP API, do the following:

1. In the `CreationOfRecords` class, add the `CreateStockItem()` method that is shown in the following code.

```
//Creating a stock item with attributes
public static void CreateStockItem(DefaultSoapClient soapClient)
{
    Console.WriteLine("Creating a stock item with attributes...");

    //Stock item data
    string inventoryID = "BASESERV2";
    string itemDescription = "Baseline level of performance";
    //Item class that has attributes defined
    string itemClass = "STOCKITEM";
    //An attribute of the item class (STOCKITEM)
    string attributeName1 = "Operation System";
    string attributeValue1 = "Windows";
    //An attribute of the item class (STOCKITEM)
    string attributeName2 = "Version Of Software";
    string attributeValue2 = "Server 2012 R2";

    //Specify the values of the new stock item
    StockItem stockItemToBeCreated = new StockItem
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        InventoryID = new StringValue { Value = inventoryID },
        Description = new StringValue { Value = itemDescription },
        ItemClass = new StringValue { Value = itemClass },
        Attributes = new[]
        {
            new AttributeValue
            {
                AttributeID = new StringValue { Value = attributeName1 },
                Value = new StringValue { Value = attributeValue1 }
            },
            new AttributeValue
            {
                AttributeID = new StringValue { Value = attributeName2 },
                Value = new StringValue { Value = attributeValue2 }
            }
        }

    };

    //Create a stock item with the specified values
    StockItem newStockItem = (StockItem)soapClient.Put(stockItemToBeCreated);

    //Display the summary of the created stock item
    Console.WriteLine("Inventory ID: " + newStockItem.InventoryID.Value);
    foreach (AttributeValue attr in newStockItem.Attributes)
    {
        Console.WriteLine("Attribute name: " + attr.AttributeID.Value);
```

```
            Console.WriteLine("Attribute value: " + attr.Value.Value);
        }
        Console.WriteLine();
        Console.WriteLine("Press Enter to continue");
        Console.ReadLine();
    }
```

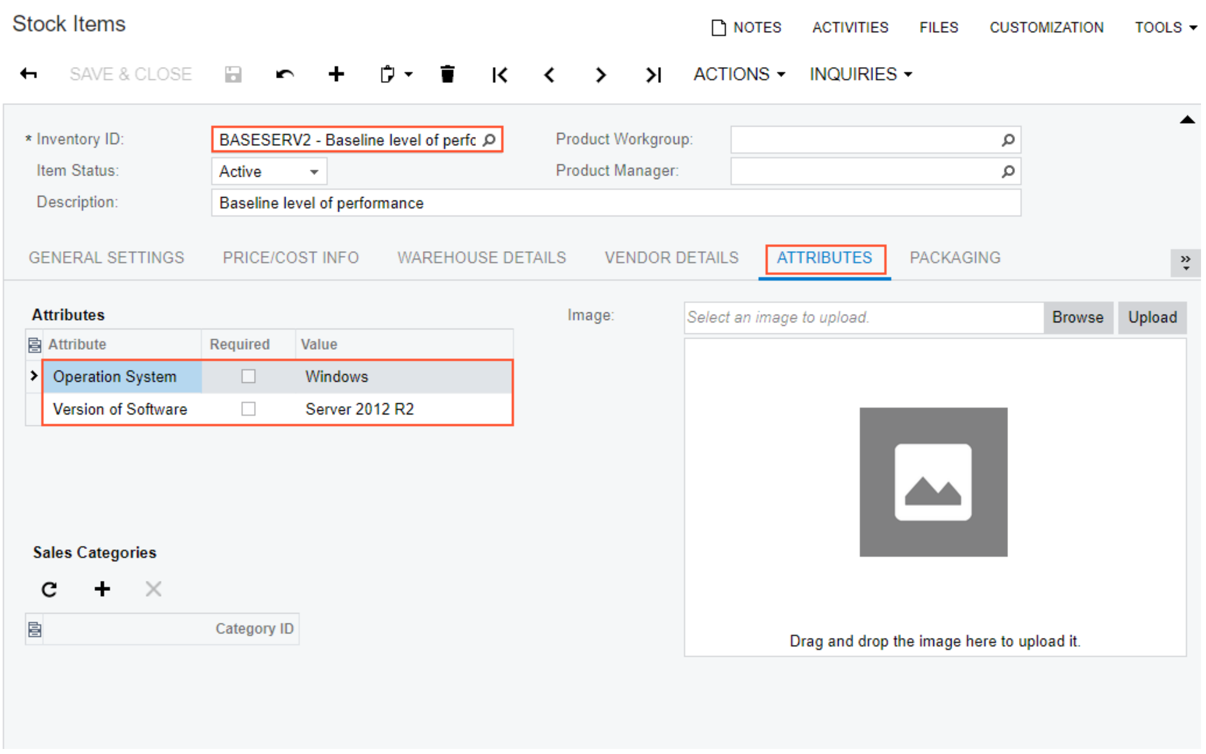
2. In the `try` block of the `Main()` method of the `Program` class, call the `CreateStockItem()` method of the `CreationOfRecords` class, as the following code shows.

   You can comment the code of the previous example for the contract-based SOAP API in the `Program.Main()` method.

```
//Create a stock item with attributes
CreationOfRecords.CreateStockItem(soapClient);
```

3. Rebuild the project, and run the application. The inventory ID and the values of the added attributes are displayed in the console application window.

   On the Stock Items (IN202500) form, select the *BASESERV2* stock item, and make sure the item has the attributes specified, as shown in the following screenshot.



* Inventory ID:	BASESERV2 - Baseline level of perfc
Item Status:	Active
Description:	Baseline level of performance
Product Workgroup:	
Product Manager:	

Attributes

Attribute	Required	Value
Operation System	☐	Windows
Version of Software	☐	Server 2012 R2

Figure: Attributes on the form

Related Links

Attributes Property
Put() Method

Additional Information: Creation of Records with Attributes with the Screen-Based SOAP API

This scenario is outside of the scope of this course but may be useful to some readers.

To specify the values of attributes through the screen-based SOAP API, you use the same approach as you use to specify the values of any other box on the Acumatica ERP form. To create a record, you use the `Submit()` method of a `Screen` object that corresponds to the needed Acumatica ERP form. For details about this method, see *Submit() Method*. To specify the value of an attribute, you use the `Value` command, as described in *Commands for Setting the Values of Elements*.

Lesson Summary

In this lesson, you have learned how to create records with attributes through the contract-based APIs. To specify the values of the attributes of a stock item, you have used the `Attributes` field of the `StockItem` entity. To identify the attribute whose value you need to specify, in the `AttributeID` field of the `AttributeValue` entity, you have specified the attribute name.

You have also reviewed how to create records with attributes through the screen-based SOAP API.

Part 2: Update of Records

In this part of the course, you will update records in Acumatica ERP through the web services APIs. You will update a customer record by using one request. You will update the detail lines of a sales order by retrieving the sales order with detail lines from Acumatica ERP in one request and then submitting the updated sales order in another request.

As a result of completing the lessons of this part, you will know which techniques to use when you are updating a record or its detail lines through the contract-based APIs.

Lesson 2.1: Updating a Customer Account

In the online store of the MyStore company, a customer can order goods if an account has been created for the customer in the online store. The customer can view and edit the information in the customer account. When a customer account is created or updated, the online store needs to pass customer data to Acumatica ERP.

The MyStore company wants to use email addresses for the authorization of customers to use the online store. Therefore, to search for a customer record in Acumatica ERP, the MyStoreIntegration application will use the customer's email address. If a registered customer needs to update some customer data, the online store will submit the email address of the customer and the updated information to Acumatica ERP, where customer account data is entered and maintained on the *Customers* (AR303000) form.

A customer of the MyStore online store should also be able to specify a billing contact in addition to the main contact, which the customer specifies during initial registration to the online store. In Acumatica ERP, this contact is specified on the **Billing Settings** tab of the Customers form.

In this lesson, you will update a customer record in Acumatica ERP by using the contract-based API: You will specify a different customer class for an existing customer record and add a new billing contact for this customer. You will search for the needed record by using the email address of the customer.

You will use the `Customer` entity of the *Default/18.200.001* endpoint. This entity is mapped to the Customers form. You will use one request to update both the customer class and the billing contact.

The implementation of this scenario with the screen-based SOAP API is outside of the scope of this course. For a summary of this implementation, see *Additional Information: Update of a Customer Record with the Screen-Based SOAP API*.

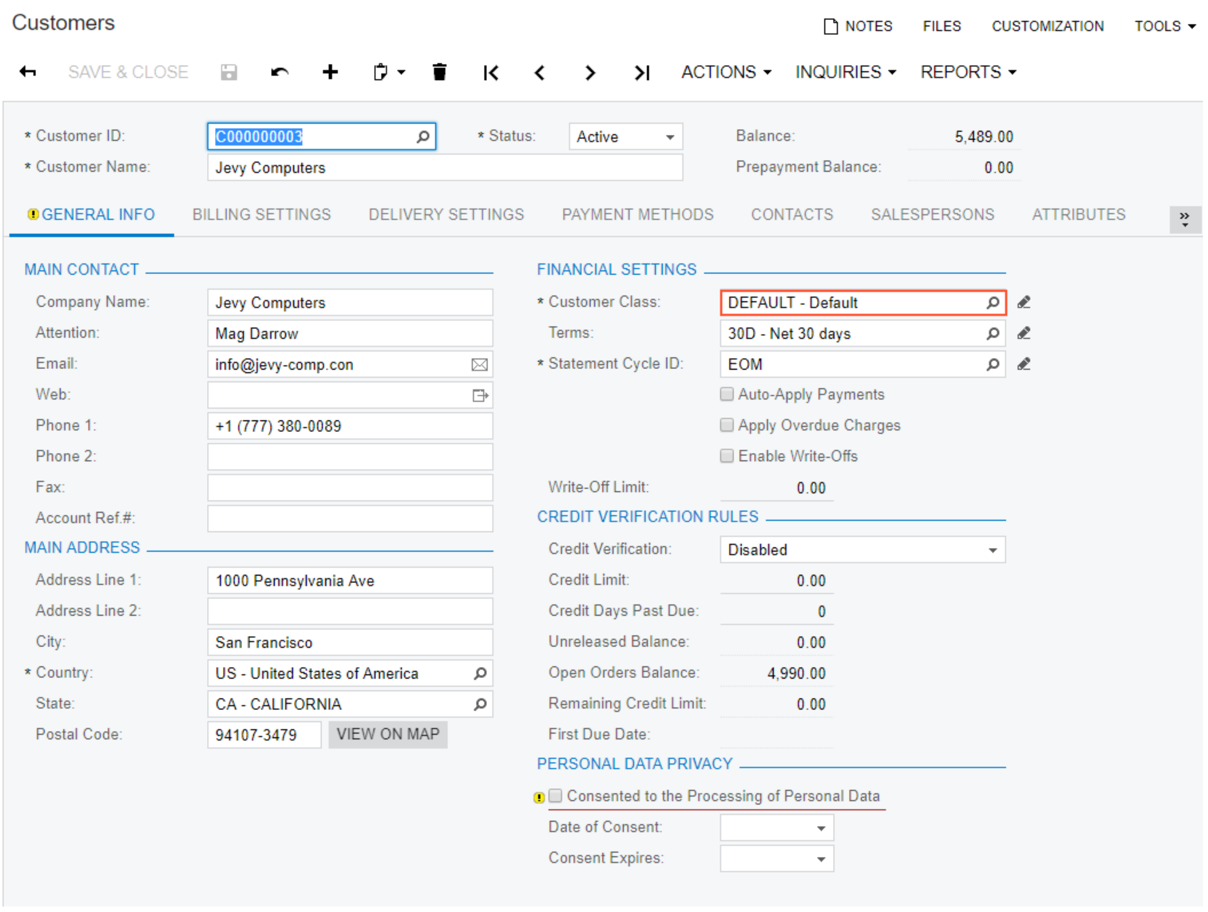
The removal of a record is outside of the scope of this course. For basic information about this scenario, see *Additional Information: Removal of a Record*.

Lesson Objective

In this lesson, you will learn how to update an existing record by using the contract-based APIs.

Prerequisites

In this lesson, you will update the customer class of the customer record that has the *info@jevy-comp.con* email address. Before you complete the examples of this lesson, on the *Customers* (AR303000) form, select this record, which has the *C000000003* customer ID, and view its settings. Notice that this customer record currently is assigned the *DEFAULT* customer class, as shown in the following screenshot.



* Customer ID:	C000000003	* Status:	Active	Balance:	5,489.00
* Customer Name:	Jevy Computers			Prepayment Balance:	0.00

MAIN CONTACT

Company Name:	Jevy Computers
Attention:	Mag Darrow
Email:	info@jevy-comp.con
Web:	
Phone 1:	+1 (777) 380-0089
Phone 2:	
Fax:	
Account Ref.#:	

MAIN ADDRESS

Address Line 1:	1000 Pennsylvania Ave
Address Line 2:	
City:	San Francisco
* Country:	US - United States of America
State:	CA - CALIFORNIA
Postal Code:	94107-3479

FINANCIAL SETTINGS

* Customer Class:	DEFAULT - Default
Terms:	30D - Net 30 days
* Statement Cycle ID:	EOM
	☐ Auto-Apply Payments
	☐ Apply Overdue Charges
	☐ Enable Write-Offs
Write-Off Limit:	0.00

CREDIT VERIFICATION RULES

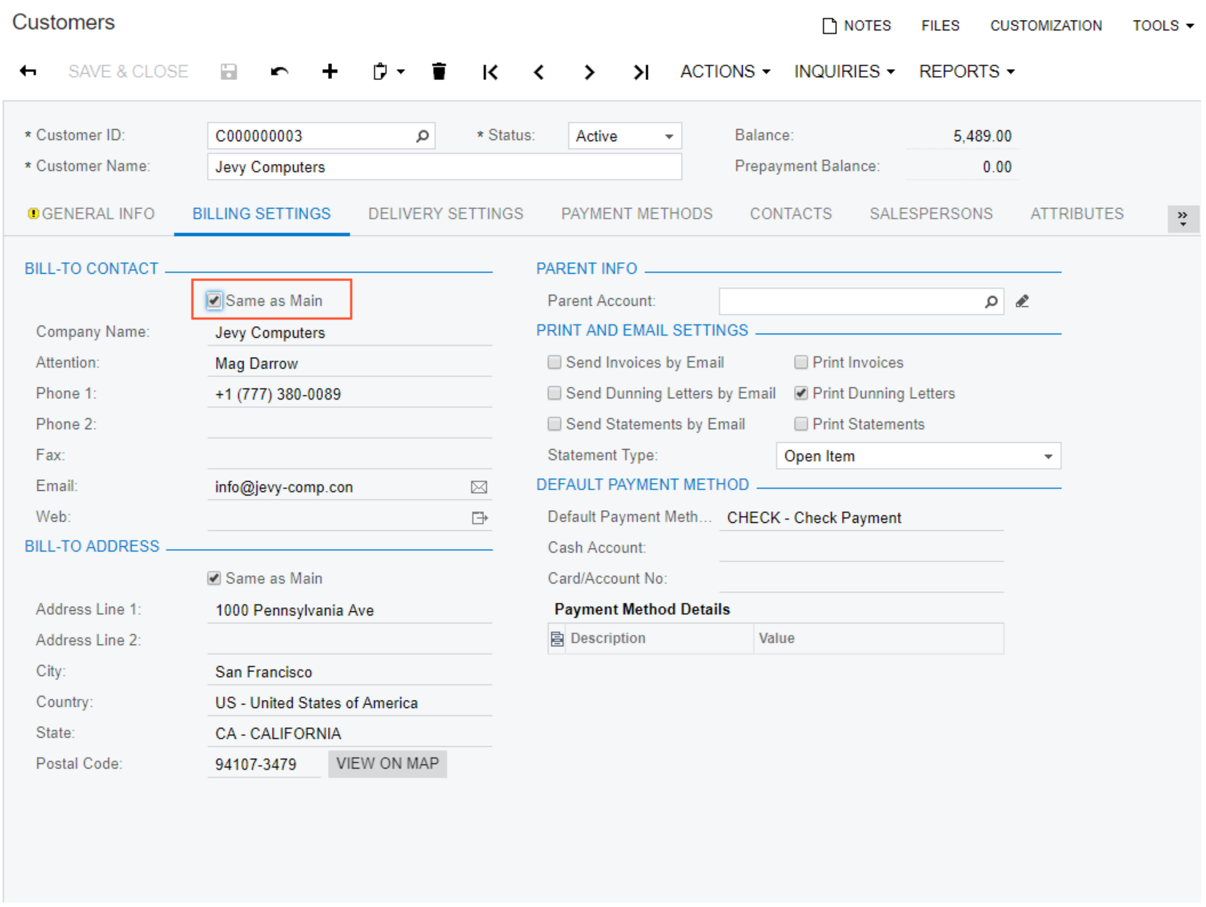
Credit Verification:	Disabled
Credit Limit:	0.00
Credit Days Past Due:	0
Unreleased Balance:	0.00
Open Orders Balance:	4,990.00
Remaining Credit Limit:	0.00
First Due Date:	

PERSONAL DATA PRIVACY

	☐ Consented to the Processing of Personal Data
Date of Consent:	
Consent Expires:	

Figure: Customer record

On the **Billing Settings** tab, you can see that this customer currently has the same billing contact as the main contact of the customer. That is, the **Same as Main** check box is selected, as shown in the following screenshot.



* Customer ID:	C000000003
* Customer Name:	Jevy Computers
* Status:	Active
Balance:	5,489.00
Prepayment Balance:	0.00

BILL-TO CONTACT

	✓ Same as Main
Company Name:	Jevy Computers
Attention:	Mag Darrow
Phone 1:	+1 (777) 380-0089
Phone 2:	
Fax:	
Email:	info@jevy-comp.con
Web:	

BILL-TO ADDRESS

	✓ Same as Main
Address Line 1:	1000 Pennsylvania Ave
Address Line 2:	
City:	San Francisco
Country:	US - United States of America
State:	CA - CALIFORNIA
Postal Code:	94107-3479

PARENT INFO

Parent Account:

PRINT AND EMAIL SETTINGS

☐ Send Invoices by Email	☐ Print Invoices
☐ Send Dunning Letters by Email	✓ Print Dunning Letters
☐ Send Statements by Email	☐ Print Statements
Statement Type:	Open Item

DEFAULT PAYMENT METHOD

Default Payment Meth...	CHECK - Check Payment
Cash Account:	
Card/Account No:	

Payment Method Details

Description	Value

Figure: Customer billing contact

If you want to perform an example of this lesson multiple times with the same data or complete both the REST example and the SOAP example of this lesson, after you complete the example, you need to restore the customer data on the Customers form before you attempt to update the customer once again. To restore the customer data, change the customer class to *Default* in the **Customer Class** box on the **General Info** tab and select the **Same as Main** check box on the **Billing Settings** tab.

When you complete an example in this lesson, you can also use any other customer record that exists in the system. If you use a different record, you need to change the email address in the example to the email address of the needed customer.

Example 2.1.1: Using PUT and $filter (REST)

This example shows how you can update a customer record by using the contract-based REST API.

You will use the `PUT` HTTP method to update the record. You will specify the `$filter` parameter to find the needed customer record by using the email address.

When the Acumatica ERP contract-based web services receive a `PUT` request that contains at least one `$filter` parameter, Acumatica ERP tries to search for the record by using the specified search value or values. If a record that satisfies the specified conditions is found, Acumatica ERP updates the fields of the record that are specified in the body of the request. If no record that satisfies the specified conditions is found, a new record is created. If multiple records that satisfy the specified conditions are found, Acumatica ERP returns an error.

In the `$filter` parameter, you will specify a field of the `MainContact` child entity. Thus, you will specify the `MainContact` entity in the `$expand` parameter.

Updating a Customer Record

To update a customer record by using the contract-based REST API, do the following:

1. In the Postman collection, add a request with the following settings:
    - HTTP method: `PUT`
    - URL: *https://localhost/MyStoreInstance/entity/Default/18.200.001/Customer*
    - Parameters of the request:

Parameter	Value
`$filter`	`MainContact/Email eq 'info@jevy-comp.con'`
`$expand`	`MainContact`
`$select`	`CustomerID,CustomerClass`

    - Headers:

Key	Value
`Accept`	`application/json`
`Content-Type`	`application/json`

    - Body of the request:

```
{
 "CustomerClass":{"value":"INTL"},
 "BillingContactSameAsMain":{"value":false},
 "BillingContact":{
  "Email":{"value":"green@jevy-comp.con"},
  "Attention":{"value":"Mr. Jack Green"},
  "JobTitle":{"value":""}
 }
}
```

2. Send the request. If the request is successful, the response contains the `200 OK` status code and includes the list of the requested fields of the customer record in JSON format. The following code example shows a fragment of the response.

```
{
    "id": "12f61a4c-2776-43a8-aca6-a676fe475572",
    "rowNumber": 1,
    "note": "",
    "BillingContact": {
        ...
    },
    "BillingContactSameAsMain": {
        "value": false
    },
    "CustomerClass": {
        "value": "INTL"
    },
    "CustomerID": {
        "value": "C000000003"
    },
    "custom": {},
    "files": []
}
```

3. Save the request.

Related Links

Update of a Record

Example 2.1.2: Using Put() and StringSearch (SOAP)

This example shows how you can update a customer record by using the contract-based SOAP API.

You will use the `Put()` method of the `DefaultSoapClient` object to update the record. You will pass to the method the `Customer` object that contains the email address specified with the `StringSearch` object.

When the Acumatica ERP contract-based web services receive a `Put` request that contains at least one `Search` object, Acumatica ERP tries to search for a record by using the specified search value or values. If a record that satisfies the specified conditions is found, Acumatica ERP updates the fields of the record that are specified by using the `Value` and `Search` objects. If no record that satisfies the specified conditions is found, a new record is created. If multiple records that satisfy the specified conditions are found, Acumatica ERP returns an error.

For details on how the `Put()` method works, see *Put() Method* in the documentation.

Updating a Customer Record

To update a customer record by using the contract-based SOAP API, do the following:

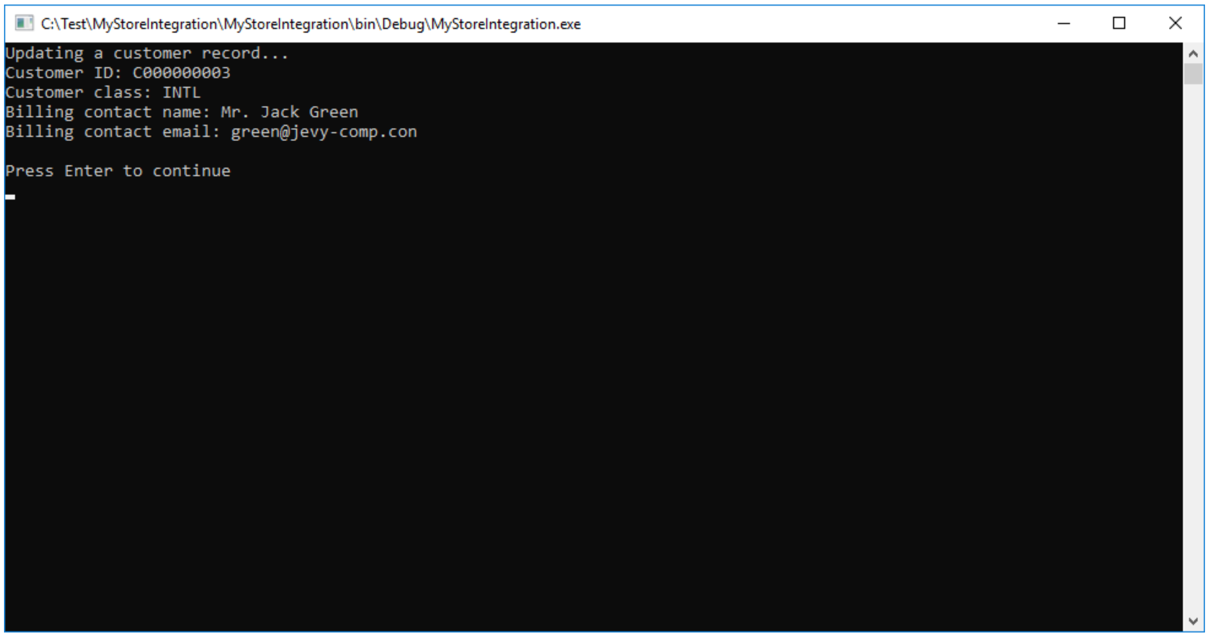
1. In the `Integration` folder of the *MyStoreIntegration* project, create the `UpdateOfRecords` class and add the `MyStoreIntegration.Default` `using` directive in the created file.
2. In the `UpdateOfRecords` class, define the `UpdateCustomer()` method as shown in the following code.

```
//Updating a customer record
public static void UpdateCustomer(DefaultSoapClient soapClient)
{
    Console.WriteLine("Updating a customer record...");

    //Customer data
    //Specify the email address of a customer that exists in the system
    string customerMainContactEmail = "info@jevy-comp.con";
    //Specify one of the customer classes that are configured in the system
    string customerClass = "INTL";
    string contactTitle = "Mr.";
    string contactFirstName = "Jack";
    string contactLastName = "Green";
    string contactEmail = "green@jevy-comp.con";

    //Select the needed customer record and
    //specify the values that should be updated
    Customer customerToBeUpdated = new Customer
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        CustomerID = new StringReturn(),
        MainContact = new Contact
        {
            //Search for the customer record by email address
            Email = new StringSearch { Value = customerMainContactEmail },
        },
        CustomerClass = new StringValue { Value = customerClass },
        //Specify the values of the customer billing contact
        BillingContactSameAsMain = new BooleanValue { Value = false },
        BillingContact = new Contact
        {
            Email = new StringValue { Value = contactEmail },
            Attention = new StringValue { Value = contactTitle + " " +
              contactFirstName + " " + contactLastName }
        }
    };

    //Update the customer record with the specified values
    Customer updCustomer = (Customer)soapClient.Put(customerToBeUpdated);
    //Display the ID and customer class of the updated record
    Console.WriteLine("Customer ID: " + updCustomer.CustomerID.Value);
    Console.WriteLine("Customer class: " + updCustomer.CustomerClass.Value);
    Console.WriteLine("Billing contact name: " +
      updCustomer.BillingContact.Attention.Value);
    Console.WriteLine("Billing contact email: " +
      updCustomer.BillingContact.Email.Value);
    Console.WriteLine();
    Console.WriteLine("Press Enter to continue");
    Console.ReadLine();
}
```

3. In the `try` block of the `Main()` method of the `Program` class, call the `UpdateCustomer()` method of the `UpdateOfRecords` class, as the following code shows.

```
//Update a customer record
UpdateOfRecords.UpdateCustomer(soapClient);
```

4. Rebuild the project, and run the application. The updated customer class is displayed in the console application window, as shown in the following screenshot.


C:\Test\MyStoreIntegration\MyStoreIntegration\bin\Debug\MyStoreIntegration.exe

```
Updating a customer record...
Customer ID: C000000003
Customer class: INTL
Billing contact name: Mr. Jack Green
Billing contact email: green@jevy-comp.con

Press Enter to continue
```

Figure: Console application window

Related Links

Put() Method

Additional Information: Update of a Customer Record with the Screen-Based SOAP API

This scenario is outside of the scope of this course but may be useful to some readers.

To update the customer class of a customer record identified in the system by its email address, in an array of commands, you use the `FilterEmail` service command to search for the needed record. You also use the `DialogAnswer` service command to answer the question in the dialog box that appears when you update a customer class.

For details about the `FilterEmail` service command, see *Commands for Record Searching: Filter Service Command*. For details about the `DialogAnswer` service command, see *Commands for Pop-Up Dialog Boxes and Pop-Up Forms*.

Additional Information: Removal of a Record

This scenario is outside of the scope of this course but may be useful to some readers.

By using all types of the web services APIs, you can remove records from Acumatica ERP. For details about how to remove a record, see the following topics in the documentation:

- For the contract-based REST API, *Removal of a Record*
- For the contract-based SOAP API, *Delete() Method*
- For the screen-based SOAP API, *Commands for Clicking Buttons on a Form*

Lesson Summary

In this lesson, you have learned how to update an existing customer record and find the record for update by using a non-key field. You have used one request for the update.

You have also briefly reviewed how the update of a record can be implemented with the screen-based SOAP API and how to remove a record with different types of APIs.

Lesson 2.2: Updating the Detail Lines of a Sales Order

The MyStore online store assigns a customer number to each sales order. This number (which is different than the sales order number assigned by Acumatica ERP) is stored in Acumatica ERP in the **Customer Order** box of the *Sales Orders*(SO301000) form. Before submitting a sales order for processing, a customer of the online store can select the needed order by using this customer number and then edit and submit the order.

This lesson shows how you can implement this integration scenario by using the contract-based APIs. In this lesson, you will select a sales order in Acumatica ERP by using the value of the customer number of the order and then update the detail lines of the selected sales order.

You will use the `SalesOrder` entity of the *Default/18.200.001* endpoint. This entity is mapped to the Sales Orders form. You will retrieve the needed sales order, update the detail lines of the sales order locally, and submit the updated sales order to Acumatica ERP.

You will identify the detail lines to be updated by using the entity IDs of the detail lines.

The entity ID is a GUID that is assigned to each entity you work with during an Acumatica ERP session. You can obtain the value of the entity ID from the `ID` property of an entity returned from Acumatica ERP.

The records of top-level entities that you retrieve through the contract-based API have persistent IDs, which are the values in the `NoteID` column of the corresponding database tables. That is, you can use the value from the `ID` property of a top-level entity returned from Acumatica ERP throughout different sessions with Acumatica ERP. However, if a record does not have a note ID (which could be the case for detail entities, entities that correspond to generic inquiries, or custom entities), this record is assigned the entity ID that is new for each new session. That is, after a new login to Acumatica ERP, you cannot use the entity ID that you received in the previous session to work with the entity.

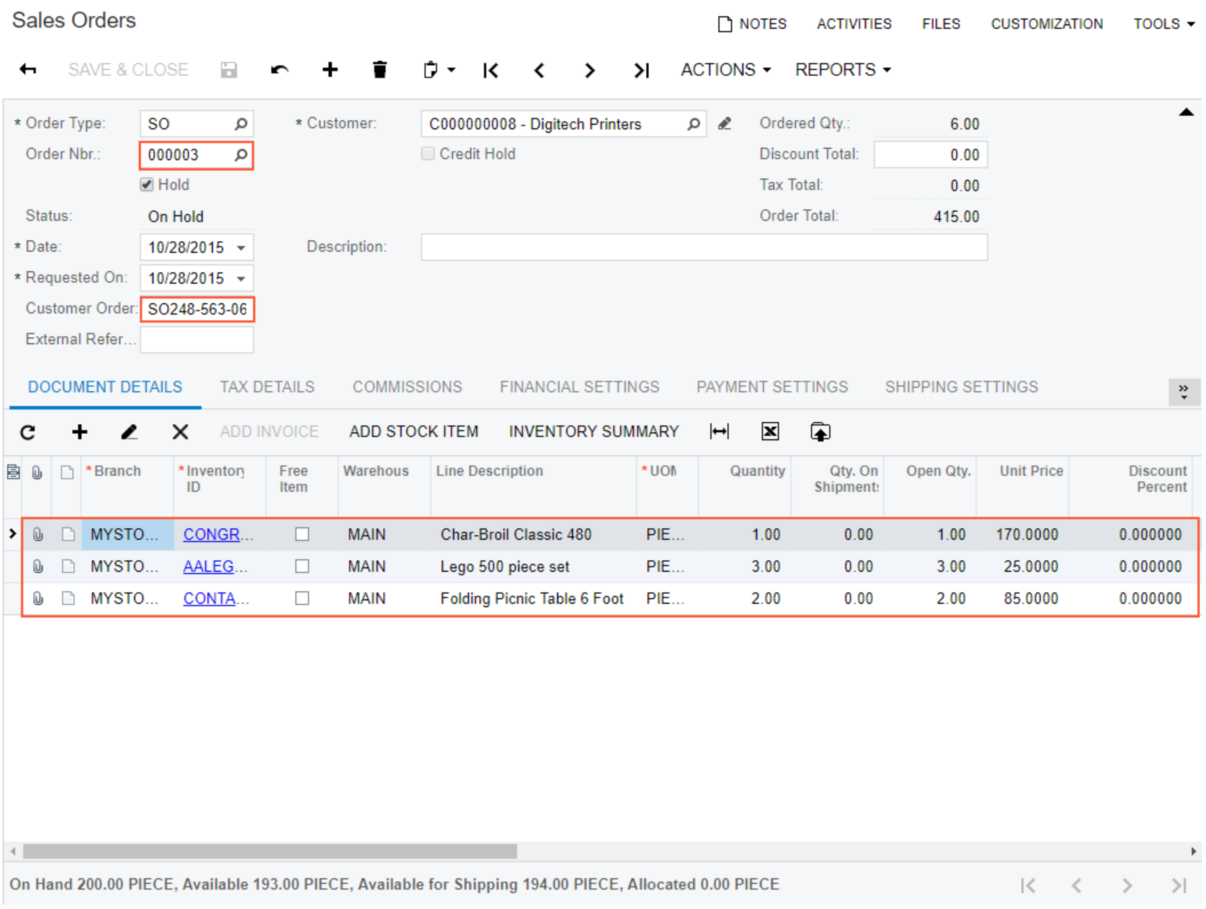
You will delete a detail line by setting the `delete` system filed of the detail entity to `true`.

Lesson Objective

In this lesson, you will learn how to update the detail lines of a document by using the contract-based APIs.

Prerequisites

In the examples of this lesson, you will update the sales order with customer order number *SO248-563-06*; this sales order has been preconfigured. Before you complete the examples of this lesson, on the *Sales Orders* (SO301000) form, make sure that the sales order with this customer order number exists. (In the system, this sales order has the order number *000003*.) Notice that this sales order currently has three detail lines, as shown in the following screenshot.



* Order Type:	SO	* Customer:	C000000008 - Digitech Printers
Order Nbr.:	000003	Credit Hold	
Hold	✓		
Status:	On Hold		
* Date:	10/28/2015	Description:	
* Requested On:	10/28/2015		
Customer Order:	SO248-563-06		
External Refer...			

Ordered Qty.:	6.00
Discount Total:	0.00
Tax Total:	0.00
Order Total:	415.00

*Branch	*Inventory ID	Free Item	Warehouse	Line Description	*UOM	Quantity	Qty. On Shipments	Open Qty.	Unit Price	Discount Percent
MYSTO...	CONGR...	☐	MAIN	Char-Broil Classic 480	PIE...	1.00	0.00	1.00	170.0000	0.000000
MYSTO...	AALEG...	☐	MAIN	Lego 500 piece set	PIE...	3.00	0.00	3.00	25.0000	0.000000
MYSTO...	CONTA...	☐	MAIN	Folding Picnic Table 6 Foot	PIE...	2.00	0.00	2.00	85.0000	0.000000

On Hand 200.00 PIECE, Available 193.00 PIECE, Available for Shipping 194.00 PIECE, Allocated 0.00 PIECE

Figure: Sales order to be updated

If you want to perform an example of this lesson multiple times for the same sales order or complete both the REST example and the SOAP example of this lesson, you should make sure that the sales order that you are going to update contains the lines that are updated in the example. You can restore the sales order to the state that is described in this topic and shown above (that is, add the deleted lines and update the quantity of the items in the order) or modify the values that are used for the update of detail lines in the code examples.

When you complete the examples in this lesson, you can also use the data of any other sales order with the *On Hold* or *Open* status. If you do this, you need to update the values that are used in the examples accordingly.

Example 2.2.1: Using GET with $filter and PUT (REST)

In this example, you will update the detail lines of a sales order record by using the contract-based REST API.

You will retrieve the sales order by using the values of the customer order number (*SO248-563-06*) and the order type (*SO*). Because you do not use the full set of key fields to find the needed record, you will use the `$filter` parameter of the `GET` request to specify the values that are used for the search (instead of specifying the values of key fields in the URL). You will also use the `$expand` and `$select` parameters to request only the key fields and the fields to be updated.

To update the detail lines, you will use the `PUT` request and identify the detail entities by the session IDs of the details returned in the `GET` response. You will delete the detail line for the *CONGRILL* item by setting the `delete` system field of the detail entity to `true`. You will also update the quantity of the *AALEGO500* item.

Updating the Detail Lines of the Sales Order

To update the detail lines of a sales order by using the contract-based REST API, do the following:

1. Retrieve the needed sales order by using the value of the customer order number as follows:
    a. In the Postman collection, add a new request with the following settings:
        - HTTP method: `GET`
        - URL: *https://localhost/MyStoreInstance/entity/Default/18.200.001/SalesOrder*
        - Parameters of the request:

Parameter	Value
`$expand`	`Details`
`$select`	`OrderNbr,OrderType,Details/InventoryID,Details/WarehouseID`
`$filter`	`OrderType eq 'SO' and CustomerOrder eq 'SO248-563-06'`

        - Headers:

Key	Value
`Accept`	`application/json`
`Content-Type`	`application/json`

    b. Send the request. If the request is successful, the response contains the `200 OK` status code and includes the list of requested fields of the sales order record in JSON format. The following code shows an example of the response.

```
[
    {
        "id": "c52bd7ac-c715-4ce3-8565-50463570b7d9",
        "rowNumber": 1,
        "note": "",
        "CustomerOrder": {
            "value": "SO248-563-06"
        },
        "Details": [
            {
                "id": "988988a5-3bc0-4645-a884-8a9ba6a400b4",
                "rowNumber": 1,
                "note": "",
                "InventoryID": {
                    "value": "AALEGO500"
                },
                "WarehouseID": {
                    "value": "MAIN"
                },
                "custom": {},
                "files": []
            },
            {
                "id": "983f9831-b139-489c-8ad0-86d50f6e535d",
                "rowNumber": 2,
                "note": "",
                "InventoryID": {
                    "value": "CONTABLE1"
                },
                "WarehouseID": {
                    "value": "MAIN"
                },
                "custom": {},
                "files": []
            },
            {
                "id": "19193380-63b2-445c-a50b-fd6d57f176a0",
                "rowNumber": 3,
                "note": "",
                "InventoryID": {
                    "value": "CONGRILL"
                },
                "WarehouseID": {
                    "value": "MAIN"
                },
                "custom": {},
                "files": []
            }
        ],
        "OrderNbr": {
            "value": "000003"
        },
        "OrderType": {
            "value": "SO"
        },
        "custom": {},
        "files": []
    }
]
```

The sales order record in this code example contains three detail lines. You will delete the line for the *CONGRILL* item (which, in the session of this request, has the *19193380-63b2-445c-a50b-fd6d57f176a0* identifier) and update the quantity of the *AALEGO500* item (which has the *988988a5-3bc0-4645-a884-8a9ba6a400b4* session identifier).

2. Update the sales order record as follows:

a. In the Postman collection, add a new request with the following settings:

- HTTP method: `PUT`
- URL: *https://localhost/MyStoreInstance/entity/Default/18.200.001/SalesOrder*
- Parameters of the request:

Parameter	**Value**
`$expand`	`Details`

Parameter	Value
$select	OrderType,OrderNbr,Details/OrderQty,Details/InventoryID,OrderedQty,OrderTotal

- Body of the request:

```
{
 "OrderType":{"value":"SO"},
 "OrderNbr":{"value":"000003"},
 "Hold":{"value":false},
 "Details":[
  {
   "id":"19193380-63b2-445c-a50b-fd6d57f176a0",
   "delete":true
  },
  {
   "id":"988988a5-3bc0-4645-a884-8a9ba6a400b4",
   "OrderQty":{"value":5.0}
  }
 ]
}
```

  In the body, you specify the IDs of the items from the previous response: the ID of the *CONGRILL* item to delete the item, and the ID of the *AALEGO500* item to update the order quantity.

- Headers:

Key	Value
Accept	application/json
Content-Type	application/json

b. Send the request. If the request is successful, the response contains the `200 OK` status code and includes the list of requested fields of the updated sales order record in JSON format. The following code shows an example of the response.

```
{
    "id": "28fd1be7-7b78-45d1-a7ba-7ae3ea6b8a0a",
    "rowNumber": 1,
    "note": "",
    "Details": [
        {
            "id": "23c24e55-e92e-47e4-aec2-9213da02f823",
            "rowNumber": 1,
            "note": null,
            "InventoryID": {
                "value": "AALEGO500"
            },
            "OrderQty": {
                "value": 5
            },
            "custom": {},
            "files": []
        },
        {
            "id": "f86410ce-9807-4fce-90a8-7af8872572fc",
            "rowNumber": 2,
            "note": "",
            "InventoryID": {
                "value": "CONTABLE1"
            },
            "OrderQty": {
                "value": 2
            },
            "custom": {},
            "files": []
        }
    ],
    "Hold": {
        "value": false
    },
    "OrderedQty": {
        "value": 7
    },
    "OrderNbr": {
        "value": "000003"
    },
    "OrderTotal": {
        "value": 295
    },
    "OrderType": {
        "value": "SO"
    },
    "custom": {},
    "files": []
}
```

c. Save the request.

Related Links

Update of a Record

Example 2.2.2: Using Get() with StringSeach and Put() (SOAP)

In this lesson, you will add to the MyStoreIntegration application the method that updates the detail lines of an existing sales order through the contract-based SOAP API.

In this method, you will retrieve the sales order by using the values of the customer order number (*SO248-563-06*) and the order type (*SO*). To retrieve the sales order, you will use the `Get()` method of the `DefaultSoapClient` object. You will use the `StringSearch` objects to specify the values that the system should use to find the sales order to be updated. You will use `ReturnBehavior.All` to obtain the values of all detail fields of the sales order.

You will use the `Single()` method to find the needed detail lines of the sales order locally. You will delete the detail line for the *CONTABLE1* item by setting the `Delete` property of the `SalesOrderDetail` object to `true`. You will also update the quantity of the *AALEGO500* item. You will send to the service the updated record by using the `Put()` method.

Updating the Detail Lines of the Sales Order

To update the detail lines of the existing sales order by using the contract-based SOAP API, do the following:

1. Add to the `UpdateOfRecords` class the `UpdateSO()` method by using the following code.

```
//Updating the detail lines of a sales order
public static void UpdateSO(DefaultSoapClient soapClient)
{
    Console.WriteLine("Updating a sales order...");

    //Sales order data
    string orderType = "SO";
    //The unique value that identifies the sales order
    string customerOrder = "SO248-563-06";

    string firstItemInventoryID = "CONTABLE1";
    string firstItemWarehouse = "MAIN";
    string secondItemInventoryID = "AALEGO500";
    string secondItemWarehouse = "MAIN";
    decimal secondItemQuantity = 4;

    //Find the sales order to be updated
    SalesOrder soToBeFound = new SalesOrder
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        OrderType = new StringSearch { Value = orderType },
        CustomerOrder = new StringSearch { Value = customerOrder },
        OrderNbr = new StringReturn(),
        Details = new SalesOrderDetail[]
        {
            new SalesOrderDetail
            {
                ReturnBehavior = ReturnBehavior.All
            }
        }
    };
    SalesOrder order = (SalesOrder)soapClient.Get(soToBeFound);

    //Find the line to be deleted and mark it for deletion.
    //The Single method makes the program find
    //the only SalesOrderDetail of order.Details
    //that has the specified InventoryID and WarehouseID.
    SalesOrderDetail orderLine = order.Details.Single(
        orderLineToBeDeleted =>
            orderLineToBeDeleted.InventoryID.Value == firstItemInventoryID &&
            orderLineToBeDeleted.WarehouseID.Value == firstItemWarehouse);
    orderLine.Delete = true;

    //Find the line to be updated and update the quantity in it
    orderLine = order.Details.Single(
    orderLineToBeUpdated =>
            orderLineToBeUpdated.InventoryID.Value == secondItemInventoryID &&
            orderLineToBeUpdated.WarehouseID.Value == secondItemWarehouse);
    orderLine.OrderQty = new DecimalValue { Value = secondItemQuantity };

    //Clear the Hold check box
    order.Hold = new BooleanValue { Value = false };

    //Specify the additional values to be returned
    order.OrderedQty = new DecimalReturn();
    order.OrderTotal = new DecimalReturn();

    //Update the sales order
    order = (SalesOrder)soapClient.Put(order);

    //Display the summary of the updated record
    Console.WriteLine("Order type: " + order.OrderType.Value);
    Console.WriteLine("Order number: " + order.OrderNbr.Value);
    Console.WriteLine("Ordered quantity: " + order.OrderedQty.Value);
    Console.WriteLine("Order total: " + order.OrderTotal.Value);
    Console.WriteLine();
    Console.WriteLine("Press Enter to continue");
    Console.ReadLine();
}
```

2. In the `try` block of the `Main()` method of the `Program` class, call the `UpdateSO()` method of the `UpdateOfRecords` class, as the following code shows.

```
//Update detail lines of a sales order
UpdateOfRecords.UpdateSO(soapClient);
```

3. Rebuild the project, and run the application. The summary information on the updated sales order (which has the *000003* order number) is displayed in the console application window.

   On the *Sales Orders* (SO301000) form, you can see that the updated sales order now does not have a detail line for the *CONTABLE1* item and has a quantity of 4 for the *AALEGO500* item.

Related Links

Put() Method

Additional Information: Update of the Detail Lines of a Sales Order with the Screen-Based SOAP API

This scenario is outside of the scope of this course but may be useful to some readers.

To select the needed detail lines of a sales order, you use the `Key` command. For details about the command, see *Commands for Record Searching: Key Command*.

To delete a detail line, you use the `DeleteRow` service command. To add a detail line, you use the `NewRow` service command, described in *Commands for Adding Detail Lines*.

Lesson Summary

In this lesson, you have learned how to delete and update the detail lines of a sales order document through the contract-based APIs. You have retrieved the sales order in one request, updated the detail lines of the sales order locally, and submitted the order to Acumatica ERP in another request. You have identified the detail lines for the update by the entity IDs of the lines. You have used the `delete` system field of the detail entity to remove the detail line.

You have also reviewed how the update of detail lines of a record can be implemented with the screen-based SOAP API.

Part 3: Execution of Actions

In this part of the course, you will perform actions on records in Acumatica ERP through the web services APIs. You will update an invoice and invoke the release of the invoice by using one request. Because the release of an invoice is a long-running operation, you will monitor the status of the release operation and retrieve the released invoice once the operation is completed.

As a result of completing the lesson of this part, you will know how to execute long-running operations through the contract-based APIs.

Lesson 3.1: Releasing an Invoice

The MyStore online store needs to provide an invoice for each shipped order to the customer so that the customer can review it and confirm the invoice. To make it possible for this invoice to be created and confirmed, the online store passes the data of the customer and the shipped orders to Acumatica ERP and releases the invoice in Acumatica ERP. In Acumatica ERP, the information about whether the invoice was confirmed is reflected by the **Status** setting of the invoice on the *Invoices* (SO303000) form. As a result of the method being called, the invoice has the *Open* status.

In this lesson, you will add to the MyStoreIntegration REST or SOAP application a request that, in one call to the contract-based API, changes the status of an invoice from *On Hold* to *Balanced* and then invokes the release of the invoice. You do not need to perform a separate request to update the data of an entity if you then perform an action on this entity. You can combine these two requests into one, which improves the performance of the application.

To release an invoice, you will use the `ReleaseSalesInvoice` action of the `SalesInvoice` entity of the *Default/18.200.001* endpoint. Because the release of an invoice is a long-running operation, you will monitor the status of this operation in the method and get the result of processing only after the operation is completed.

The implementation of this scenario with the screen-based SOAP API is outside of the scope of this course. For a summary of this implementation, see *Additional Information: Release of an Invoice with the Screen-Based SOAP API*.

In this lesson, you will release a sales order invoice. The processing of other types of documents (such as direct sales invoices and pro forma invoices) is ouside of the scope of this course. For basic information about processing of other types of documents, see *Additional Information: Processing of Other Types of Invoices*.

Lesson Objective

In this lesson, you will learn how to handle long-running operations, such as the releasing of the invoice, by using the contract-based APIs.

Prerequisites

Before you proceed with any examples in this lesson, clear the **Validate Document Totals on Entry** check box on the *Accounts Receivable Preferences* (AR101000) form. With this check box cleared, the invoice amount does not have to be entered in the **Amount** box on the *Invoices* (SO303000) form, which can be used to validate data during manual entry.

On the Invoices form, make sure that the invoices with the *INV000046* and *INV000047* reference numbers exist and that these invoices have the *On Hold* status.

If you want to perform an example of this lesson multiple times, after you complete the example, on the Invoices form, you need to create an invoice that has the *On Hold* status. You also need to update the reference number of the invoice in the code example to that of the invoice you created. To create an invoice, you can use the procedure described in *To Prepare an Invoice for a Sales Order* in the documentation or perform a web services API call.

Example 3.1.1: Using POST to Release an Invoice (REST)

In this example, by using one `POST` request in the contract-based REST API, you will invoke the release of the *INV000046* invoice, which has the *On Hold* status in the system.

If the request returns the `400 Bad Request`, `401 Unauthorized`, or `500 Internal Server Error` response, the operation has failed.

A response to the `POST` request with the `202 Accepted` status has the `Location` header, which contains a URL that you will use to check the status of the operation by using the `GET` HTTP method. While the status returned by the request is `202 Accepted`, the operation is in progress. You should have a delay between the checks of the status of the operation so that the performance of the application is not impaired.

When the `GET` HTTP method with this URL returns `204 No Content`, the operation is completed. You will then use the `GET` request to retrieve the data of the released invoice from Acumatica ERP.

Releasing an Invoice

To release an invoice by using the contract-based REST API, do the following:

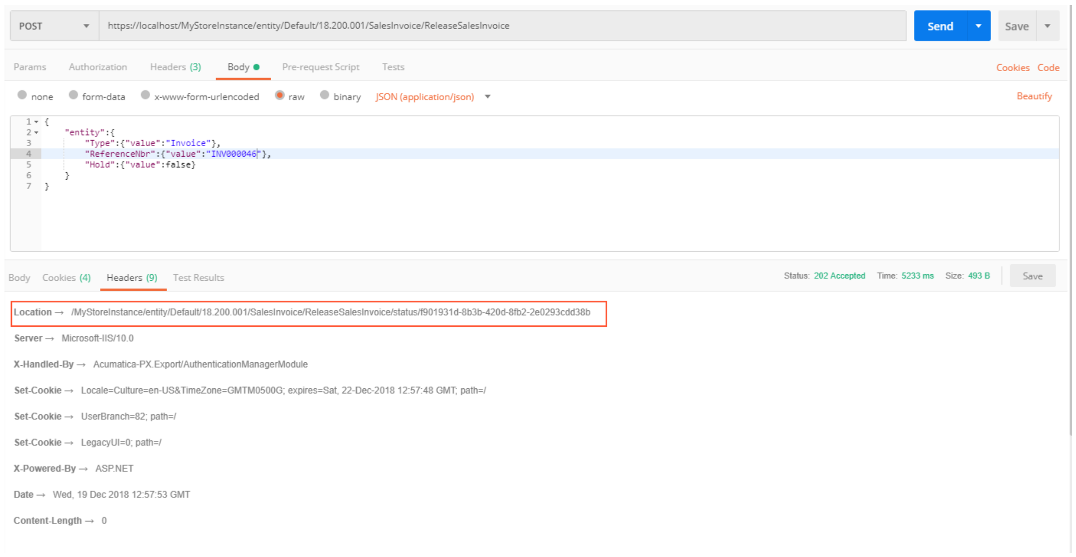
1. Invoke the release of the invoice as follows:
    a. In the Postman collection, add a new request with the following settings:
        - HTTP method: `POST`
        - URL: *https://localhost/MyStoreInstance/entity/Default/18.200.001/SalesInvoice/ReleaseSalesInvoice*
        - Body of the request:

```
{
 "entity":{
  "Type":{"value":"Invoice"},
  "ReferenceNbr":{"value":"INV000046"},
  "Hold":{"value":false}
    }
}
```

        - Headers:

Key	**Value**
Accept	application/json
Content-Type	application/json

    b. Send the request, and make sure the response contains the `202 Accepted` status, as shown in the following screenshot. Find the URL in the `Location` header.



https://localhost/MyStoreInstance/entity/Default/18.200.001/SalesInvoice/ReleaseSalesInvoice

```
{
    "entity":{
        "Type":{"value":"Invoice"},
        "ReferenceNbr":{"value":"INV000046"},
        "Hold":{"value":false}
    }
}
```

Status: 202 Accepted   Time: 5233 ms   Size: 493 B

```
Location → /MyStoreInstance/entity/Default/18.200.001/SalesInvoice/ReleaseSalesInvoice/status/f901931d-8b3b-420d-8fb2-2e0293cdd38b
Server → Microsoft-IIS/10.0
X-Handled-By → Acumatica-PX.Export/AuthenticationManagerModule
Set-Cookie → Locale=Culture=en-US&TimeZone=GMTM0500G; expires=Sat, 22-Dec-2018 12:57:48 GMT; path=/
Set-Cookie → UserBranch=82; path=/
Set-Cookie → LegacyUI=0; path=/
X-Powered-By → ASP.NET
Date → Wed, 19 Dec 2018 12:57:53 GMT
Content-Length → 0
```

Figure: The status and the Location header

c. Save the request.

2. Monitor the status of the release operation by using a request with the following settings:

- HTTP method: `GET`
- URL: *https://localhost/MyStoreInstance/entity/Default/18.200.001/SalesInvoice/ReleaseSalesInvoice/status/f901931d-8b3b-420d-8fb2-2e0293cdd38b* (You copy this URL from the `Location` header of the previous `POST` request.)
- Headers:

Key	**Value**
`Accept`	`application/json`
`Content-Type`	`application/json`

Once the `GET` HTTP method returns `204 No Content`, the operation is completed.

3. Retrieve the released invoice by using a request with the following settings:

- HTTP method: `GET`
- URL: *https://localhost/MyStoreInstance/entity/Default/18.200.001/SalesInvoice/Invoice/INV000046*
- Parameters of the request:

Parameter	**Value**
`$select`	`ReferenceNbr,Type,Status`

- Headers:

Key	**Value**
`Accept`	`application/json`
`Content-Type`	`application/json`

A successful response contains the `200 OK` status. The status of the invoice is *Open*, as the following code example shows.

```
{
    "id": "e22bc7f2-16a6-4025-98c3-d8cb2eaac18a",
    "rowNumber": 1,
    "note": "",
    "ReferenceNbr": {
        "value": "INV000046"
    },
    "Status": {
        "value": "Open"
    },
    "Type": {
        "value": "Invoice"
    },
    "custom": {},
    "files": []
}
```

Related Links

Execution of an Action

Example 3.1.2: Using Invoke() to Release an Invoice (SOAP)

In this example, by using the `Invoke()` method of the `DefaultSoapClient` object through the contract-based SOAP API, you will release the *INV000047* invoice, which has the *On Hold* status in the system.

You will use an `Invoke()` call to invoke the release of the invoice. You will use the `GetProcessResult` method of the `LongRunProcessor` object to monitor the status of a long-running processing operation. While the status returned by the method is *InProcess*, the operation is in progress. You should have a delay between the checks of the status of the operation so that the performance of the application is not impaired.

When the status of the process is *Completed*, the method returns the `ProcessResult` object. You will then pass the ID of the released invoice to the `Get()` method to obtain the invoice from Acumatica ERP.

If a check for the status of the operation returns the *Aborted* status, the operation has failed.

Releasing an Invoice

To release the invoice by using the contract-based SOAP API, do the following:

1. In the Visual Studio project, add the `Actions` class and the `MyStoreIntegration.Default using` directive.
2. In the project, add the `Helper` folder, and add the `LongRunProcessor` class to the folder. In the class, add the following code.

```
using System;
using MyStoreIntegration.Default;
using System.Threading;

namespace MyStoreIntegration.Helper
{
    class LongRunProcessor
    {
        //A supplementary method for monitoring of the status of
        //long-running operations
        public static ProcessResult GetProcessResult(
          DefaultSoapClient soapClient, InvokeResult invokeResult)
        {
            while (true)
            {
                var processResult = soapClient.GetProcessStatus(invokeResult);
                switch (processResult.Status)
                {
                    case ProcessStatus.Aborted:
                        throw new SystemException("Process status: " +
                        processResult.Status + "; Error: " +
 processResult.Message);
                    case ProcessStatus.NotExists:
                    case ProcessStatus.Completed:
                        //Go to normal processing.
                        return processResult;
                    case ProcessStatus.InProcess:
                        //Pause for 1 second.
                        Thread.Sleep(1000);
                        if (processResult.Seconds > 30)
                            throw new TimeoutException();
                        continue;
                    default:
                        throw new InvalidOperationException();
                }
            }
        }
    }
}
```

3. In the `Actions` class, add the `using MyStoreIntegration.Helper;` directive and the `ReleaseSOInvoice()` method, as the following code shows.

```
//Releasing an invoice on the Invoices form (SO303000)
public static void ReleaseSOInvoice(DefaultSoapClient soapClient)
{
    Console.WriteLine("Releasing an invoice...");

    //Invoice data
    string invoiceType = "Invoice";
    string referenceNbr = "INV000047";

    //Specify the values of a new invoice.
    SalesInvoice soInvoice = new SalesInvoice
    {
        Type = new StringValue { Value = invoiceType },
        ReferenceNbr = new StringValue { Value = referenceNbr },
        Hold = new BooleanValue { Value = false }
    };

    //Release the invoice.
    InvokeResult invokeResult =
      soapClient.Invoke(soInvoice, new ReleaseSalesInvoice());

    //Monitor the status of the process.
    ProcessResult processResult =
      LongRunProcessor.GetProcessResult(soapClient, invokeResult);

    //Get the confirmed shipment.
    soInvoice = (SalesInvoice)soapClient.Get(
      new SalesInvoice { ID = processResult.EntityId });
    //Display the summary of the invoice
    Console.WriteLine("Invoice type: " + soInvoice.Type.Value);
    Console.WriteLine("Invoice number: " + soInvoice.ReferenceNbr.Value);
    Console.WriteLine("Invoice status: " + soInvoice.Status.Value);
    Console.WriteLine();
    Console.WriteLine("Press Enter to continue");
    Console.ReadLine();
}
```

4. In the `try` block of the `Main()` method of the `Program` class, call the `ReleaseSOInvoice()` method of the `Actions` class, as the following code shows.

```
//Release an SO invoice
Actions.ReleaseSOInvoice(soapClient);
```

5. Rebuild the project, and run the application. The type, number, and status of the invoice are displayed in the console application window.

Related Links

Invoke() Method

GetProcessStatus() Method

Additional Information: Processing of Other Types of Invoices

The following scenarios, which describe the processing of types of invoices other than sales orders invoices, are outside of the scope of this course but may be useful to some readers.

Processing of Direct Sales Invoices

Direct sales invoices are invoices for which neither a sales order nor a shipment has been created. The creation of a direct sales invoice is a typical scenario of integration of Acumatica ERP with point-of-sale (POS) systems. The POS system creates a direct sales invoice and a payment, applies this payment to the invoice, and releases the invoice. You can find examples of integration of Acumatica ERP with POS systems in the *Contract-Based API Examples* in the documentation.

Processing of Pro Forma Invoices

A pro forma invoice is a draft invoice for project billing. You may need to create a pro forma invoice through the API if you implement integration of Acumatica ERP projects with external systems.
For an example of the creation of a pro forma invoice, see the *Contract-Based API Examples* in the documentation.

Additional Information: Release of an Invoice with the Screen-Based SOAP API

This scenario is outside of the scope of this course but may be useful to some readers.

To release the invoice, you use the `Release` action of the `Content` object that corresponds to the *Invoices* (SO303000) form. For details about the use of actions, see *Commands for Clicking Buttons on a Form*.

To monitor the status of the long-running release operation, you use the `GetProcessStatus()` method. For more information about the method, see *GetProcessStatus() Method*.

Lesson Summary

In this lesson, you have learned how to handle long-running operations, such as the releasing of the invoice, by using the contract-based APIs. You have updated and released an invoice in one call. You have monitored the status of the long-running operation, and when the operation completed, you have retrieved the released invoice from Acumatica ERP.

You have also reviewed how the release of an invoice can be implemented with the screen-based SOAP API, and how other types of invoices can be processed through the contract-based APIs.

Part 4: Processing of Payments by Credit Cards

Acumatica ERP supports the processing of credit card payments through the Authorize.Net payment gateway.

You can also use other payment gateways for the processing of credit card payments. You can implement a plug-in for the needed payment gateway by yourself or use an existing plug-in available in *https://www.acumatica.com/extensions/*. For details about the creation of custom plug-ins, see *Implementing Plug-Ins for Processing Credit Card Payments* in the documentation.

In this part of the course, you will create a credit card payment method for a customer credit card by using the customer profile ID and the payment profile ID of the customer in Authorize.Net. You will not pass any credit card-specific information to Acumatica ERP.

As a result of completing the lesson of this part, you will learn how to register a customer credit card in Acumatica ERP through the contract-based APIs.

Lesson 4.1: Registering a Customer Credit Card

The MyStore online store needs to process customer payments that are made by using a credit card. To process these payments, the online store uses Acumatica ERP, which supports the processing of credit card payments through the Authorize.Net payment gateway. Acumatica ERP interacts with the payment gateway in PCI DSS–compliant mode.

In this lesson, you will add to the MyStoreIntegration REST or SOAP application a request that creates a customer payment method, which represents a particular credit card of a customer.

To create this customer payment method, you will submit the customer profile ID and the payment profile ID of the customer in Authorize.Net to the *Customer Payment Methods* (AR303010) form through the contract-based APIs. The customer profile ID and the payment profile ID are used to identify the customer and the customer's card, respectively, across the payment gateway and Acumatica ERP. The payment profile ID, instead of any information specific to the card, is saved to the customer payment method in Acumatica ERP.

You will use the `CustomerPaymentMethod` entity of the *Default/18.200.001* endpoint.

This scenario can also be implemented with the screen-based SOAP API, which is outside of the scope of this course. For a summary of this implementation, see *Additional Information: Registration of a Customer Credit Card with the Screen-Based SOAP API*.

Capturing of the credit card payments is outside of the scope of this course. For basic information about capturing of credit card payments, see *Additional Information: Processing of Credit Card Payments*.

Lesson Objective

In this lesson, you will learn how to create a customer payment method in Acumatica ERP through the contract-based APIs.

Prerequisites

Before you proceed with this lesson, make sure that you have configured the HTTPS connection, as described in *Configuring a Website for HTTPS*.

Additionally, to be able to process credit card payments in the system, you need to do the following before you complete the examples of the lesson:

1. To get a test account at the Authorize.Net payment gateway, create a sandbox account at *http://developer.authorize.net*. After you create an account, you will get the credentials to be used in payment processing (API login ID and transaction key).
2. Set up the connection with the payment gateway by using your credentials as follows:
   a. On the *Processing Centers* (CA205000) form, select the *AUTHNETUSD* processing center, which has been preconfigured in the system.
   b. To use the payment gateway in test mode (as opposed to live mode), on the **Settings** tab, specify the API login ID of your sandbox account as the value of the *MERCNAME* setting and the transaction key of your sandbox account as the value of the *TRANKEY* setting. Save your changes.

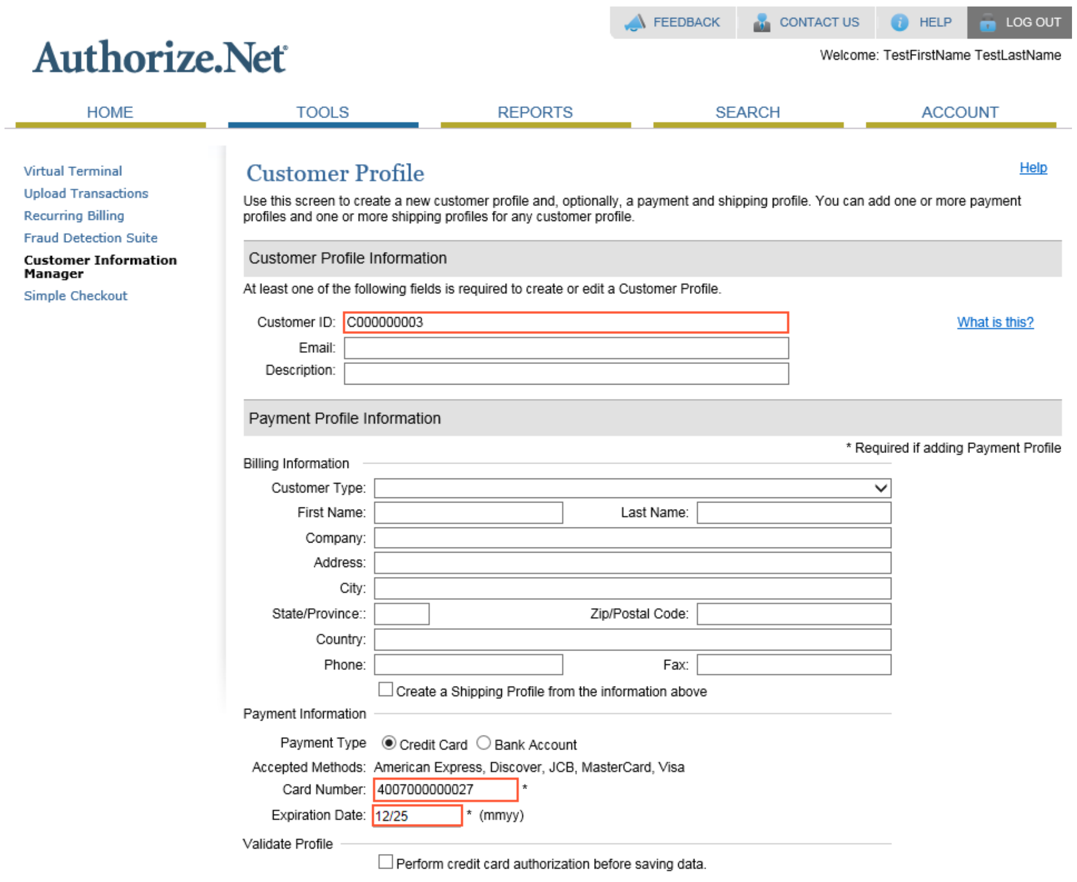
      The payment processing credentials API login ID and transaction key are not the login ID and password that you receive from Authorize.Net to access the Merchant Interface. By using the sandbox account's login ID and password, you can sign in to the sandbox Authorize.Net Merchant Interface (*https://sandbox.authorize.net/*), where you can review the transactions that have been processed at the gateway and manage the registered credit cards.

      For information on settings for live mode, see *Setup of Card Payment Processing* in the documentation.
   c. On the form toolbar, click **Test Credentials** to check the connection settings.

   If the test credentials are accepted by the processing center, the connection to the payment gateway is ready.
3. To process the credit card payments of the customer you will work with in the examples of this lesson by using Authorize.Net, create a customer profile for this customer in your Authorize.Net developer sandbox account as follows:
   a. In the Customer Information Manager tool of Authorize.Net on *https://sandbox.authorize.net/*, add a new customer profile. In the **Customer ID** box of the new customer profile, specify the ID of the customer in Acumatica ERP: *C000000003*, as shown in the following screenshot.

      In your client application, you can create a customer profile by using the API that is provided by Authorize.Net. The integration between your client application and Authorize.Net is outside of the scope of this guide.
   b. In the **Card Number** box, type `4007000000027` as the card number (which is the demo Visa card number you can use with Authorize.Net), and in the **Expiration Date** box, specify any date that is later than the current date.

      You can use other test credit card numbers, which you have received with your Authorize.Net sandbox account.



Welcome: TestFirstName TestLastName

Customer Profile

Use this screen to create a new customer profile and, optionally, a payment and shipping profile. You can add one or more payment profiles and one or more shipping profiles for any customer profile.

Customer Profile Information

At least one of the following fields is required to create or edit a Customer Profile.

Customer ID:	C000000003
Email:	
Description:	

Payment Profile Information

* Required if adding Payment Profile

Billing Information

Customer Type:			
First Name:		Last Name:	
Company:			
Address:			
City:			
State/Province::		Zip/Postal Code:	
Country:			
Phone:		Fax:	

☐ Create a Shipping Profile from the information above

Payment Information

Payment Type	[✓ Credit Card] Bank Account
Accepted Methods:	American Express, Discover, JCB, MasterCard, Visa
Card Number:	4007000000027 *
Expiration Date:	12/25 * (mmyy)

Validate Profile

☐ Perform credit card authorization before saving data.

Figure: Customer Information Manager

Do not use the data of a real credit card in this example! If you do, the money for the testing operations will be charged to this card.

You will use the payment profile ID that has been generated by Authorize.Net as the input value of the method that you create in this lesson.

If you plan to complete both the REST API example and the SOAP API example, you should register two credit cards in Authorize.Net.

Example 4.1.1: Using PUT and the CustomerPaymentMethod Entity (REST)

In this example, through the REST API, you will create a customer payment method that corresponds to a customer credit card. To create this customer payment method, you will use the `PUT` HTTP method.

Creating a Customer Payment Method for a Credit Card

To create a customer payment method for a credit card by using the contract-based REST API, do the following:

1. In the Postman collection, add a new request with the following settings:
    - HTTP method: `PUT`
    - URL: *https://localhost/MyStoreInstance/entity/Default/18.200.001/CustomerPaymentMethod*
    - Parameters of the request:`$select=CardAccountNbr`
    - Body of the request:

    ```
    {
     "PaymentMethod":{"value":"VISA"},
     "CustomerID":{"value":"C000000003"},
     "CustomerProfileID":{"value":"37100472"},
     "CashAccount":{"value":"102050MYST"},
     "Details":[
      {
       "Name":{"value":"Payment Profile ID"},
       "Value":{"value":"35596199"}
      }
     ]
    }
    ```

    In the body of the request, replace the values of the `CustomerProfileID` field and the `Payment Profile ID` detail with the values that you have received during the registration of the customer payment method in Authorize.Net.
    - Headers:

Key	Value
Accept	application/json
Content-Type	application/json

2. Send the request. If the request is successful, the response contains the `200 OK` status and includes the created customer payment method in JSON format. The following code shows an example of the response. The `CardAccountNbr` field contains the card identifier in the system, which consists of the payment method ID and the last four numbers of the credit card. This format complies with PCI DSS.

```
{
    "id": "8dfdc822-8323-4e3c-b349-30285a75b6ec",
    "rowNumber": 1,
    "note": null,
    "CardAccountNbr": {
        "value": "VISA:****-****-****-0027"
    },
    "CashAccount": {
        "value": "102050MYST"
    },
    "CustomerID": {
        "value": "C000000003"
```

```
    },
    "CustomerProfileID": {
        "value": "37100472"
    },
    "Details": [
        {
            "id": "b752b864-e923-4195-b3a7-e546e78ceca5",
            "rowNumber": 1,
            "note": null,
            "Name": {
                "value": "CCPID"
            },
            "Value": {
                "value": "35596199"
            },
            "custom": {},
            "files": []
        }
    ],
    "PaymentMethod": {
        "value": "VISA"
    },
    "custom": {},
    "files": []
}
```

3. Save the request.

Related Links

Creation of a Record

Setup of Card Payment Processing

Example 4.1.2: Using Put() and the CustomerPaymentMethod Entity (SOAP)

This example shows how you can implement the creation of a customer payment method by using the contract-based SOAP API.

To create a customer payment method for a credit card, you will use the `Put()` method with the `CustomerPaymentMethod` object as a parameter.

Creating a Customer Payment Method for a Credit Card

Create a customer payment method for a credit card by using the contract-based SOAP API as follows:

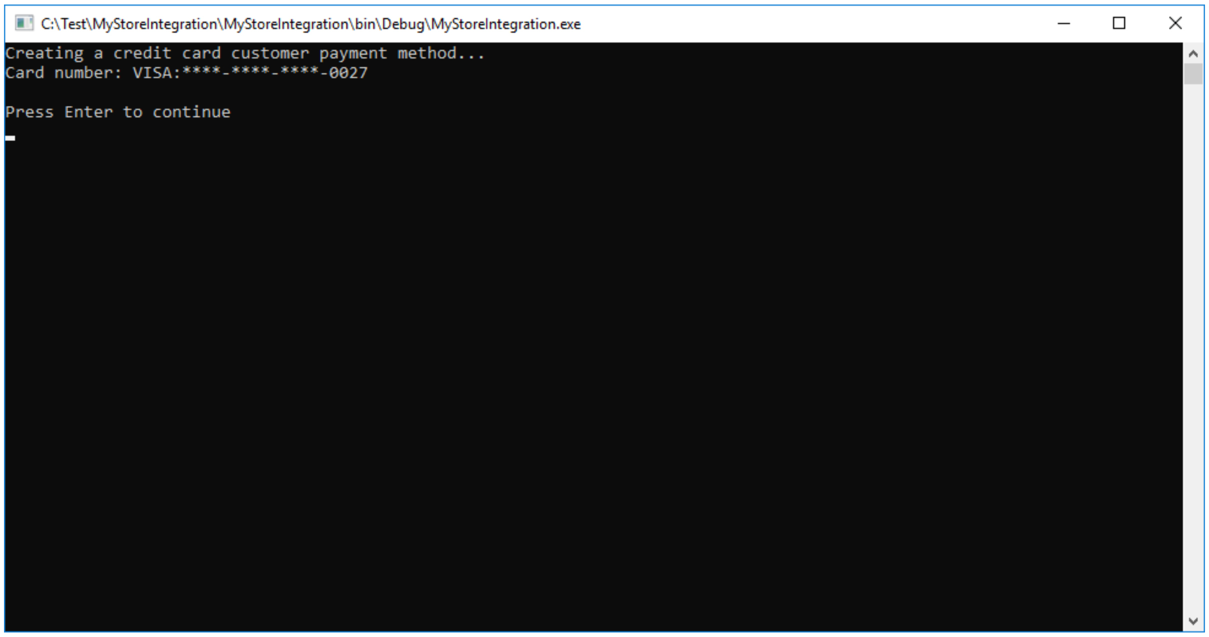
1. In the `Integration` folder of the *MyStoreIntegration* project, create a `CreditCardPayments` class, and add the `MyStoreIntegration.Default using` directive in the created file.
2. In the `CreditCardPayments` class, define the `CreateCustomerPaymentMethod()` method as follows.

```
//Creating a customer payment method by credit card
public static void CreateCustomerPaymentMethod(DefaultSoapClient soapClient)
{
    Console.WriteLine("Creating a credit card customer payment method...");

    //Customer payment method data
    string customerID = "C000000003";
    string paymentMethod = "VISA";
    //Payment profile ID value, which was obtained from Authorize.Net
    string paymentProfileID = "35596199";
    string cashAccount = "102050MYST";
    //Customer profile ID value, which was obtained from Authorize.Net
    string customerProfileID = "37100472";

    //Specify the parameters of the created customer payment method
    CustomerPaymentMethod paymMethToBeCreated = new CustomerPaymentMethod
    {
        CustomerID = new StringValue { Value = customerID },
        PaymentMethod = new StringValue { Value = paymentMethod },
        CashAccount = new StringValue { Value = cashAccount },
        CustomerProfileID = new StringValue { Value = customerProfileID },
        Details = new[]
                {
                    new CustomerPaymentMethodDetail
                    {
                        Name = new StringValue{Value = "Payment Profile ID"},
                        Value = new StringValue{Value = paymentProfileID}
                    }
                }
    };

    //Create the customer payment method
    CustomerPaymentMethod createdMethod =
      (CustomerPaymentMethod)soapClient.Put(paymMethToBeCreated);

    //Display the card number of the created payment method
    Console.WriteLine("Card number: " + createdMethod.CardAccountNbr.Value);
    Console.WriteLine();
    Console.WriteLine("Press Enter to continue");
    Console.ReadLine();
}
```

3. In the `try` block of the `Main()` method of the `Program` class, call the `CreateCustomerPaymentMethod()` method, as the following code shows.

```
//Create a customer payment method
CreditCardPayments.CreateCustomerPaymentMethod(soapClient);
```

4. Rebuild the project, and run the application. The value of the **Card/Account No** box of the *Customer Payment Methods* (AR303010) form is displayed in the console application window, as shown in the following screenshot. The **Card/Account No.** box contains the card identifier in the system, which consists of the payment method ID and the last four numbers of the credit card. This format complies with PCI DSS.


C:\Test\MyStoreIntegration\MyStoreIntegration\bin\Debug\MyStoreIntegration.exe

```
Creating a credit card customer payment method...
Card number: VISA:****-****-****-0027

Press Enter to continue
```

Figure: Console application window

Related Links

Put() Method

Setup of Card Payment Processing

Additional Information: Registration of a Customer Credit Card with the Screen-Based SOAP API

This scenario is outside of the scope of this course but may be useful to some readers.

The *Customer Payment Methods* (AR303010) form does not have key elements among the elements of the form; instead, the form has an internal key field. That is, you cannot create a record on this form by specifying the unique values of key elements on the form. Instead, you should use the `Insert` action and disable the linked commands (that is, set the linked commands to `null`) of the elements that are linked to the internal key field (these elements are `Customer`, `PaymentMethod`, and `CustomerProfileID`).

You find the needed detail line on the form to specify the payment profile ID by using the `Key` command.

For details about the commands, see the following topics:

- *Commands for Clicking Buttons on a Form*
- *Commands for Setting the Values of Elements*
- *Commands for Record Searching: Key Command*

Additional Information: Processing of Credit Card Payments

The following scenarios are outside of the scope of this course but may be useful to some readers.

Selecting a Customer Credit Card During the Creation of a Sales Order

When you create a sales order through a contract-based API, you can select the credit card to be used for the payment of the sales order by specifying the value of the `PaymentProfileID` field of the `SalesOrder` entity. The selection of the credit card by the payment profile ID guarantees that you select the correct credit card, which cannot be guaranteed if you select the credit card by the card number. This is because, for example, multiple credit cards of a customer can have the same card number but different expiration dates; these cards would have different payment profile IDs.

Capturing Credit Card Payments

Through the Acumatica ERP web services APIs, you can capture credit card payments.

To capture a payment through the contract-based APIs, you need to use the `CaptureCreditCardPayment` action available for the `Payment` entity of the *Default/18.200.001* endpoint. Because the capturing of a credit card payment is a long-running operation, you need to monitor the status of the long-running operation before retrieving the result of capturing. (You have learned how to run a long-running operation and monitor its status in *Lesson 3.1: Releasing an Invoice*.)

To capture a credit card payment through the screen-based SOAP API, you need to configure the sequence of API commands, which includes selecting the needed payment on the *Payments and Applications* (AR302000) form and clicking **Actions** > **Capture CC Payment** for the selected payment. You submit this sequence of commands to Acumatica ERP through the `Submit()` method that corresponds to the Payments and Applications form, and monitor the status of the long-running operation before retrieving the result of the capture.

Lesson Summary

In this lesson, you have learned how to create a customer payment method for a customer credit card. You have used the customer profile ID and the payment profile ID of the customer in Authorize.Net to create a customer payment method in Acumatica ERP.

You have reviewed how the creation of a customer payment method can be implemented with the screen-based SOAP API. You have also reviewed how you can capture credit card payments through the different kinds of APIs of Acumatica ERP.

Part 5: Attachment of Files and Notes

In this part of the guide, you will add notes and attachments to records in Acumatica ERP through the web services APIs. You will add a note to a stock item record and attach a file to a stock item record.

As a result of completing the lessons of this part, you will know to which records you can add notes and attachments through the web services APIs, and how to add these notes and attachments.

Lesson 5.1: Adding a Note to a Stock Item Record

The administrator of the MyStore company's online store can add notes to inventory items by using the administrator interface of the online store. The online store passes the notes about the stock items to Acumatica ERP by using the web services API.

In this lesson, you will attach a note to a stock item. You will use the `StockItem` entity of the *Default/18.200.001* endpoint; this entity is mapped to the *Stock Items* (IN202500) form. You will find the needed stock item by using the value of the key field (inventory ID). To add a note, you will use the `note` system field of the `StockItem` entity.

The implementation of this scenario with the screen-based SOAP API is outside of the scope of this course. For a summary of this implementation, see *Additional Information: Addition of Notes to Records with the Screen-Based SOAP API*.

In this lesson, you will not attach notes to detail entities (such as warehouse details of a stock item); this scenario is described briefly in *Additional Information: Addition of Notes to Detail Lines*.

Lesson Objective

In this lesson, you will learn how to add a note to a record through the contract-based APIs.

Example 5.1.1: Using PUT and the note Field (REST)

In this example, you will add a note to the *AALEGO500* stock item record by using the `PUT` method and the REST API.

Adding a Note to a Stock Item Record

To add a note to the stock item record by using the contract-based REST API, do the following:

1. In the Postman collection, add a new request with the following settings:
   - HTTP method: `PUT`
   - URL: *https://localhost/MyStoreInstance/entity/Default/18.200.001/StockItem*
   - Parameters of the request: `$select=InventoryID`
   - Body of the request:

     ```
     {
      "InventoryID":{"value":"AALEGO500"},
      "note": "My note"
     }
     ```

   - Headers:

Key	Value
Accept	application/json
Content-Type	application/json

2. Send the request. If the request is successful, the response contains the `200 OK` status. The following code shows an example of the response.

   ```
   {
       "id": "2d1cac03-bfb4-4934-8203-172a4d948400",
       "rowNumber": 1,
       "note": "My note",
       "InventoryID": {
           "value": "AALEGO500"
       },
       "custom": {},
       "files": []
   }
   ```

3. Save the request.

Example 5.1.2: Using PUT and the Note Field (SOAP)

In this example, you will add a note to the *CONGRILL* stock item record by using the `Put()` method of the `DefaultSoapClient` object through the contract-based SOAP API.

Adding a Note to a Stock Item Record

To add a note to the stock item record by using the contract-based SOAP API, do the following:

1. In the `Integration` folder of the *MyStoreIntegration* project, add the `Attachment` class and add the `using` directives shown in the following code sample to it, or use the existing `Attachment` class, which you have created in the *I310 Web Services: Advanced | Data Retrieval* training course.

```
using MyStoreIntegration.Default;
using System.IO;
```

2. In the `Attachment` class, add the following code.

```
//Adding a note to a stock item record
public static void AddNoteToStockItem(DefaultSoapClient soapClient)
{
    Console.WriteLine("Adding a note to a stock item record...");

    //Stock item data
    string inventoryID = "CONGRILL";
    string noteText = "My note";

    //Find the stock item in the system, and specify the note text
    StockItem stockItemToBeUpdated = new StockItem
    {
        InventoryID = new StringSearch { Value = inventoryID },
        Note = noteText
    };
    StockItem stockItem = (StockItem)soapClient.Put(stockItemToBeUpdated);

    //Display the summary of the created stock item
    Console.WriteLine("Inventory ID: " + stockItem.InventoryID.Value);
    Console.WriteLine("Note text: " + stockItem.Note);
    Console.WriteLine();
    Console.WriteLine("Press Enter to continue");
    Console.ReadLine();
}
```

3. In the `try` block of the `Main()` method of the `Program` class, call the `AddNoteToStockItem()` method of the `Attachment` class, as the following code shows.

```
//Add a note to a stock item
Attachment.AddNoteToStockItem(soapClient);
```

4. Rebuild the project, and run the application. The inventory ID and the note text of the stock item record are displayed in the console application window.

Additional Information: Addition of Notes to Records with the Screen-Based SOAP API

This scenario is outside of the scope of this course but may be useful to some readers.

You add a note to a record on an Acumatica ERP form by using the `Value` command and the `NoteText` field of the object that corresponds to the summary area of the target form. For example, to add a note to a stock item record, you use the `NoteText` field of the `StockItemSummary` object.

Additional Information: Addition of Notes to Detail Lines

This scenario is outside of the scope of this course but may be useful to some readers.

To add a note to a detail line of a document, you use the same approach that you use to add notes to top-level entities, as described in this lesson. That is, you do the following:

- For the contract-based REST and SOAP API, you specify the value of the `note` system field of the detail entity. For example, to add a note to a warehouse detail line of a stock item, you specify the value of the `note` system field of the `StockItemWarehouseDetail` entity, which is a detail entity of the `StockItem` entity. You identify the detail line that should be updated by the values of key fields or the entity ID.
- For the screen-based SOAP API, you specify the note by using the `Value` command and the `NoteText` field of the object that corresponds to the detail tab on the target form. For example, to add a note to a warehouse detail line of a stock item, you use the `NoteText` field of the `WarehouseDetails` object. You identify the detail line that should be updated by using the `Key` command.

Lesson Summary

In this lesson, you have learned how to add notes to records in Acumatica ERP by using the contract-based APIs. To add a note, you have used the `note` system field of the entity.

You have also reviewed how to add notes to records through the screen-based SOAP API, and how to add notes to detail lines of documents.

Lesson 5.2: Attaching a File to a Stock Item Record

The administrator of the MyStore company's online store can attach files to inventory items by using the administrator interface of the online store. The online store passes these files to Acumatica ERP by using the web services API.

Through the contract-based APIs, files cannot be attached to records that are not available for editing, such as closed AR invoices on the *Invoices and Memos* (AR301000) form. However, the files can be attached to these records through the UI.

In this lesson, you will attach a file to a stock item. You will use the `StockItem` entity of the *Default/18.200.001* endpoint; this entity is mapped to the *Stock Items* (IN202500) form. You will find the needed stock item by using the value of the key field (inventory ID).

The implementation of this scenario with the screen-based SOAP API is outside of the scope of this course. For a summary of this implementation, see *Additional Information: Attachment of a File to a Record with the Screen-Based SOAP API*.

In this lesson, you will not attach files to detail entities (such as warehouse details of a stock item). For a brief description of this scenario, see *Additional Information: Attachment of Files to Detail Lines*.

Lesson Objective

In this lesson, you will learn how to attach a file to a record through the contract-based APIs.

Example 5.2.1: Using PUT and the Particular Endpoint (REST)

In this example, you will attach the `T2MCRO.jpg` file to the *AALEGO500* stock item record by using the `PUT` method. In the URL for the file attachment, you specify the inventory ID (which is the key field of the stock item record) and the name of the file. You have to specify the values of all key fields of the record in the URL. You pass the file in the body of the request.

Attaching a File to a Stock Item Record

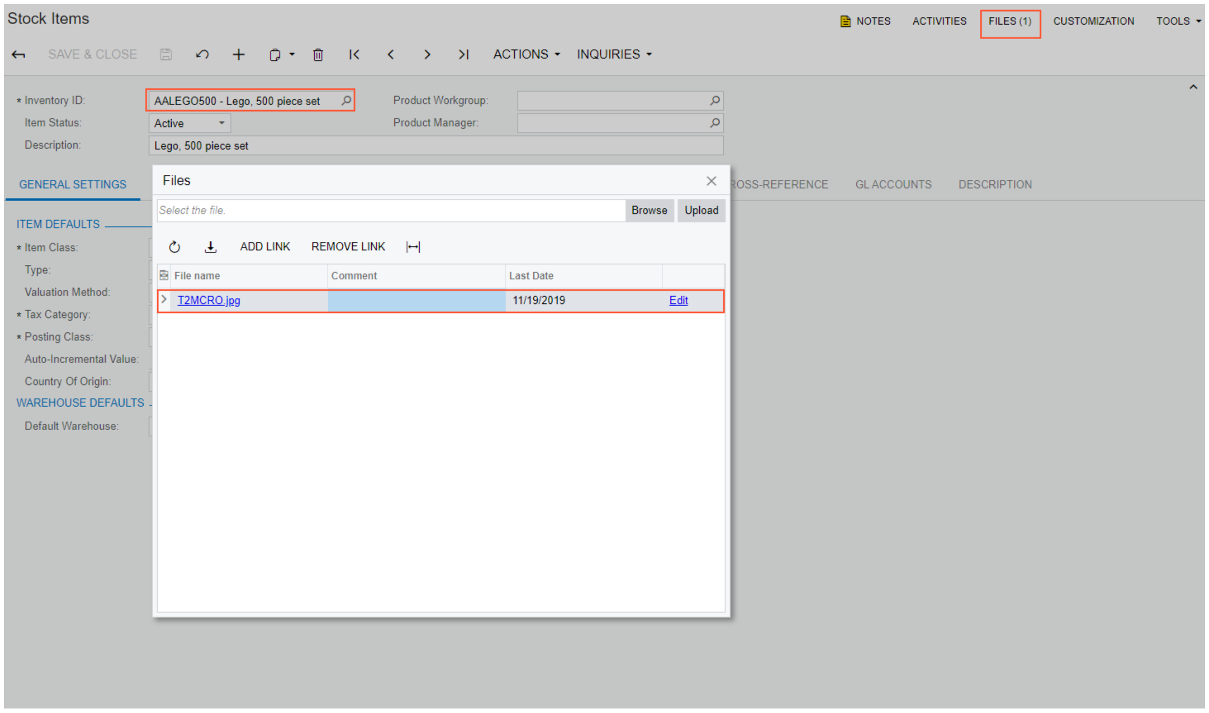
To attach a file to the stock item record by using the contract-based REST API, do the following:

1. In the Postman collection, add a new request with the following settings:
   - HTTP method: `PUT`
   - URL: *https://localhost/MyStoreInstance/entity/Default/18.200.001/StockItem/AALEGO500/files/T2MCRO.jpg*
   - Body of the request: The `T2MCRO.jpg` file in binary format
   - Headers:

Key	Value
`Accept`	`application/json`
`Content-Type`	`application/json`

2. Send the request. If the request is successful, the response contains the `204 No Content` status.

   On the *Stock Items* (IN202500) form, verify that the file is attached to the *AALEGO500* stock item.



Inventory ID: AALEGO500 - Lego, 500 piece set
Item Status: Active
Description: Lego, 500 piece set
Product Workgroup:
Product Manager:
Item Class:
Type:
Valuation Method:
Tax Category:
Posting Class:
Auto-Incremental Value:
Country Of Origin:
Default Warehouse:

Select the file.

File name	Comment	Last Date	
T2MCRO.jpg		11/19/2019	Edit

Figure: Attached file

3. Save the request.

Related Links

Attachment of a File to a Record

Example 5.2.2: Using PutFiles() (SOAP)

In this example, you will attach the `T2MCRO.jpg` file to the *CONGRILL* stock item record by using the `PutFiles()` method of the `DefaultSoapClient` object through the contract-based SOAP API. You will use the `File` entity to specify the file name and submit the file content.

Attaching a File to a Stock Item Record

To attach a file to the stock item record by using the contract-based SOAP API, do the following:

1. In the `Attachment` class, add the following method.

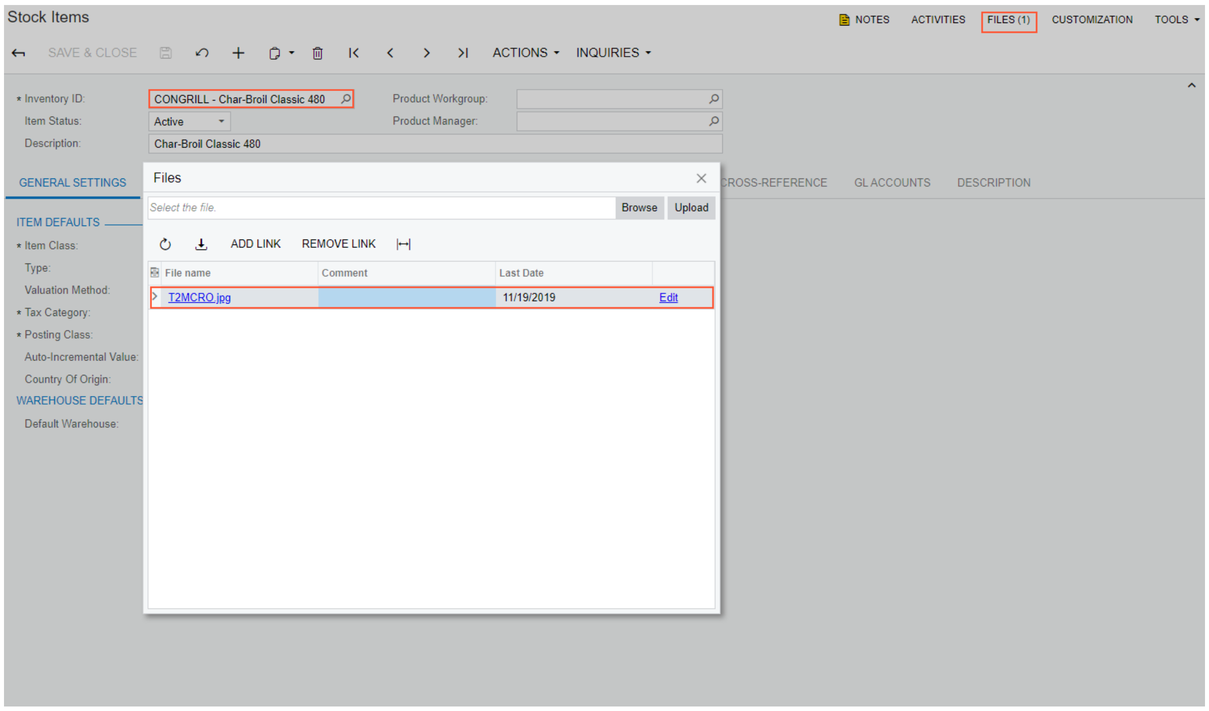
```
//Adding a file to a stock item record
public static void AddFileToStockItem(DefaultSoapClient soapClient)
{
    Console.WriteLine("Adding a file to a stock item record...");

    //Input data
    string inventoryID = "CONGRILL";
    //The path to the file that you need to attach to the stock item
    string filePath = "D:\\MyStoreIntegration\\SourceFiles\\";
    //The name of the file
    string fileName = "T2MCRO.jpg";

    //Read the file data
    byte[] filedata;
    using (FileStream file =
    System.IO.File.Open(Path.Combine(filePath, fileName), FileMode.Open))
    {
        filedata = new byte[file.Length];
        file.Read(filedata, 0, filedata.Length);
    }

    //Add the file to the stock item record
    StockItem stockItem = new StockItem
    {
        InventoryID = new StringSearch { Value = inventoryID },
    };

    Default.File[] stockItemFiles = new[]
    {
        new MyStoreIntegration.Default.File
        {
            Name = fileName,
            Content = filedata
        }
    };

    soapClient.PutFiles(stockItem, stockItemFiles);
}
```

2. In the `try` block of the `Main()` method of the `Program` class, call the `AddFileToStockItem()` method, as the following code shows.

```
//Add a file to a stock item
Attachment.AddFileToStockItem(soapClient);
```

3. Rebuild the project, and run the application.

   On the *Stock Items* (IN202500) form, the *CONGRILL* stock item now has a file attached, as shown in the following screenshot.



Inventory ID: CONGRILL - Char-Broil Classic 480
Item Status: Active
Description: Char-Broil Classic 480
Product Workgroup:
Product Manager:

File name	Comment	Last Date	
T2MCRO.jpg		11/19/2019	Edit

Figure: A file attached to the stock item

Related Links

PutFiles() Method

Additional Information: Attachment of a File to a Record with the Screen-Based SOAP API

This scenario is outside of the scope of this course but may be useful to some readers.

You attach a file to a record on an Acumatica ERP form by using the `Value` command and the `Attachment` service command of the object that corresponds to the summary area of the target form. For example, to attach a file to a stock item record, you use the `StockItemSummary.ServiceCommands.Attachment` service command. For details about the service commands, see *Commands for Working with Attachments*.

Additional Information: Attachment of Files to Detail Lines

This scenario is outside of the scope of this course but may be useful to some readers.

The attachment of files to detail lines is not supported by the available versions of system endpoints of the contract-based APIs. If you need to attach files to detail lines, you can use the screen-based SOAP API.

By using the screen-based SOAP API, you attach a file to a detail line on an Acumatica ERP form by using the `Value` command and the `Attachment` service command of the object that corresponds to the detail tab on the target form. For example, to attach a file to a warehouse detail of a stock item, you use the `WarehouseDetails.ServiceCommands.Attachment` service command. For details about the service commands, see *Commands for Working with Attachments*.

Lesson Summary

In this lesson, you have learned how to attach files to records in Acumatica ERP by using the contract-based APIs.

You have also reviewed how to attach files to records through the screen-based SOAP API.

Appendix: Web Integration Scenario Reference

In this topic, you can find reference links to the topics that describe how to implement the following integration scenarios:

- **Creating a shipment for multiple sales orders**: *Example 1.1.2: Using One Call of the Put() Method (SOAP)*.
- **Creating a stock item with attributes**: *Lesson 1.2: Creating a Stock Item with Attributes*
- **Updating a customer record by using the email address**: *Lesson 2.1: Updating a Customer Account*
- **Updating the detail lines of a sales order** : *Lesson 2.2: Updating the Detail Lines of a Sales Order*
- **Releasing an invoice**: *Lesson 3.1: Releasing an Invoice*
- **Creating a credit card customer payment method**: *Lesson 4.1: Registering a Customer Credit Card*
- **Adding a note to a stock item**: *Lesson 5.1: Adding a Note to a Stock Item Record*
- **Attaching a file to a stock item**: *Lesson 5.2: Attaching a File to a Stock Item Record*

Appendix: Troubleshooting

When I try to create a credit card processing method, I get the error *AR Error #144: Credit card processing error. Processing Center Error: E00040: The record cannot be found.* What should I do?

Make sure that the payment profile ID that you specified when you created a credit card processing method exists for the customer in your Authorize.Net developer sandbox account.

By default, Acumatica ERP deletes the payment profile ID in Authorize.Net if you delete the corresponding payment method in Acumatica ERP. The **Synchronize Deletion** check box is selected by default on the *Processing Centers* (CA205000) form for a processing center you create. If you do not want the payment profile ID to be deleted automatically, clear the **Synchronize Deletion** check box on the Processing Centers form for the needed processing center.

When I try to attach a file to a document, I get the error *Sequence contains no matching element*. What should I do?

Make sure the document that you specified in the request is editable. Through the contract-based APIs, files cannot be attached to records that are not available for editing in Acumatica ERP, such as closed AR invoices on the *Invoices and Memos* (AR301000) form.