INTEGRATION DEVELOPMENT GUIDE

DEVELOPER GUIDE

Acumatica ERP 2019 R2

Acumatica

Contents

Copyright.....6
Integration Development Guide.....7
Limiting Connections of Integrated Applications.....8
License Restrictions for API Users..... 8
Limitation of API Connections for Integrated Applications.....12
To Limit the Number of API Connections of Integrated Applications..... 13
To Limit API Connections of a Particular Application..... 15
Authorizing Client Applications to Work with Acumatica ERP.....16
Authorization Code Flow.....16
Implicit Flow.....24
Resource Owner Password Credentials Flow..... 28
Comparison of the Flows..... 34
To Register a Client Application..... 34
To Revoke the Access of a Connected Application..... 37
Configuring the Contract-Based REST and SOAP API..... 39
Contract-Based Web Services API.....39
Endpoints and Contracts.....40
API Entities, Fields, and Actions..... 42
Custom Fields..... 45
Custom Endpoints and Endpoint Extensions..... 47
Naming Rules for Endpoints.....48
Comparison of Contract Versions..... 49
Comparison of System Endpoints..... 50
To Create a Custom Endpoint.....57
To Extend an Existing Endpoint..... 59
To Validate an Endpoint.....61
Working with the Contract-Based REST API.....63
Representation of a Record in JSON Format.....64
Signing In to the Service..... 67
Signing out from the Service..... 70
Creation of a Record.....71
Update of a Record.....74
Retrieval of a Record by Key Fields.....78
Retrieval of a Record by ID..... 80
Retrieval of Records by Conditions.....82
Retrieval of Data from an Inquiry Form.....85
Parameters for Retrieving Records.....88
Removal of a Record.....91
Execution of an Action..... 94

Attachment of a File to a Record....97
Retrieval of a File Attached to a Record....100
Retrieval of the Schema of Custom Fields....103
Retrieval of the Acumatica ERP Version and the List of Endpoints....104
Multi-Language Fields....106
Working with the Contract-Based SOAP API....108
Multi-Language Fields....108
To Configure the Client Application....110
Working with the Screen-Based SOAP API....114
Screen-Based Web Services API....114
API Objects Related to Acumatica ERP Forms....115
Screen-Based API Wrapper....117
To Generate the WSDL File of the Web Services....120
To Import the WSDL File Into the Development Environment....122
To Use the Screen-Based API Wrapper....126
Working with Commands of the Screen-Based SOAP API....127
Commands for Retrieving the Values of Elements....127
Selection of a Group of Records for Export....129
Commands for Setting the Values of Elements....131
Commands for Clicking Buttons on a Form....131
Commands for Adding Detail Lines....132
Commands for Pop-Up Dialog Boxes and Pop-Up Forms....133
Commands for Pop-Up Panels....135
Commands for Record Searching: Filter Service Command....137
Commands for Record Searching: Key Command....139
Commands for Record Searching: Custom Field....140
Commands That Require a Commit....141
Commands for Working with Attachments....142
Commands for Working with Multi-Language Fields....143
Configuring Push Notifications....146
Push Notifications....146
Recommendations for the Data Queries....148
Push Notification Destinations....148
Push Notification Format....149
To Configure Push Notifications....151
To Process Failed Notifications....153
Defining Push Notifications....155
To Create a Built-In Definition....155
To Connect to the SignalR Hub....157
To Add Additional Information to Push Notifications....160
To Create a Custom Destination Type....162

Contract-Based REST API Reference..... 165
Contract-Based SOAP API Reference.....166
Login() Method.....166
Logout() Method.....168
SetBusinessDate() Method..... 169
Get() Method..... 169
GetList() Method (Contract Version 3)..... 170
GetList() Method (Contract Version 2)..... 171
Put() Method.....172
Delete() Method..... 176
Invoke() Method.....176
GetProcessStatus() Method..... 177
GetFiles() Method..... 178
PutFiles() Method.....179
GetCustomFieldSchema() Method..... 180
Attributes Property.....181
CustomFields Property.....182
ReturnBehavior Property (Contract Version 3)..... 184
ReturnBehavior Property (Contract Version 2)..... 187
Screen-Based SOAP API Reference..... 189
Login() Method.....189
Logout() Method.....191
SetLocaleName() Method.....192
SetBusinessDate() Method..... 192
GetScenario() Method..... 193
GetSchema() Method..... 193
SetSchema() Method.....194
Export() Method..... 194
Submit() Method..... 195
Import() Method.....197
Clear() Method..... 198
GetProcessStatus() Method..... 198
Contract-Based API Examples..... 199
Integration of Acumatica ERP Projects with External Systems (REST API Examples).....199
Creation of a Pro Forma Invoice..... 200
Management of Account Groups..... 205
Running of Project Billing.....207
Creation of a General Ledger Transaction with a Project Code That Does Not Produce a Project Transaction.....210
Time Entry.....212

Integration of Acumatica ERP with POS systems (SOAP API Examples).....213
Entry of a Direct Sales Invoice..... 213
Entry of a Direct Sales Invoice Along with a Return.....218
Entry of a Credit Memo with Positive and Negative Lines.....223
Entry of a Direct Sales Invoice in a Non-Default Currency..... 227
Entry of a Direct Sales Invoice for a Shipped Order and Return.....231
Entry of a Direct Sales Invoice for an Unshipped Sales Order.....238
Entry of a Direct Sales Invoice for a Partially Shipped Sales Order.....243
Entry of a Credit Memo for an Unshipped Sales Order..... 251

Copyright

Software Version: 2019 R2

Integration Development Guide

In this guide, you can find information about how to develop client applications that work with Acumatica ERP through the web services.

In This Guide

- *Authorizing Client Applications to Work with Acumatica ERP*
- *Configuring the Contract-Based REST and SOAP API*
- *Working with the Contract-Based REST API*
- *Working with the Contract-Based SOAP API*
- *Working with the Screen-Based SOAP API*
- *Working with Commands of the Screen-Based SOAP API*
- *Configuring Push Notifications*
- *Contract-Based REST API Reference*
- *Contract-Based SOAP API Reference*
- *Screen-Based SOAP API Reference*
- *Contract-Based API Examples*

Limiting Connections of Integrated Applications

Acumatica ERP licenses include limitations that affect the external applications that are integrated with Acumatica ERP through the web services APIs. In this chapter, you can find details about these limitations and the limitations that can also be configured for integrated applications.

For general information about Acumatica ERP licenses, see *Licensing and Activating Acumatica ERP*.

In This Chapter

- *License Restrictions for API Users*
- *Limitation of API Connections for Integrated Applications*
- *To Limit the Number of API Connections of Integrated Applications*
- *To Limit API Connections of a Particular Application*

License Restrictions for API Users

API users are client applications that sign in to Acumatica ERP by using one of the following ways:

- The sign-in method of the contract-based REST API
- The sign-in method of the contract-based SOAP API
- The sign-in method of the screen-based SOAP API
- The OAuth 2.0 mechanism of authorization for applications

Each Acumatica ERP license includes the limits on the number of web services API users, the number of concurrent API requests, and the number of web services API requests per minute. You can view these limits of the Acumatica ERP license on the *License Monitoring Console* (SM604000) form. The following sections describe these limits and the life cycle of API requests.

Number of Web Services API Users

On the **License** tab of the *License Monitoring Console* (SM604000) form, the **Maximum Number of Web Services API Users** box displays the license restriction for the number of API users that can work with Acumatica ERP. When an extra API user tries to sign in to the system and the number of API user sessions exceeds your license restriction, an error message is returned and the sign-in process is interrupted. The following diagram shows an example of how this limitation works with three sign-in requests when the **Maximum Number of Web Services API Users** is set to *2*.

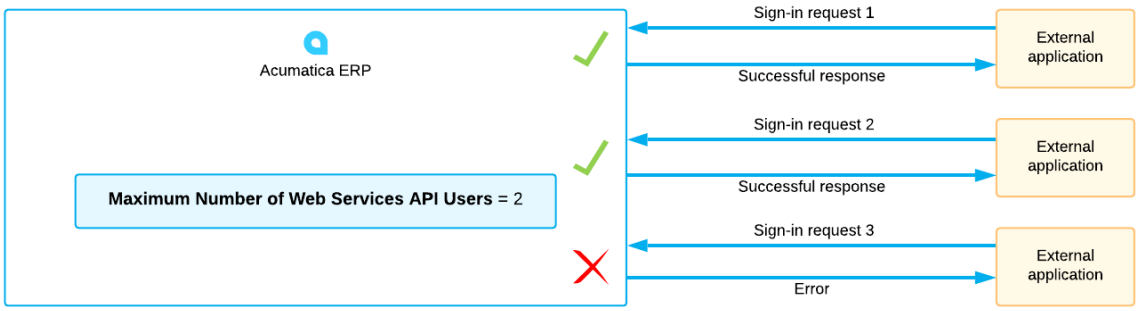


Figure: Rejection of an API user

To avoid exceeding the maximum number of API users, external applications must properly implement the signing in to and signing out from Acumatica ERP. If an external application does not close its sessions in Acumatica ERP (that is, does not sign out from Acumatica ERP in each session), this application can prevent other applications from signing in. For details about the implementation of signing in and signing out, see the following sources:

- *Signing In to the Service* for the REST API
- *Login() Method* for the contract-based SOAP API
- *Login() Method* for the screen-based SOAP API
- The descriptions of the scopes in *Authorization Code Flow*, *Implicit Flow*, and *Resource Owner Password Credentials Flow* for OAuth 2.0 authorization

Number of Concurrent Web Services API Requests

All incoming API requests (except the requests to sign out from Acumatica ERP, which are processed immediately) are placed in an internal queue. To process these requests, the server uses the API processing cores, which execute the requests in parallel. The number of cores is no more than the maximum number of concurrent web services API requests, which is specified in the **Maximum Number of Concurrent Web Services API Requests** box on the **License** tab of the *License Monitoring Console* (SM604000) form.

The API processing cores take the requests from the queue one by one. If the limit for the number of concurrent web services API requests has been reached (that is, if all API cores are processing the requests), the next concurrent request waits in the queue and is processed when one of the previous requests has completed.

The system sends a response to the request when the request is fully processed or declined. For details about the life cycle of the requests, see *Life Cycle of the Requests*.

The following diagram shows an example of how this limitation works with **Maximum Number of Concurrent Web Services API Requests** set to 3.

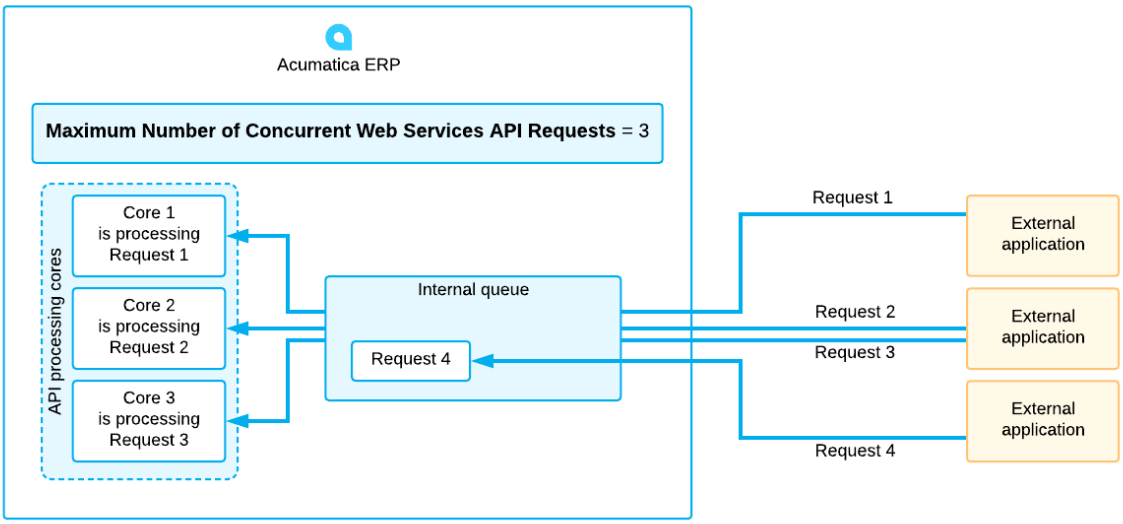


Figure: Request 4 waiting in the internal queue

Number of Web Services API Requests per Minute

The maximum number of requests that can be processed per minute is specified in the **Maximum Number of Web Services API Requests per Minute** box on the **License** tab of the *License Monitoring Console* (SM604000) form. If the number of requests in a particular minute reaches 50 percent of the limit specified for the license, the subsequent requests during this minute are delayed for the following time: 60 seconds minus the number of seconds that have passed since the beginning of the current minute, divided by the remaining number of requests that can be processed in the minute (per the specified maximum).

For example, suppose that in a particular license, the limit of the number of web services API requests per minute is 50. Since the beginning of the current minute, if the system has already processed 25 requests in 40 seconds and the system receives another request, this request is delayed in the internal queue for `(60-40)/(50-25)` seconds. After this delay, the request will be processed.

The system sends a response to the request when the request is fully processed or declined. For details about the life cycle of the requests, see *Life Cycle of the Requests*.

The following diagram shows an example of how this limitation works with **Maximum Number of Web Services API Requests per Minute** set to 50 and with 25 requests processed in the minute.

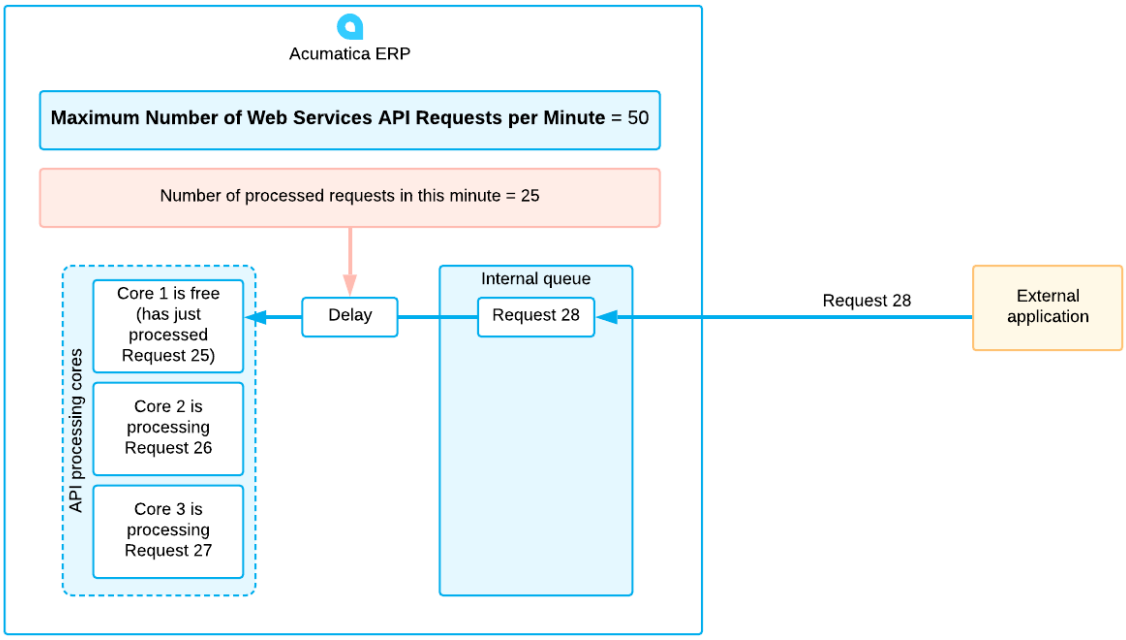


Figure: Request 28 delayed in the internal queue

Decline of the Requests

The request will be declined only if the number of requests in the internal queue is greater than 20, or the request remains in the queue for more than 10 minutes.

For example, suppose that during one second, each of 50 external applications sends a web services API request to one Acumatica ERP server. Suppose also that each request is processed for 1 second and that the maximum number of concurrent web services API requests in the license is 16. In this case, the first 16 requests will be passed to 16 API processing cores immediately. The next 20 requests will wait in the internal queue, and the last 14 requests will be declined.

Life Cycle of the Requests

The system sends a response to the request when the request is fully processed or declined. You can view the statistics of the delayed and declined requests on the **Statistics** tab of the *License Monitoring Console* (SM604000) form.

The following diagram shows the life cycle of a web services API request. For details about how the sign-in requests are processed, see *Limitation of API Connections for Integrated Applications*.

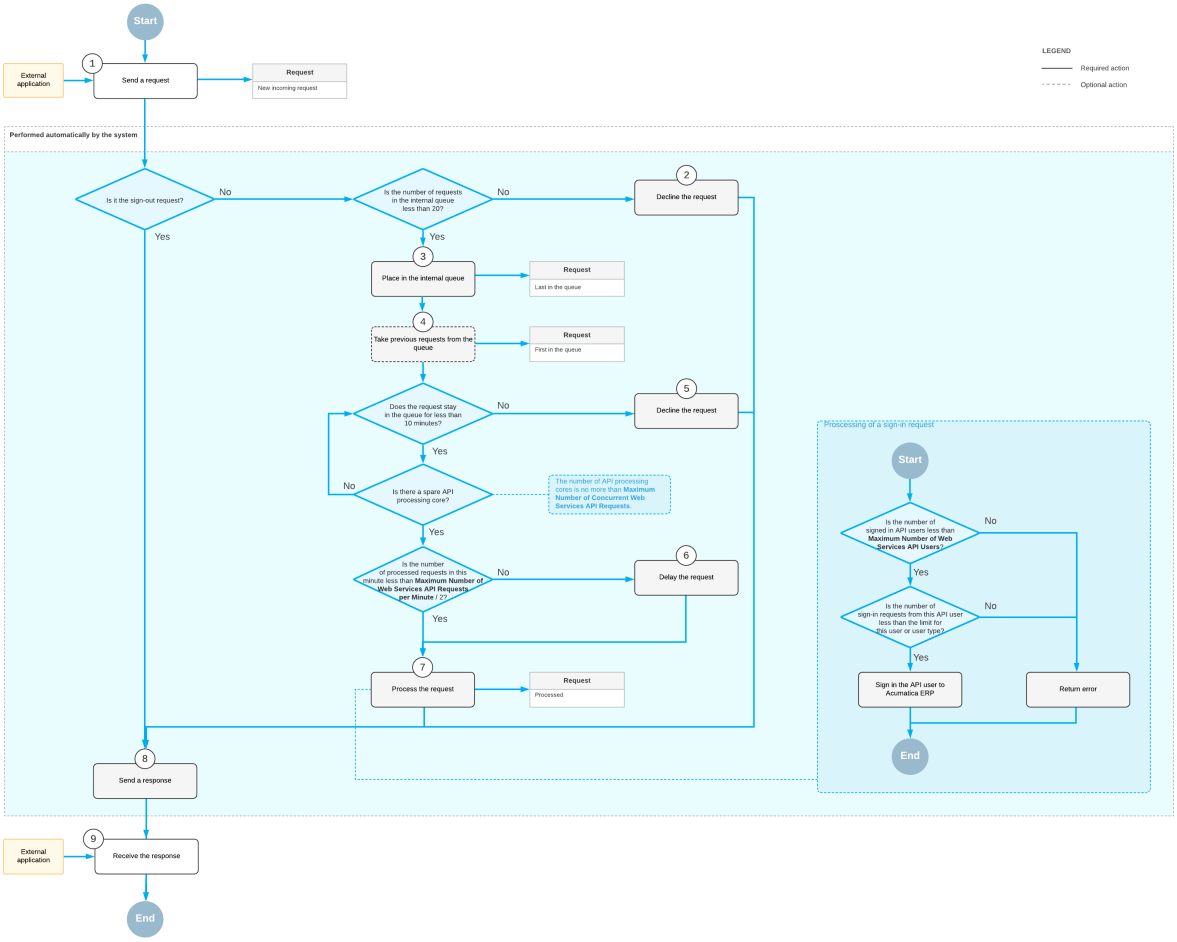


Figure: Life cycle of a request

Limitation of API Connections for Integrated Applications

If an integrated application does not properly sign out from Acumatica ERP, this application can use all of the API sessions included in the license (that is, reach the maximum number of API users, which is described in *License Restrictions for API Users*), thus preventing other integrated applications from signing in. To avoid such situations, you can limit the number of sessions for either each integration application assigned to a user type or an individual integration application.

Before configuring these limits, on the **Warnings** tab of the *License Monitoring Console* (SM604000) form, you can review whether any of the integrated application opens more than three sessions at a time (which is a predefined system value for this form). If the number of sessions has not been limited for an integrated application and this application opens more than three sessions at a time, a warning is generated and listed in the table on

this tab. This warning can indicate that you should configure a limitation for the particular integration application or review the implementation of signing in and signing out in the application.

Configuration of the Limitations

You limit the number of sessions used by each integration application on the *User Types* (EP202500) form. Also, if it is necessary to specify a different limit for a particular integration application, you can specify this limit on the *Users* (SM201010) form. For details about how to limit the number of sessions in both of these ways, see *To Limit the Number of API Connections of Integrated Applications* and *To Limit API Connections of a Particular Application*.

Life Cycle of Sign-in Requests

The life cycle of any type of requests (including sign-in requests) is described in *Life Cycle of the Requests*. This section describes peculiarities of the life cycle of sign-in requests.

The limitations for the number of API users and the limit of the number of sessions for a particular application are taken into account when one of the API processing cores takes the sign-in request. The API processing core then processes the sign-in request as follows:

1. Checks whether the maximum number of API users is exceeded. For details about this limit, see *License Restrictions for API Users*.
2. Checks whether the limit of the number of sessions specified for this API user is exceeded.

   If the number of sessions opened by the integrated application reaches the limit specified for this integrated application on either the *User Types* (EP202500) form or the *Users* (SM201010) form, the integrated application cannot open any new session (that is, cannot sign in to Acumatica ERP) until it closes one of the existing sessions. The system returns an error in the response to the sign-in request.

   If no limit is specified for the integration application on the *User Types* form or on the *Users* form, the integration application can open no more sessions than the maximum number of API users in the license.

The diagram in *Life Cycle of the Requests* shows how a sign-in request is processed by an API processing core.

Related Links

- *To Limit the Number of API Connections of Integrated Applications*
- *To Limit API Connections of a Particular Application*

To Limit the Number of API Connections of Integrated Applications

On the *User Types* (EP202500) form, you limit the number of sessions (that is, API connections) used by each of the integrated applications. You can also specify a different

limit for a particular integrated application as described in *To Limit API Connections of a Particular Application*.

Before You Proceed
On the *User Roles* (SM201005) form, create a user role that will be used by integrated applications. For details about user roles, see *Role-Based Access*.

To Limit API Connections of Each Integrated Application

1. Open the *User Types* (EP202500) form.
2. Create a user type that will be used by integrated applications as described in *To Add a User Type*. On the **Allowed Roles** tab, add the role that you created in the preliminary instructions (*Before You Proceed*).
3. On the **Login Rules** tab, specify the following settings:
   - **Allowed Login Type**:
     - *API* if your integration application uses the `Login` method of one of the web services API to sign in to Acumatica ERP
     - *Unrestricted* if your integration application uses OAuth 2.0 mechanism of authorization
   - **Allowed Number of Sessions**: The maximum number of sessions (API connections) you want to allow for each integration application
   - **Turn Off Two-Factor Authentication**: Selected
4. Save your changes.
5. On the *Users* (SM201010) form, create a user account for each integrated application that can work with this Acumatica ERP instance. For each user account, specify the type that you have created in the previous steps and add the role that you have created in the preliminary instructions (*Before You Proceed*). For details about the creation of a user account, see *To Create a Local User Account*.
6. In each of the integrated applications, implement the signing in to Acumatica ERP with its own user account (one of the user accounts created in the previous step). For details about the implementation of signing in, see the following sources:
   - *Signing In to the Service* for the REST API
   - *Login() Method* for the contract-based SOAP API
   - *Login() Method* for the screen-based SOAP API
   - The descriptions of the scopes in *Authorization Code Flow*, *Implicit Flow*, and *Resource Owner Password Credentials Flow* for OAuth 2.0 authorization

Related Links

- *Limitation of API Connections for Integrated Applications*

- *To Limit API Connections of a Particular Application*

To Limit API Connections of a Particular Application

On the *Users* (SM201010) form, you can specify a limit for the number of sessions used by a particular integrated application.

Instead of specifying a limit for a particular integrated application, on the *User Types* (EP202500) form, you can limit the number of sessions for each integrated application. For details about how to limit the number of sessions for each integrated application, see *To Limit the Number of API Connections of Integrated Applications*.

If a limit is specified on the *Users* form for a user account associated with an integrated application, this limit overrides any limit specified on the *User Types* form. If no limit is specified on the *Users* form for a user account associated with an integration application, any limit specified on the *User Types* form is used.

To Limit API Connections of a Particular Application

1. Open the *Users* (SM201010) form.
2. For the user account that the integration application uses, type the account's maximum number of sessions in the **Allowed Number of Sessions** box.

Related Links

- *Limitation of API Connections for Integrated Applications*
- *To Limit the Number of API Connections of Integrated Applications*

Authorizing Client Applications to Work with Acumatica ERP

Acumatica ERP supports the OAuth 2.0 mechanism of authorization for applications that are integrated with Acumatica ERP through application programming interfaces (APIs) and OData. When a client application of Acumatica ERP uses the OAuth 2.0 mechanism of authorization, the client application does not operate with the Acumatica ERP credentials to sign in a user to Acumatica ERP; instead, the application obtains an access token from Acumatica ERP and uses this token when it requests data from Acumatica ERP.

Depending on the OAuth 2.0 flow that the client application implements, the client application either has no information on the credentials of an Acumatica ERP user or uses this information only once to obtain the access token. The OAuth 2.0 mechanism of authorization improves the security of the Acumatica ERP data accessed by the application and simplifies the management of access rights.

The client application that implements the OAuth 2.0 authorization mechanism can use one of the OAuth 2.0 authorization flows supported by Acumatica ERP, which are the following:

- Authorization code
- Implicit
- Resource owner password credentials

In this chapter, you can find details on the OAuth 2.0 authorization flows and information about how to register the OAuth 2.0 or OpenID Connect client applications and revoke access of the applications.

In This Chapter

- *Authorization Code Flow*
- *Implicit Flow*
- *Resource Owner Password Credentials Flow*
- *Comparison of the Flows*
- *To Register a Client Application*
- *To Revoke the Access of a Connected Application*

Authorization Code Flow

When you implement OAuth 2.0 authorization in a client application to make the application work with Acumatica ERP, you can use the authorization code flow. With this authorization flow, the client application never gets the credentials of the applicable Acumatica ERP user. After the user is authenticated in Acumatica ERP, the client application receives an authorization code, exchanges it for an access token, and then uses the access token to work with data in Acumatica ERP. When the access token expires, the client application can request a new access token by providing a refresh token.

The following diagram illustrates the authorization code flow, whose steps are described in the sections of this topic.

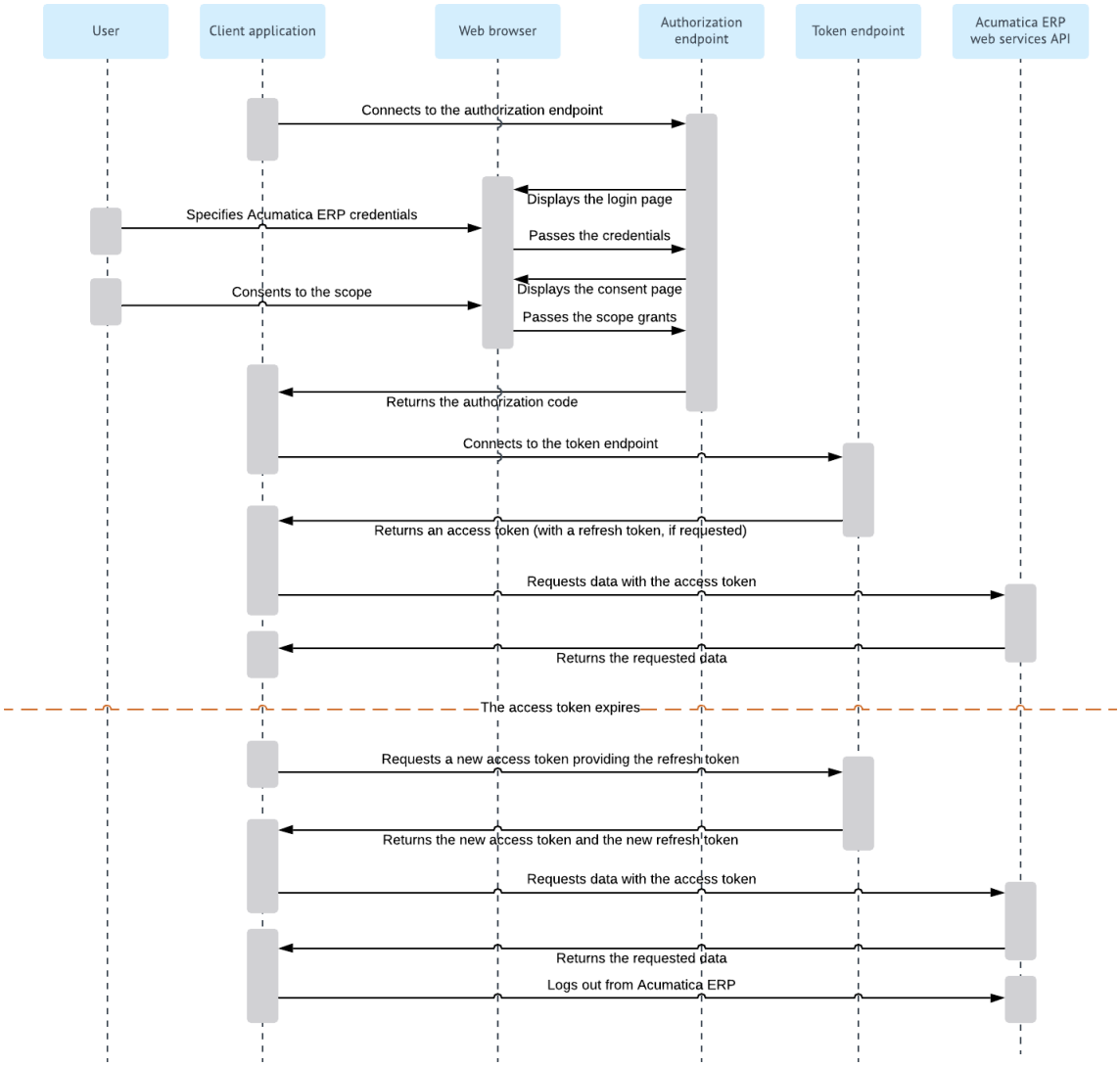


Figure: Authorization code flow

For details on the OAuth 2.0 authorization mechanism, see the specification at .

Granting Permission to a Client Application

Before an OAuth 2.0 client application can work with Acumatica ERP, you must register this application in Acumatica ERP and provide credentials to the application, as described in *To Register a Client Application with the Authorization Code Flow*. After the registration, you have the client ID and the secret value of the client application.

- According to the OAuth 2.0 specification, a secure connection between an OAuth 2.0 client application and the Acumatica ERP website with a Secure Socket Layer (SSL) certificate is required. Therefore, you have to set up the Acumatica ERP website for HTTPS before the OAuth 2.0 client application can work with data in Acumatica ERP. For more information, see *Setting Up an HTTPS Service in Web Server (IIS)*.
- When you are registering the client application, you have to be logged in to the tenant whose data the client application needs to access.

Connecting to the Authorization Endpoint

The client application connects to the authorization endpoint of Acumatica ERP by specifying the following URL with parameters:

- **URL**

  The client application can use one of the following options:

  - If the client application supports OpenID Connect Discovery, the client application can use the discovery endpoint address, which is `https://<Acumatica ERP instance URL>/identity/`. In this address, `<Acumatica ERP instance URL>` is the URL of the Acumatica ERP instance to which the client application is going to connect.

    For example, for a local Acumatica ERP instance with the name *AcumaticaDB*, the discovery endpoint address is `https://localhost/AcumaticaDB/identity/`.

    We recommend that the client application use the discovery endpoint address, which eliminates the need to change the application if the authorization or token endpoint address changes.

  - The client application can directly use the authorization endpoint address, which is `https://<Acumatica ERP instance URL>/identity/connect/authorize`. In this address, `<Acumatica ERP instance URL>` is the URL of the Acumatica ERP instance to which the client application is going to connect.

    For example, for a local Acumatica ERP instance with the name *AcumaticaDB*, the authorization endpoint address is `https://localhost/AcumaticaDB/identity/connect/authorize`.

- **URL Parameters**

  The client application should specify the following URL parameters.

Parameter	Description
`response_type`	The type of the OAuth 2.0 flow, which must be set to `code` for the authorization code flow.
`client_id`	The client ID that was assigned to the client application during the registration of the application in Acumatica ERP. The client ID must

Parameter	Description
	have the format in which the ID was generated during the registration of the application. That is, the client ID must include an auto-generated string and the ID of the company, such as `88358B02-A48D-A50E-F710-39C1636C30F6@MyCompany`. The client application will have access to the data of the company specified in the client ID.
`redirect_uri`	The URI in the client application to which the response to the request should be sent. The URI must exactly match one of the values specified for the application in the **Redirect URI** column on the **Redirect URIs** tab of the *Connected Applications* (SM303010) form.
`scope`	The access scope that is requested by the client application. The scope can be a combination of the following values, delimited by spaces: • `api`: Requests access to a web services API. If a user grants this scope to the application, the client application can work with either or both of the following types of the web services API: contract-based SOAP API or contract-based REST API. If this scope is granted and the `api:concurrent_access` scope is not granted, Acumatica ERP manages the sessions of the application through tokens. Acumatica ERP issues the first access token along with the session ID. If the client application requests a new access token by presenting a refresh token, Acumatica ERP reuses the session ID that was issued for the first access token issued with the refresh token. That is, the system uses a single session for each access granted to the client application. For details about the license limitations related to the number of sessions for client applications, see *License Restrictions for API Users*. • `offline_access`: Requests that a refresh token be granted. If a user grants this scope to the application, Acumatica ERP issues to the client application a refresh token along with the access token. (For information on issuing the access token, see *Connecting to the Token Endpoint* in this topic.) When the access token has expired, the client application can request a new access token by sending a request to the token endpoint and providing the refresh token. • `api:concurrent_access`: Requests permission for the concurrent use of multiple types of web service APIs. If a user grants this scope to the application, the client application can access data in Acumatica ERP in concurrent mode. In this case, Acumatica ERP can maintain multiple sessions for the client application, managing session IDs through cookies. We recommend that the client application request this scope only if concurrent access is required for the client application. For details about the license limitations relat-

Parameter	Description
	ed to the number of sessions for client applications, see *License Restrictions for API Users*.

An example of a URL with parameters is shown below. (Line breaks are for display purposes only.)

```
https://localhost/AcumaticaDB/identity/connect/authorize?
response_type=code
&client_id=4B1DFD71-C5EE-0B21-A6BE-9A1F060A93BD
&redirect_uri=http%3A%2F%2Flocalhost%2Fclientapp%2F
&scope=api%20offline_access
```

Authorizing a User in Acumatica ERP and Granting Access

The authorization endpoint directs the user of the client application to the login page of Acumatica ERP, where the user should enter the credentials to log in to a company configured in the Acumatica ERP instance.

The user must log in to the company that was specified in the `client_id` URL parameter passed to the authorization endpoint. (This company is selected by default on the login page.)

If the credentials are accepted by Acumatica ERP, the system displays the consent form, where the user can confirm that the application has access to the requested scopes. Only the scopes that were requested by the application are displayed on the consent form.

Receiving the Authorization Code

Once the user grants access to the requested scopes, Acumatica ERP redirects the client application to the `redirect_uri` address that was specified in the request, and adds the authorization code in the `code` URL parameter.

Connecting to the Token Endpoint

The client application connects to the token endpoint of Acumatica ERP by specifying the following URL and the following parameters in the request body:

- **URL**

  The client application can use one of the following options:

  - If the client application supports OpenID Connect Discovery, the client application can use the discovery endpoint address, which is `https://<Acumatica ERP instance URL>/identity/`. In this address, `<Acumatica ERP instance URL>` is the URL of the Acumatica ERP instance to which the client application is going to connect.

    For example, for a local Acumatica ERP instance with the name *AcumaticaDB*, the discovery endpoint address is `https://localhost/AcumaticaDB/identity/`.

We recommend that the client application use the discovery endpoint address, which eliminates the need to change the application if the authorization or token endpoint address changes.

- The client application can directly use the token endpoint address, which is `https://<Acumatica ERP instance URL>/identity/connect/token`. In this endpoint, `<Acumatica ERP instance URL>` is the URL of the Acumatica ERP instance to which the client application is going to connect.

  For example, for a local Acumatica ERP instance with the name *AcumaticaDB*, the token endpoint address is `https://localhost/AcumaticaDB/identity/connect/token`.

- **Parameters in the Request Body**

  You specify the following parameters in the request body.

Parameter	Description
`grant_type`	The type of the OAuth 2.0 flow, which must be set to `authorization_code` for the authorization code flow.
`client_id`	The client ID that was assigned to the client application during the registration of the application in Acumatica ERP. The client ID must have the format in which the ID was generated during the registration of the application. That is, the client ID must include an auto-generated string and the ID of the company, such as `88358B02-A48D-A50E-F710-39C1636C30F6@MyCompany`. The client application will have access to the data of the company specified in the client ID.
`code`	The authorization code that the client application has received from the authorization endpoint.
`client_secret`	The value of the secret that was created for the client application during the registration of the application in Acumatica ERP.
`redirect_uri`	The URI in the client application to which the response to the request should be sent. The URI must exactly match one of the values specified for the application in the **Redirect URI** column on the **Redirect URIs** tab of the *Connected Applications* (SM303010) form.

Receiving the Access Token

Acumatica ERP verifies the provided application credentials and issues the access token, which the client application should provide with each data request to Acumatica ERP. During authentication in Acumatica ERP, if the user has granted to the client application the `offline_access` scope, Acumatica ERP issues the refresh token along with the access token. A successful response includes the following parameters in the response body.

Parameter	Description
`token_type`	The type of the access token, which is *Bearer*.
`access_token`	The access token.
`expires_in`	The period of time during which the access token is valid.
`refresh_token`	The refresh token. The parameter is returned only if the `offline_access` scope was granted.

Requesting Data with the Access Token

The client application should include the access token in the `Authorization` header of each subsequent request to Acumatica ERP, as shown in the following HTTP example.

```
GET /AcumaticaDB/entity/Default/18.200.001/SalesOrder/SO/000001 HTTP/1.1
Host: localhost
Authorization: Bearer cde78a99a2dc6388eb8c7242a90cf9bc
```

Refreshing the Access Token

The access token is valid for a specific period of time, which is specified in the response that returns the access token. When the access token expires, the client application can request a new access token by providing the refresh token to the token endpoint. To request a new access token, the client application should have the following URL and the following parameters specified in the request body:

- **URL**

  The client application can use one of the following options:

  - If the client application supports OpenID Connect Discovery, the client application can use the discovery endpoint address, which is `https://<Acumatica ERP instance URL>/identity/`. In this address, `<Acumatica ERP instance URL>` is the URL of the Acumatica ERP instance to which the client application is going to connect.

    For example, for a local Acumatica ERP instance with the name *AcumaticaDB*, the discovery endpoint address is `https://localhost/AcumaticaDB/identity/`.

    We recommend that the client application use the discovery endpoint address, which eliminates the need to change the application if the authorization or token endpoint address changes.

  - The client application can directly use the token endpoint address, which is `https://<Acumatica ERP instance URL>/identity/connect/token`. In this endpoint, `<Acumatica ERP instance URL>` is the URL of the Acumatica ERP instance to which the client application is going to connect.

    For example, for a local Acumatica ERP instance with the name *AcumaticaDB*, the token endpoint address is `https://localhost/AcumaticaDB/identity/connect/token`.

- **Parameters in the Request Body**

  You specify the following parameters in the request body.

Parameter	Description
`grant_type`	The type of the request, which must be set to `refresh_token` for the request of the refresh token.
`client_id`	The client ID that was assigned to the client application during the registration of the application in Acumatica ERP. The client ID must have the format in which the ID was generated during the registration of the application. That is, the client ID must include an auto-generated string and the ID of the company, such as `88358B02-A48D-A50E-F710-39C1636C30F6@MyCompany`. The client application will have access to the data of the company specified in the client ID.
`client_secret`	The value of the secret that was created for the client application during the registration of the application in Acumatica ERP.
`refresh_token`	The refresh token that the client application received from the token endpoint along with the access token if the user granted the `offline_access` scope to the client application.

Receiving the New Access Token

Acumatica ERP verifies the provided application credentials and issues the new access token and the new refresh token. To request the access token once again, the client application should use the refresh token issued with the previous access token. A successful response includes the following parameters in the response body.

Parameter	Description
`token_type`	The type of the access token, which is *Bearer*.
`access_token`	The access token.
`expires_in`	The period of time during which the access token is valid.
`refresh_token`	The refresh token.

Logging Out from Acumatica ERP

To prevent issues with licenses that limit the number of concurrent user sessions, the client application should directly call the logout method of the Acumatica ERP web services API when the application finishes its work with Acumatica ERP.

Related Links

- *RFC 6749: The OAuth 2.0 Authorization Framework*

Implicit Flow

When you implement OAuth 2.0 authorization in a client application to make the application work with Acumatica ERP, you can use the implicit flow, which is a simplified variant of the authorization code flow.

With the implicit flow, the client application never gets the credentials of the applicable Acumatica ERP user. When the user is authenticated in Acumatica ERP, the client application does not receive an authorization code (as with the authorization code flow); instead, the client application directly receives an access token, and then uses the access token to work with data in Acumatica ERP. The access token is valid for a limited period of time and cannot be renewed.

The following diagram illustrates the implicit flow, whose steps are described in the sections later in this topic.

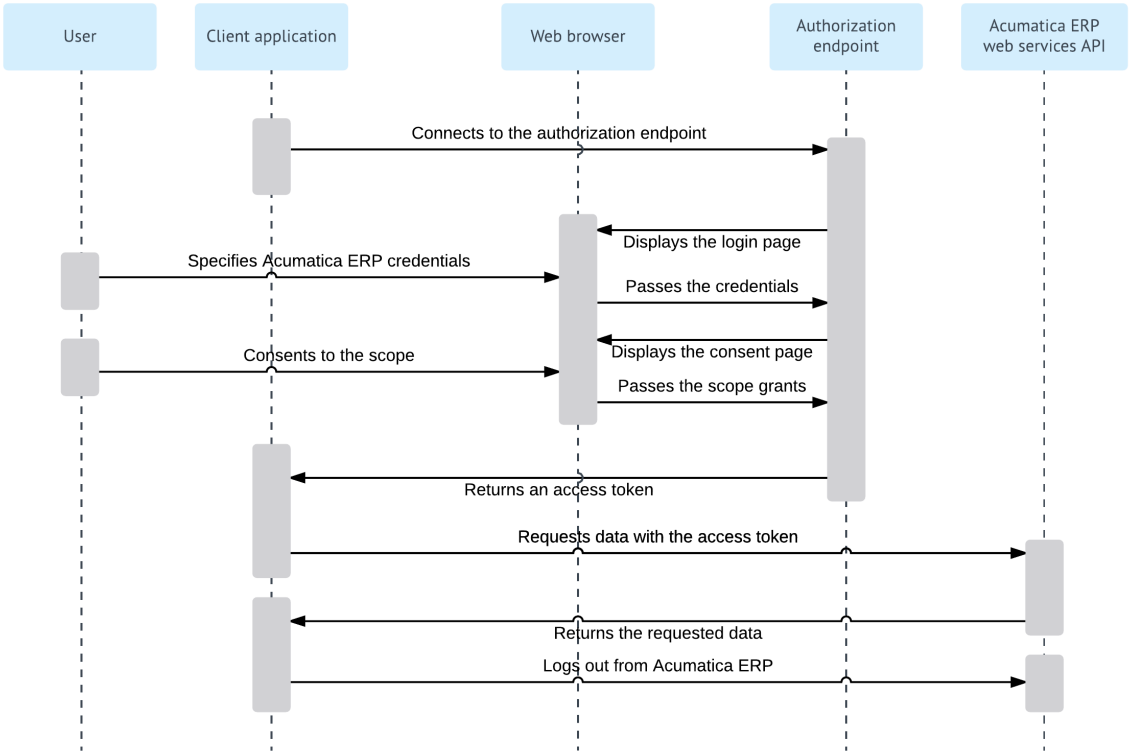


Figure: Implicit flow

This flow can be used for clients using a scripting language (such as JavaScript) or for mobile clients. For details on the OAuth 2.0 authorization mechanism, see the specification at .

Granting Permission to a Client Application

Before an OAuth 2.0 client application can work with Acumatica ERP, you must register this application in Acumatica ERP and provide credentials to the application, as described in *To Register a Client Application with the Implicit Flow*. After the registration, you have the client ID of the client application.

- According to the OAuth 2.0 specification, a secure connection between an OAuth 2.0 client application and the Acumatica ERP website with a Secure Socket Layer (SSL) certificate is required. Therefore, you have to set up the Acumatica ERP website for HTTPS before the OAuth 2.0 client application can work with data in Acumatica ERP. For more information, see *Setting Up an HTTPS Service in Web Server (IIS)*.
- When you are registering the client application, you have to be logged in to the tenant whose data the client application needs to access.

Connecting to the Authorization Endpoint

The client application connects to the authorization endpoint of Acumatica ERP by specifying the following URL and parameters:

- **URL**

  The client application can use one of the following options:

  - If the client application supports OpenID Connect Discovery, the client application can use the discovery endpoint address, which is `https://<Acumatica ERP instance URL>/identity/`. In this address, `<Acumatica ERP instance URL>` is the URL of the Acumatica ERP instance to which the client application is going to connect.

    For example, for a local Acumatica ERP instance with the name *AcumaticaDB*, the discovery endpoint address is `https://localhost/AcumaticaDB/identity/`.

    We recommend that the client application use the discovery endpoint address, which eliminates the need to change the application if the authorization or token endpoint address changes.

  - The client application can directly use the authorization endpoint address, which is `https://<Acumatica ERP instance URL>/identity/connect/authorize`. In this address, `<Acumatica ERP instance URL>` is the URL of the Acumatica ERP instance to which the client application is going to connect.

    For example, for a local Acumatica ERP instance with the name *AcumaticaDB*, the authorization endpoint address is `https://localhost/AcumaticaDB/identity/connect/authorize`.

- **URL Parameters**

  The client application should specify the following URL parameters.

Parameter	Description
`response_type`	The type of the OAuth 2.0 flow, which must be set to `token` for the implicit flow.
`client_id`	The client ID that was assigned to the client application during the registration of the application in Acumatica ERP. The client ID must have the format in which the ID was generated during the registration of the application. That is, the client ID must include an auto-generated string and the ID of the company, such as `88358B02-A48D-A50E-F710-39C1636C30F6@MyCompany`. The client application will have access to the data of the company specified in the client ID.
`redirect_uri`	The URI in the client application to which the response to the request should be sent. The URI must exactly match one of the values specified for the application in the **Redirect URI** column on the **Redirect URIs** tab of the *Connected Applications* (SM303010) form.
`scope`	The access scope that is requested by the client application. The scope can be a combination of the following values delimited by spaces: • `api`: Requests access to a web services API. If a user grants this scope to the application, the client application can work with either or both of the following types of the web services API: contract-based SOAP API or contract-based REST API. If this scope is granted and the `api:concurrent_access` scope is not granted, Acumatica ERP manages the sessions of the application through tokens. The system uses a single session for each access granted to the client application. • `api:concurrent_access`: Requests permission for the concurrent use of multiple types of web service APIs. If a user grants this scope to the application, the client application can access data in Acumatica ERP in concurrent mode. In this case, Acumatica ERP can maintain multiple sessions for the client application, managing session IDs through cookies. We recommend that the client application request this scope only if concurrent access is required for the client application. For details about the license limitations related to the number of sessions for client applications, see *License Restrictions for API Users*.  The `offline_access` scope is not supported by the implicit flow.

An example of the HTTP request is shown below. (Line breaks are for display purposes only.)

```
http://localhost/AcumaticaDB/identity/connect/authorize?
response_type=token
&client_id=4B1DFD71-C5EE-0B21-A6BE-9A1F060A93BD
&redirect_uri=http%3A%2F%2Flocalhost%2Fclientapp%2F
&scope=api
```

Authorizing a User in Acumatica ERP and Granting Access

The authorization endpoint directs the user of the client application to the login page of Acumatica ERP, where the user should enter the credentials to log in to a company configured in the Acumatica ERP instance.

The user must log in to the company that was specified in the `client_id` URL parameter passed to the authorization endpoint. (This company is selected by default on the login page.)

If the credentials are accepted by Acumatica ERP, the system displays the consent form, where the user can confirm that the application has access to the requested scopes. Only the scopes that were requested by the application are displayed on the consent form.

Obtaining the Access Token

Once the user grants access to the requested scopes, Acumatica ERP redirects the client application to the `redirect_uri` address, which was specified in the request, and adds the access token in the URL parameters. The redirect URL includes the following URL parameters.

Parameter	Description
`token_type`	The type of the access token, which is *Bearer*.
`access_token`	The access token.
`expires_in`	The period of time during which the access token is valid.

Refresh tokens are not supported by the implicit flow.

Requesting Data with the Access Token

The client application should include the access token in the `Authorization` header of each subsequent request to Acumatica ERP, as shown in the following HTTP example.

```
GET /AcumaticaDB/entity/Default/18.200.001/SalesOrder/SO/000001 HTTP/1.1
Host: localhost
Authorization: Bearer cde78a99a2dc6388eb8c7242a90cf9bc
```

Logging Out from Acumatica ERP

To prevent issues with licenses that limit the number of concurrent user sessions, the client application should directly call the logout method of the Acumatica ERP web services API when the application finishes its work with Acumatica ERP.

Related Links

- *RFC 6749: The OAuth 2.0 Authorization Framework*

Resource Owner Password Credentials Flow

When you implement OAuth 2.0 authorization in a client application to make the application work with Acumatica ERP, you can use the resource owner password credentials flow.

With the resource owner password credentials flow, the credentials (username and password) of the Acumatica ERP user are provided directly to the client application, which uses the credentials to obtain the access token. When the access token expires, the client application can request a new access token by providing a refresh token.

The following diagram illustrates the resource owner password credentials flow, whose steps are described in the sections later in this topic.

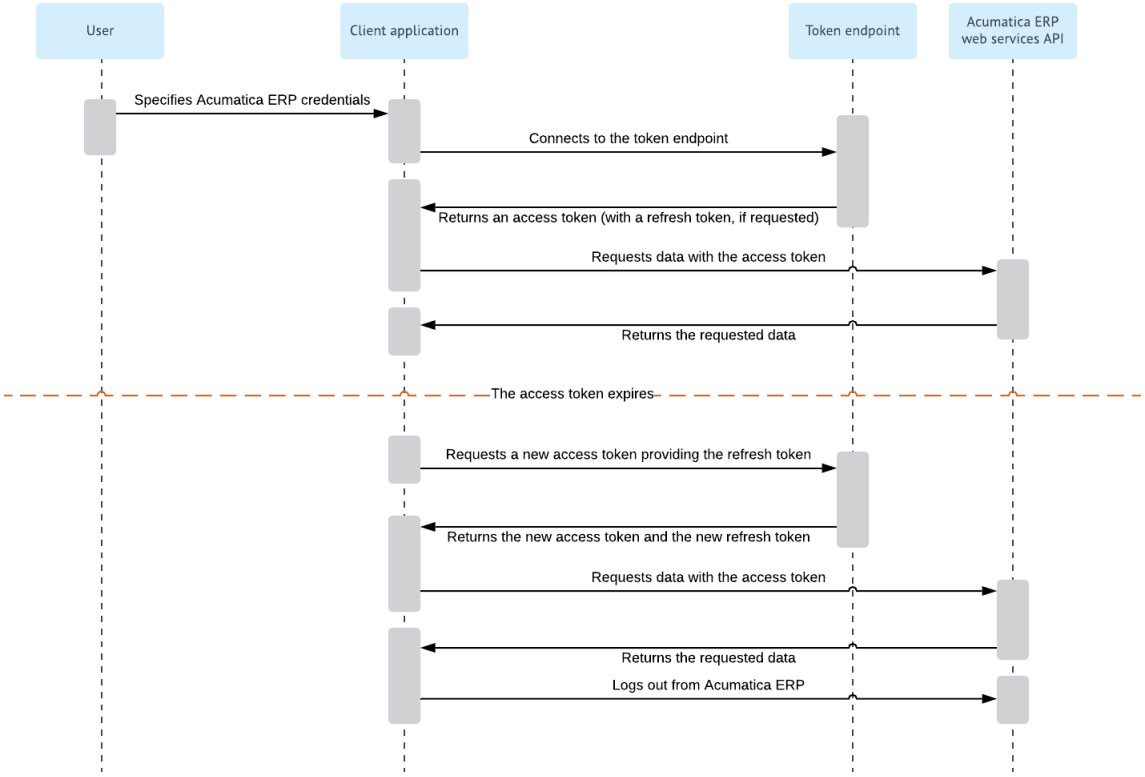


Figure: Resource owner password credentials flow

For details on the OAuth 2.0 authorization mechanism, see the specification at .

Granting Permission to a Client Application

Before an OAuth 2.0 client application can work with Acumatica ERP, you must register this application in Acumatica ERP and provide credentials to the application, as described in *To Register a Client Application with the Resource Owner Password Flow*. After the registration, you have the client ID and secret value of the client application.

- According to the OAuth 2.0 specification, a secure connection between an OAuth 2.0 client application and the Acumatica ERP website with a Secure Socket Layer (SSL) certificate is required. Therefore, you have to set up the Acumatica ERP website for HTTPS before the OAuth 2.0 client application can work with data in Acumatica ERP. For more information, see *Setting Up an HTTPS Service in Web Server (IIS)*.

- When you are registering the client application, you have to be logged in to the tenant whose data the client application needs to access.

Obtaining the Credentials of the Acumatica ERP User

The client application should obtain the username and password of the applicable Acumatica ERP user, which can then be exchanged for an access token.

Connecting to the Token Endpoint

The client application connects to the token endpoint of Acumatica ERP by specifying the following URL and parameters in the request body:

- **URL**

  The client application can use one of the following options:

  - If the client application supports OpenID Connect Discovery, the client application can use the discovery endpoint address, which is `https://<Acumatica ERP instance URL>/identity/`. In this address, `<Acumatica ERP instance URL>` is the URL of the Acumatica ERP instance to which the client application is going to connect.

    For example, for a local Acumatica ERP instance with the name *AcumaticaDB*, the discovery endpoint address is `https://localhost/AcumaticaDB/identity/`.

    We recommend that the client application use the discovery endpoint address, which eliminates the need to change the application if the authorization or token endpoint address changes.

  - The client application can directly use the token endpoint address, which is `https://<Acumatica ERP instance URL>/identity/connect/token`. In this endpoint, `<Acumatica ERP instance URL>` is the URL of the Acumatica ERP instance to which the client application is going to connect.

    For example, for a local Acumatica ERP instance with the name *AcumaticaDB*, the token endpoint address is `https://localhost/AcumaticaDB/identity/connect/token`.

- **Parameters in the Request Body**

  You specify the following parameters in the request body.

Parameter	Description
`grant_type`	The type of the OAuth 2.0 flow, which must be set to `password` for the resource owner password credentials flow.
`client_id`	The client ID that was assigned to the client application during the registration of the application in Acumatica ERP. The client ID must have the format in which the ID was generated during the registration of the application. That is, the client ID must include an au-

Parameter	Description
	to-generated string and the ID of the company, such as `88358B02-A48D-A50E-F710-39C1636C30F6@MyCompany`. The client application will have access to the data of the company specified in the client ID.
`client_secret`	The value of the secret that was created for the client application during the registration of the application in Acumatica ERP.
`username`	The username of an Acumatica ERP user.
`password`	The password for the specified `username`.
`scope`	The access scope that is requested by the client application. The scope can be a combination of the following values, delimited by spaces: • `api`: Requests access to a web services API. If a user grants this scope to the application, the client application can work with either or both of the following types of the web services API: contract-based SOAP API or contract-based REST API. If this scope is granted and the `api:concurrent_access` scope is not granted, Acumatica ERP manages the sessions of the application through tokens. Acumatica ERP issues the first access token along with the session ID. If the client application requests a new access token by presenting a refresh token, Acumatica ERP reuses the session ID that was issued for the first access token issued with the refresh token. That is, the system uses a single session for each access granted to the client application. For details about the license limitations related to the number of sessions for client applications, see *License Restrictions for API Users*. • `offline_access`: Requests that a refresh token be granted. If a user grants this scope to the application, Acumatica ERP issues to the client application a refresh token along with the access token. (For information on issuing the access token, see *Connecting to the Token Endpoint* in this topic.) When the access token has expired, the client application can request a new access token by sending a request to the token endpoint and providing the refresh token. • `api:concurrent_access`: Requests permission for the concurrent use of multiple types of web service APIs. If a user grants this scope to the application, the client application can access data in Acumatica ERP in concurrent mode. In this case, Acumatica ERP can maintain multiple sessions for the client application, managing session IDs through cookies. We recommend that the client application request this scope only if concurrent access is required for the client application. For details about the license limitations relat-

Parameter	Description
	ed to the number of sessions for client applications, see *License Restrictions for API Users*.

Receiving the Access Token

Acumatica ERP verifies the provided application credentials and issues the access token, which the client application should provide with each data request to Acumatica ERP. During authentication in Acumatica ERP, if the user has granted to the client application the `offline_access` scope, Acumatica ERP issues the refresh token along with the access token. A successful response includes the following parameters in the response body.

Parameter	Description
`token_type`	The type of the access token, which is *Bearer*.
`access_token`	The access token.
`expires_in`	The period of time during which the access token is valid.
`refresh_token`	The refresh token. The parameter is returned only if the `offline_access` scope was granted.

Requesting Data with the Access Token

The client application should include the access token in the `Authorization` header of each subsequent request to Acumatica ERP, as shown in the following HTTP example.

```
GET /AcumaticaDB/entity/Default/18.200.001/SalesOrder/SO/000001 HTTP/1.1
Host: localhost
Authorization: Bearer cde78a99a2dc6388eb8c7242a90cf9bc
```

Refreshing the Access Token

The access token is valid for a specific period of time, which is specified in the response that returns the access token. When the access token expires, the client application can request a new access token by providing the refresh token to the token endpoint. To request a new access token, the client application should have the following URL and the following parameters specified in the request body:

- **URL**

  The client application can use one of the following options:

  - If the client application supports OpenID Connect Discovery, the client application can use the discovery endpoint address, which is `https://<Acumatica ERP instance URL>/identity/`. In this address, `<Acumatica ERP instance URL>` is the URL of the Acumatica ERP instance to which the client application is going to connect.

    For example, for a local Acumatica ERP instance with the name *AcumaticaDB*, the discovery endpoint address is `https://localhost/AcumaticaDB/identity/`.

We recommend that the client application use the discovery endpoint address, which eliminates the need to change the application if the authorization or token endpoint address changes.

- The client application can directly use the token endpoint address, which is `https://<Acumatica ERP instance URL>/identity/connect/token`. In this endpoint, `<Acumatica ERP instance URL>` is the URL of the Acumatica ERP instance to which the client application is going to connect.

  For example, for a local Acumatica ERP instance with the name *AcumaticaDB*, the token endpoint address is `https://localhost/AcumaticaDB/identity/connect/token`.

- **Parameters in the Request Body**

  You specify the following parameters in the request body.

Parameter	Description
`grant_type`	The type of the request, which must be set to `refresh_token` for the request of the refresh token.
`client_id`	The client ID that was assigned to the client application during the registration of the application in Acumatica ERP. The client ID must have the format in which the ID was generated during the registration of the application. That is, the client ID must include an auto-generated string and the ID of the company, such as `88358B02-A48D-A50E-F710-39C1636C30F6@MyCompany`. The client application will have access to the data of the company specified in the client ID.
`client_secret`	The value of the secret that was created for the client application during the registration of the application in Acumatica ERP.
`refresh_token`	The refresh token that the client application received from the token endpoint along with the access token if the user granted the `offline_access` scope to the client application.

Receiving the New Access Token

Acumatica ERP verifies the provided application credentials and issues the new access token and the new refresh token. To request the access token once again, the client application should use the refresh token issued with the previous access token. A successful response includes the following parameters in the response body.

Parameter	Description
`token_type`	The type of the access token, which is *Bearer*.
`access_token`	The access token.
`expires_in`	The period of time during which the access token is valid.

Parameter	Description
`refresh_token`	The refresh token.

Logging Out from Acumatica ERP

To prevent issues with licenses that limit the number of concurrent user sessions, the client application should directly call the logout method of the Acumatica ERP web services API when the application finishes its work with Acumatica ERP.

Related Links

- *RFC 6749: The OAuth 2.0 Authorization Framework*

Comparison of the Flows

The table below summarizes the characteristics of the authorization flows supported by Acumatica ERP.

Characteristic	Authorization Code	Implicit	Resource Owner Password Credentials
The access token is returned from the authorization endpoint.	No	Yes	No
The access token is returned from the token endpoint.	Yes	No	Yes
The refresh token can be issued.	Yes	No	Yes
The client application has access to Acumatica ERP credentials (username and password).	No	No	Yes
The client application is authenticated in Acumatica ERP (that is, the client application provides the client ID and client secret).	Yes	No	Yes

Related Links

- *Authorization Code Flow*
- *Implicit Flow*
- *Resource Owner Password Credentials Flow*

To Register a Client Application

You use the *Connected Applications* (SM303010) form to register an OAuth 2.0 or OpenID Connect client application.

To register a client application in Acumatica ERP, you need to know the OAuth 2.0 flow that this application implements. For more information on the flows, see *Authorization Code Flow*, *Implicit Flow*, and *Resource Owner Password Credentials Flow*.

- According to the OAuth 2.0 specification, a secure connection between an OAuth 2.0 client application and the Acumatica ERP website with a Secure Socket Layer (SSL) certificate is required. Therefore, you have to set up the Acumatica ERP website for HTTPS before the OAuth 2.0 client application can work with data in Acumatica ERP. For more information, see *Setting Up an HTTPS Service in Web Server (IIS)*.
- When you are registering the client application, you have to be logged in to the tenant whose data the client application needs to access.

To Register a Client Application with the Authorization Code Flow

1. On the **System** tab, click **Integration**. In the navigation pane, navigate to **Configure > Connected Applications**.
2. In the **Client Name** box, type the name of the registered application.

   Leave the **Client ID** box blank. The system will fill it in when you save your changes on the form.
3. In the **OAuth 2.0 Flow** box, select *Authorization Code*.
4. On the **Secrets** tab, do the following for each client secret you want to add:
   a. On the tab toolbar, click **Add Shared Secret**. The **Add Shared Secret** dialog box opens.
   b. In the **Description** box, type the description of the shared secret.
   c. Optional: In the **Expires On (UTC)** box, enter the date and time on which the secret expires.
   d. Copy and save the value that is displayed in the **Value** box. The client application should use this client secret for authentication in Acumatica ERP.

      For security reasons, the value of the secret is displayed only once: when you create the secret by invoking this dialog box.
   e. Click **OK** to save the secret and close the dialog box.
5. On the **Redirect URIs** tab, do the following for each redirect URI you want to add:
   a. On the tab toolbar, click **Add Row**.
   b. In the **Redirect URI** column of the new row, type the exact redirect URI to which Acumatica ERP should redirect the client application after the client application has been authorized. The redirect URI must be absolute and must not have the fragment part (the part preceded with #).

6. On the form toolbar, click **Save**. Notice that the client ID has been generated in the **Client ID** box. The client application should use this client ID along with the client secret for authentication in Acumatica ERP.

To Register a Client Application with the Implicit Flow

1. On the **System** tab, click **Integration**. In the navigation pane, navigate to **Configure > Connected Applications**.
2. In the **Client Name** box, type the name of the registered application.

   Leave the **Client ID** box blank. The system will fill it in when you save your changes on the form.

3. In the **OAuth 2.0 Flow** box, select *Implicit*.
4. On the **Redirect URIs** tab, do the following for each redirect URI you want to add:
   a. On the tab toolbar, click **Add Row**.
   b. In the **Redirect URI** column of the new row, type the exact redirect URI to which Acumatica ERP should redirect the client application after the client application has been authorized. The redirect URI must be absolute and must not have the fragment part (the part preceded with #).
5. On the form toolbar, click **Save**. Notice that the client ID has been generated in the **Client ID** box. You should use this client ID to connect the client application to the authorization endpoint of Acumatica ERP.

To Register a Client Application with the Resource Owner Password Flow

1. On the **System** tab, click **Integration**. In the navigation pane, navigate to **Configure > Connected Applications**.
2. In the **Client Name** box, type the name of the registered application.

   Leave the **Client ID** box blank. The system will fill it in when you save your changes on the form.

3. In the **OAuth 2.0 Flow** box, select *Resource Owner Password Credentials*.
4. On the **Secrets** tab, do the following for each client secret you want to add:
   a. On the tab toolbar, click **Add Shared Secret**. The **Add Shared Secret** dialog box opens.
   b. In the **Description** box, type the description of the shared secret.
   c. Optional: In the **Expires On (UTC)** box, enter the date and time on which the secret expires.
   d. Copy and save the value that is displayed in the **Value** box. The client application should use this client secret for authentication in Acumatica ERP.

For security reasons, the value of the secret is displayed only once: when you create the secret by invoking this dialog box.

e. Click **OK** to save the secret and close the dialog box.

5. On the form toolbar, click **Save**. Notice that the client ID has been generated in the **Client ID** box. The client application should use this client ID along with the client secret for authentication in Acumatica ERP.

Related Links

- *Connected Applications*

To Revoke the Access of a Connected Application

To revoke the access of an OAuth 2.0 or OpenID Connect client application, you use either the *Connected Applications* (SM303010) form or the *User Profile* (SM203010) form.

On the *Connected Applications* form, you can revoke the access of any application registered in the current company. On this form, you revoke all access granted to the application.

On the *User Profile* form, you can revoke the access of any application to which you (that is, the user account to which you are logged in) have granted access. Any access granted to this application by other users remains unchanged.

To Revoke All Access of a Client Application

1. On the **System** tab, click **Integration**. In the navigation pane, navigate to **Configure > Connected Applications**.
2. In the **Client ID** box, select the application whose access you want to revoke.
3. On the form toolbar, click **Revoke Access**.
4. In the message box that opens, click **OK** to confirm that you want to revoke the access of the application.

After you have confirmed that you want to revoke access, all access tokens are removed from the Acumatica ERP database, and these tokens cannot be used to access data in Acumatica ERP. However, the client secrets remain valid until their expiration dates (if applicable), and the application can use these secrets to request a new access token.

To Revoke Access You Have Provided

1. In the info area (in the upper-right corner of the screen), click your user name, and then click **User Profile**.
2. On the toolbar of the **User Profile** form, which opens, click **View Connected Applications**. The list of applications to which you have granted access is displayed on the Client Application Permissions webpage.

3. For the application whose access you want to revoke, click **Revoke Access**.

After you have revoked access, the access tokens that were created when you granted access to the application are removed from the Acumatica ERP database, and these tokens cannot be used to access data in Acumatica ERP. However, the client secrets remain valid until their expiration dates (if applicable), and the application can use these secrets to request a new access token.

Configuring the Contract-Based REST and SOAP API

Acumatica ERP provides web services for integration with external systems. Through the web services of Acumatica ERP, external systems can get data records from Acumatica ERP, process these records, and save new or updated records to Acumatica ERP.

To access these web services, you can use the contract-based representational state transfer (REST) API, the contract-based SOAP application programming interface (API), and the screen-based SOAP API. In this chapter, you will find the main concepts that are related to the contract-based SOAP API and the contract-based REST API.

In This Chapter

- *Contract-Based Web Services API*
- *Endpoints and Contracts*
- *API Entities, Fields, and Actions*
- *Custom Fields*
- *Custom Endpoints and Endpoint Extensions*
- *Naming Rules for Endpoints*
- *Comparison of Contract Versions*
- *Comparison of System Endpoints*
- *To Create a Custom Endpoint*
- *To Extend an Existing Endpoint*
- *To Validate an Endpoint*

Contract-Based Web Services API

The contract-based web services APIs operate with business logic objects that do not depend on Acumatica ERP forms or their properties and methods. (In this context, *contract-based* means based on the object model the web services API provides.) Each contract of the web service is fixed and does not change based on system customization, localization, or any other changes made to Acumatica ERP.

For example, suppose that the contract of the web service contains the definition of the *CustomerID* field, which accesses the **Customer ID** element on the *Customers* (AR303000) form. If you have changed the name of the **Customer ID** element to **Customer Identifier** in a customization project, the contract of the web service remains fully functional and does not require update; also, your application requires no further modifications. You can access the **Customer Identifier** element on the form through the same *CustomerID* field.

REST and SOAP Interfaces of the Contract-Based Web Services

You can work with the contract-based web services through either the REST interface or the SOAP interface.

To use the contract-based REST or SOAP API in your application, first of all, you should decide which endpoint to use. You can find more information on the endpoints and their contracts in *Endpoints and Contracts*.

After that, you can use the REST API in your application. For details on the REST API, see *Working with the Contract-Based REST API*. For the REST API reference, see *Contract-Based REST API Reference*.

After you have selected the endpoint, to use the contract-based SOAP API in your application, you should obtain the WSDL description of the contract of this endpoint, import the WSDL file into your development environment (as described in *To Configure the Client Application*), and start developing your application. You can find the description of the SOAP API methods in *Contract-Based SOAP API Reference*.

You can find examples of how to use the contract-based SOAP and REST API in the *I210 Integration: Contract-Based Web Services* training course.

Related Links

- *Endpoints and Contracts*
- *API Entities, Fields, and Actions*

Endpoints and Contracts

You access the contract-based REST and SOAP API through endpoints that you configure on the *Web Service Endpoints* (SM207060) form.

Endpoints and Contracts

An *endpoint* is an entry point to the Acumatica ERP web services. For each endpoint that a web service API provides, a *contract* of the endpoint defines the entities, with their actions and fields, that are available through the endpoint and the methods that you can use to work with these entities.

The endpoint is identified by the URL that you use to access the web services API. You can see the name and version of an endpoint in its URL. For example, the endpoint `http://localhost/AcumaticaDB/entity/Default/18.200.001?wsdl` has the version *18.200.001* and the name *Default*. The version of an endpoint defines the list of entities, with their actions and fields you can work with through this endpoint.

The contract of an endpoint is identified by contract version. The version of a contract defines the list of methods for working with entities that you can use when working with Acumatica ERP through the endpoint with this version of the contract. For the difference between the contract versions, see *Comparison of Contract Versions*.

Contract Version 1 is not supported starting from Acumatica ERP 2018 R2.

System and Custom Endpoints

You can use two types of endpoints to access the web services:

- **System endpoint**: The system endpoints are precofigured in the system and have the *Default* name. Each of these endpoints has a predefined contract, which includes the API that is preconfigured in the system. You cannot change the contract of a system endpoint.

  If the API that is available in the contract of a system endpoint is sufficient for the requirements of your application, you should use the system endpoint for accessing Acumatica ERP web services. You can use the same system endpoint in future versions of Acumatica ERP. For example, if you use the system endpoint with Version 17.200.001 and Contract Version 3 to access Acumatica ERP 2017 R2, you can use the same endpoint to access future versions of Acumatica ERP.

  Acumatica ERP can include endpoints preconfigured in the system that have the names other than *Default*. The system uses these endpoints internally. We do not recommend that you use these endpoints.

- **Custom endpoint**: By default, there are no custom endpoints in the system. If the API provided by the system endpoint is not sufficient for the requirements of your application, you can create a custom endpoint. You can configure the contract of a custom endpoint by adding the needed elements of the API to the contract.

  If you need to use the same custom endpoint in future versions of Acumatica ERP, you should maintain it in future versions.

The following diagram provides an example of multiple endpoints configured in the system. The diagram shows two system endpoints with Contract Versions 2 and 3 and two custom endpoints with the names *EastEndpoint* and *WestEndpoint*.

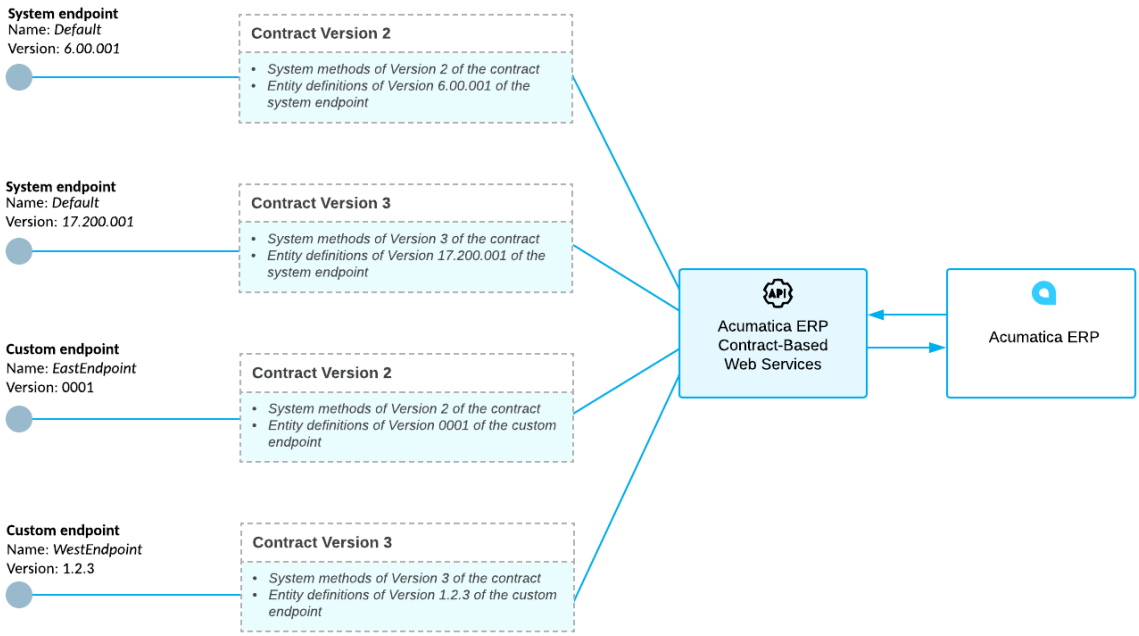


Figure: Contract-based web services

API Entities, Fields, and Actions

The contract of an endpoint defines the following elements of the contract-based web services API:

- **Entities**: An entity corresponds to a business logic object that you are going to work with. For example, the contract of a system endpoint includes the *Warehouse* entity, which represents a warehouse and holds the data related to the warehouse. This entity is associated with the *Warehouses* (IN204000) form.

  For a custom endpoint, if you are going to use an entity to transfer data to or from Acumatica ERP, you should associate this entity with a particular Acumatica ERP form. For example, you can create a *Vendor* entity, which represents a vendor. This entity is associated with the *Vendors* (AP303000) form.

- **Fields**: The fields of an entity correspond to the parameters of a business logic object. For example, the *Warehouse* entity that is available through the system endpoint has the *Description* and *WarehouseID* fields, among others. In the contract, these fields are mapped to the **Description** and the **Warehouse ID** elements of the Summary area of the *Warehouses* form.

  For a custom endpoint, if you need to connect the field with a particular element on an Acumatica ERP form, you should map the field to this element. For example, if you have created the *Vendor* entity, which designates a vendor, you can add the field *VendorID* to

the entity and connect this field with the **Vendor ID** element of the Summary area of the *Vendors* form.

- **Actions**: The actions of an entity correspond to the actions that can be applied to a business logic object. For example, the *TransferOrder* entity, which is available through the system endpoint, has the *ReleaseTransferOrder* action. This action corresponds to the **Release** button on the form toolbar of the *Transfers* (IN304000) form.

  For a custom endpoint, if you need to use an Acumatica ERP action, you should add this action to the contract of the custom endpoint with the needed parameters. For example, suppose you want to add an action that changes the customer ID of an existing customer, you can add the action *ChangeID* and map it to the **Change ID** action, which is available on the *Customers* form. The new action should have one parameter, which specifies the new ID of a customer as the **Change ID** action has.

When you add a new entity to a contract, you should specify the type of the entity, which can be one of the following:

- *Top-Level*: Entities of this type are the main entities of the contract. A top-level entity usually corresponds to an Acumatica ERP form. For example, the *Warehouse* entity of the contract of the system endpoint is a top-level entity that corresponds to the *Warehouses* form.

- *Detail*: Detail entities correspond to the detail lines of a master-detail form. A detail entity exists only as a part of a top-level entity. For example, the top-level entity *SalesOrder* of the contract of the system endpoint contains the detail entity *SalesOrderDetail*, which corresponds to a detail line on the **Document Details** tab of the *Sales Orders* (SO301000) form, as shown in the following screenshot.

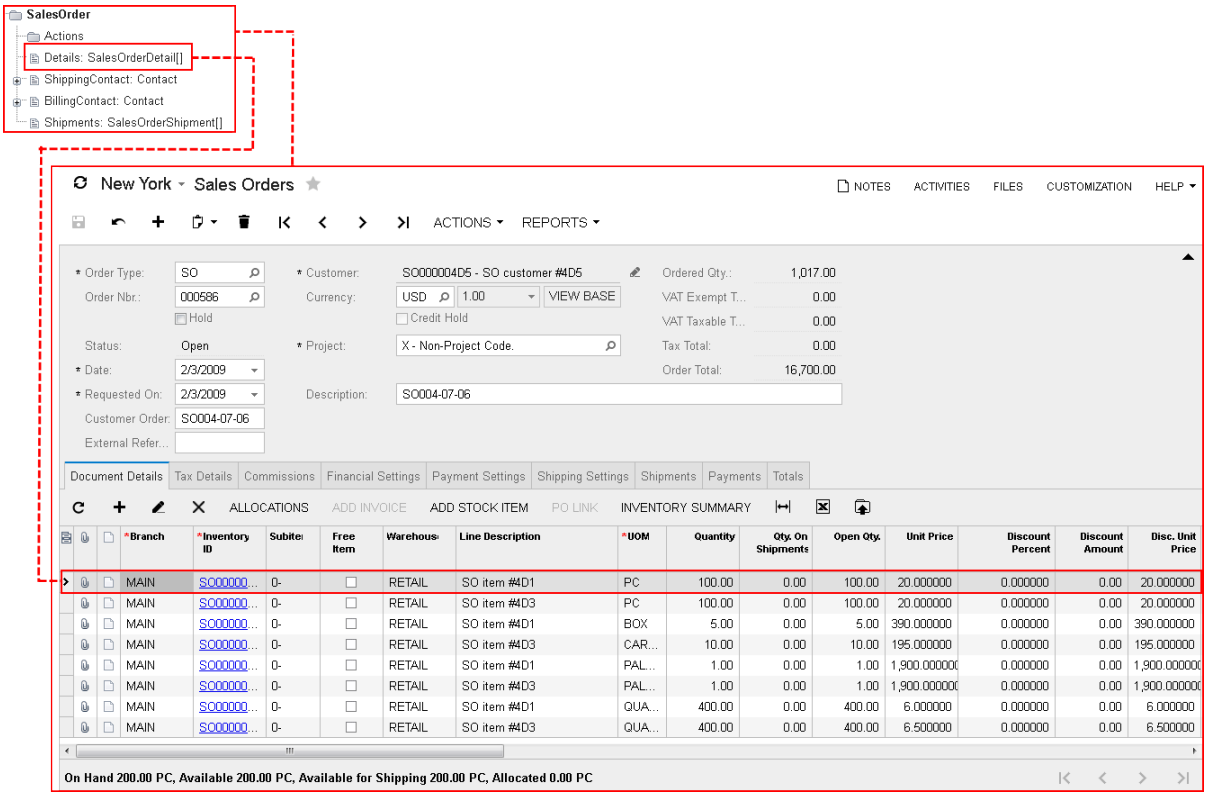


Figure: Detail entity

- *Linked*: Linked entities are supplementary entities of a contract. A linked entity usually corresponds to a part of an Acumatica ERP form and is related to one top-level entity of the contract or multiple such entities. For example, the top-level entity *Contact* of the contract of the system endpoint contains the linked entity *Address*, which corresponds to the **Address** group of fields on the **Details** tab of the *Contacts* (CR302000) form, as shown in the following screenshot.

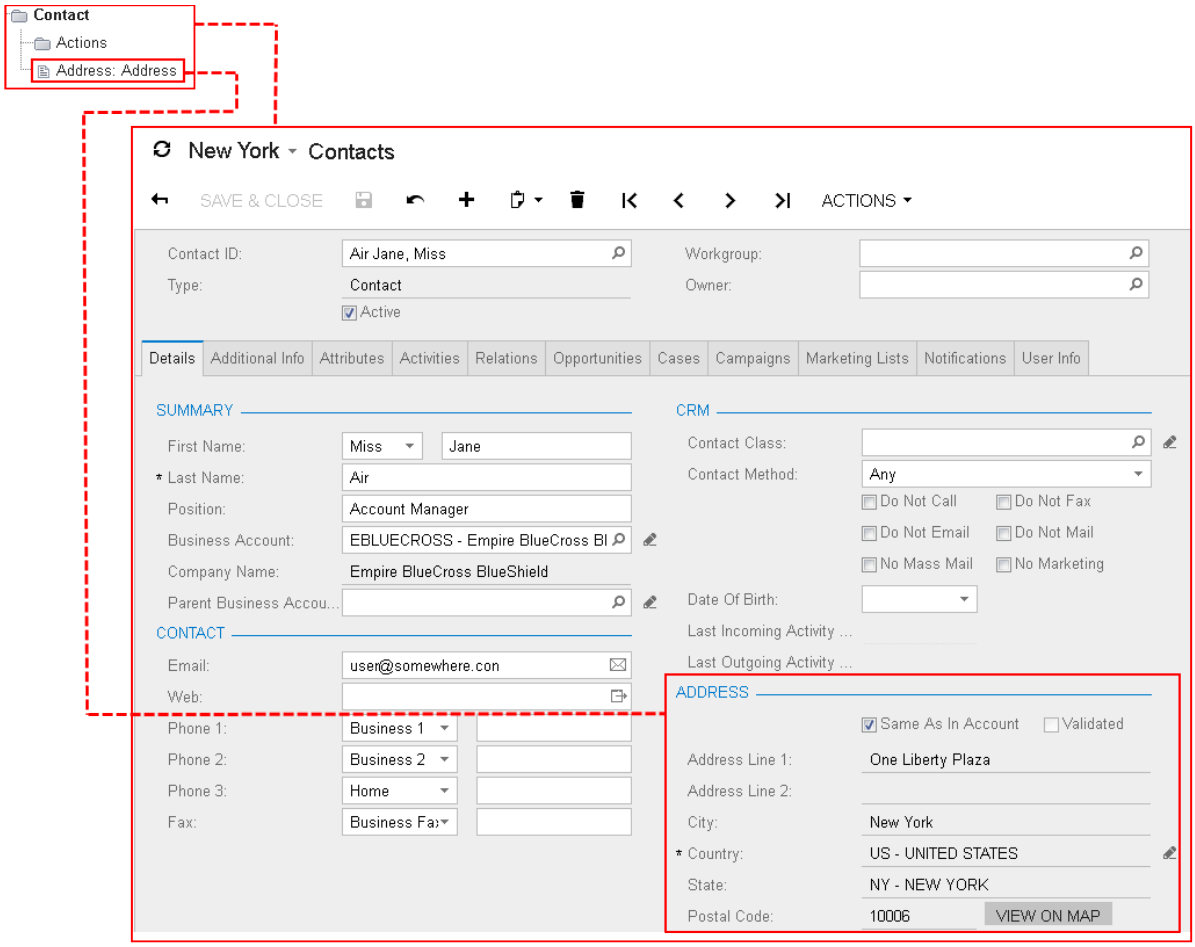


Figure: Linked entity

Custom Fields

Starting from Version 2 of the system contract of the contract-based web services application programming interface (API), you can work with the values of the custom fields that are not included in the entity definition. That is, custom fields can correspond to the predefined elements on an Acumatica ERP form that are not included in the entity definition, the elements that were added to the Acumatica ERP form in a customization project, or the user-defined fields.

To work with the needed custom field, you need to know the name of the data view that contains the corresponding custom element and the name of the field, which are described in detail below.

Field Name and View Name

A field name is the internal name of a particular element of an Acumatica ERP form. A view name is the name of the data view to which a particular element belongs. For example, the

Posting Class element on the **General Settings** tab of the *Stock Items* (IN202500) form has the `PostClassID` field name and belong to the `ItemSettings` data view.

To find out the field name and view name, on the title bar of the form, you click **Customization > Inspect Element** and click the needed element on the form. In the **Element Properties** dialog box, which opens, you find the field name in the **Data Field** element and the view name in the **View Name** element, as shown in the following screenshot.

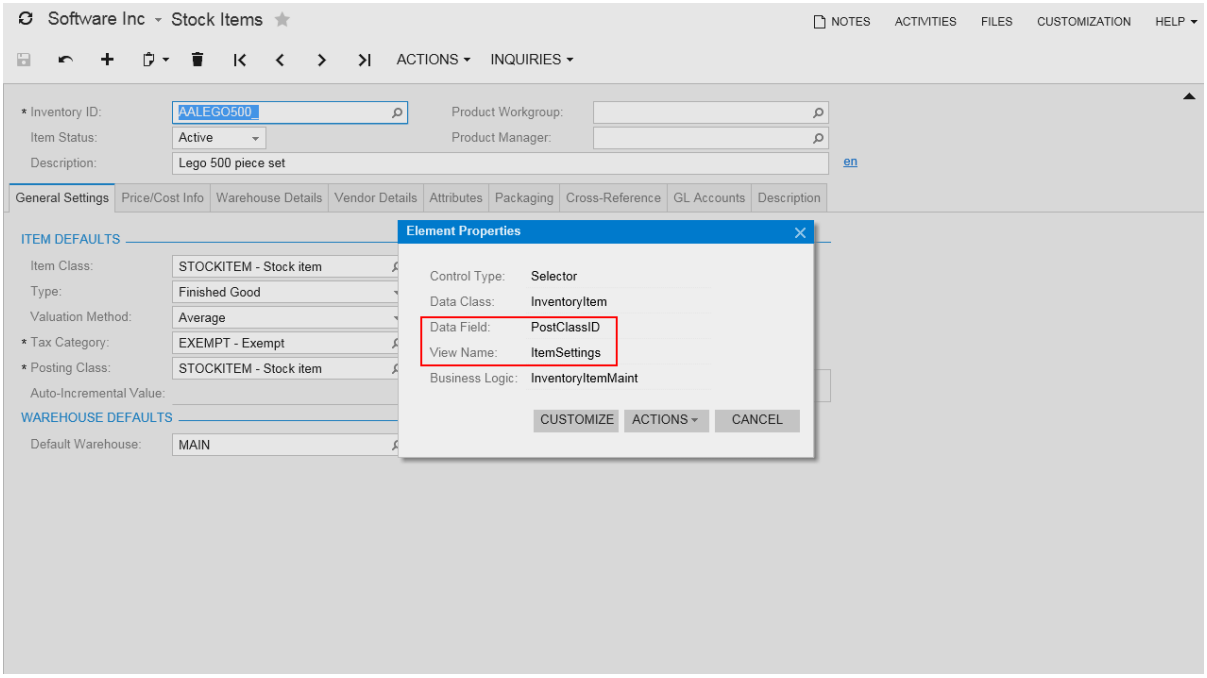


Figure: Field name and view name

In the contract-based REST API, you can also find out the field name and the view name through the special URL. For details on the URL and the HTTP method, see *Retrieval of the Schema of Custom Fields*.

In the contract-based SOAP API, you can find out the field name and the view name in code by using the `GetCustomFieldSchema()` method. For details on the method, see *GetCustomFieldSchema() Method*.

Field Name and View Name of a User-Defined Field

For any user-defined field, the view name is `Document`. The field name is `Attribute<AttributeID>`, where you replace `<AttributeID>` with the ID of the attribute that corresponds to the user-defined field.

For example, suppose that on the *Sales Orders* (SO301000) form, you have added a user-defined field for the *OPERATSYST* attribute. You work with this user-defined field by using the `Document` view name and the `AttributeOPERATSYST` field name.

Use of Custom Fields

For details on retrieving the values of custom fields by using the contract-based REST API, see the description of the *$custom* parameter in *Parameters for Retrieving Records*. For details on specifying the values of custom fields, see *Representation of a Record in JSON Format*.

For details on working with custom elements through the contract-based SOAP API, see *CustomFields Property*.

Custom Endpoints and Endpoint Extensions

If the API provided by the system endpoint of Acumatica ERP is not sufficient for the requirements of your application, you can create a custom endpoint from scratch or by extending an existing endpoint.

An Extension of an Existing Endpoint

If you are creating an endpoint as an extension of an existing endpoint, for the API elements that were inherited from the base endpoint, you cannot edit the names and types of the entities and fields, and the names, types, and parameters of the actions. In the contract of the new endpoint, you can add new top-level entities, new fields or entities to any entity, and new actions. Then you can use both the API that you added to the contract of the endpoint and the API of the base endpoint in your application. For information on how to extend an existing endpoint, see *To Extend an Existing Endpoint*.

The new endpoint that was created as an extension of an existing endpoint has the version of the contract of the base endpoint; that is, the API methods for working with entities are the same for the base endpoint and the new endpoint. See *Contract-Based REST API Reference* and *Contract-Based SOAP API Reference* for the description of the API methods of the needed contract version.

An Endpoint Created from Scratch

If you are creating an endpoint from scratch, you should add the needed elements of the API to the contract. Then you can use these API elements in your application. For information on how to create an endpoint from scratch, see *To Create a Custom Endpoint*.

The new endpoint that is created from scratch always has the latest version of the contract. For the description of the API methods for working with entities that are available in the latest version of the contract, see *Contract-Based REST API Reference* and *Contract-Based SOAP API Reference*.

Related Links

- *Comparison of Contract Versions*

Naming Rules for Endpoints

When you create a custom endpoint on the *Web Service Endpoints* (SM207060) form (either from scratch or by extending a system endpoint), for the names of the entities, fields, actions, and action parameters of the endpoint, and the endpoint name and version, you should make sure to adhere to the following rules:

- The name of the endpoint can contain only English letters, digits, underscores, and periods, and cannot start with a digit.
- The version of the endpoint can contain only English letters, digits, underscores, and periods.
- The name of the entity, field, action, or action parameter can contain only English letters, digits, and underscores, and cannot start with a digit.
- The name of the field cannot match any of the following reserved names:
  - `ID`
  - `RowNumber`
  - `Note`
  - `Delete`
  - `CustomFields`
  - `ReturnBehavior`
  - `Entity`
  - `Action`
- The name of the field must be unique among the names of the fields of the entity.
- The name of the parameter must be unique among the names of the parameters of the action.
- The name of the entity or action must be unique among the names of the entities and actions of the endpoint.

The system checks whether the names used in the endpoint satisfy these rules each time you enter the name of a new entity, field, action, or action parameter. You can also validate the endpoint manually, as described in *To Validate an Endpoint*.

Related Links

- *To Validate an Endpoint*

Comparison of Contract Versions

Acumatica ERP 2019 R2 supports two versions of system contracts. In this topic, you can learn the main differences between the contract versions.

Contract Version 1 is not supported starting from Acumatica ERP 2018 R2.

Table: Comparison of Contract Versions

Characteristic	Contract Version 3	Contract Version 2
The REST API is supported for the endpoints with this contract version.	Yes	Yes
The SOAP API is supported for the endpoints with this contract version.	Yes	Yes
You can specify particular fields of the entity to be returned from the system.	Yes	Yes
By default, the system returns all fields of the entity (including fields of the linked and detail entities defined within the entity).	No	For the SOAP API: Yes For the REST API: No
By default, the system returns only the fields of the entity itself (without the fields of the linked and detail entities defined within the entity).	Yes	For the SOAP API: No For the REST API: Yes
Through the endpoint, you can work with the elements that were added to the Acumatica ERP form in a customization project.	Yes	Yes
Through the endpoint, you can work with the predefined elements on an Acumatica ERP form that are not included in the entity definition.	Yes	Yes
When optimization for speed of the retrieval of the list of records fails, the system behaves as follows.	The system returns an error.	The system retrieves data in an unoptimized way (slow).
Custom endpoints created from scratch have this contract version.	Yes	No
The system endpoint that has this contract version is included in Acumatica ERP 2019 R2.	Yes (Endpoint Versions *Default/18.200.001*	Yes (Endpoint Version *Default/6.00.001*)

Characteristic	Contract Version 3	Contract Version 2
	and *Default/17.200.001*)	

Related Links

- *Endpoints and Contracts*

Comparison of System Endpoints

Acumatica ERP 2019 R2 supports three system endpoints. In this topic, you can learn about the differences between these endpoints.

Contract Versions of the Endpoints

The following table specifies the contract versions of the endpoints. For details about the differences between the contract versions, see *Comparison of Contract Versions*.

Endpoint	Contract Version
Default/18.200.001	3
Default/17.200.001	3
Default/6.00.001	2

Changes to the Entities, Fields, and Actions of the Default/18.200.001 Endpoint as Compared to the Default/17.200.001 Endpoint

The following tables contain the new, modified, or removed elements of the *Default/18.200.001* endpoint as compared to the *Default/17.200.001* endpoint.

Table: New Entities

The entities listed in the following table (except `ProFormaInvoice`) can be created, retrieved, updated, and deleted through the standard API methods. `ProFormaInvoice` can be created only by the invocation of the `RunProjectBilling` action of `Project`.

Entity	Related Form Name and ID
`AccountGroup`	*Account Groups* (PM201000)
`Activity`	*Activity* (CR306010)
`AllocationRule`	*Allocation Rules* (PM207500)
`ChangeOrder`	*Change Orders* (PM308000)
`ChangeOrderClass`	*Change Order Classes* (PM203000)
`CommonTask`	*Common Tasks* (PM208030)
`CompanyFinancialPeriod`	*Company Financial Calendar* (GL201100)

Entity	Related Form Name and ID
CostCode	*Cost Codes* (PM209500)
ExpenseClaim	*Expense Claim* (EP301000)
ExpenseReceipt	*Expense Receipt* (EP301020)
ExternalCommitment	*External Commitments* (PM209000)
LaborCostRate	*Labor Cost Rates* (PM209900)
ManageFinancialPeriods	*Manage Financial Periods* (GL503000)
ProFormaInvoice	*Pro Forma Invoices* (PM307000)
Project	*Projects* (PM301000)
ProjectBilling	*Run Project Billing* (PM503000)
ProjectBillingRules	*Billing Rules* (PM207000)
ProjectBudget	*Project Budget* (PM309000)
ProjectTask	*Project Tasks* (PM302000)
ProjectTemplate	*Project Templates* (PM208000)
ProjectTemplateTask	*Project Template Tasks* (PM208010)
TimeEntry	Time Entry (PM209100)
UnionLocal	*Union Locals* (PM209700)
WorkClassCompensationCode	*Work Class Compensation Codes* (PM209800)

Table: Changed Entities

Entity	Related Form Name and ID	Change
Email.TimeActivity	*Email Activity* (CR306015)	The object name has been changed from EmailTimeActivity to TimeActivity.
Employee	*Employees* (EP203000)	The mapping of the entity has been completely changed.

Table: New Fields and Actions

Field or Action Name	Related Form Name and ID
AccountLocation.Address.Validated	*Account Locations* (CR303010)
Bill.Details.CalculateDiscountsOnImport	*Bills and Adjustments* (AP301000)
Bill.Details.CostCode	*Bills and Adjustments* (AP301000)

Field or Action Name	Related Form Name and ID
Bill.Details.InventoryID	*Bills and Adjustments* (AP301000)
Bill.Details.POLine	*Bills and Adjustments* (AP301000)
Bill.Details.POReceiptLine	*Bills and Adjustments* (AP301000)
Bill.Details.POReceiptNbr	*Bills and Adjustments* (AP301000)
Bill.Project	*Bills and Adjustments* (AP301000)
BusinessAccount.Activities.NoteID	*Business Accounts* (CR303000)
BusinessAccount.LastModifiedDateTime	*Business Accounts* (CR303000)
BusinessAccount.MainAddress.Validated	*Business Accounts* (CR303000)
BusinessAccount.ShippingAddress.Validated	*Business Accounts* (CR303000)
Case.Activities.CostCode	*Cases* (CR306000)
Case.Activities.NoteID	*Cases* (CR306000)
CashSale.Details.CostCode	*Cash Sales* (AR304000)
CashSale.Details.ProjectTask	*Cash Sales* (AR304000)
CashSale.Project	*Cash Sales* (AR304000)
Contact.Activities.NoteID	*Contacts* (CR302000)
Contact.Address.Validated	*Contacts* (CR302000)
Contact.LastModifiedDateTime	*Contacts* (CR302000)
Email.TimeActivity.CostCode	*Email Activity* (CR306015)
FinancialPeriod.Details.Status	*Master Financial Calendar* (GL201000)
JournalTransaction.Details.CostCode	*Journal Transactions* (GL301000)
JournalTransaction.Details.IsNonPM	*Journal Transactions* (GL301000)
JournalTransaction.Details.ProjectTransactionID	*Journal Transactions* (GL301000)
InventoryReceipt.Details.CostCode	*Receipts* (IN301000)
InventoryReceipt.Details.Project	*Receipts* (IN301000)
InventoryReceipt.Details.ProjectTask	*Receipts* (IN301000)

Field or Action Name	Related Form Name and ID
Invoice.Details.CalculateDiscountsOnImport	*Invoices and Memos* (AR301000)
Invoice.Details.CostCode	*Invoices and Memos* (AR301000)
Invoice.DiscountDetails	*Invoices and Memos* (AR301000)
Lead.Activities.NoteID	*Leads* (CR301000)
Lead.Address.Validated	*Leads* (CR301000)
Opportunity.Activities.NoteID	*Opportunities* (CR304000)
Opportunity.Address.Validated	*Opportunities* (CR304000)
Payment.PaymentLoadDocuments	*Payments and Applications* (AR302000)
Payment.PaymentLoadOrders	*Payments and Applications* (AR302000)
PhysicalInventoryReview.CreatedDateTime	*Payments and Applications* (AR302000)
ProjectTransaction.Details.Billed	*Project Transactions* (PM304000)
ProjectTransaction.Details.CostCode	*Project Transactions* (PM304000)
ProjectTransaction.Details.ExternalRefNbr	*Project Transactions* (PM304000)
ProjectTransaction.ReleaseTransactions	*Project Transactions* (PM304000)
PurchaseOrder.Branch	*Purchase Orders* (PO301000)
PurchaseOrder.Details.CalculateDiscountsOnImport	*Purchase Orders* (PO301000)
PurchaseOrder.Details.CostCode	*Purchase Orders* (PO301000)
PurchaseOrder.Details.Project	*Purchase Orders* (PO301000)
PurchaseOrder.Details.ProjectTask	*Purchase Orders* (PO301000)
PurchaseOrder.EnterAPBill	*Purchase Orders* (PO301000)
PurchaseOrder.EnterPOReceipt	*Purchase Orders* (PO301000)
PurchaseOrder.LastModifiedDateTime	*Purchase Orders* (PO301000)
PurchaseOrder.Project	*Purchase Orders* (PO301000)
PurchaseOrder.ShippingInstructions.ShipToAddress.Validated	*Purchase Orders* (PO301000)

Field or Action Name	Related Form Name and ID
PurchaseOrder.TaxDetails	*Purchase Orders* (PO301000)
PurchaseOrder.Terms	*Purchase Orders* (PO301000)
PurchaseOrder.VendorTaxZone	*Purchase Orders* (PO301000)
PurchaseReceipt.BillDate	*Purchase Receipts* (PO302000)
PurchaseReceipt.Branch	*Purchase Receipts* (PO302000)
PurchaseReceipt.CreateAPBill	*Purchase Receipts* (PO302000)
PurchaseReceipt.Description	*Purchase Receipts* (PO302000)
PurchaseReceipt.TransferOrderNbr	*Purchase Receipts* (PO302000)
PurchaseReceipt.TransferOrderType	*Purchase Receipts* (PO302000)
PurchaseReceipt.TransferShipmentNbr	*Purchase Receipts* (PO302000)
PurchaseReceipt.UnbilledQuantity	*Purchase Receipts* (PO302000)
PurchaseReceipt.Warehouse	*Purchase Receipts* (PO302000)
SalesInvoice.ApplicationsCreditMemo	*Invoices* (SO303000)
SalesInvoice.ApplicationsInvoice	*Invoices* (SO303000)
SalesInvoice.BillToSettings.BillToAddress.Validated	*Invoices* (SO303000)
SalesInvoice.BillToSettings.CustomerLocation	*Invoices* (SO303000)
SalesInvoice.CashDiscount	*Invoices* (SO303000)
SalesInvoice.Commissions	*Invoices* (SO303000)
SalesInvoice.CreditHold	*Invoices* (SO303000)
SalesInvoice.Currency	*Invoices* (SO303000)
SalesInvoice.Details.CostCode	*Invoices* (SO303000)
SalesInvoice.Details.ProjectTask	*Invoices* (SO303000)
SalesInvoice.Details.CalculateDiscountsOnImport	*Invoices* (SO303000)
SalesInvoice.Details.DiscountAmount	*Invoices* (SO303000)
SalesInvoice.Details.DiscountPercent	*Invoices* (SO303000)
SalesInvoice.Details.ExpirationDate	*Invoices* (SO303000)
SalesInvoice.Details.InventoryDocType	*Invoices* (SO303000)
SalesInvoice.Details.InventoryRefNbr	*Invoices* (SO303000)
SalesInvoice.Details.Location	*Invoices* (SO303000)
SalesInvoice.Details.LotSerialNbr	*Invoices* (SO303000)

Field or Action Name	Related Form Name and ID
SalesInvoice.Details.ManualDiscount	*Invoices* (SO303000)
SalesInvoice.Details.OrderLineNbr	*Invoices* (SO303000)
SalesInvoice.Details.OrigInvLineNbr	*Invoices* (SO303000)
SalesInvoice.Details.OrigInvNbr	*Invoices* (SO303000)
SalesInvoice.Details.OrigInvType	*Invoices* (SO303000)
SalesInvoice.Details.Subitem	*Invoices* (SO303000)
SalesInvoice.Details.TaxCategory	*Invoices* (SO303000)
SalesInvoice.Details.TransactionDescr	*Invoices* (SO303000)
SalesInvoice.Details.WarehouseID	*Invoices* (SO303000)
SalesInvoice.DiscountDetails	*Invoices* (SO303000)
SalesInvoice.FinancialDetails	*Invoices* (SO303000)
SalesInvoice.Project	*Invoices* (SO303000)
SalesInvoice.SalesInvoiceAddOrder	*Invoices* (SO303000)
SalesInvoice.SalesInvoiceAutoApply	*Invoices* (SO303000)
SalesInvoice.TaxDetails	*Invoices* (SO303000)
SalesInvoice.Totals	*Invoices* (SO303000)
SalesInvoice.VATExemptTotal	*Invoices* (SO303000)
SalesInvoice.VATTaxableTotal	*Invoices* (SO303000)
SalesOrder.AutoRecalculateDiscounts	*Sales Orders* (SO301000)
SalesOrder.BillToAddress.Validated	*Sales Orders* (SO301000)
SalesOrder.Details.CalculateDiscountsOnImport	*Sales Orders* (SO301000)
SalesOrder.Details.CostCode	*Sales Orders* (SO301000)
SalesOrder.DisableAutomaticDiscountUpdate	*Sales Orders* (SO301000)
SalesOrder.DiscountDetails.Description	*Sales Orders* (SO301000)
SalesOrder.DiscountDetails.ExternalDiscountCode	*Sales Orders* (SO301000)
SalesOrder.OpenSalesOrder	*Sales Orders* (SO301000)
SalesOrder.PaymentProfileID	*Sales Orders* (SO301000)
SalesOrder.ReleaseFromCreditHoldSalesOrder	*Sales Orders* (SO301000)
SalesOrder.SalesOrderAddInvoice	*Sales Orders* (SO301000)
SalesOrder.SalesOrderAddStockItem	*Sales Orders* (SO301000)
SalesOrder.SalesOrderCreatePurchaseOrder	*Sales Orders* (SO301000)

Field or Action Name	Related Form Name and ID
SalesOrder.SalesOrderCreateReceipt	*Sales Orders* (SO301000)
SalesOrder.SalesOrderCreateShipment	*Sales Orders* (SO301000)
SalesOrder.ShipToAddress.Validated	*Sales Orders* (SO301000)
Shipment.PrepareInvoice	*Shipments* (SO302000)
Shipment.ShipToSettings.ShipToAddress.Validated	*Shipments* (SO302000)
Task.RelatedActivities.NoteID	*Task* (CR306020)
Task.TimeActivity.CostCode	*Task* (CR306020)

Table: Renamed Actions

Action Name in *Default/18.200.001* (Action Name in *Default/17.200.001*)	Related Form Name and ID
Email.CreateContactFromEmail (Email.CreateContactEmail)	*Email Activity* (CR306015)
Email.CreateEventFromEmail (Email.CreateEventEmail)	*Email Activity* (CR306015)
Email.CreateLeadFromEmail (Email.CreateLeadEmail)	*Email Activity* (CR306015)
Email.CreateOpportunityFromEmail (Email.CreateOpportunityEmail)	*Email Activity* (CR306015)
Email.CreateExpenseReceiptFromEmail (Email.CreateExpenseReceiptEmail)	*Email Activity* (CR306015)
Email.CreateCaseFromEmail (Email.CreateCaseEmail)	*Email Activity* (CR306015)
Email.CreateTaskFromEmail (Email.CreateTaskEmail)	*Email Activity* (CR306015)
Email.LinkEntityToEmail (Email.SelectRelatedEntityEmail)	*Email Activity* (CR306015)
Event.LinkEntityToEvent (Event.SelectRelatedEntityEvent)	*Event* (CR306030)
Task.LinkEntityToTask (Task.SelectRelatedEntityTask)	*Task* (CR306020)

Table: Removed Fields and Actions

Field or Action Name	Related Form Name and ID
Email.SelectSourceEmail	*Email Activity* (CR306015)
FinancialPeriod.Details.Active	*Master Financial Calendar* (GL201000)
FinancialPeriod.Details.ClosedInGL	*Master Financial Calendar* (GL201000)
FinancialPeriod.Details.ClosedInPR	*Master Financial Calendar* (GL201000)
PurchaseReceipt.Details.Project	*Purchase Receipts* (PO302000)

Field or Action Name	Related Form Name and ID
`PurchaseReceipt.Details.ProjectTask`	*Purchase Receipts* (PO302000)

To Create a Custom Endpoint

You use the *Web Service Endpoints* (SM207060) form to create a custom endpoint.

If you need to use a custom endpoint, you can either create an endpoint from scratch or extend an existing endpoint with the needed API. This procedure describes how to create a custom endpoint from scratch. To learn how to extend an existing endpoint, see *To Extend an Existing Endpoint*.

To Create an Endpoint from Scratch

1. Open the *Web Service Endpoints* (SM207060) form.

   To open any form, you can navigate to it or search for it (by its name or by its form ID without periods). For more information about search capabilities, see *Search*.

2. In the **Endpoint Name** box, type the name of the new endpoint.

   For details on the characters that can be used in the endpoint name and version, see *Naming Rules for Endpoints*.

3. In the **Endpoint Version** box, type the version of the new endpoint.
4. Add the needed entities, fields, and actions to the contract of the created endpoint, as described in the sections below.
5. Click **Save** on the form toolbar.

To Add a Top-Level Entity to the Contract of the Endpoint

1. In the **Endpoint Name** box, select the name of the endpoint to which you want to add an entity.
2. In the **Endpoint Version** box, select the version of the endpoint to which you want to add an entity.
3. In the left pane, select the *Endpoint* node.
4. On the toolbar of the left pane, click **Insert**, and in the **Create Entity** dialog box, specify the values as follows, and click **OK**:

   a. In the **Object Name** box, type the name of the entity. This is the name of the API object that you will use in the code of your application to work with the entity.

For details on the characters that can be used in the entity names, see *Naming Rules for Endpoints*.

b. In the **Screen ID** lookup box, select the form to which the entity should correspond.

5. Add the needed fields, actions, or nested entities to the entity, as described in the sections below.

To Add a Linked or Detail Entity to Another Entity

1. In the **Endpoint Name** box, select the name of the endpoint to which you want to add an entity.

2. In the **Endpoint Version** box, select the version of the endpoint to which you want to add an entity.

3. In the left pane, select the entity node to which you want to add a linked or detail entity.

4. On the toolbar of the left pane, click **Insert**.

5. In the **Field Name** box of the **Create Entity** dialog box, which opens, type the name of the field that should be used to access the nested entity, and specify the values of other elements in one of the following ways:

- If you want to insert an entity that already exists in the contract, select the **Use Existing Entity** check box, and select the needed entity in the **Entity Type** box.
- If you want to insert a new entity, in the **Object Name** box, type the name of the entity, and in the **Object Type** box, select the type of the entity: *Top-Level*, *Linked*, or *Detail*. If you have selected the top-level entity to be inserted, in the **Screen ID** lookup box, specify the form to which the entity should correspond.

For details on the characters that can be used in the entity names, see *Naming Rules for Endpoints*.

6. Click **OK**. The new entity appears in the contract.

7. Add fields to the created entity, as described in the following section.

To Add Fields to an Entity

1. In the **Endpoint Name** box, select the name of the endpoint to which you want to add an entity.

2. In the **Endpoint Version** box, select the version of the endpoint to which you want to add an entity.

3. In the left pane, select the entity node to which you want to add fields.

4. On the **Fields** tab of the right pane, do one of the following:

- Click **Populate** on the tab toolbar. In the **Populate Fields** dialog box, select the Acumatica ERP object whose fields you want to include in the entity and the fields that you want to include, and click **OK**. The selected fields are added to the contract.
- Click **Add Row** on the tab toolbar; then type the name of the new field in the **Field Name** column of the added row, select the Acumatica ERP object whose field you want to include in the entity in the **Mapped Object** column, and select the field that you want to include in the **Mapped Field** column.

- For some fields to be included in the entity, the corresponding Acumatica ERP feature or features must be enabled on the *Enable/Disable Features* (CS100000) form. For information on Acumatica ERP basic functionality and add-on features, see *Overview of the Acumatica ERP Features*.
- For details on the characters that can be used in the field names, see *Naming Rules for Endpoints*.

5. Click **Save** on the form toolbar.

To Add an Action to an Entity

1. In the **Endpoint Name** box, select the name of the endpoint to which you want to add an entity.
2. In the **Endpoint Version** box, select the version of the endpoint to which you want to add an entity.
3. On the left pane, select the *Actions* node in the needed entity.
4. On the toolbar, click **Insert**.
5. In the **Create Action** dialog box, which opens, select the needed Acumatica ERP action, type the name that should be used to invoke this action through the API, and click **OK**. The new action is added to the contract.

For details on the characters that can be used in the action names, see *Naming Rules for Endpoints*.

6. Click **Save** on the form toolbar.

To Extend an Existing Endpoint

You use the *Web Service Endpoints* (SM207060) form to create an endpoint as an extension of an existing endpoint.

You may need to create an extension of an endpoint if you want to use the entities that are defined in the contract of the existing endpoint but you also need some additional entities, fields, and actions in the contract. For example, the contract of the system endpoint with the name *Default* and Version *6.00.001* contains the `Address` entity, which includes the following fields: `AddressLine1`, `AddressLine2`, `City`, `Country`, `PostalCode`, and `State`. Suppose that

you want to add the new `GPSCoordinates` field to the `Address` entity of the contract and use it with other API of the contract. You cannot edit the contract of the system endpoint; instead, you should create an endpoint that is based on this system endpoint, and add the new `GPSCoordinates` field to the `Address` entity of the contract of the new endpoint.

This procedure describes how to create an endpoint that is based on an existing endpoint.

To Extend an Existing Endpoint

1. Open the *Web Service Endpoints* (SM207060) form.

   To open any form, you can navigate to it or search for it (by its name or by its form ID without periods). For more information about search capabilities, see *Search*.

2. Select the endpoint that you want the new endpoint to be based on as follows:

   a. Select the name of the base endpoint in the **Endpoint Name** box.

   b. Select the version of the base endpoint in the **Endpoint Version** box.

3. Click **Extend Endpoint** on the form toolbar.

4. In the **Extend Current Endpoint** dialog box, which opens, make sure the correct name and version of the base endpoint are specified in the **Base Endpoint Name** and **Base Endpoint Version** boxes. Specify the name of the new endpoint in the **Endpoint Name** box and the version of the new endpoint in the **Endpoint Version** box and click **OK**.

   For details on the characters that can be used in the endpoint name and version, see *Naming Rules for Endpoints*.

   The new endpoint with the name and version you specify appears on the form. On the left pane of the form, you can see the list of entities that were inherited from the base endpoint.

5. Add the needed entities, fields, and actions to the contract of the created endpoint, as described in *To Create a Custom Endpoint*, or extend the entities inherited from the base endpoint, as described in *To Extend an Existing Entity*.

6. Click **Save** on the form toolbar.

To Extend an Existing Entity

1. Select the extended endpoint in which you want to extend an entity inherited from the base endpoint as follows:

   a. In the **Endpoint Name** box, select the name of the extended endpoint.

   b. In the **Endpoint Version** box, select the version of the extended endpoint.

2. In the left pane, select the entity inherited from the base endpoint to which you want to add new fields.

3. On the toolbar of the **Fields** tab in the right pane, click **Extend Endpoint**.

4. Use the **Add Row**, **Delete Row**, and **Populate** buttons, which have become available on the tab toolbar, to add and delete fields of the entity. For more details, see *To Add Fields to an Entity*.

5. Click **Save** on the form toolbar.

To Validate an Endpoint

You use the *Web Service Endpoints* (SM207060) form to validate an endpoint, an entity, or an action. During this validation, the system makes sure the following criteria are met for the elements of the endpoint, entity, or action:

- The names of the elements satisfy the naming rules. For details on these rules, see *Naming Rules for Endpoints*.
- The elements are mapped to objects, fields, and actions that exist in the system.

The validation of the name of a new entity, field, action, or action parameter is performed automatically once you have entered the name on the form. You can validate an endpoint, entity, or action manually, as described in this topic.

To Validate an Endpoint

1. Open the *Web Service Endpoints* (SM207060) form.

   To open any form, you can navigate to it or search for it (by its name or by its form ID without periods). For more information about search capabilities, see *Search*.

2. Select the endpoint that you want to validate as follows:

   a. In the **Endpoint Name** box, select the name of the endpoint.

   b. In the **Endpoint Version** box, select the version of the endpoint.

3. On the form toolbar, click **Validate Endpoint**. The long-running validation operation starts.

   Once the validation is finished, the system displays a message with results of the validation. If the validation has failed, the error message contains the names of all fields that caused the error.

4. If any errors occur, correct the endpoint accordingly.

To Validate an Entity

1. Open the *Web Service Endpoints* (SM207060) form.

To open any form, you can navigate to it or search for it (by its name or by its form ID without periods). For more information about search capabilities, see *Search*.

2. Select the endpoint that contains the entity that you want to validate as follows:
   a. In the **Endpoint Name** box, select the name of the endpoint.
   b. In the **Endpoint Version** box, select the version of the endpoint.
3. In the left pane, click the entity that you want to validate.
4. On the toolbar of the **Fields** tab of the right pane, click **Validate Entity**.

   Once the validation is finished, the system displays a message with results of the validation. If the validation has failed, the error message contains the names of all fields that caused the error.
5. If any errors occur, correct the entity accordingly.

To Validate an Action

1. Open the *Web Service Endpoints* (SM207060) form.

   To open any form, you can navigate to it or search for it (by its name or by its form ID without periods). For more information about search capabilities, see *Search*.
2. Select the endpoint that contains the action that you want to validate as follows:
   a. In the **Endpoint Name** box, select the name of the endpoint.
   b. In the **Endpoint Version** box, select the version of the endpoint.
3. In the left pane, click the action that you want to validate.
4. On the toolbar of the **Parameters** tab of the right pane, click **Validate Action**.

   Once the validation is finished, the system displays a message with results of the validation. If the validation has failed, the error message contains the names of all fields that caused the error.
5. If any errors occur, correct the action accordingly.

Working with the Contract-Based REST API

The contract-based representational state transfer (REST) application programming interface (API) of Acumatica ERP provides the REST interface of the Acumatica ERP contract-based web services through which external systems can get data records from Acumatica ERP, process these records, and save new or updated records to Acumatica ERP.

This chapter includes the topics that are specific to the contract-based REST API. For general information on the contract-based web services, see *Configuring the Contract-Based REST and SOAP API*. You can find examples of how to use the contract-based SOAP API in the *I210 Integration: Contract-Based Web Services* training course and in *Contract-Based API Examples*. For the API reference, see *Contract-Based REST API Reference*.

In This Chapter

- *Representation of a Record in JSON Format*
- *Signing In to the Service*
- *Signing out from the Service*
- *Creation of a Record*
- *Update of a Record*
- *Retrieval of a Record by Key Fields*
- *Retrieval of a Record by ID*
- *Retrieval of Records by Conditions*
- *Retrieval of Data from an Inquiry Form*
- *Parameters for Retrieving Records*
- *Removal of a Record*
- *Execution of an Action*
- *Attachment of a File to a Record*
- *Retrieval of a File Attached to a Record*
- *Retrieval of the Schema of Custom Fields*
- *Retrieval of the Acumatica ERP Version and the List of Endpoints*
- *Multi-Language Fields*

Representation of a Record in JSON Format

By using the contract-based REST API, you obtain existing records from Acumatica ERP, create new records, update, and delete them. You work with the records in Acumatica ERP by using the entities that are defined in the contract of the endpoint that you use to access the service. You pass records to and receive them from the contract-based REST API in JavaScript object notation (JSON) format. JSON is a text format for transmitting data objects that consist of key-value pairs.

To represent a record in JSON format, you use the rules that are described in the following sections. You do not need to specify the values of all fields of an entity; you can specify the values of only the needed fields.

System Fields

You specify the value of a system field (such as `ID`, `RowNumber`, and `Note`) of an entity in the following format.

```
<Field name> : <Value>
```

For example, if you need to specify the note *Imported* for an entity, you use the following string.

```
"Note" : "Imported"
```

General Fields

You specify the value of a general field (that is, a field that is not a system field) of an entity in the following format.

```
<Field name> : {value : <Value>}
```

For example, if you need to specify *JOHNGOOD* as the customer ID of a customer record, you use the following string.

```
"CustomerID" : {value : "JOHNGOOD"}
```

Linked Entities

You specify the values of the fields of a linked entity in the following format.

```
<Field name> :
  {
    <List of fields of the linked entity with values
  }
```

For example, if you need to specify the values of an email address and the address of a customer main contact, you use the following string.

```
"MainContact" :
```

```
    {
      "Email" : {value : "demo@gmail.com" },
      "Address" :
        {
          "AddressLine1" : {value : "4030 Lake Washington Blvd NE" },
          "AddressLine2" : {value : "Suite 100" },
          "City" : {value : "Kirkland" },
          "State" : {value : "WA" },
          "PostalCode" : {value : "98033" }
        }
    }
```

Detail Entities

You specify the values of the fields of a detail entity in the following format.

```
<Field name> :
  [
    {
      <List of fields of the detail entity with the values}
    },
    {
      <List of fields of the detail entity with the values>
    },
    …
  ]
```

For example, if you need to specify the values of two detail lines of a sales order, you use the following string.

```
"Details" : [
  {
    "InventoryID" : {value: "AALEGO500"},
    "Quantity" : {value: 10},
    "UOM" : {value: "PIECE"},
  },
  {
    "InventoryID" : {value: "CONGRILL"},
    "Quantity" : {value: 1},
    "UOM" : {value: "PIECE"},
  }
]
```

Custom Fields

You specify the values of the custom fields (that is, the fields that are not included in the contract of the endpoint) in the following format.

```
"custom" :
  {
    <View name> :
     {
```

```
      <Field name> :
        {
          "type" : <value>,
          "value" : <value>
        }
    }
}
```

You use this block in the JSON representation of the entity (top-level, detail, or linked) that contains this custom field.

For details on how to find out the field name and the name of the data view, see *Custom Fields*.

For example, suppose that you added the **Personal ID** element to the **Main Contact** area of the *Customers* (AR303000) form in a customization project. The `Contact` entity, which is available through the `MainContact` property of the `Customer` entity, contains the **Personal ID** custom element. This element has the `UsrPersonalID` field name and belongs to the `DefContact` data view. The type of the element depends on the contract version (in Contract Version 2, `String`; in Contract Version 3, `CustomStringField`). Therefore, to specify the value *AB123456* of the **Personal ID** custom element for the customer with ID *JOHNGOOD* through the REST API, you use one of the following strings depending on the contract version of the endpoint:

- For Contract Version 2

```
{
  "CustomerID" : {value : "JOHNGOOD" } ,
  "MainContact" :
    {
      "custom" :
        {
          "DefContact" :
            {
              "UsrPersonalID" :
                {
                  "type" : "String",
                  "value" : "AB123456"
                }
            }
        }
    }
}
```

- For Contract Version 3

```
{
  "CustomerID" : {value : "JOHNGOOD" } ,
  "MainContact" :
    {
      "custom" :
        {
```

```
          "DefContact" :
            {
              "UsrPersonalID" :
                {
                  "type" : "CustomStringField",
                  "value" : "AB123456"
                }
            }
        }
    }
}
```

Signing In to the Service

Each time your application starts work with the Acumatica ERP contract-based REST service, you have to sign in to Acumatica ERP. To sign in to Acumatica ERP, you access the needed URL address with the `POST` HTTP method and pass the credentials in the request body. See details on the URL, parameters, HTTP method, and response format in the following sections.

Instead of directly signing in to Acumatica ERP, your application can use the OAuth 2.0 authorization. For details about OAuth 2.0, see *Authorizing Client Applications to Work with Acumatica ERP*.

URL

When you need to sign in to Acumatica ERP, you use the following URL.

```
http://<Acumatica ERP URL>/entity/auth/login
```

You replace `<Acumatica ERP URL>` with the URL of your Acumatica ERP instance.

For example, suppose that you want to sign in to a local Acumatica ERP instance with the name *AcumaticaDB*. You should use the following URL: `http://localhost/AcumaticaDB/entity/auth/login`.

Parameters

You use no parameters when you sign in to Acumatica ERP.

HTTP Method

You use the `POST` HTTP method and pass the credentials for accessing Acumatica ERP in JSON format, as shown in the following example.

```
{
  "name" : "admin",
  "password" : "123",
  "company" : "MyStore",
  "branch" : "MYSTORE",
  "locale" : "en-US"
```

```
}
```

You specify the values of the parameters as follows:

- *name*: The username that the application should use to sign in to Acumatica ERP, such as `"admin"`.
- *password*: The password for the username, such as `"123"`.
- *company*: The name of the tenant to which the application should sign in, such as `"MyStore"`. You can view the name that should be used for the tenant in the **Login Name** box of the *Tenants* (SM203520) form.
- *branch*: The ID of the branch to which the application should sign in. You can view the ID of the branch in the **Branch ID** box of the *Branches* (CS102000) form.
- *locale*: The locale that should be used in Acumatica ERP. You should specify the locale in the `System.Globalization.CultureInfo` format converted to `string`, as with `"EN-US"`.

This parameter has been developed for future use. You do not need to set its value.

Response

The response of a successful method call is *204 No Content*.

Example

The following code shows an example of a class that implements a sign in to Acumatica ERP through the REST application programming interface (API).

```
public class RestService: IDisposable
{
    private readonly HttpClient _httpClient;

    private readonly string _acumaticaBaseUrl;

    public RestService(
      string acumaticaBaseUrl, string userName, string password,
      string company, string branch, string locale)
    {
        _acumaticaBaseUrl = acumaticaBaseUrl;
        _httpClient = new HttpClient(
            new HttpClientHandler
            {
                UseCookies = true,
                CookieContainer = new CookieContainer()
            })
        {
            BaseAddress = new Uri(acumaticaBaseUrl +
              "/entity/Default/18.200.001/"),
            DefaultRequestHeaders =
```

```
            {
                Accept = {MediaTypeWithQualityHeaderValue.Parse("text/json")}
            }
        };

        //Log in to Acumatica ERP
        _httpClient.PostAsJsonAsync(
          acumaticaBaseUrl + "/entity/auth/login", new
            {
                name = userName,
                password = password,
                company = company,
                branch = branch,
                locale = locale
            }).Result
            .EnsureSuccessStatusCode();
    }

    void IDisposable.Dispose()
    {
        _httpClient.PostAsync(_acumaticaBaseUrl + "/entity/auth/logout",
          new ByteArrayContent(new byte[0])).Wait();
        _httpClient.Dispose();
    }
}
```

The following code signs in to Acumatica ERP when an instance of the `RestService` class, which is defined in the code fragment above, is created.

```
RestService rs = new RestService(
            Properties.Settings.Default.AcumaticaBaseUrl,
            Properties.Settings.Default.UserName,
            Properties.Settings.Default.Password,
            Properties.Settings.Default.Company,
            Properties.Settings.Default.Branch,
            Properties.Settings.Default.Locale
        );
```

Usage Notes

For each attempt to sign in to Acumatica ERP as described in this topic, you must implement the signing out from the service after you finish your work with Acumatica ERP to close the session. For details about the signing out, see *Signing out from the Service*.

You should also take into account Acumatica ERP license API limits. For details, see *License Restrictions for API Users*.

Signing out from the Service

Each time your application finishes work with the Acumatica ERP contract-based REST service, you have to sign out from Acumatica ERP. To sign out from Acumatica ERP, you access the needed URL address with the `POST` HTTP method and pass the credentials in the request body. See the following sections for details on the URL, parameters, HTTP method, and response format.

URL

When you need to sign out from Acumatica ERP, you use the following URL.

```
http://<Acumatica ERP URL>/entity/auth/logout
```

You replace `<Acumatica ERP URL>` with the URL of your Acumatica ERP instance.

For example, suppose that you want to sign out from a local Acumatica ERP instance with the name *AcumaticaDB*. You should use the following URL: `http://localhost/AcumaticaDB/entity/auth/logout`.

Parameters

You use no parameters when you sign out from Acumatica ERP.

HTTP Method

You use the `POST` HTTP method to sign out from Acumatica ERP.

Response

The response of a successful method call is *204 No Content*.

Example

The following code shows an example of a class that implements signing out from Acumatica ERP through the REST application programming interface (API). The signing out is performed each time an instance of the `RestService` class is released.

```
public class RestService: IDisposable
{
    private readonly HttpClient _httpClient;

    private readonly string _acumaticaBaseUrl;

    public RestService(
      string acumaticaBaseUrl, string userName, string password,
      string company, string branch)
    {
        _acumaticaBaseUrl = acumaticaBaseUrl;
        _httpClient = new HttpClient(
```

```
            new HttpClientHandler
            {
                UseCookies = true,
                CookieContainer = new CookieContainer()
            })
        {
            BaseAddress = new Uri(acumaticaBaseUrl +
              "/entity/Default/18.200.001/"),
            DefaultRequestHeaders =
            {
                Accept = {MediaTypeWithQualityHeaderValue.Parse("text/json")}
            }
        };

        _httpClient.PostAsJsonAsync(
          acumaticaBaseUrl + "/entity/auth/login", new
            {
                name = userName,
                password = password,
                company = company,
                branch = branch
            }).Result
            .EnsureSuccessStatusCode();
    }

    //Log out from Acumatica ERP
    void IDisposable.Dispose()
    {
        _httpClient.PostAsync(_acumaticaBaseUrl + "/entity/auth/logout",
          new ByteArrayContent(new byte[0])).Wait();
        _httpClient.Dispose();
    }
}
```

Creation of a Record

When you need to create a record by using the contract-based REST API, you access the needed URL address with the `PUT` HTTP method and pass the record representation in JSON format in the request body. See the following sections for details on the URL, parameters, HTTP method, and response format.

URL

If you need to create a record in Acumatica ERP, you use the following URL.

```
http://<Base endpoint URL>/<Top-level entity>
```

The URL has the following components:

- `<Base endpoint URL>` is the URL of the contract-based endpoint through which you are going to work with Acumatica ERP, which has the following format: `http://<Acumatica ERP instance URL>/entity/<Endpoint name>/<Endpoint version>/`.
- `<Top-level entity>` is the name of the entity for which you are going to create a record.

For example, suppose that you want to create a stock item record in a local Acumatica ERP instance with the name *AcumaticaDB* by using the system endpoint with the name *Default* and Version 18.200.001. You should use the following URL to create a record: `http://localhost/AcumaticaDB/entity/Default/18.200.001/StockItem`.

Parameters

You can use the following parameters when you retrieve a record from Acumatica ERP:

- *$expand*: To specify the linked and detail entities to be expanded
- *$select*: To specify the fields of the entity to be returned
- *$custom*: To specify the fields that are not defined in the contract to be returned

For detailed descriptions of the parameters, see *Parameters for Retrieving Records*.

HTTP Method

You use the `PUT` HTTP method and pass a record in JSON format in the request body. You can find details on how to represent a record in JSON format in *Representation of a Record in JSON Format*. See below for an example of a customer record representation in JSON format.

```
{
  "CustomerID" : {value : "JOHNGOOD" } ,
  "CustomerName" : {value : "John Good" },
  "MainContact" :
    {
      "Email" : {value : "demo@gmail.com" },
      "Address" :
        {
          "AddressLine1" : {value : "4030 Lake Washington Blvd NE" },
          "AddressLine2" : {value : "Suite 100" },
          "City" : {value : "Kirkland" },
          "State" : {value : "WA" },
          "PostalCode" : {value : "98033" }
        }
    }
}
```

Response

The response of a successful method call contains the created record in JSON format in the response body. The response includes only the values of the fields of the created record that

were specified during creation of the record or that were specified to be returned by using the parameters of the request.

Example

The following code shows an example of a class that implements the creation of a record in Acumatica ERP through the REST API.

```
public class RestService: IDisposable
{
    private readonly HttpClient _httpClient;

    private readonly string _acumaticaBaseUrl;

    public RestService(
      string acumaticaBaseUrl, string userName, string password,
      string company, string branch)
    {
        _acumaticaBaseUrl = acumaticaBaseUrl;
        _httpClient = new HttpClient(
            new HttpClientHandler
            {
                UseCookies = true,
                CookieContainer = new CookieContainer()
            })
        {
            BaseAddress = new Uri(acumaticaBaseUrl +
              "/entity/Default/18.200.001/"),
            DefaultRequestHeaders =
            {
                Accept = {MediaTypeWithQualityHeaderValue.Parse("text/json")}
            }
        };
        _httpClient.PostAsJsonAsync(
          acumaticaBaseUrl + "/entity/auth/login", new
            {
                name = userName,
                password = password,
                company = company,
                branch = branch
            }).Result
            .EnsureSuccessStatusCode();
    }

    void IDisposable.Dispose()
    {
        _httpClient.PostAsync(_acumaticaBaseUrl + "/entity/auth/logout",
          new ByteArrayContent(new byte[0])).Wait();
        _httpClient.Dispose();
    }
```

```
    //Data submission
    public string Put(string entityName, string parameters, string entity)
    {
        var res = _httpClient
            .PutAsync(_acumaticaBaseUrl + "/entity/Default/18.200.001/" +
                entityName + "?" + parameters,
                new StringContent(entity, Encoding.UTF8, "application/json"))
            .Result
            .EnsureSuccessStatusCode();

        return res.Content.ReadAsStringAsync().Result;
    }
}
```

The following code uses the `RestService.Put()` method, which is defined in the previous code fragment, to create a customer record.

```
public static void CreateCustomer()
{
    //Path to a source text file that contains
    //the new customer record in JSON format
    string entitySource = @"..\..\Input\Customer.txt";

    //Initialize the REST service
    RestService rs = new RestService(
            Properties.Settings.Default.AcumaticaBaseUrl,
            Properties.Settings.Default.UserName,
            Properties.Settings.Default.Password,
            Properties.Settings.Default.Company,
            Properties.Settings.Default.Branch
        );

    using (StreamReader sr = new StreamReader(entitySource))
    {
        //Read the customer record in JSON format from the file
        string entityAsString = sr.ReadToEnd().ToString();

        //Create a customer record
        string customer = rs.Put("Customer", null, entityAsString);
    }
}
```

Update of a Record

When you need to update an existing record by using the contract-based REST API, you access the needed URL with the `PUT` HTTP method and pass the record representation in JSON format in the request body. See the following sections for details on the URL, parameters, HTTP method, and response format.

URL

If you need to update a record in Acumatica ERP, you use the following URL.

```
http://<Base endpoint URL>/<Top-level entity>
```

The URL has the following components:

- `<Base endpoint URL>` is the URL of the contract-based endpoint through which you are going to work with Acumatica ERP, which has the following format: `http://<Acumatica ERP instance URL>/entity/<Endpoint name>/<Endpoint version>/`.
- `<Top-level entity>` is the name of the entity for which you are going to update a record.

For example, suppose that you want to update a stock item record in a local Acumatica ERP instance with the name *AcumaticaDB* by using the system endpoint with the name *Default* and Version 18.200.001. You would use the following URL to update a record: `http://localhost/AcumaticaDB/entity/Default/18.200.001/StockItem`.

Parameters

You can use the following parameters when you are updating a record in Acumatica ERP:

- *$filter*: To specify filtering conditions that identify the record to be updated
- *$expand*: To specify the linked and detail entities to be expanded
- *$select*: To specify the fields of the entity to be returned
- *$custom*: To specify the fields that are not defined in the contract to be returned

For detailed descriptions of the parameters, see *Parameters for Retrieving Records*.

HTTP Method

You use the `PUT` HTTP method and pass a record in JSON format in the request body. You can find details on how to represent a record in JSON format in *Representation of a Record in JSON Format*.

To make it possible for the record to be found by Acumatica ERP, you can use any of the following approaches:

- Specify the values of the key fields in the record representation in JSON format.
- Specify the value of the `ID` property in the record representation in JSON format.
- Specify the filtering conditions that identify the record in the *$filter* parameter of the method. For details on the parameter, see the *Parameters* section in this topic.

If you want to delete a detail line during update, you should specify `true` as the value of the `delete` property of the corresponding detail entity: `"delete" : true`. To identify the detail

line to be deleted, you can specify either the values of the key fields of the detail line or the value of the `ID` property.

Response

The response of a successful method call contains the updated record in JSON format in the response body. The response includes only the values of the fields of the updated record that were specified during the update or that were specified to be returned by using the parameters of the request.

Example

The following code shows an example of a class that implements the update of a record in Acumatica ERP through the REST API.

```
public class RestService: IDisposable
{
    private readonly HttpClient _httpClient;

    private readonly string _acumaticaBaseUrl;

    public RestService(
      string acumaticaBaseUrl, string userName, string password,
      string company, string branch)
    {
        _acumaticaBaseUrl = acumaticaBaseUrl;
        _httpClient = new HttpClient(
            new HttpClientHandler
            {
                UseCookies = true,
                CookieContainer = new CookieContainer()
            })
        {
            BaseAddress = new Uri(acumaticaBaseUrl +
              "/entity/Default/18.200.001/"),
            DefaultRequestHeaders =
            {
                Accept = {MediaTypeWithQualityHeaderValue.Parse("text/json")}
            }
        };
        _httpClient.PostAsJsonAsync(
          acumaticaBaseUrl + "/entity/auth/login", new
            {
                name = userName,
                password = password,
                company = company,
                branch = branch
            }).Result
            .EnsureSuccessStatusCode();
    }

    void IDisposable.Dispose()
```

```
    {
        _httpClient.PostAsync(_acumaticaBaseUrl + "/entity/auth/logout",
          new ByteArrayContent(new byte[0])).Wait();
        _httpClient.Dispose();
    }

    //Data submission
    public string Put(string entityName, string parameters, string entity)
    {
        var res = _httpClient
            .PutAsync(_acumaticaBaseUrl + "/entity/Default/18.200.001/" +
                entityName + "?" + parameters,
                new StringContent(entity, Encoding.UTF8, "application/json"))
            .Result
            .EnsureSuccessStatusCode();

        return res.Content.ReadAsStringAsync().Result;
    }
}
```

The following code uses the `RestService.PutWithFilter()` method, which is defined in the previous code fragment, to update an existing customer record that has the *demo@gmail.com* email address.

```
public static void UpdateCustomer()
{
    //Path to a source text file that contains an updated customer record in JSON format
    string entitySource = @"..\..\Input\Customer_Upd.txt";

    //REST service initialization
    RestService rs = new RestService(
            Properties.Settings.Default.AcumaticaBaseUrl,
            Properties.Settings.Default.UserName,
            Properties.Settings.Default.Password,
            Properties.Settings.Default.Company,
            Properties.Settings.Default.Branch
        );

    using (StreamReader sr = new StreamReader(entitySource))
    {
        //Read customer record in JSON format from file
        string entityAsString = sr.ReadToEnd().ToString();

        //Specify filtering parameters that identify the customer
        string parameter = "$filter=MainContact/Email eq 'demo@gmail.com'";

        //Update the customer record
        string customer = rs.Put("Customer", parameter, entityAsString);
    }
}
```

Retrieval of a Record by Key Fields

To retrieve a record by the values of its key fields from Acumatica ERP by using the contract-based REST API, you access the needed URL with the `GET` HTTP method and specify the fields that should be returned in the parameters of the method. See the following sections for details on the URL, parameters, HTTP method, and response format.

URL

If you need to obtain a particular record with the known key fields, you use the following URL

```
http://<Base endpoint URL>/<Top-level entity>/<Key value 1>/<Key value 2>
```

The URL has the following components:

- `<Base endpoint URL>` is the URL of the contract-based endpoint through which you are going to work with Acumatica ERP, which has the following format: `http://<Acumatica ERP instance URL>/entity/<Endpoint name>/<Endpoint version>/`.
- `<Top-level entity>` is the name of the entity for which you are going to retrieve a record.
- `<Key value 1>` and `<Key value 2>` are the values of the key fields of the record to be retrieved. You use the number and order of key fields as they are defined on the corresponding Acumatica ERP form.

  You can pass the key fields separated by vertical bar (|) instead of slash(/).

For example, suppose that you want to retrieve the sales order with order type *SO* and order number *000123* from a local Acumatica ERP instance with the name *AcumaticaDB* by using the system endpoint with the name *Default* and Version 18.200.001. You should use the following URL to retrieve the sales order: `http://localhost/AcumaticaDB/entity/Default/18.200.001/SalesOrder/SO/000123`.

Parameters

You can use the following parameters when you retrieve a record from Acumatica ERP:

- *$expand*: To specify the linked and detail entities to be expanded
- *$custom*: To specify the fields that are not defined in the contract to be returned
- *$select*: To specify the fields of the entity to be returned

For detailed descriptions of the parameters, see *Parameters for Retrieving Records*.

HTTP Method

You use the `GET` HTTP method to retrieve records.

Response

The response of a successful method call contains the retrieved record in JSON format in the response body. For details on record representation in JSON format, see *Representation of a Record in JSON Format*.

Example

The following code shows an example of a class that implements the retrieval of a record in Acumatica ERP through the REST API.

```
public class RestService: IDisposable
{
    private readonly HttpClient _httpClient;

    private readonly string _acumaticaBaseUrl;

    public RestService(
      string acumaticaBaseUrl, string userName, string password,
      string company, string branch)
    {
        _acumaticaBaseUrl = acumaticaBaseUrl;
        _httpClient = new HttpClient(
            new HttpClientHandler
            {
                UseCookies = true,
                CookieContainer = new CookieContainer()
            })
        {
            BaseAddress = new Uri(acumaticaBaseUrl +
              "/entity/Default/18.200.001/"),
            DefaultRequestHeaders =
            {
                Accept = {MediaTypeWithQualityHeaderValue.Parse("text/json")}
            }
        };
        _httpClient.PostAsJsonAsync(
          acumaticaBaseUrl + "/entity/auth/login", new
            {
                name = userName,
                password = password,
                company = company,
                branch = branch
            }).Result
            .EnsureSuccessStatusCode();
    }

    void IDisposable.Dispose()
    {
        _httpClient.PostAsync(_acumaticaBaseUrl + "/entity/auth/logout",
          new ByteArrayContent(new byte[0])).Wait();
```

```
        _httpClient.Dispose();
    }

    //Retrieval of a record by key fields
    public string GetByKeys(string entityName, string keys, string parameters)
    {
        var res = _httpClient.GetAsync(
          _acumaticaBaseUrl + "/entity/Default/18.200.001/" +
          entityName + "/" + keys + "?" + parameters)
            .Result
            .EnsureSuccessStatusCode();

        return res.Content.ReadAsStringAsync().Result;
    }
}
```

The following code uses the `RestService.GetByKeys()` method, which is defined in the previous code fragment, to retrieve a sales order with detail lines from Acumatica ERP.

```
public static void ExportSODetails()
{
    //Sales order data
    string orderType = "SO";
    string orderNbr = "000001";

    //Initialize the REST service
    RestService rs = new RestService(
            Properties.Settings.Default.AcumaticaBaseUrl,
            Properties.Settings.Default.UserName,
            Properties.Settings.Default.Password,
            Properties.Settings.Default.Company,
            Properties.Settings.Default.Branch
        );

    //Specify the parameter to obtain the details of a sales order
    string parameters = "$expand=Details";

    //Retrieve a sales order by keys
    string stockItems = rs.GetByKeys("SalesOrder" , orderType + "/" + orderNbr,
 parameters);
}
```

Retrieval of a Record by ID

To retrieve a record by the value of the entity ID from Acumatica ERP by using the contract-based REST API, you access the needed URL with the `GET` HTTP method and specify the fields that should be returned in the parameters of the method. See the following sections for details on the URL, parameters, HTTP method, and response format.

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.

URL

If you need to obtain a particular record with the entity ID, you use the following URL.

```
http://<Base endpoint URL>/<Top-level entity>/<Entity ID>
```

The URL has the following components:

- `<Base endpoint URL>` is the URL of a contract-based endpoint through which you are going to work with Acumatica ERP. This URL has the following format: `http://<Acumatica ERP instance URL>/entity/<Endpoint name>/<Endpoint version>/`.
- `<Top-level entity>` is the name of the entity for which you are going to retrieve a record.
- `<Entity ID>` is the ID of the record to be retrieved.

For example, suppose that you want to retrieve the sales order with entity ID *03efa858-2351-4bd5-ae06-3d9fb3b3c1e6* from a local Acumatica ERP instance with the name *AcumaticaDB* by using the system endpoint with the name *Default* and Version 18.200.001. You should use the following URL to retrieve the sales order: `http://localhost/AcumaticaDB/entity/Default/18.200.001/SalesOrder/03efa858-2351-4bd5-ae06-3d9fb3b3c1e6`.

Parameters

You can use the following parameters when you retrieve a record from Acumatica ERP:

- *$expand*: To specify the linked and detail entities to be expanded
- *$select*: To specify the fields of the entity to be returned
- *$custom*: To specify the fields that are not defined in the contract to be returned

For detailed descriptions of the parameters, see *Parameters for Retrieving Records*.

HTTP Method

You use the `GET` HTTP method to retrieve records.

Response

The response of a successful method call contains the retrieved record in JSON format in the response body. For details on record representation in JSON format, see *Representation of a Record in JSON Format*.

Retrieval of Records by Conditions

To retrieve records that satisfy the specified conditions from Acumatica ERP by using the contract-based REST API, you access the needed URL address with the `GET` HTTP method and specify filtering conditions in the parameters of the method. See the following sections for details on the URL, parameters, HTTP method, and response format.

URL

If you need to retrieve the list of records that satisfies the specified conditions, you use the following URL.

```
http://<Base endpoint URL>/<Top-level entity>
```

The URL has the following components:

- `<Base endpoint URL>` is the URL of the contract-based endpoint through which you are going to work with Acumatica ERP, which has the following format: `http://<Acumatica ERP instance URL>/entity/<Endpoint name>/<Endpoint version>/`.
- `<Top-level entity>` is the name of the entity for which you are going to retrieve the list of records.

For example, suppose that you want to retrieve the list of stock item records from a local Acumatica ERP instance with the name *AcumaticaDB* by using the system endpoint with the name *Default* and Version 18.200.001. You should use the following URL to retrieve the list of records: `http://localhost/AcumaticaDB/entity/Default/18.200.001/StockItem`.

Parameters

You can use the following parameters when you retrieve records from Acumatica ERP:

- *$filter*: To specify filtering conditions on the records to be returned
- *$skip*: To specify the number of records to be skipped from the list of returned records
- *$top*: To specify the number of records to be returned in the list
- *$expand*: To specify the linked and detail entities to be expanded
- *$select*: To specify the fields of the entity to be returned

- *$custom*: To specify the fields that are not defined in the contract to be returned

For detailed descriptions of the parameters, see *Parameters for Retrieving Records*.

HTTP Method

You use the `GET` HTTP method to retrieve records.

Response

The response of a successful method call contains the retrieved records in JSON format in the response body. For details on record representation in JSON format, see *Representation of a Record in JSON Format*.

Example

The following code shows an example of a class that implements the retrieval of records from Acumatica ERP through the REST API.

```
public class RestService: IDisposable
{
    private readonly HttpClient _httpClient;

    private readonly string _acumaticaBaseUrl;

    public RestService(
      string acumaticaBaseUrl, string userName, string password,
      string company, string branch)
    {
        _acumaticaBaseUrl = acumaticaBaseUrl;
        _httpClient = new HttpClient(
            new HttpClientHandler
            {
                UseCookies = true,
                CookieContainer = new CookieContainer()
            })
        {
            BaseAddress = new Uri(acumaticaBaseUrl +
              "/entity/Default/18.200.001/"),
            DefaultRequestHeaders =
            {
                Accept = {MediaTypeWithQualityHeaderValue.Parse("text/json")}
            }
        };
        _httpClient.PostAsJsonAsync(
          acumaticaBaseUrl + "/entity/auth/login", new
            {
                name = userName,
                password = password,
                company = company,
                branch = branch
```

```
            }).Result
            .EnsureSuccessStatusCode();
    }

    void IDisposable.Dispose()
    {
        _httpClient.PostAsync(_acumaticaBaseUrl + "/entity/auth/logout",
          new ByteArrayContent(new byte[0])).Wait();
        _httpClient.Dispose();
    }

    public string Get(string entityName, string parameters)
    {
        var res = _httpClient.GetAsync(
          _acumaticaBaseUrl + "/entity/Default/18.200.001/"
          + entityName + "?" + parameters).Result
            .EnsureSuccessStatusCode();

        return res.Content.ReadAsStringAsync().Result;
    }
}
```

The following code uses the `RestService.Get()` method, which is defined in the previous code fragment, to retrieve the list of stock item records that have the *Active* status in Acumatica ERP and have been modified within the past month. The code uses the *$expand* parameter to retrieve the vendor details of each record.

```
public static void ExportStockItems()
{
    //Initialize the REST service
    RestService rs = new RestService(
            Properties.Settings.Default.AcumaticaBaseUrl,
            Properties.Settings.Default.UserName,
            Properties.Settings.Default.Password,
            Properties.Settings.Default.Company,
            Properties.Settings.Default.Branch
        );

    //Specify the parameter to filter records by status and last modified date
    string parameters1 =
      "$filter=ItemStatus eq 'Active' and LastModified gt datetimeoffset'" +
        WebUtility.UrlEncode(new DateTimeOffset(DateTime.Now.AddMonths(-1))
        .ToString("yyyy-MM-ddTHH:mm:ss.fffK")) + "'";

    //Specify the parameter to obtain vendor details
    string parameters2 = "$expand=VendorDetails";

    //Retrieve the list of stock items
    string stockItems = rs.Get("StockItem", parameters1 + "&" + parameters2);
}
```

Usage Notes for Endpoints with Contract Version 3

When multiple records are retrieved from Acumatica ERP through an endpoint with Contract Version 3, the system tries to optimize the retrieval of the records and obtain all needed records in one request to the database (instead of requesting the records one by one). If the optimization fails, the system returns an error, which specifies the entities or fields that caused the failure of the optimized request. To prevent the error from occurring, you can do any of the following:

- If you do not need to retrieve the entities or fields that caused the failure, you can exclude these entities or fields from the request as follows:
  - Exclude the entities from the entities specified in the *$expand* parameter.
  - Explicitly specify the other fields to be returned (while excluding the fields that caused the failure) by using the *$select* parameter.
- If you need to retrieve the entities or fields that caused the failure, you can retrieve the needed records one by one either *by key fields*, or *by IDs*.

Retrieval of Data from an Inquiry Form

To retrieve data from an inquiry form of Acumatica ERP by using the contract-based REST API, you access the needed URL with the `PUT` HTTP method and pass the parameters of the inquiry in JSON format in the request body. See the following sections for details on and examples of the URL, parameter, HTTP method, and response format.

URL

If you need to retrieve data from an inquiry form, you use the following URL.

```
http://<Base endpoint URL>/<Top-level entity>
```

The URL includes the following components:

- `<Base endpoint URL>` is the URL of the contract-based endpoint through which you are going to work with Acumatica ERP, which has the following format: `http://<Acumatica ERP instance URL>/entity/<Endpoint name>/<Endpoint version>/`.
- `<Top-level entity>` is the name of the entity that corresponds to the inquiry form from which you are going to retrieve data.

For example, suppose that you want to retrieve data from the *Inventory Summary* (IN401000) form in a local Acumatica ERP instance with the name *AcumaticaDB* by using the system endpoint with the name *Default* and Version 18.200.001. You would use the following URL to retrieve data: `http://localhost/AcumaticaDB/entity/Default/18.200.001/InventorySummaryInquiry`.

Parameter

When you are retrieving data from an inquiry form, you should use the *$expand* parameter to expand the detail entity, which contains the results of the inquiry. For a detailed description of the parameter, see *Parameters for Retrieving Records*.

HTTP Method

You use the `PUT` HTTP method and pass parameters of the inquiry in JSON format in the request body. See below for an example of the representation of parameters of the *Inventory Summary* inquiry form in JSON format.

```
{
  "InventoryID" : {value : "AALEGO500" } ,
  "WarehouseID" : {value : "MAIN" }
}
```

Response

The response of a successful method call contains the data returned from an inquiry form in JSON format in the response body.

Example

The following code shows an example of a class that implements the retrieval of data from an inquiry form of Acumatica ERP through the REST API.

```
public class RestService: IDisposable
{
    private readonly HttpClient _httpClient;

    private readonly string _acumaticaBaseUrl;

    public RestService(
      string acumaticaBaseUrl, string userName, string password,
      string company, string branch)
    {
        _acumaticaBaseUrl = acumaticaBaseUrl;
        _httpClient = new HttpClient(
            new HttpClientHandler
            {
                UseCookies = true,
                CookieContainer = new CookieContainer()
            })
        {
            BaseAddress = new Uri(acumaticaBaseUrl +
              "/entity/Default/18.200.001/"),
            DefaultRequestHeaders =
            {
                Accept = {MediaTypeWithQualityHeaderValue.Parse("text/json")}
            }
```

```
        };
        _httpClient.PostAsJsonAsync(
          acumaticaBaseUrl + "/entity/auth/login", new
            {
                name = userName,
                password = password,
                company = company,
                branch = branch
            }).Result
            .EnsureSuccessStatusCode();
    }

    void IDisposable.Dispose()
    {
        _httpClient.PostAsync(_acumaticaBaseUrl + "/entity/auth/logout",
          new ByteArrayContent(new byte[0])).Wait();
        _httpClient.Dispose();
    }

    //Data submission
    public string Put(string entityName, string parameters, string entity)
    {
        var res = _httpClient
            .PutAsync(_acumaticaBaseUrl + "/entity/Default/18.200.001/" +
                entityName + "?" + parameters,
                new StringContent(entity, Encoding.UTF8, "application/json"))
            .Result
            .EnsureSuccessStatusCode();

        return res.Content.ReadAsStringAsync().Result;
    }
}
```

The following code uses the `RestService.Put()` method, which is defined in the previous code fragment, to retrieve the quantities of a stock item from the *Inventory Summary* inquiry form.

```
public static void ExportItemQty()
{
    //Path to a source text file that contains the parameters of
    //the inquiry in JSON format
    string entitySource = @"..\..\Input\InventorySummaryInquiry.txt";

    //REST service initialization
    RestService rs = new RestService(
            Properties.Settings.Default.AcumaticaBaseUrl,
            Properties.Settings.Default.UserName,
            Properties.Settings.Default.Password,
            Properties.Settings.Default.Company,
            Properties.Settings.Default.Branch
        );
```

```
    using (StreamReader sr = new StreamReader(entitySource))
    {
        //Read inquiry parameters in JSON format from a file
        string entityAsString = sr.ReadToEnd().ToString();

        //Specify the parameter to expand results of the inquiry
        string parameters = "$expand=Results";

        //Retrieve data from the inquiry
        string stockItems =
            rs.Put("InventorySummaryInquiry", parameters, entityAsString);
    }
}
```

Parameters for Retrieving Records

When you retrieve records from Acumatica ERP by using the contract-based REST API, you can use the URL parameters, which are described in this topic, to filter records by the specified conditions, expand particular entities, and retrieve the values of the fields that are not defined in the contract of the endpoint.

$filter Parameter

You use this parameter to specify the conditions that determine which records should be selected from Acumatica ERP. You use *OData URI conventions* to specify the value of the parameter. The following examples illustrate the use of this parameter:

- *$filter=ItemStatus eq 'Active'*: Obtains stock item records that have the *Active* status in Acumatica ERP.
- *$filter=MainContact/Email eq 'demo@gmail.com'*: Obtains a customer record that has the *demo@gmail.com* email address. (The `Email` field is defined in a linked entity, which is available through the `MainContact` property.)
- *$filter=ItemStatus eq 'Active' and LastModified gt datetimeoffset'2016-07-15T10%3A31%3A28.402%2B03%3A00'*: Obtains stock item records that have the *Active* status in Acumatica ERP and have been modified later than July 15, 2016.

  You should encode date and time values in URL format before passing them as the value of the parameter. For example, you can encode the current date and time by using the `System.Net.WebUtility.URLEncode()` method as follows: `WebUtility.UrlEncode(new DateTimeOffset(DateTime.Now).ToString("yyyy-MM-ddTHH:mm:ss.fffK"))`.

For one field, multiple conditions cannot be specified in the *$filter* parameter. The only exception is the *ge X and le Y* condition, such as *$filter=CreatedDateTime ge*

DateTimeOffset'2015-12-31T00%3A00%3A00%2B000' and CreatedDateTime le DateTimeOffset'2015-01-01T00%3A00%3A00%2B000'.

When you specify the value of the parameter, you can use the following functions as they are defined in OData:

- `substringof`
- `startswith`
- `endswith`

You can also use the following custom function to filter records by the values of the elements that are not defined in the contract of the endpoint: `cf.<Type name>(f='<View name>.<Field name>')`, where `<Type name>` is the type of the custom element, `<View name>` is the name of the data view that contains the element, and `<Field name>` is the name of the element.

For example, suppose that, in an extension of the *Default/18.200.001* endpoint, you added the `RepairItemType` field to the top-level `StockItem` entity. This field corresponds to the **Repair Item Type** custom element added to the **Item Defaults** section of the **General Setting** tab of the *Stock Items* (IN202500) form. If you want to obtain all records on the *Stock Items* form that have the value of the custom **Repair Item Type** element equal to *Battery*, you would use the following parameter string: *$filter=cf.CustomStringField(f='ItemSettings.UsrRepairItemType) eq 'Battery*.

For details on how to find out the name of a custom element and the name of its data view, see *Custom Fields*.

$top Parameter

You use this parameter to specify the number of records to be returned from Acumatica ERP. That is, if you specify *N* as the value of this parameter, the first *N* records will be returned from Acumatica ERP. For example, if you want to obtain only first five records from the list, you use the following parameter string: *$top=5*.

$skip Parameter

You use this parameter to specify the number of records to be skipped from the list of returned records. That is, if you specify *N* as the value of this parameter, the first *N* records will be skipped from the list of returned records. For example, if you do not want to obtain the first five records from the list, you use the following parameter string: *$skip=5*.

If you use the *$skip* and *$top* parameters together, the *$skip* parameter is applied first.

$expand Parameter

You use this parameter to specify the linked and detail entities that should be expanded. By default, no linked or detail entities are expanded; that is, only fields of the top-level entity are returned.

You use *OData URI conventions* to specify the value of this parameter. For example, if you want to obtain the values of the warehouse detail lines of stock item records, you use the following parameter string: *$expand=WarehouseDetails*.

You explicitly specify each linked or detail entity to be expanded. For example, if you specify *$expand=MainContact* for the `Customer` entity, only the `Contact` linked entity of the `Customer` entity is expanded, but the `Address` linked entity within `MainContact` is not. To expand the `Address` entity, you should explicitly specify the `Address` entity to be expanded: *$expand=MainContact/Address*.

$select Parameter

You use this parameter to specify the fields of the entity to be returned from Acumatica ERP. By default, all fields of the entity are returned.

You use *OData URI conventions* to specify the value of this parameter. For example, if you want to obtain only the order types and order numbers of sales orders, you use the following parameter string: *$select=OrderType,OrderNbr*.

$custom Parameter

You use this parameter to specify the fields that are not defined in the contract of the endpoint to be returned from Acumatica ERP. That is, you can use this parameter to obtain the values of the predefined elements on an Acumatica ERP form that are not included in the entity definition, the values of user-defined fields, and the values of elements that were added to the Acumatica ERP form in a customization project.

You use one of the following formats to specify the element whose value should be returned:

- If a top-level entity contains the custom field that corresponds to the element: `<View name>.<Field name>`, where you replace `<View name>` with the name of the data view that contains the element and `<Field name>` with the internal name of the element.

  For example, suppose that in an extension of the *Default/18.200.001* endpoint, you added the `RepairItemType` field to the top-level `StockItem` entity. This field corresponds to the **Repair Item Type** custom element added to the **Item Defaults** section of the **General Setting** tab of the *Stock Items* (IN202500) form. If you want to obtain the value of this element, you use the following parameter string: *$custom=ItemSettings.UsrRepairItemType*.

- If a linked or detail entity contains the custom field that corresponds to the element: `<Entity name>/<View name>.<Field name>`, where you replace `<Entity name>` with the name of the linked or detail entity that contains the field, `<View name>` with the name of the data view that contains the element, and `<Field name>` with the internal name of the element.

  For example, suppose that in an extension of the *Default/18.200.001* endpoint, you added the `RepairItemType` field to the `SalesOrderDetail` detail entity. This field corresponds to the **Repair Item Type** custom column added to the **Document Details** tab of the *Sales Orders* (SO301000) form. If you want to obtain the value of this element, you use the following parameter string: *$custom=Details/Transactions.UsrRepairItemType*.

If you want to obtain the value of a custom field of a linked or detail entity, in addition to specifying the *$custom* parameter, you have to specify this entity in the *$expand* parameter.

- If the element is a user-defined field: `Document.Attribute<AttributeID>`, where you replace `<AttributeID>` with the ID of the attribute that corresponds to the user-defined field.

  For example, suppose that on the *Sales Orders* (SO301000) form, you have added a user-defined field for the *OPERATSYST* attribute. If you want to obtain the value of this user-defined field, you use the following parameter string: *$custom=Document.AttributeOPERATSYST*.

  For details about user-defined fields, see *User-Defined Fields*.

If you want to obtain the values of multiple custom elements, you specify the custom elements to be returned, separated by commas. For details on how to find out the field name and the name of the data view, see *Custom Fields*.

Related Links

- *OData URI conventions*
- *Custom Fields*

Removal of a Record

In the contract-based REST API, you can delete the record by the value of its key fields or by its session identifier. To delete a record from Acumatica ERP, you access the needed URL address with the `DELETE` HTTP method. See the following sections for details on the possible URL, parameters, HTTP method, and response format.

If you need to delete a detail line of a record, you should use the `PUT` HTTP method, as described in *Update of a Record*.

URL for Removing by Key Fields

If you need to delete a record with known key fields, you use the following URL.

```
http://<Base endpoint URL>/<Top-level entity>/<Key value 1>/<Key value 2>
```

The URL has the following components:

- `<Base endpoint URL>` is the URL of the contract-based endpoint through which you are going to work with Acumatica ERP, which has the following format: `http://<Acumatica ERP instance URL>/entity/<Endpoint name>/<Endpoint version>/`.
- `<Top-level entity>` is the name of the entity for which you are going to delete a record.

- `<Key value 1>` and `<Key value 2>` are the values of the key fields of the record to be deleted. You use the number and order of key fields as they are defined on the corresponding Acumatica ERP form.

  You can pass the key fields separated by vertical bar (|) instead of slash(/).

For example, suppose that you want to delete the sales order with order type *SO* and order number *000123* from a local Acumatica ERP instance with the name *AcumaticaDB* by using the system endpoint with the name *Default* and Version 18.200.001. You should use the following URL to delete the sales order: `http://localhost/AcumaticaDB/entity/Default/18.200.001/SalesOrder/SO/000123`.

URL for Removing by ID

If you need to delete a record with a known entity ID, you use the following URL. For details about entity IDs, see *Retrieval of a Record by ID*.

```
http://<Base endpoint URL>/<Top-level entity>/<Entity ID>
```

You replace `<Base endpoint URL>` with the URL of the contract-based endpoint through which you are going to work with Acumatica ERP, which has the following format: `http://<Acumatica ERP instance URL>/entity/<Endpoint name>/<Endpoint version>/`. You replace `<Top-level entity>` with the name of the entity for which you are going to retrieve the list of records. You replace `<Entity ID>` with the entity ID (which is the GUID that you can obtain from the `ID` property of an entity returned from Acumatica ERP).

For example, suppose that you want to delete the sales order with entity ID *03efa858-2351-4bd5-ae06-3d9fb3b3c1e6* from a local Acumatica ERP instance with the name *AcumaticaDB* by using the system endpoint with the name *Default* and Version 18.200.001. You should use the following URL to delete the sales order: `http://localhost/AcumaticaDB/entity/Default/18.200.001/SalesOrder/03efa858-2351-4bd5-ae06-3d9fb3b3c1e6`.

Parameters

You use no parameters when deleting a record.

HTTP Method

You use the `DELETE` HTTP method to retrieve records. You pass no content in the request body.

Response

The response of a successful method call is *204 No Content*.

Example

The following code shows an example of a class that implements the removal of a record from Acumatica ERP through the REST API.

```
public class RestService: IDisposable
{
    private readonly HttpClient _httpClient;

    private readonly string _acumaticaBaseUrl;

    public RestService(
      string acumaticaBaseUrl, string userName, string password,
      string company, string branch)
    {
        _acumaticaBaseUrl = acumaticaBaseUrl;
        _httpClient = new HttpClient(
            new HttpClientHandler
            {
                UseCookies = true,
                CookieContainer = new CookieContainer()
            })
        {
            BaseAddress = new Uri(acumaticaBaseUrl +
              "/entity/Default/18.200.001/"),
            DefaultRequestHeaders =
            {
                Accept = {MediaTypeWithQualityHeaderValue.Parse("text/json")}
            }
        };
        _httpClient.PostAsJsonAsync(
          acumaticaBaseUrl + "/entity/auth/login", new
            {
                name = userName,
                password = password,
                company = company,
                branch = branch
            }).Result
            .EnsureSuccessStatusCode();
    }

    void IDisposable.Dispose()
    {
        _httpClient.PostAsync(_acumaticaBaseUrl + "/entity/auth/logout",
          new ByteArrayContent(new byte[0])).Wait();
        _httpClient.Dispose();
    }

    //Removal of a record
    public string Delete(string entityName, string keys)
    {
        var res = _httpClient.DeleteAsync(_acumaticaBaseUrl
          + "/entity/Default/18.200.001/" + entityName + "/" + keys).Result
            .EnsureSuccessStatusCode();
```

```
        return res.Content.ReadAsStringAsync().Result;
    }
}
```

The following code uses the `RestService.Delete()` method, which is defined in the previous code fragment, to delete a stock item record.

```
public static void DeleteStockItem()
{
    //Stock item data
    string inventoryID = "ASDF";

    //Initialize the REST service
    RestService rs = new RestService(
            Properties.Settings.Default.AcumaticaBaseUrl,
            Properties.Settings.Default.UserName,
            Properties.Settings.Default.Password,
            Properties.Settings.Default.Company,
            Properties.Settings.Default.Branch
        );

    //Remove the stock item
    string stockItem = rs.Delete("StockItem", inventoryID);
}
```

Execution of an Action

To perform an action by using the contract-based REST API, you access the needed URL address with the `POST` HTTP method and pass the record representation in JSON format and parameters of the action in the request body. See the following sections for details on the URL, parameters, HTTP method, and response format.

URL

If you need to perform an action on an Acumatica ERP form, you use the following URL.

```
http://<Base endpoint URL>/<Top-level entity>/<Action name>
```

The URL has the following components:

- `<Base endpoint URL>` is the URL of the contract-based endpoint through which you are going to work with Acumatica ERP, which has the following format: `http://<Acumatica ERP instance URL>/entity/<Endpoint name>/<Endpoint version>/`.
- `<Top-level entity>` is the name of the entity for which you are going to perform an action.
- `<Action name>` is the name of the action that you are going to perform.

For example, suppose that you want to confirm a shipment in a local Acumatica ERP instance with the name *AcumaticaDB* by using the system endpoint with the name *Default* and Version 18.200.001. You should use the following URL to confirm a shipment: `http://localhost/AcumaticaDB/entity/Default/18.200.001/Shipment/ConfirmShipment`.

Parameters

You use no parameters when performing an action.

HTTP Method

You use the `POST` HTTP method and pass the record to which the action should be applied and the parameters of the action in the request body in JSON format as follows.

```
{
  "entity" : <record in JSON format>,
  "parameters" : <parameters in JSON format>
}
```

You can find details on how to represent a record in JSON format in *Representation of a Record in JSON Format*.

Response

If the long-running operation that was initiated by the action is completed or wasn't created, the response is *204 No Content*. If the long-running operation is in progress, the response is *202 Accepted*; it has the `Location` header, which contains a URL that can be used to check the status of the operation by using the `GET` HTTP method. When the `GET` HTTP method with this URL returns *204 No Content*, the operation is completed.

Example

The following code shows an example of a class that implements the execution of an action in Acumatica ERP through the REST API.

```
public class RestService: IDisposable
{
    private readonly HttpClient _httpClient;

    private readonly string _acumaticaBaseUrl;

    public RestService(
      string acumaticaBaseUrl, string userName, string password,
      string company, string branch)
    {
        _acumaticaBaseUrl = acumaticaBaseUrl;
        _httpClient = new HttpClient(
            new HttpClientHandler
            {
                UseCookies = true,
                CookieContainer = new CookieContainer()
            })
```

```
        {
            BaseAddress = new Uri(acumaticaBaseUrl +
              "/entity/Default/18.200.001/"),
            DefaultRequestHeaders =
            {
                Accept = {MediaTypeWithQualityHeaderValue.Parse("text/json")}
            }
        };
        _httpClient.PostAsJsonAsync(
          acumaticaBaseUrl + "/entity/auth/login", new
            {
                name = userName,
                password = password,
                company = company,
                branch = branch
            }).Result
            .EnsureSuccessStatusCode();
    }

    void IDisposable.Dispose()
    {
        _httpClient.PostAsync(_acumaticaBaseUrl + "/entity/auth/logout",
          new ByteArrayContent(new byte[0])).Wait();
        _httpClient.Dispose();
    }

    //Invocation of an action
    public string Post(string entityName, string actionName,
      string entityAndParameters)
    {
        var result = _httpClient
            .PostAsync(_acumaticaBaseUrl + "/entity/Default/18.200.001/"
              + entityName + "/" + actionName,
              new StringContent(entityAndParameters,
                Encoding.UTF8, "application/json"));
            var res = result.Result;
            var cont = res.Content.ReadAsStringAsync().Result;
            res.EnsureSuccessStatusCode();

        var dt = DateTime.Now;
        while (true)
        {
            switch (res.StatusCode)
            {
                case HttpStatusCode.NoContent:
                    return "No content";
                case HttpStatusCode.Accepted:
                    if ((DateTime.Now - dt).Seconds > 30)
                        throw new TimeoutException();
                    Thread.Sleep(500);
                    res = _httpClient.GetAsync(res.Headers.Location)
```

```
                    .Result.EnsureSuccessStatusCode();
                continue;
            default:
                throw new InvalidOperationException(
                  "Invalid process result: " + res.StatusCode);
        }
    }
}
```

The following code uses the `RestService.Post()` method, which is defined in the previous code fragment, to release a sales order invoice.

```
public static void ReleaseSOInvoice()
{
    //Invoice to be released
    string invoice = "{\"Type\" : {value : \"Invoice\"}, "
      + "\"ReferenceNbr\" : {value : \"INV000045\"} }";

    //Initialize the REST service
    RestService rs = new RestService(
            Properties.Settings.Default.AcumaticaBaseUrl,
            Properties.Settings.Default.UserName,
            Properties.Settings.Default.Password,
            Properties.Settings.Default.Company,
            Properties.Settings.Default.Branch
        );

    //Release the invoice
    invoice = rs.Post("SalesInvoice", "ReleaseSalesInvoice",
      "{\"entity\" : " + invoice + ", \"parameters\" : null}");
}
```

Attachment of a File to a Record

When you need to attach a file to a record by using the contract-based REST API, you access the needed URL address with the `PUT` HTTP method and pass the file in the request body. See the following sections for details on the URL, parameters, HTTP method, and response format.

URL

If you need to attach a file to a record in Acumatica ERP, you use the following URL.

```
http://<Base endpoint URL>/<Top-level entity>/<Key value 1>/<Key value 2>/files/<File name>
```

The URL has the following components:

- `<Base endpoint URL>` is the URL of the contract-based endpoint through which you are going to work with Acumatica ERP, which has the following format: `http://<Acumatica ERP instance URL>/entity/<Endpoint name>/<Endpoint version>/`.
- `<Top-level entity>` is the name of the entity to which you are going to attach a file.
- `<Key value 1>` and `<Key value 2>` are the values of the key fields of the record to which you are going to attach a file. You use the number and order of key fields as they are defined on the corresponding Acumatica ERP form.

  You can pass the key fields separated by vertical bar (|) instead of slash(/).

- `<File name>` is the name of the file that you are going to attach with the extension.

For example, suppose that you want to attach the **Sample.jpg** file to the stock item record with inventory ID *AALEGO500* in a local Acumatica ERP instance with name *AcumaticaDB* by using the system endpoint with the name *Default* and Version 18.200.001. You should use the following URL to attach the file: `http://localhost/AcumaticaDB/entity/Default/18.200.001/StockItem/AALEGO500/files/Sample.jpg`.

Parameters

You use no parameters when you attach a file to a record.

HTTP Method

You use the `PUT` HTTP method and pass the file to be attached in the request body.

Response

The response of a successful method call is *204 No Content*.

Example

The following code shows an example of a class that implements the attachment of a file to a record in Acumatica ERP through the REST API.

```
public class RestService: IDisposable
{
    private readonly HttpClient _httpClient;

    private readonly string _acumaticaBaseUrl;

    public RestService(
      string acumaticaBaseUrl, string userName, string password,
      string company, string branch)
    {
        _acumaticaBaseUrl = acumaticaBaseUrl;
        _httpClient = new HttpClient(
            new HttpClientHandler
            {
```

```
                UseCookies = true,
                CookieContainer = new CookieContainer()
            })
        {
            BaseAddress = new Uri(acumaticaBaseUrl +
              "/entity/Default/18.200.001/"),
            DefaultRequestHeaders =
            {
                Accept = {MediaTypeWithQualityHeaderValue.Parse("text/json")}
            }
        };
        _httpClient.PostAsJsonAsync(
          acumaticaBaseUrl + "/entity/auth/login", new
            {
                name = userName,
                password = password,
                company = company,
                branch = branch
            }).Result
            .EnsureSuccessStatusCode();
    }

    void IDisposable.Dispose()
    {
        _httpClient.PostAsync(_acumaticaBaseUrl + "/entity/auth/logout",
          new ByteArrayContent(new byte[0])).Wait();
        _httpClient.Dispose();
    }

    //Attachment of a file to a record
    public string PutFile(string entityName, string keys,
      string fileName, System.IO.Stream file)
    {
        var res = _httpClient.PutAsync(_
          acumaticaBaseUrl + "/entity/Default/18.200.001/"
          + entityName + "/" + keys + "/files/" + fileName,
          new StreamContent(file)).Result
            .EnsureSuccessStatusCode();
        return res.Content.ReadAsStringAsync().Result;
    }
}
```

The following code uses the `RestService.PutFile()` method, which is defined in the previous code fragment, to attach a file to a stock item record.

```
public static void PutFile()
{
    //Input data
    string inventoryID = "AALEGO500";
    string fileName = "T2MCRO.jpg";
    string entitySource = @"..\..\Input\T2MCRO.jpg";
```

```
    //Initialize the REST service
    RestService rs = new RestService(
            Properties.Settings.Default.AcumaticaBaseUrl,
            Properties.Settings.Default.UserName,
            Properties.Settings.Default.Password,
            Properties.Settings.Default.Company,
            Properties.Settings.Default.Branch
        );

    using (StreamReader sr = new StreamReader(entitySource))
    {
        //Attach a file to a stock item record
        string stockItem = rs.PutFile("StockItem", inventoryID, fileName,
          sr.BaseStream);
    }

}
```

Retrieval of a File Attached to a Record

To retrieve a file that is attached to a record from Acumatica ERP by using the contract-based REST API, you access the URL address of the file with the `GET` HTTP method. See the following sections for details on the URL, parameters, HTTP method, and response format.

URL

If you need to obtain a file attached to a record, you use the following URL.

```
http://<Base endpoint URL>/files/<File identifier>
```

The URL has the following components:

- `<Base endpoint URL>` is the URL of the contract-based endpoint through which you are going to work with Acumatica ERP, which has the following format: `http://<Acumatica ERP instance URL>/entity/<Endpoint name>/<Endpoint version>/`.
- `<File identifier>` is the internal identifier of the file in the system.

To get this URL for a particular file attached to a record, you should obtain the record from Acumatica ERP and, in the returned JSON representation of the record, find the value of the `href` property of the needed file in the `files` array. For information on how to retrieve a record from Acumatica ERP, see *Retrieval of a Record by Key Fields* and *Retrieval of a Record by ID*.

For example, suppose that you retrieved the stock item record that contains the following `files` array from a local Acumatica ERP instance with the name *AcumaticaDB*.

```
{
  ...,
```

```
  "files":[
    {
      "id":"9be45eb7-f97d-400b-96a5-1c4cf82faa96",
      "filename":"Stock Items (AAMACHINE1)\\T2MCRO.jpg",
      "href":
"/AcumaticaDB/entity/Default/18.200.001/files/9be45eb7-f97d-400b-96a5-1c4cf82faa96"
    }
  ]
}
```

You should use the following URL to retrieve the **T2MCRO.jpg** file attached to the stock item record: `http://localhost/AcumaticaDB/entity/Default/18.200.001/files/9be45eb7-f97d-400b-96a5-1c4cf82faa96`.

Parameters

You use no parameters when retrieving a file.

HTTP Method

You use the `GET` HTTP method to retrieve a file.

Response

The response of a successful method call contains the retrieved file in the response body.

Example

The following code shows an example of a class that implements the retrieval of a file attached to a record in Acumatica ERP through the REST API.

```
public class RestService: IDisposable
{
    private readonly HttpClient _httpClient;

    private readonly string _acumaticaBaseUrl;

    public RestService(
      string acumaticaBaseUrl, string userName, string password,
      string company, string branch)
    {
        _acumaticaBaseUrl = acumaticaBaseUrl;
        _httpClient = new HttpClient(
            new HttpClientHandler
            {
                UseCookies = true,
                CookieContainer = new CookieContainer()
            })
        {
            BaseAddress = new Uri(acumaticaBaseUrl +
              "/entity/Default/18.200.001/"),
            DefaultRequestHeaders =
            {
```

```
                Accept = {MediaTypeWithQualityHeaderValue.Parse("text/json")}
            }
        };
        _httpClient.PostAsJsonAsync(
          acumaticaBaseUrl + "/entity/auth/login", new
            {
                name = userName,
                password = password,
                company = company,
                branch = branch
            }).Result
            .EnsureSuccessStatusCode();
    }

    void IDisposable.Dispose()
    {
        _httpClient.PostAsync(_acumaticaBaseUrl + "/entity/auth/logout",
          new ByteArrayContent(new byte[0])).Wait();
        _httpClient.Dispose();
    }

    //Retrieval of a record by key fields
    public string GetByKeys(string entityName, string keys, string parameters)
    {
        var res = _httpClient.GetAsync(
          _acumaticaBaseUrl + "/entity/Default/18.200.001/" +
          entityName + "/" + keys + "?" + parameters)
            .Result
            .EnsureSuccessStatusCode();

        return res.Content.ReadAsStringAsync().Result;
    }

    //Retrieving of a file
    public System.IO.Stream GetFile(string href)
    {
        var res = _httpClient.GetAsync(href).Result.EnsureSuccessStatusCode();
        return res.Content.ReadAsStreamAsync().Result;
    }
}
```

The following code uses the `RestService.GetByKeys()` and `RestService.GetFile()` methods, which are defined in the previous code fragment, to retrieve a file attached to a stock item record from Acumatica ERP.

```
public static void GetFile()
{
    //Specify stock item data
    string inventoryID = "AAMACHINE1";

    //Initialize the REST service
```

```
    RestService rs = new RestService(
            Properties.Settings.Default.AcumaticaBaseUrl,
            Properties.Settings.Default.UserName,
            Properties.Settings.Default.Password,
            Properties.Settings.Default.Company,
            Properties.Settings.Default.Branch
        );

    //Retrieve the stock item
    string stockItem = rs.GetByKeys("StockItem", inventoryID, null);

    //Find href and the file name of the needed file
    //(using Newtonsoft.Json.Linq and System.IO)
    JObject jItem = JObject.Parse(stockItem);
    JArray jFiles = jItem.Value<JArray>("files");
    string fileRef = jFiles[0].Value<string>("href");
    string fullFileName = jFiles[0].Value<string>("filename");
    string fileName = Path.GetFileName(fullFileName);

    //Obtain the file
    Stream file = rs.GetFile(fileRef);

    using (var outputFile = File.Create(@"..\..\Output\" + fileName))
    {
        file.Seek(0, SeekOrigin.Begin);
        file.CopyTo(outputFile);
    }
}
```

Retrieval of the Schema of Custom Fields

To retrieve the schema of custom fields of an entity—that is, the field name, view name, and type of the fields that are not defined in the contract of the endpoint for this entity—by using the contract-based REST API, you access the needed URL with the `GET` HTTP method. See the following sections for details on and examples of the URL, parameters, HTTP method, and response format.

URL

If you need to obtain the schema of custom fields of an entity, you use the following URL.

```
http://<Base endpoint URL>/<Top-level entity>/$adHocSchema
```

The URL includes the following components:

- `<Base endpoint URL>` is the URL of the contract-based endpoint through which you are going to work with Acumatica ERP, which has the following format: `http://<Acumatica ERP instance URL>/entity/<Endpoint name>/<Endpoint version>/`.

- `<Top-level entity>` is the name of the entity for which you are going to retrieve the schema of custom fields.

For example, suppose that you want to obtain the schema of custom fields of a stock item entity from a local Acumatica ERP instance with the name *AcumaticaDB* by using the system endpoint with the name *Default* and Version 18.200.001. You would use the following URL to retrieve the schema: `http://localhost/AcumaticaDB/entity/Default/18.200.001/StockItem/$adHocSchema`.

Parameters

You use no parameters when retrieving the schema of custom fields of an entity.

HTTP Method

You use the `GET` HTTP method to retrieve the schema of custom fields.

Response

The response of a successful method call contains the schema of custom fields in JSON format in the response body.

Retrieval of the Acumatica ERP Version and the List of Endpoints

To obtain the Acumatica ERP version and the list of contract-based endpoints available in this version by using the REST API, you access the needed URL with the `GET` HTTP method. The remainder of this topic provides details on and examples of the URL, the parameters, the headers, and the format of the response.

If the request is sent without the authentication information, the list contains only the endpoints available in the system by default. If the request is sent with the authentication information for a particular tenant, the list also includes the custom endpoints configured in this tenant of the Acumatica ERP instance.

HTTP Method

You use the `GET` HTTP method to retrieve the Acumatica ERP version and the list of contract-based endpoints available in this version.

URL

You use the following URL.

```
http://<Acumatica ERP instance URL>/entity
```

In this URL, *<Acumatica ERP instance URL>* is the URL of the Acumatica ERP instance for which you want to obtain information about the version and endpoints.

For example, suppose that you work with a local Acumatica ERP instance with the name *AcumaticaDB*. You would use the following URL to retrieve the information: `http://localhost/AcumaticaDB/entity`.

Parameters

You use no parameters when you retrieve the Acumatica ERP version and the list of contract-based endpoints available in this version.

Headers

You can specify the following header in the request.

Header	Description
`Accept`	Optional. Specifies the format of the response body, which can be one of the following: • `application/json` (default) • `text/json` • `application/xml` • `text/xml`

Response

The response of a successful method call contains the Acumatica ERP version and the list of contract-based endpoints available in this version in the response body. The list of endpoints in the response includes the custom endpoints configured in a particular tenant of the Acumatica ERP instance if the request is sent along with the authentication information. You can obtain the response in JSON or XML format.

The following example shows the response in JSON format.

```
{
    "version": {
        "acumaticaBuildVersion": "19.191.0093",
        "databaseVersion": "19.191.0093"
    },
    "endpoints": [
        {
            "name": "Default",
            "version": "17.200.001",
            "href": "/AcumaticaERP/entity/Default/17.200.001/"
        },
        {
            "name": "DeviceHub",
            "version": "17.200.001",
            "href": "/AcumaticaERP/entity/DeviceHub/17.200.001/"
        },
```

```
        {
            "name": "POS",
            "version": "17.200.001",
            "href": "/AcumaticaERP/entity/POS/17.200.001/"
        },
        {
            "name": "Default",
            "version": "18.200.001",
            "href": "/AcumaticaERP/entity/Default/18.200.001/"
        },
        {
            "name": "Default",
            "version": "6.00.001",
            "href": "/AcumaticaERP/entity/Default/6.00.001/"
        }
    ]
}
```

OpenAPI Specification

You can retrieve the **swagger.json** file with the OpenAPI specification of the endpoint by using the following URL.

```
http://<Acumatica ERP instance URL>/entity/swagger.json
```

In this URL, *<Acumatica ERP instance URL>* is the URL of the Acumatica ERP instance for which you want to obtain information about the version and endpoints.

Multi-Language Fields

For some text boxes on Acumatica ERP forms, users can type values in multiple languages if multiple locales are configured in Acumatica ERP. For example, if your Acumatica ERP instance has English and French locales activated and multilingual user input configured, you can specify the value of the **Description** box on the *Stock Items* (IN202500) form in English and French. For the list of elements that support multiple languages, see *Boxes that Have Multi-Language Support*. For details on how to turn on multilingual user input, see *Enabling Multilingual User Input*.

Specifying Localized Values of a Multi-Language Field

When you need to specify localized values of a text box by using the contract-based REST API, you specify the value of the field that corresponds to the box as a string in JSON format with the localized values. In this string, you use the two-letter ISO code of the language with which the value should be associated.

In the example that is mentioned at the beginning of the topic, if you need to specify values in English and French in the **Description** box on the *Stock Items* form, you specify the value of the `Description` field of the `StockItem` entity in the following format: `[{en:English description},{fr:French description}]`. See below for an example of a stock item record with localized `Description` field values in JSON format. For details on how to pass the record to the service, see *Creation of a Record* and *Update of a Record*.

```
{
  "InventoryID" : {value : "BASESERV" } ,
  "Description" : {value : "[{en:Item},{fr:Pièce}]" }
}
```

In the JSON-formatted string, you should specify the actual values of the field in all languages that are configured for multilingual user input. If you specify the values of the field in particular languages, the values of the field in other languages configured for multilingual user input become empty. For example, suppose that in your instance of Acumatica ERP, multi-language fields can have values in English and French. If you pass the value of a field in the following format `[{en:English description}]`, the French value of the field becomes empty.

If you specify the value of a multi-language field as plain text, this text is saved as the value of the corresponding box in the current language of Acumatica ERP (that is, the language that you specified when you logged in to Acumatica ERP). For details on how to specify the language on login through the contract-based REST API, see *Signing In to the Service*.

Retrieving Localized Values of a Multi-Language Field

If you need to retrieve localized values of a text box that supports multiple input languages, you retrieve the value a special custom field that contains all localized values of the text box and has the *Translations* suffix in its field name.

To find out the field name and the view name of the needed custom field with localized values, you find out the field name and the view name of the multi-language text box and append *Translations* to the field name. (For details on how to find out the field name and the view name of an element on the form, see *Custom Fields*.) For example, the multi-language **Description** box on the *Stock Items* form has the *Descr* field name and the *Item* view name; therefore, the custom field that contains the localized descriptions of a stock item has the *DescrTranslations* field name and the *Item* view name.

You obtain the value of the needed custom field by using the *$custom* parameter. For example, suppose that you need to obtain the localized values of the **Description** element of the *Stock Items* form. In this case, you should use the following parameter string in the request URL: *$custom=Item.DescrTranslations*. For details on how to retrieve a record, see *Retrieval of a Record by Key Fields* and *Retrieval of Records by Conditions*.

The returned value of a `Translations` custom field is a string in JSON format with the available localized values of the field. The language to which the value belongs is identified by the two-letter ISO code of the language. For example, suppose that the **Description** element of the *Stock Items* form has the value *Item* in English and *Pièce* in French. In this case, the value of the `DescrTranslations` custom field, which corresponds to the **Description** element, is the following string: `[{en:Item},{fr:Pièce}]`.

Related Links

- *Locales and Languages*
- *Boxes that Have Multi-Language Support*
- *Custom Fields*

Working with the Contract-Based SOAP API

The contract-based SOAP application programming interface (API) of Acumatica ERP provides the SOAP interface of the Acumatica ERP contract-based web services through which external systems can get data records from Acumatica ERP, process these records, and save new or updated records to Acumatica ERP.

This chapter includes the topics that are specific for the contract-based SOAP API. For general information on the contract-based web services, see *Configuring the Contract-Based REST and SOAP API*. You can find examples of how to use the contract-based SOAP API in the *I210 Integration: Contract-Based Web Services* training course and in *Contract-Based API Examples*. For the API reference, see *Contract-Based SOAP API Reference*.

In This Chapter

- *Multi-Language Fields*
- *To Configure the Client Application*

Multi-Language Fields

For some text boxes on Acumatica ERP forms, users can type values in multiple languages if multiple locales are configured in Acumatica ERP. For example, if your Acumatica ERP instance has English and French locales activated and multilingual user input configured, you can specify the value of the **Description** box on the *Stock Items* (IN202500) form in English and French. For the list of elements that support multiple languages, see *Boxes that Have Multi-Language Support*. For details on how to turn on multilingual user input, see *Enabling Multilingual User Input*.

Specifying Localized Values of a Multi-Language Field

When you need to specify localized values of a text box by using the contract-based SOAP API, you specify the value of the field that corresponds to the box as a string in JSON format with the localized values. In this string, you use the two-letter ISO code of the language with which the value should be associated.

In the example that is mentioned at the beginning of the topic, if you need to specify values in English and French in the **Description** box on the *Stock Items* form, you specify the value of the `Description` field of the `StockItem` entity in the following format: `[{en:English description},{fr:French description}]`, as shown in the following code fragment.

```
public static void CreateStockItem(DefaultSoapClient soapClient)
{
    //Specify the values of the new stock item
    StockItem stockItemToBeCreated = new StockItem
    {
        InventoryID = new StringValue { Value = "BASESERV" },
        Description = new StringValue { Value = "[{en:Item},{fr:Pièce}]" },
        ItemClass = new StringValue { Value = "STOCKITEM" },
    };
```

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

In the JSON-formatted string, you should specify the actual values of the field in all languages that are configured for multilingual user input. If you specify the values of the field in particular languages, the values of the field in other languages configured for multilingual user input become empty. For example, suppose that in your instance of Acumatica ERP, multi-language fields can have values in English and French. If you pass the value of a field in the following format `[{en:English description}]`, the French value of the field becomes empty.

If you specify the value of a multi-language field as plain text, this text is saved as the value of the corresponding box in the current language of Acumatica ERP (that is, the language that you specified when you logged in to Acumatica ERP). For details on how to specify the language on login through the contract-based SOAP API, see *Login() Method*.

Retrieving Localized Values of a Multi-Language Field

If you need to retrieve localized values of a text box that supports multiple input languages, you retrieve the value of a special custom field that contains all localized values of the text box and has the *Translations* suffix in its field name.

To find out the field name and the view name of the needed custom field with localized values, you find out the field name and the view name of the multi-language text box and append *Translations* to the field name. (For details on how to find out the field name and the view name of an element on the form, see *Custom Fields*.) For example, the multi-language **Description** box on the *Stock Items* form has the *Descr* field name and the *Item* view name; therefore, the custom field that contains the localized descriptions of a stock item has the *DescrTranslations* field name and the *Item* view name.

The following code shows how to retrieve the localized values of the **Description** element of the *Stock Items* form. (The code below uses Contract Version 2.)

```
public static void ExportStockItem(DefaultSoapClient soapClient)
{
    StockItem stockItem = new StockItem
    {
        InventoryID = new StringValue { Value = "BASESERV" },
        //Specify the localized values to be returned
        CustomFields = new[]
        {
          new CustomStringField
          {
            fieldName = "DescrTranslations",
            viewName = "Item",
            Value = new StringReturn(),
          }
        }
    };
    //Retrieve the stock item record
    StockItem stockItemToRetrieve = (StockItem)client.Get(stockItem);
```

```
}
```

The returned value of a `Translations` custom field is a string in JSON format with the available localized values of the field. The language to which the value belongs is identified by the two-letter ISO code of the language. For example, suppose that the **Description** element of the *Stock Items* form has the value *Item* in English and *Pièce* in French. In this case, the value of the `DescrTranslations` custom field, which corresponds to the **Description** element, is the following string: `[{en:Item},{fr:Pièce}]`.

Related Links

- *Locales and Languages*
- *Boxes that Have Multi-Language Support*
- *Custom Fields*

To Configure the Client Application

In this topic, you will learn how to import the WSDL description of the Acumatica ERP web services into a Visual Studio project.

The procedure below is described based on an example of the creation of a console project with the *MyBIIntegration* name that uses the *Default* endpoint of the *18.200.001* version. You can create another type of Visual Studio project with another name and use any other endpoint in this procedure.

The instructions in this topic have been created for Visual Studio 2019.

Importing the WSDL File into a Visual Studio Project

To configure the client application, do the following:

1. Open Visual Studio, and create a new Visual C# console application with the *MyBIIntegration* name.

   To create a console application, click **File > New > Project** on the main menu of Visual Studio. In the **Create a new project** dialog box, which appears, select the *Console App (.Net Framework)* C# template and click **Next**. In the **Configure your new project** dialog box, specify the name and location of the solution, select the *.NET Framework 4.8* framework version, and click **Create**.

   The MyBIIntegration application that you have created contains the **Program.cs** file with the `Program` class.

2. Add to the project a reference to the Acumatica ERP web service as follows:

   a. On the main menu of Visual Studio, click **Project > Add Service Reference**.

   b. In the **Address** box of the **Add Service Reference** dialog box, which opens, specify the URL of the *Default* endpoint of the *18.200.001* version (see Item 1 in the following screenshot).

To copy the URL of the service, on the *Web Service Endpoints* (SM207060) form, select *Default* in the **Endpoint Name** box and *18.200.001* in the **Endpoint Version** box, click **View Endpoint Service > WSDL** on the form toolbar, and copy the URL from the address line in the browser for the page that opens. In this example, the URL is *http://localhost/MyStoreInstance/entity/Default/18.200.001?wsdl*.

c. Click **Go** (Item 2) to make Visual Studio connect to the web service.

If your Acumatica ERP website uses a self-signed certificate, Visual Studio displays security alert windows with warnings on the certificate. Click **Yes** in these windows to proceed.

d. In the **Namespace** box, type `Default` (3). This name will be used as a namespace for the web service classes generated by Visual Studio based on the WSDL description of the service.

e. Click **OK** (4) to close the dialog box and add to the project the reference to the specified service.

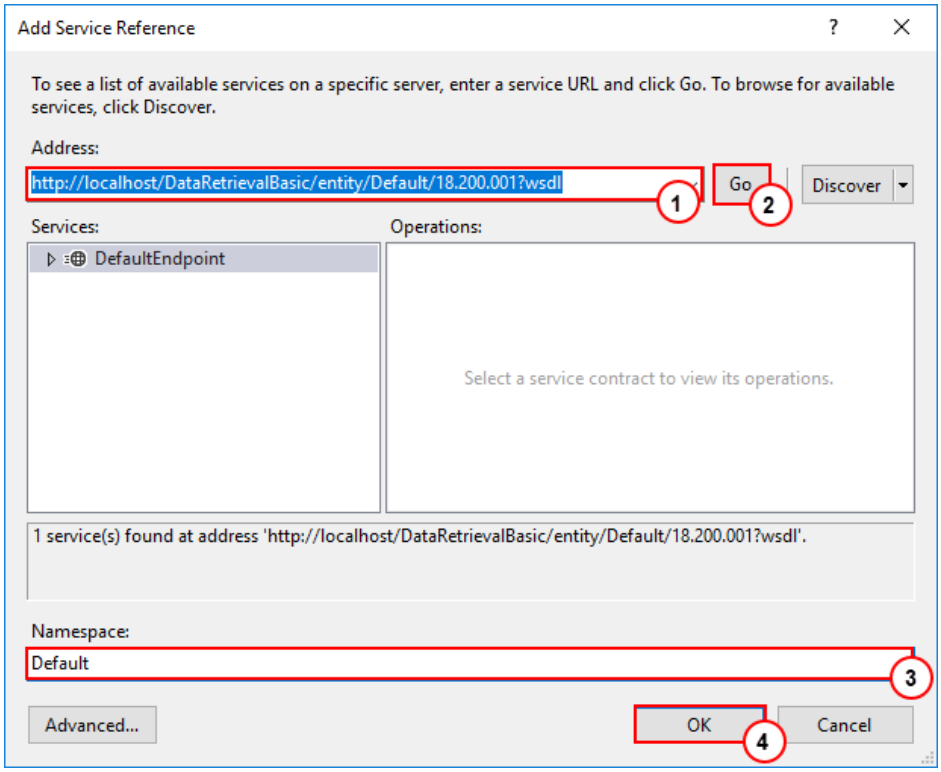


Figure: Add Service Reference dialog box

Visual Studio adds to the project the *Default* service reference in the **Connected Services** project folder, as shown in the following screenshot. Double-click the *Default* service reference to open the Object Browser and view the list of objects and methods available through the service.

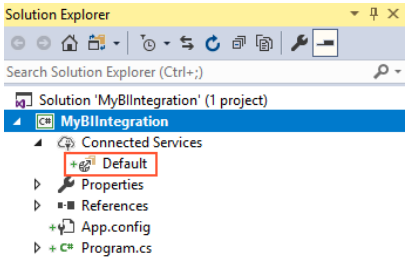


Figure: Solution Explorer

3. Modify the **app.config** file of the project as follows. Cookies are required for the client application to sign in to Acumatica ERP.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  ...
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="Acumatica" allowCookies="true"
         maxReceivedMessageSize="6553600">
        </binding>
      </basicHttpBinding>
    </bindings>
    <client>
      <endpoint address=
       "http://localhost/MyStoreInstance/entity/Default/18.200.001"
       binding="basicHttpBinding" bindingConfiguration="Acumatica"
       contract="Default.DefaultSoap" name="DefaultSoap" />
    </client>
  </system.serviceModel>
</configuration>
```

To make API calls to Acumatica ERP through HTTPS, you can use the following configuration in the **app.config** file. (Here you use the HTTPS address of the endpoint instead of the HTTP address, and use the `Transport` security mode, which indicates that API calls to Acumatica ERP are made through HTTPS.)

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  ...
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="Acumatica" allowCookies="true"
         maxReceivedMessageSize="6553600">
          <security mode="Transport" />
```

```
        </binding>
        ...
      </basicHttpBinding>
    </bindings>
    <client>
      <endpoint address=
       "https://localhost/MyStoreInstance/entity/Default/18.200.001"
       binding="basicHttpBinding" bindingConfiguration="Acumatica"
       contract="Default.DefaultSoap" name="DefaultSoap" />
    </client>
  </system.serviceModel>
</configuration>
```

4. Rebuild the project.

You have created a Visual Studio application and added to it the reference to the Acumatica ERP web service. Now you can start developing your application. For the description of the SOAP API methods, see *Contract-Based SOAP API Reference*.

Related Links

- *Contract-Based SOAP API Reference*

Working with the Screen-Based SOAP API

The screen-based SOAP API of Acumatica ERP provides the SOAP interface of the Acumatica ERP contract-based web services through which external systems can get data records from Acumatica ERP, process these records, and save new or updated records to Acumatica ERP.

In This Chapter

- *Screen-Based Web Services API*
- *API Objects Related to Acumatica ERP Forms*
- *Screen-Based API Wrapper*
- *To Generate the WSDL File of the Web Services*
- *To Import the WSDL File Into the Development Environment*
- *To Use the Screen-Based API Wrapper*

Screen-Based Web Services API

The screen-based web services API is a part of Acumatica ERP integration services, which provides integration with external data sources and third-party systems by using a SOAP interface.

External applications and systems that use the Acumatica ERP web services API can access the data managed by Acumatica ERP and the business functionality of Acumatica ERP. For example, you can integrate Acumatica ERP with eCommerce or an online store system, so that an external system pushes all information about customers, sales orders, and payments to Acumatica ERP, and Acumatica ERP provides information on the availability of stock items and processes all incoming data.

The screen-based web services API works with Acumatica ERP forms. That is, it provides API objects and methods for working with elements on Acumatica ERP forms.

To upload data to and from Acumatica ERP by using the screen-based web services API, you define the sequence of commands for the system to work with elements on an Acumatica ERP form. This sequence of commands reflects the sequence of actions to be executed for a data record as if the record is being manipulated by a user through an Acumatica ERP form. That is, when you enter data into the system manually, you perform a sequence of actions. You open the needed data entry form and start entering data. As you add a new record, you use the UI elements one by one—you type text, select values from combo boxes, clear or select check boxes, and click buttons. In the sequence of commands for the web services, you compose exactly the same sequence of actions by specifying a command for each user action on the form. For more information on the commands you can use, see *Working with Commands of the Screen-Based SOAP API*.

This sequence of commands is similar to the sequence of commands you configure when you create import and export scenarios. You can find more information on

import and export scenarios in *Configuring Import Scenarios* and *Configuring Export Scenarios*.

To use screen-based web services API in your application, you should generate the WSDL file of the web service, as described in *To Generate the WSDL File of the Web Services*, and import this file to your development project as described in *To Import the WSDL File Into the Development Environment*. After that, you can start developing your application. You can find the description of the API methods in *Screen-Based SOAP API Reference*.

You can find more details on screen-based web services API and examples of use of the API in the I200 Screen-Based Web Services training course.

Related Links

- *API Objects Related to Acumatica ERP Forms*
- *Screen-Based SOAP API Reference*

API Objects Related to Acumatica ERP Forms

The main object, which provides access to all other objects and methods of the Acumatica ERP web services API, is a `Screen` object. By using the methods of a `Screen` object, you can log in to Acumatica ERP and retrieve, insert, update, and delete data. You can also use the methods to perform any actions that are exposed by Acumatica ERP forms available through the web service.

Screen Object

After you have logged in to Acumatica ERP by using the web services API, you can access data on Acumatica ERP forms available through the web service. The `Screen` class provides the same set of API methods for working with all Acumatica ERP forms available through the service. You can find out which form a method accesses by noting the prefix in the name of the method, which is the form ID. For example, the `Export()` method that you use to export data from the *Stock Items* (IN202500) form is `IN202500Export()`.

Content Object

To get the description of the structure (schema) of a form, you should use the `GetSchema()` method of the `Screen` object. This method is specific for each Acumatica ERP form, and you should use the method with the ID of the needed form in the prefix of the method name. The method returns the schema of the form as the corresponding `Content` object, which is specific for each form. For example, to get the schema of the *Stock Items* form, you should call the `IN202500GetSchema()` method of the `Screen` object. You will receive the result as a `IN202500Content` object, as the following code shows.

```
Screen context = new Screen();
...
IN202500Content stockItemsSchema = context.IN202500GetSchema();
```

Command Object

By using subobjects of a `Content` object, you configure the sequence of commands that should be executed during data import, data export, or data processing through the web service. You configure the sequence of commands inside an array of objects of the `Command` type. When you are reflecting the selection of an element on a form in the sequence of commands, you have to select the object of the Acumatica ERP form whose element you want to access. The objects that are available inside a `Content` object have names that are similar to the names of the UI elements that you see on the form and have the ID of the form as a prefix.

For example, to be able to select the **Item Class** element, which is located in the **Item Defaults** group on the **General Settings** tab of the *Stock Items* form (shown in the following screenshot), you would select the `GeneralSettingsItemDefaults` property of the `IN202500Content` object. This property provides access to the `IN202500GeneralSettingsItemDefaults` object, which includes the `ItemClass` property.

Each element on an Acumatica ERP form (such as a text box, combo box, or table column) is associated with a particular web services API class and is available through corresponding property of this class. The property has a similar name to that of the corresponding box on the form, such as `ItemClass`, as the following screenshot shows.

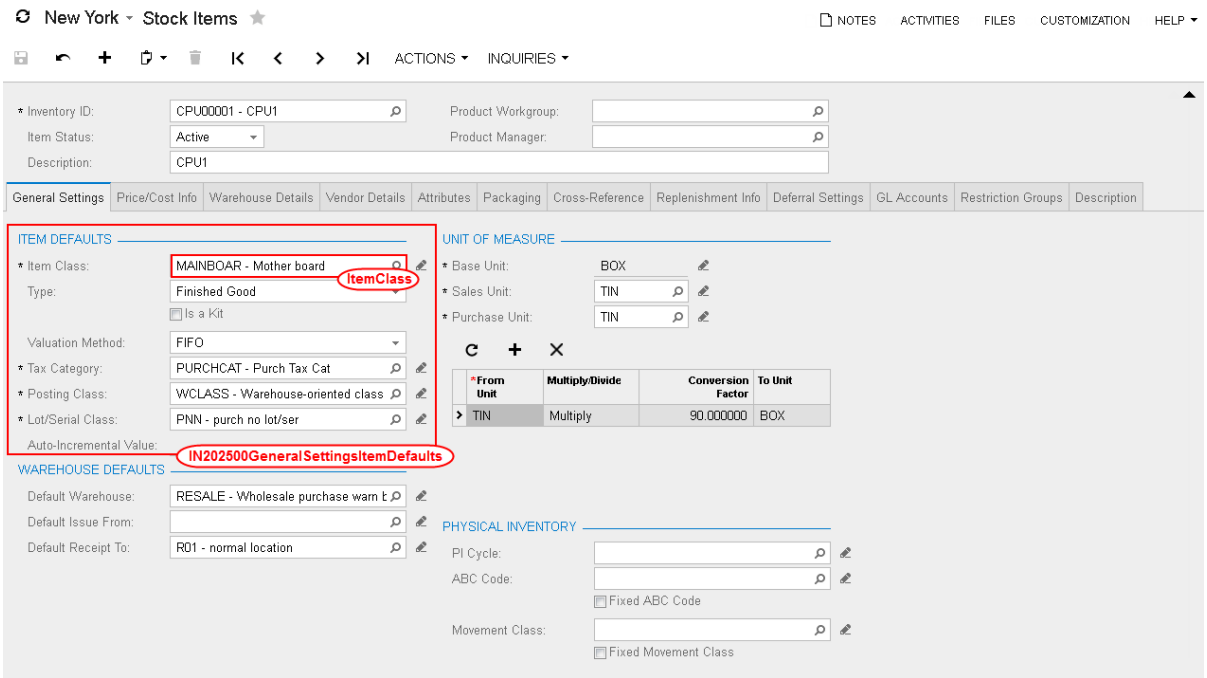


Figure: API object on an Acumatica ERP form

The following code shows an example of configuring a list of commands for the *Stock Items* form inside an array of `Command` objects.

```
//stockItemsSchema is a IN202500Content object
var commands = new Command[]
```

```
{
    stockItemsSchema.StockItemSummary.ServiceCommands.EveryInventoryID,
    stockItemsSchema.StockItemSummary.InventoryID,
    stockItemsSchema.StockItemSummary.Description,
    stockItemsSchema.GeneralSettingsItemDefaults.ItemClass,
    stockItemsSchema.GeneralSettingsUnitOfMeasureBaseUnit.BaseUnit
};
```

For more information on the commands, see *Working with Commands of the Screen-Based SOAP API*.

Actions Object

To insert an action to a sequence of commands, such as clicking a button, you use the corresponding property of the special API object `Actions` (which has a prefix with the ID of the form in the name). The `Actions` object is available through the `Actions` property of the `Content` object that corresponds to the form. The properties of the `Actions` object have names that are similar to the names of corresponding buttons on the form, such as `Delete`.

You can view the classes available through the web services API by using Object Browser in Visual Studio.

Related Links

- *Working with Commands of the Screen-Based SOAP API*

Screen-Based API Wrapper

Because of the connection of the screen-based SOAP API with Acumatica ERP forms, the applications that are developed based on this API are sensitive to the UI changes in the system. That is, any changes made to the UI after the application is created require the application to be updated and recompiled. If you want your application to not depend on the UI changes in the system, you should use the screen-based API wrapper, which is described in this topic.

How a Client Application Based on the Screen-Based API Works

A client application that uses the screen-based web services API includes the WSDL description of the service, which contains the API elements that the application can use to work with the service. The API elements have names that are similar to the names of the elements in the UI of Acumatica ERP. For example, the **Customer ID** element of the *Customers* (AR303000) form corresponds to the `Customer.CustomerSummary.CustomerID` property.

When the application calls the `Screen.GetSchema()` method, it retrieves from Acumatica ERP the schema of an Acumatica ERP form. The schema of the form is a `Content` object, which defines the correspondence between the API elements and the internal data fields that are used for operations with data by Acumatica ERP. If something has been changed in the schema of an Acumatica ERP form, the `Content` object that is returned by the `Screen.GetSchema()` method contains a different correspondence between the API elements

and internal data fields than the correspondence for which the application is compiled, and the application fails.

For example, suppose that a client application requests the customer ID by the `Customer.CustomerSummary.CustomerID` property. Suppose also that in an update of Acumatica ERP, the **Customer ID** element of the *Customers* form was renamed to **Customer**, and therefore it should be requested by using the `Customer.CustomerSummary.Customer` property from the API. The client application requests the customer ID by using the `Customer.CustomerSummary.CustomerID` property and fails.

What the Screen-Based API Wrapper Is

The screen-based API wrapper is a special wrapper designed to prevent the UI changes in the system from causing application failure. The wrapper works with any changes in the schema of an Acumatica ERP form. That is, the wrapper makes the application work regardless of the changes in the names of UI elements and the changes in the internal names of data fields and objects.

The wrapper is distributed as the **PX.Soap.dll** file, which is installed automatically during Acumatica ERP installation. You can find the **PX.Soap.dll** file in the **ScreenBasedAPIWrapper** folder of your Acumatica ERP installation folder.

The `PX.Soap` library, which the wrapper provides, includes the `Helper.GetSchema()` method, which you should use instead of the `Screen.GetSchema()` method of the screen-based web services API. For information on how to use the screen-based API wrapper, see *To Use the Screen-Based API Wrapper*.

How the Screen-Based API Wrapper Works

When the client application is executed for the first time and requests the schema of an Acumatica ERP form by using the `Helper.GetSchema()` method, the wrapper requests the schema from the Acumatica ERP screen-based web service by using the `Screen.GetSchema()` method of the screen-based API. The web service interacts with the Acumatica ERP import and export engine and returns the current schema of the form. The wrapper saves the schema in an XML file and returns the schema to the client application as a `Content` object. The client application uses this schema to work with Acumatica ERP.

Instead of the `Helper.GetSchema()` method you can use the `Helper.ReuseStoredSchema()` method to upload a schema that was saved earlier by the wrapper.

The following diagram illustrates the way the screen-based API wrapper works during the first execution of a client application.

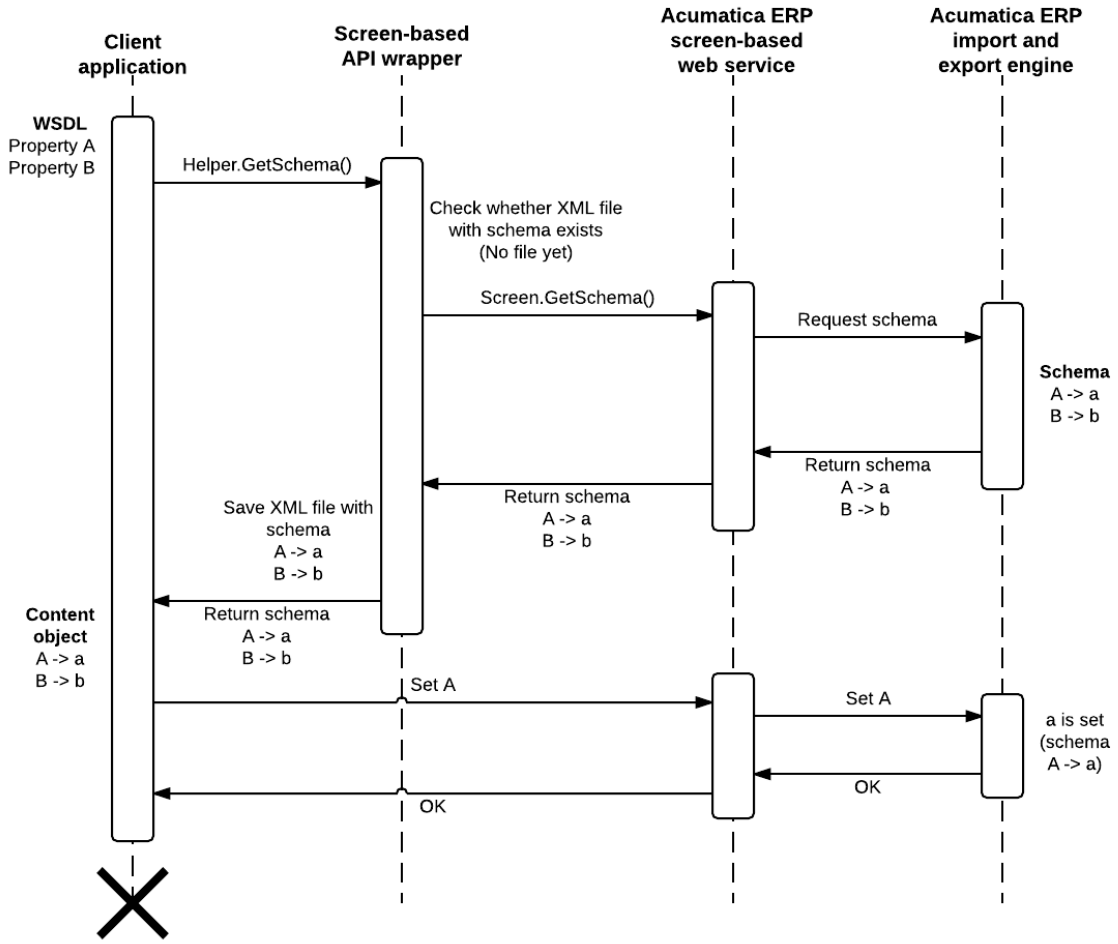


Figure: First execution of an application

When the client application is executed for the second time and all subsequent times and it requests the schema of an Acumatica ERP form by using the `Helper.GetSchema()` method, the wrapper retrieves the XML schema that is saved locally and submits this schema to the Acumatica ERP screen-based web service by using the `Screen.SetSchema()` method of the screen-based API. The web service interacts with the Acumatica ERP import and export engine, which replaces the current schema of the form that is stored on the server with the one that was saved locally. The wrapper returns the schema that was saved locally to the client application. The client application uses the local schema to work with Acumatica ERP.

The schema that is submitted to Acumatica ERP by using the screen-based API wrapper is used on the server during the current session and is discarded after the end of the session.

The following diagram illustrates the way the screen-based API wrapper works during the second execution and all subsequent executions of a client application.

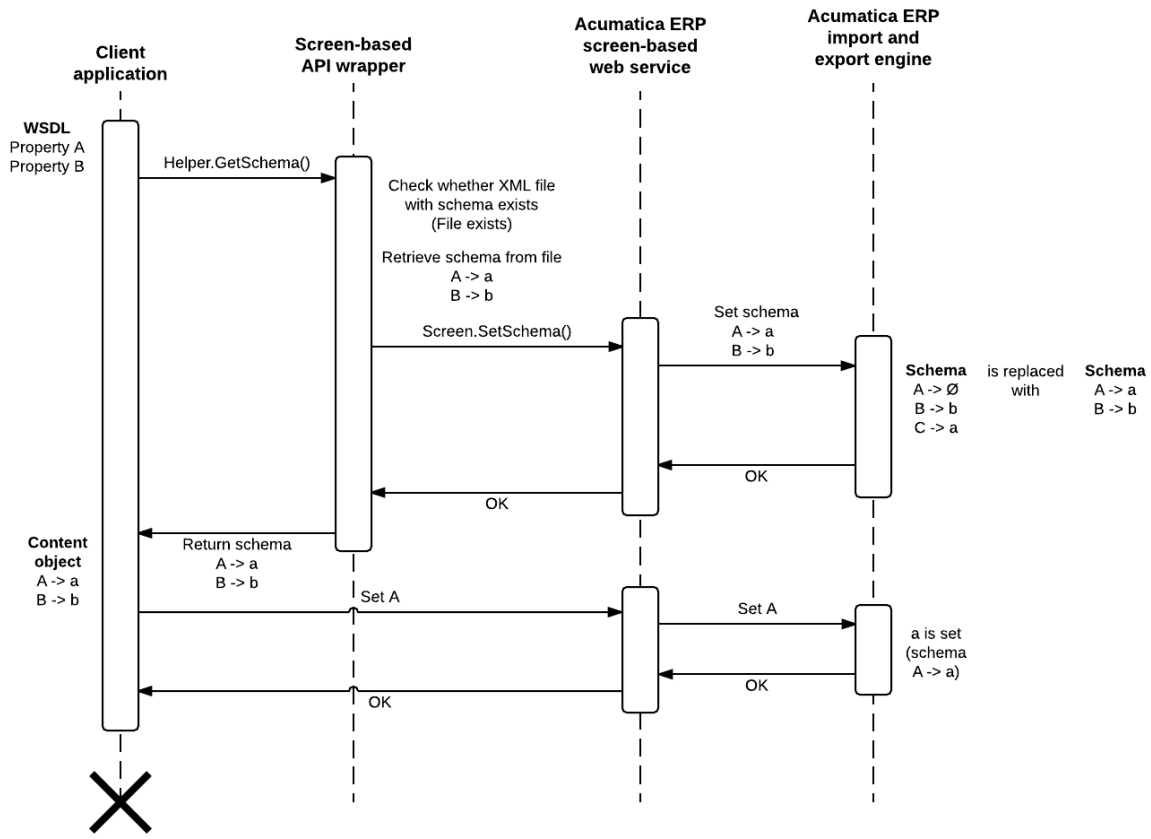


Figure: Subsequent executions of the application

You should distribute the XML file with the schema along with your client application. If the wrapper has not found the XML file, it requests the new schema.

The name of the XML file with the schema contains a hash code that depends on the WSDL description. If you update the WSDL description in your application, the wrapper works in the same way as it did during the first execution of the application and creates a new XML file with the schema.

Related Links

- *To Use the Screen-Based API Wrapper*

To Generate the WSDL File of the Web Services

The WSDL description of the Acumatica ERP screen-based web services API contains the descriptions of the API objects and methods that you can use to access Acumatica ERP forms.

Because of the connection of the API with Acumatica ERP forms, each generated WSDL description of this API reflects the current state of the system. That is, the WSDL description does not include any changes made to the system after the WSDL file was generated, and each time you change the system, you should regenerate the WSDL description and update your application accordingly.

You can prevent breaking changes in your application and omit regeneration of the WSDL description for each change in the system by using the screen-based API wrapper. For details on how to use the wrapper, see *To Use the Screen-Based API Wrapper*.

For example, suppose that you generated a WSDL description of the web service that contains the definition of the `CustomerID` property, which corresponds to the **Customer ID** element on the *Customers* (AR303000) form. Further, suppose that you changed the name of the **Customer ID** element to **Customer Identifier** in a customization project. To access the **Customer Identifier** element on the form through the screen-based web services API, you need to regenerate the WSDL description so that it contains the definition of the `CustomerIdentifier` property that corresponds to the **Customer Identifier** element, and update your application accordingly.

You can generate the WSDL file of an Acumatica ERP web service in one of the following ways.

To Generate a WSDL File for One Acumatica ERP Form

If your application needs to work with only one Acumatica ERP form, you can generate the web service that provides access to only this form.

To generate a WSDL file for a form, on the title bar of this form, click **Tools > Web Service** in the UI. This opens the page that contains the description of the web service, with the following URL: `http(s)://<ApplicationPath>/Soap/<FormID>.asmx`. In this URL, `<ApplicationPath>` is replaced with the actual URL of your application, and `<FormID>` is replaced with the ID of the form. For example, suppose that you generated a web service for the *Customers* form of the *WebServiceAPITest* application, which is accessed under a secure connection and is running on a local computer. The URL of this service is `https://localhost/WebServiceAPITest/Soap/AR303000.asmx`.

To Generate a WSDL File for Multiple Acumatica ERP Forms

If your application needs to work with multiple Acumatica ERP forms, you can generate one web service that provides access to all needed forms.

To generate a WSDL file for multiple forms, on the *Web Services* (SM207040) form, you should do the following:

1. Type the ID of the web service in the **Service ID** box of the Summary area of the form. This ID will be used in the URL of the generated service.
2. Select the Acumatica ERP forms that your application needs to use on the left pane of the form, and click **Add to Grid** for each form.

3. Select the types of integration these forms should provide (which can include importing data, exporting data, and submitting data) by selecting the appropriate check boxes (any combination of **Import**, **Export**, or **Submit**) for corresponding rows on the right pane of the form.

4. Click **Generate**.

The modules that expose the selected forms should be properly configured and the forms should open in a web browser. If a form could not be opened in a web browser, the web service definition will not be generated for this form.

After the web service is generated, you can view the WSDL description by clicking **View Generated** on the form toolbar. This opens the page that contains the description of the web service with the following URL: `http(s)://<ApplicationPath>/Soap/<ServiceID>.asmx`. In this URL, `<ApplicationPath>` is replaced with the actual URL of your application, and `<ServiceID>` is replaced with the ID of the service that you specified in the **Service ID** box when configuring the service. For example, suppose that you generated a web service for multiple forms with the `MYSTORE` service ID for the *WebServiceAPITest* application, which is accessed under a secure connection and is running on a local computer. The URL of this service is `https://localhost/WebServiceAPITest/Soap/MYSTORE.asmx`.

To Import the WSDL File Into the Development Environment

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

The procedure below is described based on an example of the creation of a console project with the *MyBIIntegrationSBAPI* name that uses the *MYBIINTEGRATION* web service. You can create another type of Visual Studio project with another name and use any other web service define on the *Web Services* (SM207040) form in this procedure.

In this topic, you will find instructions on how to implement the proxy classes by using Visual Studio 2019.

Importing the WSDL File into a Visual Studio Project

To import the WSDL description into your development environment, proceed as follows:

1. Open Visual Studio, and create a new Visual C# console application with the *MyBIIntegrationSBAPI* name and the *.NET Framework 4.8* target framework.

To create a console application, click **File > New > Project** on the main menu of Visual Studio. In the **Create a new project** dialog box, which appears, select the *Console App (.Net Framework)* C# template and click **Next**. In the **Configure**

your new project dialog box, specify the name and location of the solution, select the *.NET Framework 4.8* framework version, and click **Create**.

The MyBIIntegration application that you have created contains the **Program.cs** file with the `Program` class.

2. Add to the project a reference to the Acumatica ERP web service as follows:

a. On the main menu of Visual Studio, click **Project > Add Service Reference**.

b. In the **Add Service Reference** dialog box, which appears, click **Advanced**.

c. In the **Service Reference Settings** dialog box, which appears, click **Add Web Reference**.

d. In the **URL** box of the **Add Web Reference** dialog box, which appears, specify the URL of the *MYBIINTEGRATION* web service (see item 1 in the following screenshot).

To copy the URL of the service, on the *Web Services* (SM207040) form, select *MYBIINTEGRATION* in the **Service ID** box, click **Generate** and then **View Generated** on the form toolbar, and copy the URL from the address line in the browser for the page that opens. In this example, the URL is *http://localhost/MyStoreInstance/Soap/MYBIINTEGRATION.asmx*.

e. Click **Go** (2) to make Visual Studio connect to the web service.

If your Acumatica ERP website uses a self-signed certificate, Visual Studio displays security alert windows with warnings on the certificate. Click **Yes** in these windows to proceed.

f. In the **Web reference name** box, type `MyBIIntegration` (3). This name will be used as a namespace for the web service classes generated by Visual Studio based on the WSDL description of the service.

g. Click **Add Reference** (4) to close the dialog box and add to the project the reference to the specified service.

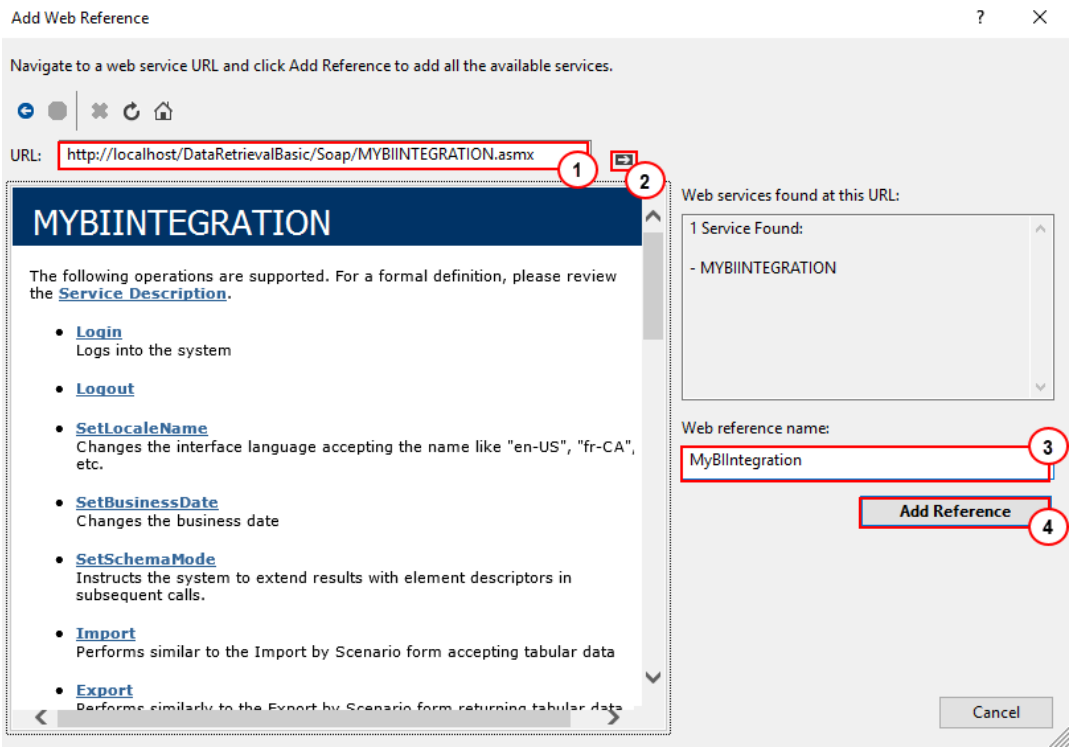


Figure: Add Web Reference dialog box

Visual Studio adds to the project the **Web References** project folder with the *MyBIIntegration* web service reference in it, as shown in the following screenshot.

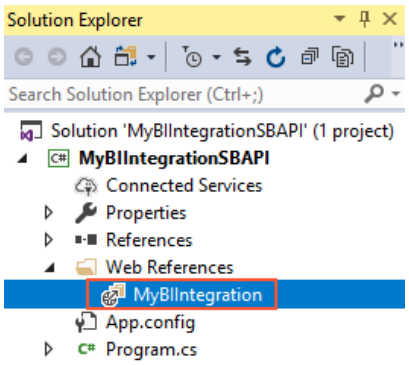


Figure: Solution Explorer

If you need to update a web reference to the Acumatica ERP web service in a Visual Studio project, you can update the WSDL description in Acumatica ERP and update the web reference in the project by right-clicking the web reference in Solution Explorer and selecting **Update Web Reference**, as shown in the following screenshot.

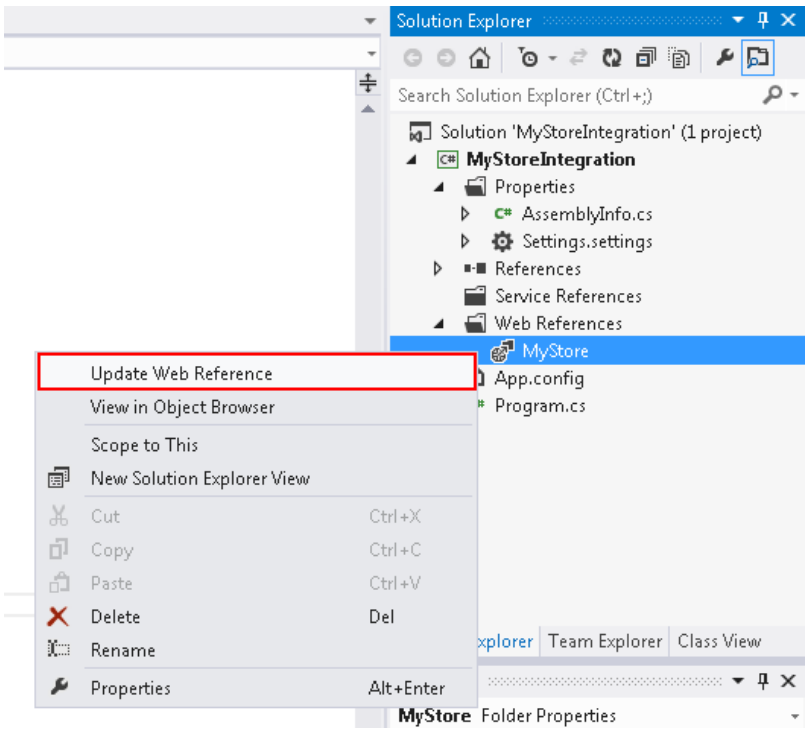


Figure: Update Web Reference menu item

3. Rebuild the project.

To Use the Screen-Based API Wrapper

The screen-based SOAP API depends on the user interface of Acumatica ERP forms. That is, each generated WSDL description of this API includes the API elements that correspond to the current names of UI elements of the system. Therefore, if any changes have been made to the user interface of the system after the WSDL file was generated, the WSDL description will not include these changes, and the client application that uses this WSDL will fail when it works with the system.

To prevent application failures and omit the regeneration of the WSDL description for each change of the user interface of the system, you can use the screen-based API wrapper, as described in this topic. You can find more information on the screen-based API wrapper in *Screen-Based API Wrapper*.

To Use the Screen-Based Web Services API Wrapper in a Client Application

1. Add to your project a reference to **PX.Soap.dll**.

   The **PX.Soap.dll** file is installed automatically during Acumatica ERP installation. You can find this file in the **ScreenBasedAPIWrapper** folder of your Acumatica ERP installation folder.

2. Use the `Helper.GetSchema()` method from the `PX.Soap` library in the code of your application instead of the corresponding `Screen.GetSchema()` method to obtain the schema of each form. For example, the following code retrieves the schema of the *Customers* (AR303000) form by using the screen-based API wrapper.

```
Screen context = new Screen();
context.CookieContainer = new System.Net.CookieContainer();
context.Url = Properties.Settings.Default.MyStoreIntegration_MyStore_Screen;
context.Login
(
  Properties.Settings.Default.Login,
  Properties.Settings.Default.Password
);

AR303000Content custSchema =
  PX.Soap.Helper.GetSchema<AR303000Content>(context);
```

3. Work with the retrieved `Content` object by using the standard screen-based web services API methods.

Related Links

- *Screen-Based API Wrapper*
- *To Update a Client Application that Uses Screen-Based Web Services*

Working with Commands of the Screen-Based SOAP API

To upload data to and from Acumatica ERP by using the screen-based SOAP API, you should define the sequence of commands for the system as it works with elements on an Acumatica ERP form. This sequence of commands reflects the sequence of actions to be executed for a data record as if the record is being manipulated by a user through an Acumatica ERP form.

In This Chapter

- *Commands for Retrieving the Values of Elements*
- *Selection of a Group of Records for Export*
- *Commands for Setting the Values of Elements*
- *Commands for Clicking Buttons on a Form*
- *Commands for Adding Detail Lines*
- *Commands for Pop-Up Dialog Boxes and Pop-Up Forms*
- *Commands for Pop-Up Panels*
- *Commands for Record Searching: Filter Service Command*
- *Commands for Record Searching: Key Command*
- *Commands for Record Searching: Custom Field*
- *Commands That Require a Commit*
- *Commands for Working with Attachments*
- *Commands for Working with Multi-Language Fields*

Commands for Retrieving the Values of Elements

You configure the sequence of commands that should be executed during data import, data export, or data processing through the web service by using an array of objects of the `Command` type.

As you specify this sequence of commands, when you need to reflect obtaining the value of an element on a form, you should use a `Field` object. To specify the element whose value you need to obtain, you can do one of the following:

- Select the needed `Field` object from the subobjects of the `Field` type of a `Content` object that corresponds to the needed Acumatica ERP form
- Create an object of the `Field` type and specify its properties

Both of these ways are described in detail in the following sections of this topic.

Selection of the Fields Available in a Content Object

If you want to obtain the value of an element on an Acumatica ERP form, you can select the needed `Field` object from the subobjects of the `Field` type of the `Content` object that corresponds to the form. For example, if you want to export the values of the **Inventory ID** and **Description** elements on the *Stock Items* (IN202500) form, you can use the following code.

```
//stockItemsSchema is an IN202500Content object
var commands = new Command[]
{
   ...
   stockItemsSchema.StockItemSummary.InventoryID,
   stockItemsSchema.StockItemSummary.Description,
   ...
};
```

Field Object Creation

You can retrieve the values of not only the fields that are available on the Acumatica ERP form, but also the data fields of the data access classes (DACs) underlying the form. Some of these data fields are not available directly through the elements of the form. That is, you cannot select the needed `Field` object among the subobjects of the `Content` object.

If you want to retrieve the value of an element that is not available on the form, you can create a `Field` object and specify its properties so that it specifies the needed data field of the DAC. You should specify the name of the object that corresponds to the needed DAC as the `ObjectName` and type the name of the data field in `FieldName`. The following code illustrates the creation of a `Field` object for the `LastModifiedDateTime` data field (which specifies the date and time when a record was modified) that is available through the DAC underlying the `StockItemSummary` object of the *Stock Items* form.

To find the names of the data fields that belong to DACs, you should read the applicable Acumatica ERP code. You can find the source code of Acumatica ERP on the *Source Code* (SM204570) form or in <ApplicationFolder>\App_Data\CodeRepository\PX.Objects\, where <ApplicationFolder> is replaced with the path to the folder of the Acumatica ERP application instance. You can learn the details of working with the source code of Acumatica ERP in the T300 Acumatica Customization Platform training course.

```
//stockItemsSchema is an IN202500Content object
var commands = new Command[]
{
    ...
    new Field
    {
         ObjectName = stockItemsSchema.StockItemSummary.InventoryID.ObjectName,
         FieldName = "LastModifiedDateTime"
    },
    ...
```

```
};
```

You can create `Field` objects for the elements that are available on a form. If you want to create a `Field` object for an element available on the form, set the `ObjectName` property to the `ObjectName` property of the needed subobject of the `Field` type of a `Content` object, and set the `FieldName` property to the `FieldName` property of this `Field` object. The following code illustrates the creation of a `Field` object for the `InventoryID` field of the *Stock Items* form.

```
new Field
{
     ObjectName = stockItemsSchema.StockItemSummary.InventoryID.ObjectName,
     FieldName = stockItemsSchema.StockItemSummary.InventoryID.FieldName
}
```

Selection of a Group of Records for Export

Each record in the system is identified by the values of the key elements on the applicable Acumatica ERP form. For example, a record on the *Stock Items* (IN202500) form is identified by the value of the **Inventory ID** key element. Key elements are available through the Summary object of a form. You can use the key element or elements of the form to select all records of the form for export. The other way to specify a group of records for export is to use the elements of the Summary area of the form in custom filters. Both ways of selecting records are described in detail below.

Export of All Records from a Form

If you want to export every record from an Acumatica ERP form, in the array of `Command` objects that you pass to the `Export()` method, you should insert the service command of the `EveryValue` type for the corresponding key element on the form. The `EveryValue` service command is available through the `ServiceCommands` subobject of the Summary object of the `Content` object that corresponds to the form. This service command specifies that every record of the specific type should be processed during export.

For example, if you want to export all stock item records available in the system, you should insert the `EveryInventoryID` service command, as the following code shows.

```
//stockItemsSchema is an IN202500Content object
var commands = new Command[]
{
   stockItemsSchema.StockItemSummary.ServiceCommands.EveryInventoryID,
   ...
};
```

Export of a Group of Records from a Form

You can filter the data available in the Acumatica ERP database to select the records for export. For example, you can configure the system to export only the records that have a particular status.

To define a filter for the data being exported, you should create an array of `Filter` objects and add the needed filters to it. To define a filter, you should specify:

- The UI element whose value should be used for filtering (in the `Field` property of the `Filter` object).
- The value or values with which the value of the element should be compared (in the `Value` property or `Value` and `Value2` properties of the `Filter` object).
- The condition of comparison (in the `Condition` property of the `Filter` object).

If you define multiple filters in the array, you should also specify the logical operator (either `And` or `Or`) that defines how to apply these filters. If filters are passed to an `Export()` method, during export, the system selects only the records that conform to the specified condition (or conditions) and exports only these records.

For example, suppose that you want to export only the stock item records that have the *Active* status. In this case, you can specify the filtering condition that the **Item Status** element should be equal to *Active*, as the following code shows. During export, the system processes the records that match the filtering conditions and exports only the records with the *Active* status.

```
var filters = new Filter[]
{
    new Filter
    {
         Field = stockItemsSchema.StockItemSummary.ItemStatus,
         Condition = FilterCondition.Equals,
         Value = "Active",
    }
};
```

For filtering records, you can use either the data fields of the Summary object of the form from which you are exporting data, or the data fields of the data access class (DAC) underlying the Summary object. (In Acumatica Framework, this DAC is called the main DAC of the primary data view.) If a data field of the DAC is not available directly through the elements of the Summary object, you cannot select the needed `Field` object among the subobjects of a `Content` object, as the previous code example shows for the `ItemStatus` data field. Instead, you should create a new `Field` object and specify its properties as follows: Specify the name of the Summary object as `ObjectName` (that is, the object name that corresponds to the object to which the key data field belongs), and type the name of the data field as `FieldName` in the code directly.

The following example shows filtering by the `LastModifiedDateTime` data field (which specifies the date and time when a record was modified) that is available through the DAC underlying the `StockItemSummary` object of the *Stock Items* form.

```
new Filter
{
    Field = new Field
    {
         ObjectName = stockItemsSchema.StockItemSummary.InventoryID.ObjectName,
```

```
        FieldName = "LastModifiedDateTime"
    },
    Condition = FilterCondition.Greater,
    Value = DateTime.Now.AddMonths(-1).ToLongDateString()
}
```

Commands for Setting the Values of Elements

As you specify the sequence of commands in an array of `Command` objects, when you need to specify the value of an element on a form, you should use `Value` commands.

To set the value of an element on a form, you should do the following:

1. Create a `Value` object.
2. Specify the value of the element on the form in the `Value` property of the created `Value` object.
3. Specify the element on the form whose value should be set by using the `LinkedCommand` property of the `Value` object.

The following code illustrates setting the value of the **Customer Name** element on the *Customers* (AR303000) form.

```
//custSchema is an AR303000Content object
var commands = new Command[]
{
    ...
    new Value
    {
        Value = "John Good",
        LinkedCommand = custSchema.CustomerSummary.CustomerName
    },
    ...
}
```

Commands for Clicking Buttons on a Form

As you specify the sequence of commands in an array of `Command` objects, when you need to reflect the clicking of a button on a form (such as clicking **Save**, **Delete**, or **Release** to perform the action on a document), you should use the corresponding `Action` command. Actions are available for all buttons on the form.

In an array of `Command` objects, to use an `Action` command, you have to select the needed action in the `Actions` subobject of the `Content` object that corresponds to the form. Actions have names that are similar to the names of the buttons on the form.

The following code reflects the clicking of the **Save** button on the *Customers* (AR303000) form in a command.

```
//custSchema is an AR303000Content object
var commands = new Command[]
{
    ...
    custSchema.Actions.Save,
    ...
};
```

Commands for Adding Detail Lines

When you need to add a detail line to an Acumatica ERP form, you can use one of the following approaches:

- Add detail lines one by one on the Details tab of the form. For example, on the *Sales Orders* (SO301000) form, you can click **Add Row** on the **Document Details** tab and specify the values of the elements of each detail line.
- Add detail lines by using a pop-up panel. For example, on the *Shipments* (SO302000) form, you can use the **Add Sales Order** pop-up panel, which is opened when you click **Add Order** on the table toolbar of the **Document Details** tab.

In this topic, you will find the description of the `NewRow` command, which imitates the first approach listed above in the screen-based API. The second approach is described in *Commands for Pop-Up Panels*.

NewRow Service Command

When you are specifying the sequence of commands in an array of `Command` objects for a processing method and you need to add a new detail line to a document, you should use commands as follows:

1. To add a new row, use the `NewRow` service command, which is an available service command of the Details subobject of a `Content` object.
2. To specify the values of the elements of the created row, use the `Value` commands corresponding to the elements.

The following code shows an example of an order line being added to a sales order.

```
//orderSchema is an SO301000Content object
var commands = new Command[]
{
    ...
    orderSchema.DocumentDetails.ServiceCommands.NewRow,
    new Value
    {
        Value = "AALEGO500",
        LinkedCommand = orderSchema.DocumentDetails.InventoryID
    },
    new Value
```

```
    {
        Value = "10.0",
        LinkedCommand = orderSchema.DocumentDetails.Quantity
    },
    new Value
    {
        Value = firstItemUOM,
        LinkedCommand = orderSchema.DocumentDetails.UOM
    },
    ...
}
```

Commands for Pop-Up Dialog Boxes and Pop-Up Forms

In this topic, you will learn how to enter data to pop-up dialog boxes and pop-up forms by using the screen-based SOAP API.

Pop-Up Dialog Boxes

When you update specific fields on some forms under certain circumstances, the system displays pop-up dialog boxes where you need to respond to a question (by clicking a button) in order to proceed. For example, when you update the **Customer Class** value on the *Customers* (AR303000) form for an existing customer, the system displays a **Warning** dialog box with the text *Please confirm if you want to update current customer settings with the customer class defaults. Otherwise, original settings will be preserved.* and the **Yes** and **No** buttons. You should click **Yes** to proceed with changing the customer class.

When you are specifying the sequence of commands in an array of `Command` objects for a processing method and you need to specify an answer to a question that would appear in a pop-up dialog box if the data was being entered manually, you should create a `Value` command and set its properties as follows:

- In the `Value` property, specify the answer that you select in the dialog box during manual entry of a record.
- In the `LinkedCommand` property, use a `DialogAnswer` service command, which is available through the `ServiceCommands` subobject of an object that invokes the appearance of the pop-up dialog box.

You should insert this `Value` command directly before the field that causes the appearance of the dialog box.

The following code shows how you would update the customer class in an existing customer record on the *Customers* form.

```
//custSchema is an AR303000Content object
var commands = new Command[]
{
    ...
    new Value
    {
        Value = "Yes",
```

```
        LinkedCommand = custSchema.CustomerSummary.ServiceCommands.DialogAnswer
    },
    new Value
    {
        Value = "INTL",
        LinkedCommand = custSchema.GeneralInfoFinancialSettings.CustomerClass
    },
    ...
};
```

Pop-Up Forms

When you click specific buttons on some forms, the system opens a pop-up window with another Acumatica ERP form where you can specify or edit the values of elements as needed. For example, if you click **Add Contact** on the **Contacts** tab of the *Customers* form, the system displays the *Contacts* (CR302000) form.

Do not confuse a situation when the system opens a pop-up window that contains an Acumatica ERP form with a situation when the system opens a pop-up panel (where you can specify needed settings but no Acumatica ERP form is shown). A pop-up window that contains a form has an address line in the browser where you can see the ID of the form. A pop-up panel looks like a dialog box and does not have an address line.

When you are specifying a sequence of commands in an array of `Command` objects for a processing method and you need to reflect the setting of values of elements of a pop-up form in these commands, you should perform the following steps:

1. Call an action that invokes a pop-up form as follows:
    a. By using the `GetSchema()` method of the `PX.Soap.Helper` class, get the `Content` object that corresponds to the form that invokes a pop-up form.
    b. Specify the command that invokes a pop-up form in the sequence of commands by using the corresponding `Action` command.
    c. Submit this sequence of commands to the form that invokes the pop-up form by using the corresponding `Submit()` method.
2. Specify the values on the pop-up form as follows:
    a. By using the `GetSchema()` method of the `PX.Soap.Helper` class, get the `Content` object that corresponds to the form that appears as a pop-up.
    b. Specify the list of commands that specifies the values of needed elements of the pop-up form.
    c. Add the `Save` action to the list of commands.
    d. Submit this sequence of commands to the pop-up form by using the corresponding `Submit()` method.

The following code illustrates the setting of the values of the *Contacts* form, which appears as a pop-up after the user clicks **Add Contact** on the **Contacts** tab of the *Customers* form.

```
//context is a Screen object
//custSchema is an AR303000Content object
var commands = new Command[]
{
    new Value
    {
        Value = customerID,
        LinkedCommand = custSchema.CustomerSummary.CustomerID
    },

    custSchema.Actions.NewContact
};
context.AR303000Submit(commands);

//contSchema is a CR302000Content object
commands = new Command[]
{
    new Value
    {
        Value = "Green",
        LinkedCommand = contSchema.DetailsSummary.LastName
    },
    contSchema.Actions.Save,
};
context.CR302000Submit(commands);
```

Commands for Pop-Up Panels

In some instances, when you click a specific button on some form, the system opens a pop-up panel, where you can set the values of needed elements. This pop-up panel looks like a dialog box and does not contain an Acumatica ERP form. For example, if you click **Add Order** on the table toolbar of the **Document Details** tab of the *Shipments* (SO302000) form, the system displays the **Add Sales Order** pop-up panel.

Do not confuse a situation when the system opens a pop-up panel (where you can set needed settings but no Acumatica ERP form is shown) with a situation when the system opens a pop-up window that contains an Acumatica ERP form. A pop-up window that contains a form has an address line in the browser where you can see the ID of the form. A pop-up panel looks like a dialog box and does not have an address line.

When you are specifying a sequence of commands in an array of `Command` objects for a processing method and you need to reflect the setting of values of elements on a pop-up panel in these commands, you should perform the following steps:

1. Insert the `DialogAnswer` service command of the pop-up panel object before the action that opens the panel, and set the `Commit` property to `true` for this command.
2. Specify the action that opens the pop-up panel, and set the `Commit` property of the action to `true`.
3. Specify the values of elements as needed on the pop-up panel.
4. Specify the action that closes the panel, and set the `Commit` property of the action to `true`.

For some pop-up panels, you need to specify only one action to select values from the pop-up panel. For example, you need to specify only one action if you are creating a shipment by using the **Create Shipment** action on the *Sales Orders* (SO301000) form, which displays the **Specify Shipment Parameters** pop-up panel.

The following code illustrates the setting of the values on the **Add Sales Order** pop-up panel, which appears after a user clicks **Add Order** on the toolbar of the **Document Details** tab of the *Shipments* form.

```
//shipmentSchema is a SO302000Content object
//Force a commit for the SelectSO action
var selectSOwithCommit = shipmentSchema.Actions.SelectSO;
selectSOwithCommit.Commit = true;

//Force a commit for the AddSO action
var addSOwithCommit = shipmentSchema.Actions.AddSO;
addSOwithCommit.Commit = true;

//Configure the list of commands
var commands = new Command[]
{
  ...
  //Open the Add Sales Order panel
  new Value
  {
    Value = "OK",
    LinkedCommand =
     shipmentSchema.AddSalesOrderOperation.ServiceCommands.DialogAnswer,
    Commit = true
  },
  selectSOwithCommit,

  //Specify the first order number on the Add Sales Order panel
  //and get the values of item elements
  new Value
  {
    Value = firstOrderNbr,
    LinkedCommand = shipmentSchema.AddSalesOrderOperation.OrderNbr
  },
```

```
  new Value
  {
    Value = "True",
    LinkedCommand = shipmentSchema.AddSalesOrder.Selected
  },
  shipmentSchema.AddSalesOrder.InventoryID,
  shipmentSchema.AddSalesOrder.Quantity,
  shipmentSchema.AddSalesOrder.OpenQty,
  shipmentSchema.AddSalesOrder.LineDescription,
};
//context is a Screen object
//Submit the commands to the form
var soLines = context.SO302000Submit(commands);

//Select all items of the first order for shipment
List<Command> commandList = new List<Command>();
for (int index = 0; index < soLines.Length; index++)
{
  commandList.Add(new Value
  {
    Value = index.ToString(),
    LinkedCommand = shipmentSchema.AddSalesOrder.ServiceCommands.RowNumber
  });
  commandList.Add(new Value
  {
    Value = "True",
    LinkedCommand = shipmentSchema.AddSalesOrder.Selected,
    Commit = index < soLines.Length - 1
  });
}

//Add items to the shipment
commandList.Add(addSOwithCommit);
context.SO302000Submit(commandList.ToArray());
```

Commands for Record Searching: Filter Service Command

The system uses the key element or elements on a form to find records that belong to different documents. For example, on the *Invoices and Memos* (AR301000) form, there are two key elements: **Type** and **Reference Nbr**.

If you know the values of the key element or elements of the needed record, you can select this record for update by specifying the key values in the sequence of commands that you pass to the processing method of the web services API. In the sequence of commands, you should first specify key element or elements to identify the record that you are going to update. After you have specified the values of key element or elements, you should specify the values of other elements in the order in which you would specify them on the form.

If you do not know the values of the key element or elements of the needed record, you can update records in the system by searching for them using their unique field values that you know. For example, you can identify customers by email addresses or phone numbers. To search for a record, you have to imitate the use of a column of a **Select** dialog box, declare a custom key, or declare a custom field in the sequence of commands that you pass to a processing method. In this topic, you will find a detailed description of the `Filter` service command, which imitates the use of a selector column. You can find a description of two other approaches in *Commands for Record Searching: Key Command* and *Commands for Record Searching: Custom Field*.

Filter Service Command

Selector columns on an Acumatica ERP form appear when a user clicks the Magnifier icon of the key element of the form to bring up the **Select** dialog box. Service commands for selector columns have the `Filter` prefix in their names. For example, to search for a customer record, you can use the `FilterCity`, `FilterCountry`, `FilterEmail`, and `FilterPhone1` service commands.

To use a column of a **Select** dialog box for a search, you have to do the following:

1. Create a `Field` object, and initialize its properties with the values of the properties of the key field.
2. Concatenate the `FieldName` property of this object (which is now equal to the value of the `FieldName` property of the key field) with `!` and the `FieldName` property of the needed `Filter` service command.
3. In the `Value` command in the array of `Command` objects, set the `Value` property to the value that should be used for the search and the `LinkedCommand` property to the created `Field` object.

For example, the following code searches for a customer record by email address.

```
//custSchema is an AR303000Content object
Field customerIDSelector = custSchema.CustomerSummary.CustomerID;
customerIDSelector.FieldName += "!" +
        custSchema.CustomerSummary.ServiceCommands.FilterEmail.FieldName;

var commands = new Command[]
{
    new Value
    {
        Value = "demo@gmail.com",
        LinkedCommand = customerIDSelector
    },
    ...
};
```

If you need to get the value of the field that was used for a search as a result of the processing, you should assign an initial `FieldName` to the field before getting the

value. For example, the following code shows how to get the value of the Customer ID element after you have modified the corresponding field for the search.

```
//custSchema is an AR303000Content object
//Save the initial field name of the CustomerID field
string initialCustomerIDFieldName =
custSchema.CustomerSummary.CustomerID.FieldName;

//Configure the command that searches for a customer record
//by using the FilterEmail service command
Field customerIDSelector = custSchema.CustomerSummary.CustomerID;
customerIDSelector.FieldName += "!" +
custSchema.CustomerSummary.ServiceCommands.FilterEmail.FieldName;

//Configure the list of commands
var commands = new Command[]
{
    //Search for the needed customer record
    new Value
    {
         Value = customerMainContactEmail,
         LinkedCommand = customerIDSelector
    },

    //Do the needed modifications and save changes on the form
    ...
};

//context is a Screen object
//Submit commands to the form
context.AR303000Submit(commands);

//Assign an initial field name to the CustomerID field
custSchema.CustomerSummary.CustomerID.FieldName = initialCustomerIDFieldName;

//Get the customer ID
commands = new Command[]
{
    custSchema.CustomerSummary.CustomerID
};

//Submit commands to the form
AR303000Content customer = context.AR303000Submit(commands)[0];
```

Commands for Record Searching: Key Command

To search for a record, you have to imitate the use of a column of a **Select** dialog box, declare a custom key, or declare a custom field in the sequence of commands that you pass to a processing method. In this topic, you will find a detailed description of the use of the `Key` command, which you use to declare a custom key. You can find descriptions of two other

approaches in *Commands for Record Searching: Filter Service Command* and *Commands for Record Searching: Custom Field*.

Key Command

If you specify the value of key elements on an Acumatica ERP form, the system does not change the current record in the system; instead, it searches for the record, which is identified by the values of the key elements, and selects this record. By using custom keys, you can make some elements on an Acumatica ERP form work as key elements.

You can specify custom keys to search for a record or a detail line. You can use the fields of the summary object and the detail objects, but not the fields that belong to other objects, as custom key fields. For example, you can find the needed customer record in the system by using the `CustomerName` field of the `AR303000CustomerSummary` object of the *Customers* (AR303000) form as the custom key, but you cannot find the record by using the `Email` field of the `AR303000GeneralInfoMainContact` object as the custom key.

To specify a custom key, you have to define the key by using the `Key` command as follows:

1. Create a `Key` command by using the operator `new`.
2. To specify the element that should be used as a custom key, set the `ObjectName` and `FieldName` properties of the created `Key` command to the values of the `ObjectName` and `FieldName` properties of the field corresponding to the element.
3. To specify the value of the custom key, set the `Value` property of the created `Key` command to the needed value in the format `='<Key value>'`, where you should replace `<Key value>` with the needed value of the key.

Below is an example of the `Key` command that declares the **Warehouse** column as the custom key on the **Document Details** tab of the *Sales Orders* (SO301000) form.

```
new Key
{
    ObjectName = orderSchema.DocumentDetails.Warehouse.ObjectName,
    FieldName = orderSchema.DocumentDetails.Warehouse.FieldName,
    Value = "='MAIN'"
},
```

Commands for Record Searching: Custom Field

To search for a record, you have to imitate the use of a column of a **Select** dialog box, declare a custom key, or declare a custom field in the sequence of commands that you pass to a processing method. In this topic, you will find a detailed description of the declaration of a custom field. You can find descriptions of two other approaches in *Commands for Record Searching: Filter Service Command* and *Commands for Record Searching: Key Command*.

Custom Field

Acumatica ERP does not include `Filter` service commands for all selector columns that are available on Acumatica ERP forms. To use the needed selector column for record searching, you can create a custom field as follows:

1. Initialize the properties of a `Field` object with the values of the properties of the key field.
2. Concatenate the `FieldName` property of this object (which is now equal to the value of the `FieldName` property of the key field) with `!` and the internal name of the selector column that you are going to use for the search. The internal name of the selector column is equal to the value of the `FieldName` property of the corresponding element on the form.
3. In the `Value` command in the array of `Command` objects, set the `Value` property to the value that should be used for the search and the `LinkedCommand` property to the created `Field` object.

For example, the following code searches for a sales order by order number.

```
//orderSchema is an SO301000Content object
var searchCustomerOrder = orderSchema.OrderSummary.OrderNbr;
searchCustomerOrder.FieldName +=
  "!" + orderSchema.OrderSummary.CustomerOrder.FieldName;

var commands = new Command[]
{
  new Value
  {
    Value = "SO",
    LinkedCommand = orderSchema.OrderSummary.OrderType
  },
  new Value
  {
    Value = "SO248-563-06",
    LinkedCommand = searchCustomerOrder
  },
  ...
}
```

Commands That Require a Commit

There are two types of elements on an Acumatica ERP form: elements with a commit, and elements without a commit. A commit is an action initiated by the form that, when triggered, sends the data to the server. On the server, the commit causes all data on the form to be updated, including the insertion of default values and the recalculation of the values of elements on the form.

A commit is a costly operation that causes the browser to make requests to the server and takes server time. As such, a commit is invoked for only a limited number of elements: mainly the key elements and the elements the other elements depend on.

In a Visual Studio project, if you specify the value of an element for which the system performs a commit by using a `Value` command, the `Commit` property of the `LinkedCommand` property (which specifies this element) is automatically set to `true`. You can check the value

of the `Commit` property when you are debugging an application if you insert a breakpoint in the code after an array of commands has been configured. In the following screenshot, you can see that for the command that sets the value of the **Customer ID** element on the *Customers* (AR303000) form, the `Commit` property of the `LinkedCommand` is set to `true`. (The **Customer ID** element has the internal field name `AcctCD`.)

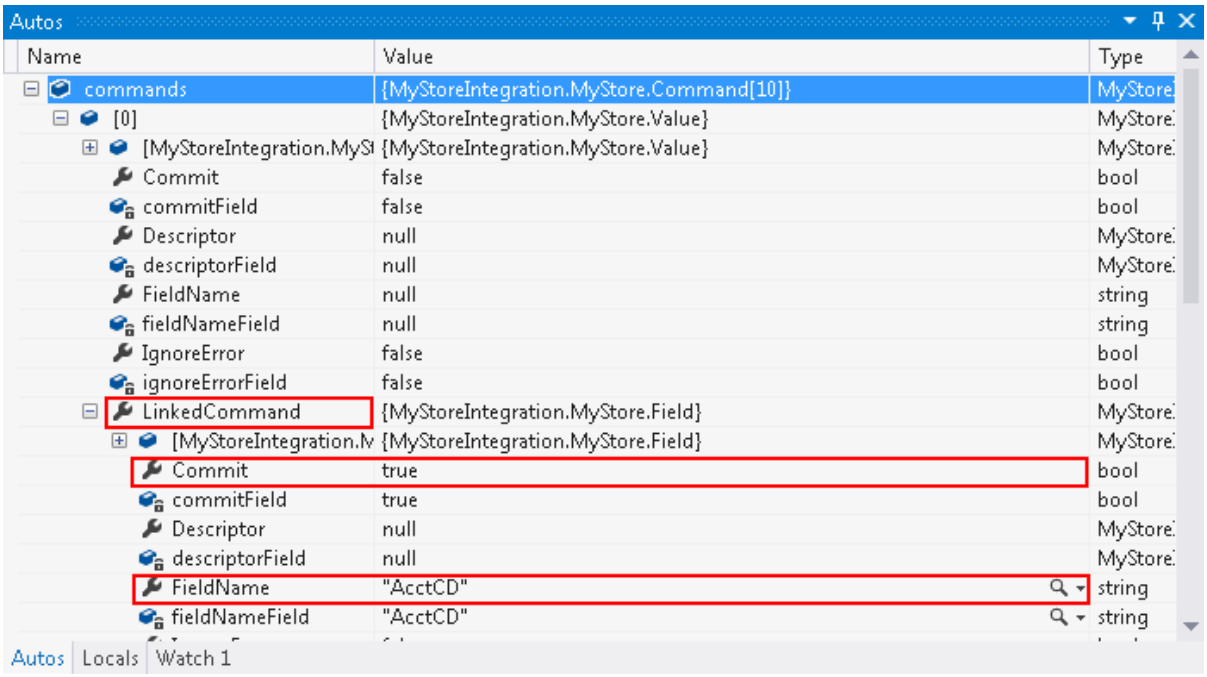


Figure: Commit property

If the `Commit` property is `false`, the element is filled in with data but no update of the form is invoked. If the `Commit` property is `true`, after the element is filled in, the commit is invoked on the server and the form is updated.

For example, when you select a customer class on the *Customers* form, the system assigns values to the elements related to customer class settings, such as **Statement Cycle ID** and **Country**. When you configure a sequence of commands for setting the values of elements on the *Customers* form, for elements such as **Customer**, the system automatically sets the `Commit` property of `LinkedCommand` to `true`.

In most cases, when you compose the sequence of commands, you can use the values of the `Commit` property that are set by default. However, you may need to force the system to invoke a commit to the server—for example, when you need to commit the value of the field before entering another field value. In such cases, you should set the `Commit` property to `true` for the command or action.

Commands for Working with Attachments

In Acumatica ERP, you can attach files to records on Acumatica ERP forms and to detail lines on master-detail forms.

To obtain a file attached to the selected record or detail line through the web services API, you specify the `Value` command in the array of `Command` objects as follows:

- In the `FieldName` property, you specify the name of the file that should be obtained.
- In the `LinkedCommand` property, you specify the needed `Attachment` service command.

  To work with a file attached to a form, you use the `Attachment` service command of the object that corresponds to the Summary object. For example, to obtain a file attached to a stock item record on the *Stock Items* (IN202500) form, you use the `Attachment` service command of the `IN202500StockItemSummary` object.

  To work with the file attached to a detail line of a master-detail form, you use the `Attachment` service command of the object that corresponds to the Details object. For example, to obtain the file attached to a warehouse detail line of a stock item on the *Stock Items* form, you use the `Attachment` service command of the `IN202500WarehouseDetails` object.

The following code shows how to retrieve the file attached to a stock item record on the *Stock Items* form.

```
//stockItemsSchema is an IN202500Content object
var commands = new Command[]
{
  new Value
  {
    Value = "AAMACHINE1",
    LinkedCommand = stockItemSchema.StockItemSummary.InventoryID
  },
  new Value
  {
    FieldName = "T2MCRO.jpg",
    LinkedCommand =
     stockItemSchema.StockItemSummary.ServiceCommands.Attachment
  }
};
//context is a Screen object
var stockItemAttachment =
 context.IN202500Export(commands, null, 1, false, true);
```

Commands for Working with Multi-Language Fields

For some text boxes on Acumatica ERP forms, users can type values in multiple languages if multiple locales are configured in Acumatica ERP. For example, if your Acumatica ERP instance has English and French locales activated and multilingual user input configured, you can specify the value of the **Description** box on the *Stock Items* (IN202500) form in English and French. For the list of elements that support multiple languages, see *Boxes that Have Multi-Language Support*. For details on how to turn on multilingual user input, see *Enabling Multilingual User Input*.

Specifying Localized Values of a Multi-Language Field

When you need to specify localized values of a text box by using the screen-based SOAP API, you specify the value of the field that corresponds to the box as a string in JSON format with the localized values. In this string, you use the two-letter ISO code of the language with which the value should be associated.

In the example that is mentioned at the beginning of the topic, if you need to specify values in English and French in the **Description** box on the *Stock Items* form, you specify the value of the `Description` field of the `StockItem` entity in the following format: `[{en:English description},{fr:French description}]`, as shown in the following code fragment.

```
//stockItemSchema is an IN202500Content object
new Value
{
    Value = "[{en:Item},{fr:Pièce}]",
    LinkedCommand = stockItemSchema.StockItemSummary.Description
}
```

In the JSON-formatted string, you should specify the actual values of the field in all languages that are configured for multilingual user input. If you specify the values of the field in particular languages, the values of the field in other languages configured for multilingual user input become empty. For example, suppose that in your instance of Acumatica ERP, multi-language fields can have values in English and French. If you pass the value of a field in the following format `[{en:English description}]`, the French value of the field becomes empty.

If you specify the value of a multi-language field as plain text, this text is saved as the value of the corresponding box in the current language of Acumatica ERP (that is, either the default language of the instance or the language that you have specified by using the `SetLocaleName()` method). For details on how to specify the locale through the screen-based SOAP API, see *SetLocaleName() Method*.

Retrieving Localized Values of a Multi-Language Field

If you need to retrieve localized values of a text box that supports multiple input languages, you retrieve the value of an internal field that contains all localized values of the text box and has the *Translations* suffix in its field name.

To specify the field name and the object name of the needed internal field with localized values, you specify the field name and the object name of the multi-language text box and append *Translations* to the field name. For example, the following code shows the command that you should use to retrieve the localized values of the **Description** element of the *Stock Items* form.

```
//stockItemSchema is an IN202500Content object
new Field
{
    ObjectName = stockItemsSchema.StockItemSummary.Description.ObjectName,
    FieldName = stockItemsSchema.StockItemSummary.Description.FieldName +
      "Translations"
```

```
}
```

The returned value of a `Translations` field is a string in JSON format with the available localized values of the field. The language to which the value belongs is identified by the two-letter ISO code of the language. For example, suppose that the **Description** element of the *Stock Items* form has the value *Item* in English and *Pièce* in French. In this case, the value of the `DescrTranslations` field, which corresponds to the **Description** element, is the following string: `[{en:Item},{fr:Pièce}]`.

Related Links

- *Locales and Languages*
- *Boxes that Have Multi-Language Support*

Configuring Push Notifications

Acumatica ERP provides push notifications that make it possible for the external applications to track the changes in the data in Acumatica ERP. That is, you can configure Acumatica ERP to send notifications to a destination (such as an HTTP address) when specific data changes in Acumatica ERP. The external application can receive these notifications and process the information on the data changes, if necessary. With push notifications, the external application doesn't need to continually poll for the data to find out whether there are any changes in this data, which helps improve the performance of the external application.

In this chapter, you can find information on how to configure Acumatica ERP to send push notifications.

In This Chapter

- *Push Notifications*
- *Recommendations for the Data Queries*
- *Push Notification Destinations*
- *Push Notification Format*
- *To Configure Push Notifications*
- *To Process Failed Notifications*

Push Notifications

Push notifications are notifications in JSON format that are sent by Acumatica ERP to notification destinations when specific data changes occur in Acumatica ERP. External applications can receive the notifications and process them to retrieve the information about the changes.

If you have installed a new version of Acumatica ERP or updated your Acumatica ERP instance by using the Acumatica ERP Configuration Wizard, push notifications are enabled in Acumatica ERP automatically.

If you have updated your Acumatica ERP instance through the web interface, you need to manually enable push notifications in Acumatica ERP, as described in *To Enable Push Notifications Manually* in the Installation Guide.

If you need to enable push notifications on the Acumatica ERP instances in a cluster, you should follow the instructions in *To Enable Push Notifications in a Cluster* in the Installation Guide.

To work with Acumatica ERP push notifications, you need to configure the following items:

- The data query that defines the data changes for which Acumatica ERP should send notifications
- The destination to which Acumatica ERP should send notifications
- The way the external application processes the notifications

- The definition of the push notification in Acumatica ERP, which specifies the data query and the notification destination

The following diagram illustrates the sending of a push notification. In the diagram, rectangles with a red border indicate the items that you need to configure to receive push notifications when changes occur.

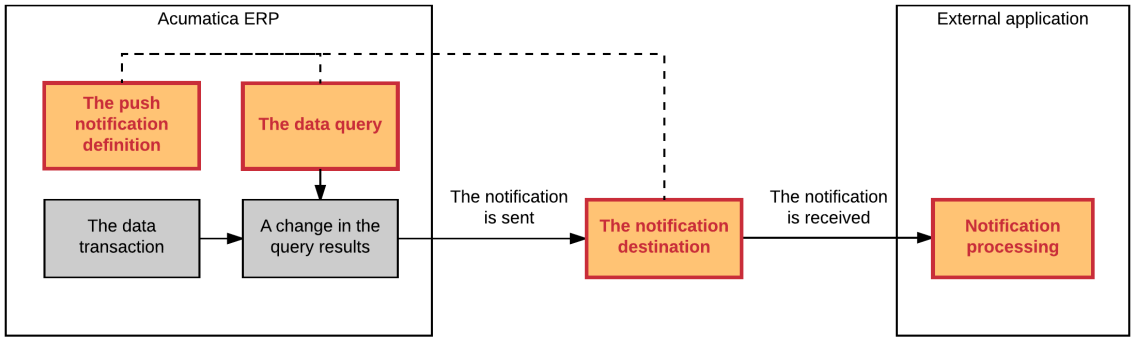


Figure: Sending a push notification

Data Query

The data query can be defined by either a generic inquiry or a built-in definition (which is a data query defined in code). For details on generic inquiries, see *Managing Generic Inquiries*. For information on how to create a built-in definition, see *To Create a Built-In Definition*. You can define multiple queries for one notification destination.

The data query should adhere to the recommendations described in *Recommendations for the Data Queries*.

Notification Destination

The following predefined notification destinations are provided: webhook (HTTP address), message queue, or SignalR hub. For more information on the predefined notification destinations, see *Push Notification Destinations*. You can also create your own destination type, as described in *To Create a Custom Destination Type*.

Processing of the Notifications in the External Application

The external application can process the notifications and extract the information about the data changes. Acumatica ERP sends notifications to notification destinations in JSON format. For details on the format of the notifications, see *Push Notification Format*. If your application watches notifications in the SignalR hub, you need to connect to the hub, as described in *To Connect to the SignalR Hub*.

Definition of the Push Notification

In the definition of the push notification on the *Push Notifications* (SM302000) form, you specify the notification destination and the data queries for which the notifications should be sent. For details on the setup of the push notifications, see *To Configure Push Notifications*.

Related Links

- *To Configure Push Notifications*

Recommendations for the Data Queries

For optimal results, follow these recommendations when you create each data query for which you want to configure push notifications:

- Do not use aggregation and grouping in the query; Acumatica ERP does not guarantee that push notifications will work correctly with such queries.
- Do not use joins of multiple detail tables in the query because this may cause the system to hang.
- If you need to join multiple tables, use a left join or an inner join in the data query. If you use an inner join, the query execution may be slower than for a left join.
- Use as simple a data query as possible.
- For a query defined by using a generic inquiry, do not use a formula on the **Results Grid** tab of the *Generic Inquiry* (SM208000) form.

Related Links

- *To Configure Push Notifications*

Push Notification Destinations

When you configure a push notification on the *Push Notifications* (SM302000) form of Acumatica ERP, you select the type of the notification destinations, which can be any of the predefined types described in this topic. You can also create your own destination type, as described in *To Create a Custom Destination Type* in the Acumatica Framework Developer Guide.

Webhook

A webhook is an HTTP address to which Acumatica ERP sends HTTP `POST` requests with notification information. For this destination type, you specify a valid HTTP address (such as `http://localhost:80/main.aspx?pushqueue`) in the **Address** box on the *Push Notifications* (SM302000) form. For security reasons, you can specify a header of the HTTP request in the **Header Name** and **Header Value** boxes.

Do not specify the `Accept` and `Content-Type` headers for the request. The values of these headers are specified automatically by the system.

If Acumatica ERP cannot send notifications to the HTTP address, Acumatica ERP logs information on the failed notifications and displays these notifications on the *Process Push Notifications* (SM502000) form. You can resend the failed notifications for two days, after which the notifications are removed from the Acumatica ERP database. For details on how to resend notifications, see *To Process Failed Notifications*.

Message Queue

The message queue is a local or remote private Microsoft message queue. You specify the address of the message queue (such as `MyComputer\private$\TestQueueForPushNotificatons`) in the **Address** box on the *Push Notifications* (SM302000) form. For information on how to configure a private Microsoft message queue, see the Microsoft documentation.

The message queue is the most reliable destination type protected from network failures. However, if Acumatica ERP cannot send notifications to the message queue for some reason, Acumatica ERP logs information about the failed notifications and displays these notifications on the *Process Push Notifications* (SM502000) form. You can resend the failed notifications for two days, after which the notifications are removed from the Acumatica ERP database. For information on how to resend notifications, see *To Process Failed Notifications*.

SignalR Hub

The SignalR hub is the destination type implemented in Acumatica ERP by using the ASP.NET SignalR library. The address of this destination type is `PushNotificationsHub`, which is filled in automatically in the **Address** box on the *Push Notifications* form. This destination type can be used if you can expose neither an HTTP address (webhook) nor a message queue to receive push notifications. If Acumatica ERP is configured to send notifications to the SignalR hub, the external application can connect to Acumatica ERP through websoket or a long-polling mechanism and receive notifications through this connection. If multiple external applications are connected to the SignalR hub, they receive notifications simultaneously. For information on how to connect your application to the SignalR hub of Acumatica ERP, see *To Connect to the SignalR Hub* in the Acumatica Framework Developer Guide.

The SignalR hub destination type is not reliable: If the connection fails or there are no clients connected to the SignalR hub when a notification comes, this notification will not be sent and it cannot be resent later.

Push Notification Format

Acumatica ERP sends push notifications in JSON format. This topic describes the structure of the notifications.

Elements

The push notifications that Acumatica ERP sends include the following elements in JSON format.

Element	Description
`Inserted`	The rows that are new in the results of the query execution.
`Deleted`	The rows that were in the results of the query execution but are missing after the latest data transaction. You can compare the `Inserted` and `Deleted` rows to identify the rows that have been updated.
`Query`	The query for which Acumatica ERP has produced the notification. The value of the element can be either the name of the generic inquiry or the name of the class with the built-in definition.
`CompanyId`	The name of the company.
`Id`	The unique identifier of the data transaction in Acumatica ERP that has initiated the notification. The external application can use this identifier to omit duplicated notifications.
`TimeStamp`	The `long` value that corresponds to the date and time when the transaction that initiated the notification happened in Acumatica ERP. By using the value of this element, the external application can define the order of notifications.
`AdditionalInfo`	Any additional information that is added to the notification. This element can contain additional information added by the system as well as the custom information. For more information on how to add custom additional information to push notifications, see *To Add Additional Information to Push Notifications*.

Example

Suppose that push notifications are configured for the *Stock Items: Last Modified Date* generic inquiry (which displays the **InventoryID**, **StockItem**, **ItemStatus**, and **InventoryItem_lastModifiedDateTime** columns). Acumatica ERP sends the following notification when the status of the *AACOMPUT01* inventory item has been changed from *Active* to *Inactive*.


```
{
  "Inserted":
  [{
    "InventoryID":"AACOMPUT01",
    "StockItem":true,
    "ItemStatus":"Inactive",
    "InventoryItem_lastModifiedDateTime":"2017-05-05T15:16:23.1"
  }],
  "Deleted":
  [{
    "InventoryID":"AACOMPUT01",
    "StockItem":true,
    "ItemStatus":"Active",
    "InventoryItem_lastModifiedDateTime":"2017-05-05T15:16:23.103"
  }],
  "Query":"Stock Items: Last Modified Date",
```

```
  "CompanyId":"Company",
  "Id":"1af4d140-5321-41f2-a2ec-50b67f577c6c",
  "TimeStamp":636295833829493672,
  "AdditionalInfo":{}
}
```

To Configure Push Notifications

To configure push notifications, you use the *Push Notifications* (SM302000) form.

Before You Proceed

If you have installed a new version of Acumatica ERP or updated your Acumatica ERP instance by using the Acumatica ERP Configuration Wizard, push notifications are enabled in Acumatica ERP automatically.

If you have updated your Acumatica ERP instance through the web interface, you need to manually enable push notifications in Acumatica ERP, as described in *To Enable Push Notifications Manually* in the Installation Guide.

If you need to enable push notifications on the Acumatica ERP instances in a cluster, you should follow the instructions in *To Enable Push Notifications in a Cluster* in the Installation Guide.

You need to define the data query or data queries for which Acumatica ERP should send notifications on data changes. Each data query can be defined by either a generic inquiry or a built-in definition. For details on generic inquiries, see *Managing Generic Inquiries*. For information on how to create a built-in definition, see *To Create a Built-In Definition* in the Acumatica Framework Development Guide. For details on defining the data queries, see *Recommendations for the Data Queries*.

You should also identify the notification destination that your external application will scan, which can be one of the following: webhook (HTTP address), message queue, or SignalR hub. For more information on the notification destinations, see *Push Notification Destinations*. You should configure your external application so that it can process notifications sent to the destination.

To Send Notifications to an HTTP Address

1. On the **System** tab, click **Integration**. In the navigation pane, navigate to **Configure > Push Notifications** (SM302000).
2. In the **Destination Name** box, type the name of the target notification destination, which can be the name of your external application.
3. In the **Destination Type** box, select *Webhook*.
4. In the **Address** box, type the HTTP address to which Acumatica ERP should send the push notifications, such as `http://localhost:80/main.aspx?pushqueue`.
5. Optional: In the **Header Name** and **Header Value** boxes, specify the header of the HTTP `POST` request in which Acumatica ERP sends the notification.

Do not specify the `Accept` and `Content-Type` headers for the request. The values of these headers are specified automatically by the system.

6. For each generic inquiry for which you want Acumatica ERP to send notifications on changes in the inquiry results, do the following:
   a. On the table toolbar of the **Generic Inquiries** tab, click **Add Row**. The new row has the **Active** check box selected.
   b. In the **Inquiry Title** column of the added row, select the generic inquiry for which you want Acumatica ERP to send notifications.
7. For each built-in definition for which you want Acumatica ERP to send notifications on changes in the results, do the following:
   a. On the table toolbar of the **Built-In Definitions** tab, click **Add Row**. The new row has the **Active** check box selected.
   b. In the **Class Name** column of the added row, select the class that defines the data query for which you want Acumatica ERP to send notifications.
8. On the form toolbar, click **Save**.

To Send Notifications to a Message Queue

1. On the **System** tab, click **Integration**. In the navigation pane, navigate to **Configure > Push Notifications** (SM302000).
2. In the **Destination Name** box, type the name of the target notification destination, which can be the name of your external application.
3. In the **Destination Type** box, select *Message Queue*.
4. In the **Address** box, type the address of the local or remote private Microsoft message queue that you have configured for receiving messages from Acumatica ERP, such as `MyComputer\private$\TestQueueForPushNotificatons`.
5. For each generic inquiry for which you want Acumatica ERP to send notifications on changes in the inquiry results, do the following:
   a. On the table toolbar of the **Generic Inquiries** tab, click **Add Row**. The new row has the **Active** check box selected.
   b. In the **Inquiry Title** column of the added row, select the generic inquiry for which you want Acumatica ERP to send notifications.
6. For each built-in definition for which you want Acumatica ERP to send notifications on changes in the results, do the following:
   a. On the table toolbar of the **Built-In Definitions** tab, click **Add Row**. The new row has the **Active** check box selected.

   b. In the **Class Name** column of the added row, select the class that defines the data query for which you want Acumatica ERP to send notifications.

7. On the form toolbar, click **Save**.

To Send Notifications to a SignalR Hub

1. On the **System** tab, click **Integration**. In the navigation pane, navigate to **Configure > Push Notifications** (SM302000).
2. In the **Destination Name** box, type the name of the target notification destination, which can be the name of your external application.
3. In the **Destination Type** box, select *SignalR Hub*. The **Address** box is filled in automatically with `PushNotificationsHub`.
4. For each generic inquiry for which you want Acumatica ERP to send notifications on changes in the inquiry results, do the following:

   a. On the table toolbar of the **Generic Inquiries** tab, click **Add Row**. The new row has the **Active** check box selected.

   b. In the **Inquiry Title** column of the added row, select the generic inquiry for which you want Acumatica ERP to send notifications.

5. For each built-in definition for which you want Acumatica ERP to send notifications on changes in the results, do the following:

   a. On the table toolbar of the **Built-In Definitions** tab, click **Add Row**. The new row has the **Active** check box selected.

   b. In the **Class Name** column of the added row, select the class that defines the data query for which you want Acumatica ERP to send notifications.

6. On the form toolbar, click **Save**.

Related Links

- *Push Notifications*

To Process Failed Notifications

To process the push notifications that Acumatica ERP could not send, you use the *Process Push Notifications* (SM502000) form. On this form, you can view the notifications that Acumatica ERP has failed to send, try to resend them, or delete the notifications.

If a notification has failed in Acumatica ERP before it was sent (for example, if Acumatica ERP could not retrieve the data for the notification), this notification is not displayed in the table on the *Process Push Notifications* form. Acumatica ERP saves the information about these notifications in the `PushNotificationsErrors` database table

if the `api:push-notifications:enable-dead-message-log` key is set to `true` in the **web.config** file of the Acumatica ERP instance.

To View a Notification

1. On the **System** tab, click **Integration**. In the navigation pane, navigate to **Process > Process Push Notifications** (SM502000).
2. In the table on the form, select the row that contains the notification you want to view.
3. On the form toolbar, click **Show Notification**.
4. In the **Notification Event** dialog box, which opens, view the notification in JSON format. For details on the format of the notification, see *Push Notification Format*.

To Resend Notifications

1. On the **System** tab, click **Integration**. In the navigation pane, navigate to **Process > Process Push Notifications** (SM502000).
2. Do one of the following:
   - If you want to resend particular notifications, in the table on the form, select the check boxes in the rows that correspond to the notification you want to send, and click **Send** on the form toolbar.
   - If you want to resend all notifications, on the form toolbar, click **Send All** on the form toolbar.

To Delete Notifications

1. On the **System** tab, click **Integration**. In the navigation pane, navigate to **Process > Process Push Notifications** (SM502000).
2. Do one of the following:
   - If you want to delete particular notifications, in the table on the form, select the unlabeled check boxes in the rows that correspond to the notifications you want to delete, and click **Delete** on the form toolbar.
   - If you want to delete all notifications, on the form toolbar, click **Delete All**.
3. On the form toolbar, click **Save**.

Related Links

- *Process Push Notifications*

Defining Push Notifications

In Acumatica ERP or an Acumatica Framework-based application, you can configure the system to send push notifications to a destination (such as an HTTP address) when specific data changes in the system. The external application can receive these notifications and process the information on the data changes if necessary. For more information on push notifications, see *Configuring Push Notifications*.

If this chapter, you can find information on how to configure push notifications in code.

In This Chapter

- *To Create a Built-In Definition*
- *To Connect to the SignalR Hub*
- *To Add Additional Information to Push Notifications*
- *To Create a Custom Destination Type*

To Create a Built-In Definition

In Acumatica ERP or an Acumatica Framework-based application, you can configure push notifications for a query that is defined as a class in the source code of the application—that is, for a built-in definition of the query. The built-in definition can be implemented in a project of your Acumatica ERP extension library, in a *Code* item of an Acumatica ERP customization project, or in a project of your Acumatica Framework-based application. (For the differences between the use of extension libraries and *Code* items in a customization project, see *Extension Library (DLL) Versus Code in a Customization Project* in the Customization Guide.)

To create a built-in definition, follow the instructions in this topic. For more information on the push notifications, see *Configuring Push Notifications* in the Integration Development Guide.

To Create a Built-In Definition

1. Define a class that implements the `PX.PushNotifications.Sources.IInCodeNotificationDefinition` interface in a project of your Acumatica ERP extension library, in a *Code* item of an Acumatica ERP customization project, or in a project of your Acumatica Framework-based application. The following code demonstrates the definition of such a class.

```
using PX.PushNotifications.Sources;

public class TestInCodeDefinition : IInCodeNotificationDefinition
{
}
```

2. In the class that implements the `IInCodeNotificationDefinition` interface, implement the `GetSourceSelect()` method of the interface so that the method satisfies the following requirements:

   - The method must return a tuple of a `BqlCommand` object, which defines the data query, and a `PXDataValue` array, which defines the parameters that should be passed to the query.
   - The data query that the method defines should adhere to *Recommendations for the Data Queries* in the Integration Development Guide.

   The following example shows the `GetSourceSelect()` method implementation.

```
using PX.Data;
using PX.PushNotifications.Sources;
using PX.PushNotifications.UI.DAC;

public class TestInCodeDefinition : IInCodeNotificationDefinition
{
  public Tuple<BqlCommand, PXDataValue[]> GetSourceSelect()
  {
    return
      Tuple.Create(
        PXSelectJoin<PushNotificationsHook,
        LeftJoin<PushNotificationsSource,
          On<PushNotificationsHook.hookId,
            Equal<PushNotificationsSource.hookId>>>>
        .GetCommand(), new PXDataValue[0]);
  }
}
```

3. In the class that implements the `IInCodeNotificationDefinition` interface, implement the `GetRestrictedFields()` method of the interface so that the method satisfies the following requirements:

   - The method must return an array of `IBqlField`-derived types, which contains the fields that should be returned from the query.
   - If you need to return all fields, the method must return `null`.

   The following code shows an example of the implementation of the `GetRestrictedFields()` method.

```
using PX.Data;
using PX.PushNotifications.Sources;
using PX.PushNotifications.UI.DAC;

public class TestInCodeDefinition : IInCodeNotificationDefinition
{
  ...
```

```
  public Type[] GetRestrictedFields()
  {
    return new []
    {
      typeof(PushNotificationsHook.address),
      typeof(PushNotificationsHook.type),
      typeof(PushNotificationsSource.designID),
      typeof(PushNotificationsSource.inCodeClass),
      typeof(PushNotificationsSource.lineNbr)
    };
  }
}
```

4. Compile your Acumatica ERP extension library or Acumatica Framework-based application, or publish the customization project with the *Code* item that contains the class implementation.

5. Run Acumatica ERP or your Acumatica Framework-based application, and make sure that you can select the new built-in definition by its class name on the **Built-In Definitions** tab of the *Push Notifications* (SM302000) form. The class of the built-in definition, which implements the `IInCodeNotificationDefinition` interface, is detected by the system and automatically added to the list of classes available for selection on the tab.

You can obtain the results of the data query defined with a built-in definition by using the following endpoint: `http(s)://<Acumatica ERP instance URL>/PushNotifications/<full class name of the built-in definition>`. For example, suppose that you want to retrieve the results of the data query defined with the **PX.PushNotifications.Sources.TestInCodeDefinition** class from a local Acumatica ERP instance with the name *AcumaticaDB*. You should use the following URL to obtain the data: `http(s)://localhost/AcumaticaDB/PushNotifications/PX.PushNotifications.Sources.TestInCodeDefinition`. The endpoint returns the data in JSON format.

Related Links

- *To Configure Push Notifications*
- *IInCodeNotificationDefinition Interface*

To Connect to the SignalR Hub

If you want your external application to receive push notifications from Acumatica ERP or an Acumatica Framework-based application but you cannot publish a web hook (for example, for security reasons), the system can send notifications to the SignalR hub, from which any connected application can receive the notifications.

To connect the external application to the SignalR hub, follow the instructions in this topic. For more information on push notifications, see *Configuring Push Notifications*.

To Connect to the SignalR Hub

1. In your external application, set up a Basic authentication token to authenticate the application in Acumatica ERP or an Acumatica Framework-based application, as shown in the following code.

```
using System;
using System.Text;

class Program
{
  static void Main(string[] args)
  {
    var login = "admin";
    var tenant = "Tenant";
    var password = "123";
    // Set up a Basic authentication token
    var basicAuthToken = Convert.ToBase64String(
     Encoding.UTF8.GetBytes(login+"@"+tenant+":"+password));
  }
}
```

2. In your external application, connect to an instance of Acumatica ERP or an Acumatica Framework-based application, as shown in the following code.

```
using System;
using System.Text;
using Microsoft.AspNet.SignalR.Client.Hubs;

class Program
{
  static void Main(string[] args)
  {
    ...
    //Connect to an Acumatica ERP instance
    var connection = new HubConnection("http://localhost:8081/AcumaticaDB/");
    connection.Headers.Add("Authorization", "Basic "+basicAuthToken);
  }
}
```

3. In your external application, create a proxy to the SignalR hub, based on the name of the hub that was specified in the **Destination Name** box when the push notification was defined on the *Push Notifications* (SM302000) form.

```
using System;
using System.Text;
using Microsoft.AspNet.SignalR.Client.Hubs;

class Program
{
```

```
  static void Main(string[] args)
  {
    ...
    //Create a proxy to hub
    //Use "PushNotificationsHub" as the address of the hub
    var myHub = connection.CreateHubProxy("PushNotificationsHub");
    connection.Start().ContinueWith(task =>
    {
      if(task.IsFaulted)
      {
        Console.WriteLine(
          "There was an error during open of the connection:{0}",
          task.Exception.GetBaseException());
      }
      else
      {
        //Instead of "TestSignalR", specify the name
        //that you specified on the Push Notifications form
        myHub.Invoke<string>("Subscribe", "TestSignalR").Wait();
      }
    }).Wait();
  }
}
```

4. In your external application, define the class for a notification, as shown in the following code. For details on the format of the push notifications, see *Push Notification Format*.

```
public class NotificationResult
{
  public object[] Inserted { get; set; }
  public object[] Deleted { get; set; }
  public string Query { get; set; }
  public string CompanyId { get; set; }
  public Guid Id { get; set; }
  public long TimeStamp { get; set; }
  public Dictionary<string, object> AdditionalInfo { get; set; }
}
```

5. In your external application, process notifications. The following code displays the number of inserted and updated records specified in the notification in the console application window.

```
using System;
using System.Text;
using Microsoft.AspNet.SignalR.Client.Hubs;

class Program
{
  static void Main(string[] args)
  {
    ...
```

```
    //Process the notifications
    myHub.On<NotificationResult>("ReceiveNotification", nr =>
    {
      Console.WriteLine("Inserted {0}", nr.Inserted.Length);
      Console.WriteLine("Deleted {0}", nr.Deleted.Length);
    });
    Console.Read();
    connection.Stop();
  }
}
```

6. Compile and test your application.

Related Links

- *To Configure Push Notifications*

To Add Additional Information to Push Notifications

In Acumatica ERP or an Acumatica Framework-based application, you can add additional information to push notifications, such as the username of the user that performed the data change or the business date when the data transaction is performed. The additional information can be added to the `AdditionalInfo` element of notifications in JSON format. For more information on the format of notifications, see *Push Notification Format* in the Integration Development Guide.

The additional information that you add to push notifications is included in all notifications that are sent from the Acumatica ERP or Acumatica Framework instance on which the additional information is configured.

The dictionary of additional information can be implemented in a project of your Acumatica ERP extension library, in a *Code* item of an Acumatica ERP customization project, or in a project of your Acumatica Framework-based application. (For the differences between the use of extension libraries and *Code* items in a customization project, see *Extension Library (DLL) Versus Code in a Customization Project* in the Customization Guide.)

To add additional information to push notifications, follow the instructions in this topic. For more information on the push notifications, see *Configuring Push Notifications* in the User Guide.

To Add Additional Information to Push Notifications

1. Define a class that implements the `PX.Data.PushNotifications.ICommitEventEnricher` interface in a project of your Acumatica ERP extension library, in a *Code* item of an Acumatica ERP customization project, or in a project of your Acumatica Framework-based application.

2. In the class that implements the `ICommitEventEnricher` interface, implement the `Enrich()` method of the interface. In the method, add the properties that you want to be returned in push notifications.

The `Enrich()` method is called in the `PX.Data.PXTransactionScope.Dispose()` method. Therefore, the `Enrich()` method must not return data that is not accessible in this scope.

The following code shows a sample implementation of the `ICommitEventEnricher` interface, which adds the business date and the name of the user to the `AdditionalInfo` element of notifications in JSON format.

```
using PX.Data;
using PX.Data.PushNotifications;

public class CommitEventEnricher : ICommitEventEnricher
{
  public void Enrich(IQueueEvent commitEvent)
  {
    var businessDate = PXContext.PXIdentity?.BusinessDate;
    var userName = PXContext.PXIdentity?.IdentityName;
    commitEvent.AdditionalInfo.Add(nameof(businessDate), businessDate);
    commitEvent.AdditionalInfo.Add(nameof(userName), userName);
  }
}
```

3. Compile your Acumatica ERP extension library or Acumatica Framework-based application, or publish the customization project with the *Code* item that contains the class implementation.

4. Run Acumatica ERP or your Acumatica Framework-based application and make sure that when the results of the query change, the application includes the additional information in the `AdditionalInfo` element of the notifications in JSON format, as shown in the following notification example.

```
{
  ...
  "TimeStamp":636295833829493672,
  "AdditionalInfo":
  {
    "businessDate":"2017-05-05T15:16:23.1",
    "userName":"admin"
  }
}
```

Related Links

- *To Configure Push Notifications*
- *Push Notification Format*

To Create a Custom Destination Type

In Acumatica ERP or an Acumatica Framework-based application, you can send push notifications to the notification destinations, which can be of a predefined type (which is webhook, message queue, or SignalR hub) or of a custom type, which you can implement in the code of the application.

The notification destination of a custom type can be implemented in a project of your Acumatica ERP extension library, in a *Code* item of an Acumatica ERP customization project, or in a project of your Acumatica Framework-based application. (For the differences in the use of extension libraries and *Code* items in a customization project, see *Extension Library (DLL) Versus Code in a Customization Project* in the Customization Guide.)

To create a custom destination, follow the instructions in this topic. For more information on the push notifications, see *Configuring Push Notifications* in the Integration Development Guide.

To Create a Custom Destination Type

1. Define a class that implements the `PX.PushNotifications.NotificationSenders.IPushNotificationSender` interface, which is a sender of push notifications, in a project of your Acumatica ERP extension library, in a *Code* item of an Acumatica ERP customization project, or in a project of your Acumatica Framework-based application.
2. In the class that implements the `IPushNotificationSender` interface, implement the following methods and properties of the interface:
   - The `Address` property, which is the address to which the system should send notifications. A user specifies the value of this property in the **Address** box on the *Push Notifications* (SM302000) form. The property uses the following syntax.

     ```
     string Address { get; }
     ```

   - The `Name` property, which is the name of the notification destination. A user specifies the value of this property in the **Destination Name** box on the *Push Notifications* (SM302000) form. Use the following syntax for the property.

     ```
     string Name { get; }
     ```

   - The `Send` method, which sends a notification synchronously and uses as the parameters the notification to be sent and a cancellation token. The method uses the following syntax.

     ```
     void Send(
       NotificationResultWrapper results,
       CancellationToken cancellationToken
     );
     ```

- The `SendAndForget` method, which sends a notification without blocking the current thread. We recommend that you use `HostingEnvironment.QueueBackgroundWorkItem` in the method implementation to delegate the execution to a parallel thread. The following code shows a sample implementation of the method.

```
using System;
using System.Threading;
using PX.PushNotifications;
using PX.PushNotifications.NotificationSenders;

public void SendAndForget(
  NotificationResultWrapper result,
  CancellationToken cancellationToken,
  Action onSendingFailed,
  Action finalizer)
{
  try
  {
    Send(result, cancellationToken);
  }
  catch (Exception e)
  {
    onSendingFailed($"Send to target {Name} failed: ({e.Message})");
  }
  finally
  {
    finalizer();
  }
}
```

3. Define a class that implements the `PX.PushNotifications.NotificationSenders.IPushNotificationSenderFactory` interface, which creates a sender of push notifications, in a project of your Acumatica ERP extension library, in a *Code* item of an Acumatica ERP customization project, or in a project of your Acumatica Framework-based application.

4. In the class that implements the `IPushNotificationSenderFactory` interface, implement the following methods and properties of the interface:

   - The `Create` method, which creates a sender and uses as the parameters the destination address, the name of the notification destination, and the additional parameters (such as a header for an HTTP address). Use the following syntax for the method.

```
IPushNotificationSender Create(
  string address,
  string name,
  IDictionary<string, object> additionalParameters
);
```

- The `Type` property, which is a `string` identifier of the destination type that is exactly four characters long. The value of this property is stored in the database. The property uses the following syntax.

```
string Type { get; }
```

- The `TypeDescription` property, which is a `string` label of the destination type. A user selects the value of this property in the **Destination Type** box on the *Push Notifications* (SM302000) form. Use the following syntax for the property.

```
string TypeDescription { get; }
```

5. Compile your Acumatica ERP extension library or Acumatica Framework-based application, or publish the customization project that contains the *Code* item or items with the implementation of the classes.
6. Run Acumatica ERP or your Acumatica Framework-based application, and test the new destination type.

Related Links

- *To Configure Push Notifications*

Contract-Based REST API Reference

Acumatica ERP provides the reference information for the methods of the contract-based REST API endpoint in the **swagger.json** file, which is an OpenAPI 2.0 (formerly known as Swagger 2.0) file. (For more information about the OpenAPI specification, see .) You can use this file to review the API of the endpoint and build the client applications of Acumatica ERP based on this file.

Acumatica ERP 2019 R2 does not provide a user interface to view the **swagger.json** file. You can use external tools to view the file.

You can retrieve the **swagger.json** file by clicking **View Endpoint Service > OpenAPI 2.0** on the *Web Service Endpoints* (SM207060) form, or by using the following URL.

```
http://<Base endpoint URL>/swagger.json
```

In this URL, `<Base endpoint URL>` is the URL of the contract-based endpoint through which you are going to work with Acumatica ERP. This URL has the following format: `http://<Acumatica ERP instance URL>/entity/<Endpoint name>/<Endpoint version>/`.

You can specify the `company` URL parameter to obtain information on the API of the endpoint available in a particular company. For example, suppose that you wanted to retrieve the API reference of the custom endpoint with the name *MyEndpoint* and Version 18.200.001 available in the *MyCompany* company from a local Acumatica ERP instance with the name *AcumaticaDB*. You would use the following URL to obtain the **swagger.json** file.

```
http://localhost/AcumaticaDB/entity/MyEndpoint/18.200.001/swagger.json?company=MyCompany
```

If no company is specified, the API of the endpoint available in the company to which the user is currently logged in is returned.

Related Links

- *Working with the Contract-Based REST API*
-

Contract-Based SOAP API Reference

In this chapter, you will find the reference information for the main objects and methods of the contract-based SOAP API; these objects and methods are used to transfer data to and from Acumatica ERP. This chapter covers the following methods, which are exposed by the `DefaultSoapClient` class:

Semantics and syntax of some methods and properties are different depending on the version of the system contract.

- *Login() Method*
- *Logout() Method*
- *SetBusinessDate() Method*
- *Get() Method*
- *GetList() Method (Contract Version 3)*
- *GetList() Method (Contract Version 2)*
- *Put() Method*
- *Delete() Method*
- *Invoke() Method*
- *GetProcessStatus() Method*
- *GetFiles() Method*
- *PutFiles() Method*
- *GetCustomFieldSchema() Method*

You will also find descriptions of the following properties:

- *Attributes Property*
- *CustomFields Property*
- *ReturnBehavior Property (Contract Version 3)*
- *ReturnBehavior Property (Contract Version 2)*

Login() Method

You use the `Login()` method of a `DefaultSoapClient` object to make the client application sign in to Acumatica ERP.

Instead of directly signing in to Acumatica ERP, your application can use the OAuth 2.0 authorization. For details about OAuth 2.0, see *Authorizing Client Applications to Work with Acumatica ERP*.

Syntax

```
public void Login(string name, string password, string company, string branch, string
 locale)
```

Parameters

- *name*: The username that the application should use to sign in to Acumatica ERP, such as `"admin"`.
- *password*: The password for the username, such as `"123"`.
- *company*: The name of the tenant to which the application should sign in, such as `"MyStore"`. You can view the name that should be used for the tenant in the **Login Name** box of the *Tenants* (SM203520) form.
- *branch*: The ID of the branch to which the application should sign in. You can view the ID of the branch in the **Branch ID** box of the *Branches* (CS102000) form.
- *locale*: The locale that should be used in Acumatica ERP. You should specify the locale in the `System.Globalization.CultureInfo` format converted to `string`, as with `"EN-US"`.

  This parameter has been developed for future use. You do not need to set its value.

Example

The following code causes the client to sign in to Acumatica ERP by using the parameters that are specified in the application settings.

```
using (var soapClient = new DefaultSoapClient())
{
  //Log in to Acumatica ERP
  soapClient.Login
  (
    Properties.Settings.Default.UserName,
    Properties.Settings.Default.Password,
    Properties.Settings.Default.TenantName,
    Properties.Settings.Default.CompanyName,
    null
  );
}
```

Usage Notes

For each call of the `Login()` method, you must call the `Logout()` method after you finish your work with Acumatica ERP to close the session.

You should take into account Acumatica ERP license API limits when using the `Login()` method. For details, see *License Restrictions for API Users*.

Related Links

- *Logout() Method*

Logout() Method

You use the `Logout()` method of a `DefaultSoapClient` object to make the client application sign out from Acumatica ERP.

Syntax

```
public void Logout()
```

Example

The following code shows how to make the client application sign out from Acumatica ERP.

```
using (var soapClient = new DefaultSoapClient())
{
  //Sign in to Acumatica ERP
  ...
  try
  {
    //Work with Acumatica ERP through the web services API
  }
  finally
  {
    //Sign out from Acumatica ERP
    soapClient.Logout();
  }
}
```

Usage Notes

For each call of the `Login()` method, you must call the `Logout()` method after you finish your work with Acumatica ERP to close the session. Therefore, we recommend that you call the `Logout()` method within the `finally` block.

Related Links

- *Login() Method*

SetBusinessDate() Method

You use the `SetBusinessDate()` method to specify the business date in Acumatica ERP. You can set the business date to any date to make the system insert this date into the date fields by default. The business date is inserted into any new document that you create and is used in the default selection parameters that appear on processing and inquiry screens.

Syntax

```
public void SetBusinessDate(System.DateTime businessDate)
```

Parameter

- *businessDate*: The business date that should be used in Acumatica ERP.

Usage Notes

The business date resets to the current date of your computer after each login. Therefore, if you need to specify a business date in your application, you should call the `SetBusinessDate()` method after each client application login.

Get() Method

You use the `Get()` method to get one record that satisfies the specified conditions from Acumatica ERP. The conditions must specify only one record in Acumatica ERP; otherwise, an error is returned. If you need to get multiple records that satisfy the specified conditions, use the `GetList()` method instead.

Syntax

```
public Entity Get(Entity entity)
```

Parameter

- *entity*: The entity that specifies the record that should be obtained from Acumatica ERP.

Return Value

The method returns the `Entity` object that corresponds to the specified record.

Example

The following code gets a customer record with the specified customer ID.

```
Customer customer = new Customer
{
  CustomerID = new StringSearch { Value = "C000000003" },
};
```

```
Customer customerData = (Customer)soapClient.Get(customer);
```

GetList() Method (Contract Version 3)

This topic describes the `GetList()` method that is available in Version 3 of the system contract.

You use the `GetList()` method to retrieve from an Acumatica ERP data entry form a list of records that satisfy the specified conditions.

Do not use the `GetList()` method to retrieve the records from an inquiry form; instead, use the `Put()` method.

Syntax

```
public Entity[] GetList(Entity entity)
```

Parameter

- *entity*: The entity that specifies the conditions that must be met for the records to be returned from Acumatica ERP.

Return Value

The method returns the array of `Entity` objects that correspond to the specified records.

Example

The following example gets the list of stock items that have the *Active* status and that were modified within the past month. The code returns only the top-level fields of each stock item record (no fields of the detail or linked entities are returned).

```
//Filter the items by the last modified date and status
StockItem stockItemsToBeFound = new StockItem
{
  LastModified = new DateTimeSearch
  {
    Value = DateTime.Now.AddMonths(-1),
    Condition = DateTimeCondition.IsGreaterThan
  },
  ItemStatus = new StringSearch { Value = "Active" }
};

//Get the list of stock items
Entity[] stockItems = soapClient.GetList(stockItemsToBeFound);
```

Usage Notes

- To specify the fields whose values you need to obtain for each entity, you use the `ReturnBehavior` property of the entity.

- When the `GetList()` method is called, the system tries to optimize the retrieval of the records and obtain all needed records in one request to the database (instead of requesting the records one by one). If the optimization fails, the system returns an error, which specifies the entities or fields that caused the failure of the optimized request. To prevent the error from being generated, you can do any of the following:
  - If you do not need to retrieve the entities or fields that caused the failure, you can exclude these entities or fields from the list of requested fields by setting the `ReturnBehavior` property to `None` for the entities, or by using the `Skip` classes of the needed value type (such as `StringSkip`) for the fields.
  - If you need to retrieve the entities of fields that have caused the failure, you can retrieve the needed records one by one by using the *Get()* method.

Related Links

- *ReturnBehavior Property (Contract Version 3)*

GetList() Method (Contract Version 2)

This topic describes the `GetList()` method that is available in Version 2 of the system contract.

You use the `GetList()` method to retrieve from an Acumatica ERP data entry form a list of records that satisfy the specified conditions.

Do not use the `GetList()` method to retrieve the records from an inquiry form; instead, use the `Put()` method.

Syntax

```
public Entity[] GetList(Entity entity)
```

Parameter

- *entity*: The entity that specifies the conditions that must be met for the records to be returned from Acumatica ERP.

Return Value

The method returns the array of `Entity` objects that correspond to the specified records.

Example

The following example gets the list of stock items that have the *Active* status and that were modified within the past month. The code returns all fields of each stock item record.

```
//Filter the items by the last modified date and status
StockItem stockItemsToBeFound = new StockItem
{
```

```
  LastModified = new DateTimeSearch
  {
    Value = DateTime.Now.AddMonths(-1),
    Condition = DateTimeCondition.IsGreaterThan
  },
  ItemStatus = new StringSearch { Value = "Active" }
};

//Get the list of stock items
Entity[] stockItems = soapClient.GetList(stockItemsToBeFound);
```

Usage Notes

- To specify the fields whose values you need to obtain for each entity, you use the `ReturnBehavior` property of the entity.
- When the `GetList()` method is called, the system tries to optimize the retrieval of the records and obtain all needed records in one request to the database. If the optimization fails, the system obtains the records one by one, which may significantly increase the time of data retrieval.

Related Links

- *ReturnBehavior Property (Contract Version 2)*

Put() Method

You use the `Put()` method to create or modify a record on a data entry form of Acumatica ERP. For example, by using the `Put()` method, you can create a customer record on the *Customers* (AR303000) form.

You also use this method to retrieve data from an inquiry form. For example, by using the `Put()` method, you can retrieve data from the *Inventory Summary* (IN401000) form.

Syntax

```
public Entity Put(Entity entity)
```

Parameter

- *entity*: The entity that specifies the values of the fields of the created record, or the entity that specifies the record that should be modified and the values of the fields that should be modified in this record.

  If you use this method to create or modify a record on a data entry form, the entity must specify only one record in Acumatica ERP. If the entity specifies more

than one record or no records, an error is returned during the execution of the method.

Return Value

The method returns the `Entity` object that corresponds to the created or modified record.

Example 1: Creating a Record

The following example creates a customer record with the specified field values in Acumatica ERP.

```
//Specify the values of a new customer record
Customer customerToBeCreated = new Customer
{
  CustomerID = new StringValue { Value = "JOHNGOOD" },
  CustomerName = new StringValue { Value = John Good" },
  MainContact = new Contact
  {
    Email = new StringValue { Value = "demo@gmail.com" },
    Address = new Address
    {
      AddressLine1 = new StringValue { Value = "43 Lake Washington Blvd NE" },
      AddressLine2 = new StringValue { Value = "Suite 100" },
      City = new StringValue { Value = "Kirkland" },
      State = new StringValue { Value = "WA" },
      PostalCode = new StringValue { Value = "98033" }
    }
  }
};

//Create a customer record with the specified values
Customer newCustomer = (Customer) soapClient.Put(customerToBeCreated);
```

Example 2: Updating a Record

The following example searches for the needed customer record in Acumatica ERP by the email address and updates the record with the specified customer class value.

```
//Select the needed customer record and
//specify the values that should be updated
var customerToBeUpdated = new Customer
{
  MainContact = new Contact
  {
    //Search for the customer record by email address
    Email = new StringSearch { Value = "info@jevy-comp.con" }
  },
  CustomerClass = new StringValue { Value = "INTL" };
};
```

```
//Update the customer record with the specified values
var updCustomer = (Customer)soapClient.Put(customerToBeUpdated);
```

Example 3: Retrieving Data from an Inquiry Form

The following example retrieves the values of all elements available for a stock item from the *Inventory Summary* (IN401000) inquiry form.

```
//Filter details by warehouse
InventorySummaryInquiry stockItemsToBeFound =
new InventorySummaryInquiry
{
  InventoryID = new StringValue { Value = "AALEGO500" },
  WarehouseID = new StringValue { Value = "MAIN" }
};

//Retrieve the list of stock items from the inquiry
InventorySummaryInquiry stockItems =
 (InventorySummaryInquiry)soapClient.Put(stockItemsToBeFound);

foreach (InventorySummaryRow stockItem in stockItems.Results)
{
  //Do something with the results
  ...
}
```

Usage Notes

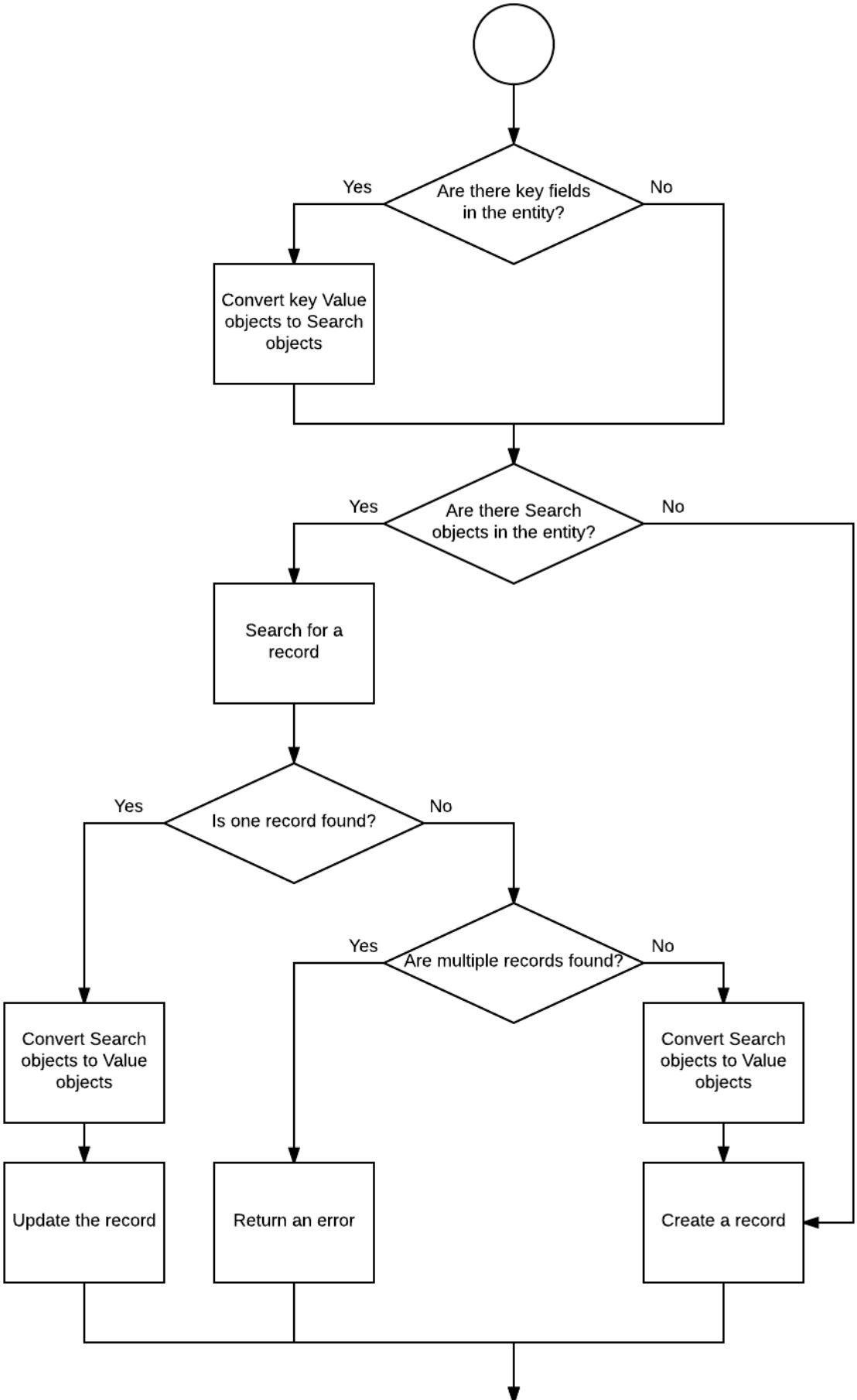
When the Acumatica ERP web services receive a `Put` request that contains an entity with at least one `Search` object, Acumatica ERP tries to search for a record by using the specified search value or values, and does one of the following:

- If the record that satisfies the specified conditions has been found, Acumatica ERP updates the fields of the record that are specified by using the `Value` objects.
- If no record has been found that satisfies the specified conditions, a new record is inserted. Note that the values that were specified with the `Search` objects are inserted into the created record in the same way as the values specified with the `Value` objects are inserted.
- If multiple records have been found, Acumatica ERP returns an error.

For the key fields of a record that are passed in the `Put` request as `Value` objects, the service converts the corresponding `Value` objects to `Search` objects.

When the Acumatica ERP web services receive a `Put` request that contains only `Value`, `Return`, and `Skip` objects without key fields specified, Acumatica ERP adds a new record with the specified values.

The workflow of a `Put` request is illustrated in the following diagram.


Are there key fields in the entity?
Yes
No
Convert key Value objects to Search objects
Are there Search objects in the entity?
Yes
No
Search for a record
Is one record found?
Yes
No
Are multiple records found?
Yes
No
Convert Search objects to Value objects
Convert Search objects to Value objects
Update the record
Return an error
Create a record

Delete() Method

You use the `Delete()` method to delete a record from Acumatica ERP.

Syntax

```
public void Delete(Entity entity)
```

Parameter

- *entity*: The entity that specifies the record that should be deleted.

  The entity must specify only one record in Acumatica ERP. If the entity specifies more than one record or no records, an error is returned during the execution of the method.

Usage Notes

The `Delete()` method is used to delete only the records that correspond to top-level entities. If you need to delete a detail line of a record, you should set the `Delete` property of the entity that corresponds to the detail line to `true` and pass the top-level entity that contains this entity to Acumatica ERP by using the `Put()` method.

Invoke() Method

You use the `Invoke()` method to invoke an action on a record in Acumatica ERP (for example, to perform the release action on a document).

Syntax

```
public InvokeResult Invoke(Entity entity, Action action)
```

Parameters

- *entity*: The entity that specifies the record on which the action should be invoked.

  The entity must specify only one record in Acumatica ERP. If the entity specifies more than one record or no records, an error is returned during method execution.

- *action*: The action that should be invoked on the record.

Return Value

The method returns an `InvokeResult` object, which you should use to monitor the status of the long-running operation by using the `GetProcessStatus()` method.

Example

The following code releases a payment.

```
//Find the payment that should be released
Payment soPaymentToBeReleased = new Payment
{
  Type = new StringSearch { Value = "Payment" },
  ReferenceNbr = new StringSearch { Value = "000001" }
};
Payment payment = (Payment)soapClient.Get(soPaymentToBeReleased);

//Release the payment
InvokeResult invokeResult = soapClient.Invoke(payment, new ReleasePayment());

//Monitor the status of the release operation
...
```

Related Links

- *GetProcessStatus() Method*

GetProcessStatus() Method

You use the `GetProcessStatus()` method to monitor the status of a long-running operation (such as the release or confirmation operation) that you invoked by using the `Invoke()` method.

Syntax

```
public ProcessResult GetProcessStatus(InvokeResult invokeResult)
```

Parameter

- *invokeResult*: An `InvokeResult` object that was returned by the `Invoke()` method.

Return Value

The method returns a `ProcessResult` object. You should use the `Status` property of this object to get the status of the processing operation. When the status of the operation is `Completed`, you can get the result of processing, which is identified by the `EntityID` property of a `ProcessResult` object.

If a synchronous operation has been invoked by the `Invoke()` method and the operation has been completed successfully, the `GetProcessStatus()` method returns the `Completed` status.

Example

The following code monitors the status of the payment release processing.

```
...
//Release payment
InvokeResult invokeResult = soapClient.Invoke(payment, new ReleasePayment());

//Monitor the status of the process
ProcessResult processResult = soapClient.GetProcessStatus(invokeResult);
while (processResult.Status == ProcessStatus.InProcess)
{
  Thread.Sleep(1000); //pause for 1 second
  processResult = soapClient.GetProcessStatus(invokeResult);
}
if (processResult.Status == ProcessStatus.Completed)
{
  //Get the released payment
  payment = (Payment)soapClient.Get(
   new Payment { ID = processResult.EntityId });
}
```

Related Links

- *Invoke() Method*

GetFiles() Method

You use the `GetFiles()` method to get the files that are attached to a record.

Syntax

```
public File[] GetFiles(Entity entity)
```

Parameter

- *entity*: The entity that specifies the record in Acumatica ERP to which the files are attached.

Return Value

The method returns an array of web service `File` objects that contain the properties for accessing the contents and names of the files.

Example

The following example retrieves the files attached to a stock item record.

```
//Filter the items by inventory ID
StockItem stockItemToBeFound = new StockItem
{
  InventoryID = new StringSearch { Value = "AAMACHINE1" }
};
```

```
//Get the stock item record
StockItem stockItem = (StockItem) soapClient.Get(stockItemToBeFound);

//Get the files that are attached to the stock item
if (stockItem != null && stockItem.ImageUrl != null)
{
  //Get the attached files
  File[] files = soapClient.GetFiles(stockItem);
  ...
};
```

PutFiles() Method

You use the `PutFiles()` method to attach files to a record in Acumatica ERP.

Syntax

```
public void PutFiles(Entity entity, File[] files)
```

Parameters

- *entity*: The entity that specifies the record to which the files should be attached.
- *files*: An array of web service `File` objects that specify the files to be attached.

Example

The following code attaches a file to a stock item record.

```
//Find the needed stock item
StockItem stockItemToBeFound = new StockItem
{
  InventoryID = new StringSearch { Value = "AALEGO500" },
};
StockItem stockItem = (StockItem)soapClient.Get(stockItemToBeFound);

//Read the file data
byte[] filedata;
using (FileStream file =
System.IO.File.Open("D:\\T2MCRO.jpg", FileMode.Open))
{
  filedata = new byte[file.Length];
  file.Read(filedata, 0, filedata.Length);
}

//Add the file to the stock item record
Default.File[] stockItemFiles = new[]
{
  //Default is the name of the service reference
```

```
  new Default.File
  {
    Name = fileName,
    Content = filedata
  }
};
soapClient.PutFiles(stockItem,stockItemFiles);
```

GetCustomFieldSchema() Method

You use the `GetCustomFieldSchema()` method to obtain the list of custom fields of an entity. You can find out the field name and view name of the needed element by using this method.

Depending on the version of the system contract, this method returns different set of fields: In Contract Version 2, custom fields correspond to both the predefined elements on an Acumatica ERP form that are not included in the entity definition and the elements that were added to the Acumatica ERP form in a customization project; in Contract Version 1, custom fields correspond to only the elements that were added to an Acumatica ERP form in a customization project.

Syntax

```
public Entity GetCustomFieldSchema(Entity entity)
```

Parameter

- *entity*: The entity for which the list of custom fields should be obtained.

Return Value

The method returns the `Entity` object, which contains the array of custom fields in its `CustomFields` property.

Example

The following example retrieves the list of custom fields of the `StockItem` entity.

```
StockItem stockItem =
  (StockItem) soapClient.GetCustomFieldSchema(new StockItem());
CustomField[] customFields = stockItem.CustomFields;
```

Related Links

- *Custom Fields*
- *CustomFields Property*

Attributes Property

By using the `Attributes` property, you can view and edit the attributes of a record in Acumatica ERP. Through this property, you work with an array of `AttributeValue` or `AttributeDetail` objects.

For the `AttributeValue` objects, to specify the value of an attribute, you specify the name of the attribute in the `AttributeID` property, and the value of the attribute in the `Value` property.

For the `AttributeDetail` objects, to specify the value of an attribute, you specify the name of the attribute in the `Attribute` property, and the value of the attribute in the `Value` property.

Syntax

```
public AttributeValue[] Attributes { set; get; }
```

Example

The following code shows how to edit the attributes of a stock item record.

```
//Specify the values of a stock item record
StockItem stockItemToBeCreated = new StockItem
{
  InventoryID = new StringValue { Value = "BASESERV" },
  ItemClass = new StringValue { Value = "STOCKITEM" },
  Attributes = new []
  {
    //Specify the values of attributes of the item class (STOCKITEM)
    new AttributeValue
    {
      AttributeID = new StringValue { Value = "Operation System" },
      Value = new StringValue { Value = "Windows" }
    },
    new AttributeValue
    {
      AttributeID = new StringValue { Value = "Version Of Software" },
      Value = new StringValue { Value = "Server 2012 R2" }
    }
  }
};

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

CustomFields Property

By using the `CustomFields` property, you can view and edit the values of the elements that are not included in the entity definition. That is, custom fields can correspond to the predefined elements on an Acumatica ERP form that are not included in the entity definition, the elements that were added to the Acumatica ERP form in a customization project, and user-defined fields.

Through the `CustomFields` property, you work with an array of `CustomField` objects. This property is exposed by the `Entity` class—that is, all entities of the web services API expose this property.

To get or set the value of an element that is not included in the entity definition, you should use the `CustomFields` property of the entity that contains this element. To work with the needed element in the `CustomFields` array, you specify the values of the `fieldName` and `viewName` properties of the `CustomField` object of the needed type. In the `viewName` property, you specify the name of the data view that contains the element, and in the `fieldName` property, you specify the internal name of the element. For details on how to find out the field name and the name of the data view, see *Custom Fields*.

For example, suppose that you added the **Personal ID** element to the **Main Contact** area of the *Customers* (AR303000) form in a customization project. To work with this customization element through the web services API, you should use the `CustomFields` property of the `Contact` entity, which is available through the `MainContact` property of the `Customer` entity. The **Personal ID** customization element has the `String` type and has the `UsrPersonalID` field name and belongs to the `DefContact` data view. Therefore, to access this element, you should set the `fieldName` property of the `CustomStringField` object to `UsrPersonalID`, and the `viewName` property of the `CustomStringField` object to `DefContact`.

Syntax

```
public CustomField[] CustomFields { set; get; }
```

Example

Suppose that you added the **Personal ID Type**, **Personal ID**, and **Credit Record Verified** elements to the *Customers* form in a customization project, as shown in the following screenshot.

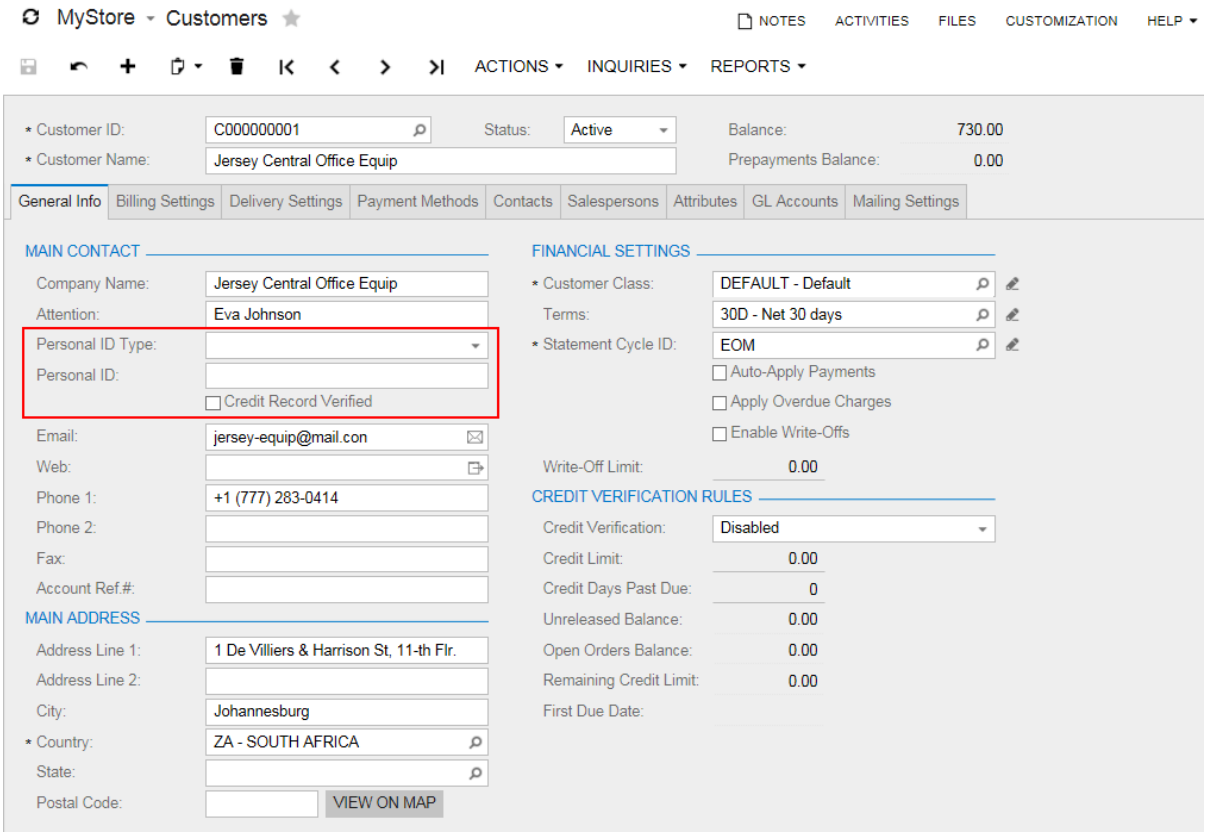


Figure: Custom elements

The following code shows how to specify the values of these custom elements through the web services API.

```
//Specify the values of a new customer record
Customer customerToBeCreated = new Customer
{
  CustomerID = new StringValue { Value = "TEDSMITH" },
  CustomerName = new StringValue { Value = "Ted Smith" },
  //Specify the values of the custom fields

  MainContact = new Contact
  {
    CustomFields = new[]
    {
      new CustomStringField
      {
        fieldName = "UsrPersonalIDType",
        viewName = "DefContact",
        Value = new StringValue { Value = "Passport" }
      },
      new CustomStringField
```

```
      {
        fieldName = "UsrPersonalID",
        viewName = "DefContact",
        Value = new StringValue { Value = customerPersonalID }
      },
      new CustomBooleanField
      {
        fieldName = "UsrCreditRecordVerified",
        viewName = "DefContact",
        Value = new BooleanValue { Value = true }
      }
    }
  }
};

//Create a customer record with the specified values
Customer newCustomer = (Customer)soapClient.Put(customerToBeCreated);
```

Usage Notes
To work with elements of an object that was added in a customization project or with a custom Acumatica ERP form, you have to create a custom endpoint for the needed form. You can create a custom endpoint on the *Web Service Endpoints* (SM207060) form. For details on creation of a custom endpoint, see *To Create a Custom Endpoint* and *To Extend an Existing Endpoint*.

Related Links

- *Custom Fields*
- *GetCustomFieldSchema() Method*

ReturnBehavior Property (Contract Version 3)

This topic describes the `ReturnBehavior` property that is available in Version 3 of the system contract.

By using the `ReturnBehavior` property, you can specify the fields of the entity whose values should be returned from the request to the service that is performed by the `Get()`, `GetList()`, or `Put()` method. This property is exposed by the `Entity` class—that is, all entities of the web services API expose this property.

For each entity, you can specify one of the following options, which determine the fields for which values should be returned:

- `None`: Values should not be returned for any fields of the entity. You can use this option for detail entities and linked entities, but not for top-level entities.
- `OnlySystem`: Values should be returned for only the system fields (that is, the fields of the `Entity` class, such as `ID`, `RowNumber`, and `Note`). You may need to obtain only the values of the system fields, for example, if you want to delete the entity.

- `OnlySpecified`: Values should be returned for only the specified fields of the entity and the system fields. You can specify the values to be returned by using the `Return` classes of the needed value type, such as `StringReturn`. The values of the fields that are specified by using the `Value` or `Search` classes of the corresponding value type, such as `StringValue` and `StringSearch`, are returned automatically.
- `Default`: Values should be returned for only the fields of the entity that are defined in the entity itself (without the fields of the linked and detail entities defined within the entity). This option is used by default.
- `All`: Values should be returned for all fields of the entity that are defined in the contract (including the fields of the linked and detail entities defined within the entity). No values of the custom fields (the Acumatica ERP fields that are not defined in the contract or the fields that are defined in a customization project) are returned. You can specify the values to be skipped by using the `Skip` classes of the needed value type, such as `StringSkip`.

In the sections below, you can find examples of how to specify the fields whose values should be returned.

Syntax

```
public ReturnBehavior ReturnBehavior { set; get; }
```

Example: Obtaining All Fields

The following example shows how to obtain the values of all fields of all stock item records in the system.

```
StockItem stockItemsToBeFound = new StockItem
{
  ReturnBehavior = ReturnBehavior.All
};

//Get the list of stock items with the values of all fields
//that are defined in the contract
Entity[] stockItems = soapClient.GetList(stockItemsToBeFound);
```

Example: Obtaining Only the Fields of the Entity Itself Without Linked and Detail Fields

In the following example, only the fields of the top-level `StockItem` entity are retrieved from the system.

```
//You can omit the setting of the ReturnBehavior property to ReturnBehavior.Default
//because this option is used by default
StockItem stockItemsToBeFound = new StockItem();

//Get the list of stock items with the values of the fields
//of the StockItem entity itself (without the fields of the detail or linked entities)
```

```
Entity[] stockItems = soapClient.GetList(stockItemsToBeFound);
```

Example: Obtaining Only Specified Fields

The following example shows how to obtain the values of the `InventoryID` field, the `ItemStatus` field, and all `WarehouseDetail` fields of all stock item records in the system.

```
StockItem stockItemsToBeFound = new StockItem
{
  ReturnBehavior = ReturnBehavior.OnlySpecified,
  InventoryID = new StringReturn(),
  ItemStatus = new StringSearch { Value = "Active" },
  WarehouseDetails = new StockItemWarehouseDetail[]
  {
    new StockItemWarehouseDetail {ReturnBehavior = ReturnBehavior.All}
  },
};

//Get the list of stock items with the values of the specified fields
Entity[] stockItems = soapClient.GetList(stockItemsToBeFound);
```

Example: Obtaining All Fields Except the Specified Ones

The following example shows how to obtain the values of all fields of all stock item records in the system except the `LastModified` field and all `WarehouseDetail` fields.

```
StockItem stockItemsToBeFound = new StockItem
{
  ReturnBehavior = ReturnBehavior.All,
  LastModified = new DateTimeSkip(),
  WarehouseDetails = new StockItemWarehouseDetail[]
  {
    new StockItemWarehouseDetail {ReturnBehavior = ReturnBehavior.None}
  },
};

//Get the list of stock items with the values of the specified fields
Entity[] stockItems = soapClient.GetList(stockItemsToBeFound);
```

Example: Obtaining Only System Fields

The following example shows how to obtain the values of only system fields of all stock item records in the system.

```
StockItem stockItemToBeFound = new StockItem
{
  ReturnBehavior = ReturnBehavior.OnlySystem,
};

//Get the list of stock items with the values of system fields
Entity[] stockItems = soapClient.GetList(stockItemToBeFound);
```

ReturnBehavior Property (Contract Version 2)

This topic describes the `ReturnBehavior` property that is available in Version 2 of the system contract.

By using the `ReturnBehavior` property, you can specify the fields of the entity whose values should be returned from the request to the service that is performed by the `Get()`, `GetList()`, or `Put()` method. This property is exposed by the `Entity` class—that is, all entities of the web services API expose this property.

For each entity, you can specify one of the following options, which determine the fields for which values should be returned:

- `None`: Values should not be returned for any fields of the entity. You can use this option for detail entities and linked entities, but not for top-level entities.
- `OnlySystem`: Values should be returned for only the system fields (that is, the fields of the `Entity` class, such as `ID`, `RowNumber`, and `Note`). You may need to obtain only the values of the system fields, for example, if you want to delete the entity.
- `OnlySpecified`: Values should be returned for only the specified fields of the entity and the system fields. You can specify the values to be returned by using the `Return` classes of the needed value type, such as `StringReturn`. The values of the fields that are specified by using the `Value` or `Search` classes of the corresponding value type, such as `StringValue` and `StringSearch`, are returned automatically.
- `All`: Values should be returned for all fields of the entity that are defined in the contract. No values of the custom fields (the Acumatica ERP fields that are not defined in the contract or the fields that are added in a customization project and are not included in the contract) are returned. You can specify the values to be skipped by using the `Skip` classes of the needed value type, such as `StringSkip`. This option is used by default.

In the sections below, you can find examples of how to specify the fields whose values should be returned.

Syntax

```
public ReturnBehavior ReturnBehavior { set; get; }
```

Example: Obtaining All Fields

The following example shows how to obtain the values of all fields of all stock item records in the system.

```
//You can omit the setting of the ReturnBehavior property to ReturnBehavior.All
//because this option is used by default
StockItem stockItemsToBeFound = new StockItem();

//Get the list of stock items with the values of all fields
//that are defined in the contract
```

```
Entity[] stockItems = soapClient.GetList(stockItemsToBeFound);
```

Example: Obtaining Only Specified Fields

The following example shows how to obtain the values of the `InventoryID` field, the `ItemStatus` field, and all `WarehouseDetail` fields of all stock item records in the system.

```
StockItem stockItemsToBeFound = new StockItem
{
  ReturnBehavior = ReturnBehavior.OnlySpecified,
  InventoryID = new StringReturn(),
  ItemStatus = new StringSearch { Value = "Active" },
  WarehouseDetails = new StockItemWarehouseDetail[]
  {
    new StockItemWarehouseDetail {ReturnBehavior = ReturnBehavior.All}
  },
};

//Get the list of stock items with the values of the specified fields
Entity[] stockItems = soapClient.GetList(stockItemsToBeFound);
```

Example: Obtaining All Fields Except the Specified Ones

The following example shows how to obtain the values of all fields of all stock item records in the system except the `LastModified` field and all `WarehouseDetail` fields.

```
StockItem stockItemsToBeFound = new StockItem
{
  ReturnBehavior = ReturnBehavior.All,
  LastModified = new DateTimeSkip(),
  WarehouseDetails = new StockItemWarehouseDetail[]
  {
    new StockItemWarehouseDetail {ReturnBehavior = ReturnBehavior.None}
  },
};

//Get the list of stock items with the values of the specified fields
Entity[] stockItems = soapClient.GetList(stockItemsToBeFound);
```

Example: Obtaining Only System Fields

The following example shows how to obtain the values of only system fields of all stock item records in the system.

```
StockItem stockItemToBeFound = new StockItem
{
  ReturnBehavior = ReturnBehavior.OnlySystem,
};

//Get the list of stock items with the values of system fields
Entity[] stockItems = soapClient.GetList(stockItemToBeFound);
```

Screen-Based SOAP API Reference

In this chapter, you will find the reference information for the main methods of the screen-based web services API; these methods are used to transfer data to and from Acumatica ERP. This chapter covers the following objects, and the following methods, which are exposed by the `Screen` class:

- *Login() Method*
- *Logout() Method*
- *SetLocaleName() Method*
- *SetBusinessDate() Method*
- *GetScenario() Method*
- *GetSchema() Method*
- *SetSchema() Method*
- *Export() Method*
- *Submit() Method*
- *Import() Method*
- *Clear() Method*
- *GetProcessStatus() Method*

Login() Method

You use the `Login()` method to make the client application sign in to Acumatica ERP.

Syntax

```
public LoginResult Login(string name, string password)
```

Parameters

- `name`: The username that the application should use to sign in to Acumatica ERP, such as `"admin"`.

  To sign in to a specific Acumatica ERP tenant, specify the `name` parameter as follows: `UserName@TenantName`, where you should specify the username instead of `UserName` and the tenant name instead of `TenantName`. For example, if you sign in to the tenant with the name *Dollar* as the user with the name *admin*, you should specify the parameter as `admin@Dollar`. You can view the name that should be used for the tenant in the **Login Name** box of the *Tenants* (SM203520) form.

To sign in to a certain branch in the tenant, specify the parameter as follows: `UserName@TenantName:BranchName`, where you should specify the username instead of `UserName`, the tenant name instead of `TenantName`, and the branch ID instead of `BranchName`. You can view the ID of the branch in the **Branch ID** box of the *Branches* (CS102000) form. For example, if you sign in to the *East* branch of the *Dollar* tenant as the user with the name *admin*, you should specify the parameter as `admin@Dollar:East`.

- `password`: The password for the username, such as `"123"`.

Return Value

The method returns the `LoginResult` object, which contains the description of errors that occurred during signing in, if any.

Example

The following code signs in to Acumatica ERP by using the parameters that are specified in the application settings.

```
Screen context = new Screen();
context.CookieContainer = new System.Net.CookieContainer();
context.Url = "https://localhost/WebServiceAPITest/Soap/MYSTORE.asmx";
context.Login("admin@MyTenant:MYSTORE", "123");
```

Usage Notes

Before you sign in to Acumatica ERP by using the `Login()` method, do the following:

1. Initialize the `CookieContainer` property of the object with a new `System.Net.CookieContainer()`. The `CookieContainer` property is a standard property of an object of the `HttpWebClientProtocol` system type. (The `Screen` class is derived from the `HttpWebClientProtocol` class.) This property is used to maintain the session state for a client.

2. Specify the URL of the web service in the `URL` property of the object. This is the same URL that you specify when you add a web reference to the Acumatica ERP web service. You can change the URL of the service dynamically in your application if you need to switch between multiple Acumatica ERP web services.

For each call of the `Login()` method, you must call the `Logout()` method after you finish your work with Acumatica ERP to close the session. Therefore, when you are working with the web services API, we recommend that you use the pattern that is shown in the following code.

```
using
(
  //Connect to the web services and log in to Acumatica ERP
  Screen context = new Screen();
  ...
)
{
```

```
  try
  {
    //Import, export, or submit data
    ...
  }
  finally
  {
    //Log out from Acumatica ERP
    context.Logout();
  }
}
```

You should take into account Acumatica ERP license API limits when using the `Login()` method. For details, see *License Restrictions for API Users*.

Logout() Method

You use the `Logout()` method to make the client application sign out from Acumatica ERP.

Syntax

```
public void Logout()
```

Usage Notes

For each call of the `Login()` method, you must call the `Logout()` method after you finish your work with Acumatica ERP to close the session. Therefore, when you are working with the web services API, we recommend that you use the pattern that is shown in the following code.

```
using
(
  //Connect to the web services and sign in to Acumatica ERP
  Screen context = new Screen();
  ...
)
{
  try
  {
    //Import, export, or submit data
    ...
  }
  finally
  {
    //Sign out from Acumatica ERP
    context.Logout();
  }
}
```

SetLocaleName() Method

You use the `SetLocaleName()` method to specify the locale for Acumatica ERP to correctly recognize the format of dates, numbers, and other country-specific data that is passed by using the web services API. By default, Acumatica ERP uses the invariant locale, which is similar to the English (United States) locale.

Syntax

```
public void SetLocaleName(string localeName)
```

Parameter

- `localeName`: The locale that should be used in Acumatica ERP. You should specify the locale in the `System.Globalization.CultureInfo` format converted to `string`, such as `"EN-US"`.

Example

The following code shows how to specify the appropriate locale with the `SetLocaleName()` method of the `Screen` object.

```
...
using System.Threading;
...
Screen context = new Screen();
context.SetLocaleName(Thread.CurrentThread.CurrentCulture.ToString());
```

SetBusinessDate() Method

You use the `SetBusinessDate()` method to specify the business date in Acumatica ERP. You can set the business date to any date to make the system insert this date into the date fields by default. The business date is inserted into any new document that you create and is used in the default selection parameters that appear on processing and inquiry screens.

Syntax

```
public void SetBusinessDate(System.DateTime businessDate)
```

Parameter

- `businessDate`: The business date that should be used in Acumatica ERP.

Usage Notes

The business date resets to the current date of your computer after each login.
Therefore, if you need to specify a business date in your application, you should call the `SetBusinessDate()` method after each client application login.

GetScenario() Method

You use the `GetScenario()` method to retrieve the list of commands of an integration scenario that is configured in the system.

Syntax

```
public Command[] GetScenario(string scenario)
```

Parameter

- `scenario`: The name of the import or export scenario for which the list of commands should be retrieved.

Return Value

The method returns the list of commands of the specified integration scenario.

GetSchema() Method

You use the `GetSchema()` method of the `Screen` object to get the description of the structure (schema) of a form. This method is specific for each Acumatica ERP form, and you should use the method with the ID of the needed form in the prefix of the method name.

To prevent application failures due to UI changes in Acumatica ERP, you can use the `GetSchema()` method of the screen-based API wrapper instead of the `GetSchema()` method of the `Screen` object. For more information on the screen-based API wrapper, see *Screen-Based API Wrapper*.

Syntax

```
public Content GetSchema()
```

Return Value

The method returns the schema of the form as the corresponding `Content` object, which is specific for each form.

Example

To get the schema of the *Stock Items* (IN202500) form, you should call the `IN202500GetSchema()` method of the `Screen` object. You will receive the result as a `IN202500Content` object, as the following code shows.

```
Screen context = new Screen();
...
IN202500Content stockItemsSchema = context.IN202500GetSchema();
```

SetSchema() Method

You use the `SetSchema()` method of the `Screen` object to change the description of the structure (schema) of a form to the one specified in the method. This method is useful when you need to work with the description of the form that was used in previous versions of Acumatica ERP. This method is specific for each Acumatica ERP form, and you should use the method with the ID of the needed form in the prefix of the method name.

Syntax

```
public void SetSchema(Content schema)
```

Parameter

- `schema`: The schema of an Acumatica ERP form.

Export() Method

You use the proper `Export()` method of a `Screen` object to export data from Acumatica ERP. You select the needed `Export()` method by using in the name of the method the prefix that contains the ID of the Acumatica ERP form from which the method exports data.

Syntax

```
public string[][] Export(
  Command[] commands,
  Filter[] filters,
  int topCount,
  bool includeHeaders,
  bool breakOnError
)
```

Parameters

- `commands`: In this parameter, you specify the UI elements of the Acumatica ERP form whose values you need to export. In the array of commands that you pass to the `commands` parameter, you can also use the `EveryValue` service command, which makes the system export all records of the specific type.
- `filters`: In this parameter, you specify any restrictions on the data to be exported. For example, you can configure the system to export only the records that have a particular status.
- `topCount`: In this parameter, you can restrict the number of records to be exported. If this parameter is set to 0, the system exports all records that are specified by the `commands` and `filters` parameters of the `Export()` method.

- `includeHeaders`: In this parameter, you specify whether the result of the export should include column headers. If this parameter is set to `true`, the result of the export includes the names of exported elements in the first row of the exported data.
- `breakOnError`: In this parameter, you specify whether the system should stop the export if an error occurs during this process. If this parameter is set to `true`, the system stops exporting data when the first error occurs during the export.

Return Value

The result of the data export is a two-dimensional string array, which represents the exported data in a table format. Thus, if an exported record contains detail lines, the values of the detail lines are translated to multiple rows of this table. The number of rows is equal to the number of detail lines of the source record. The table rows that belong to one record have the same values of the elements of the summary area specified.

For example, suppose that on the *Invoices* (SO303000) form, an invoice has three detail lines. If you export this invoice with detail lines, the data prepared for export will include three records for this invoice—one record for each detail line. These records will include identical values of the elements in the summary area of the invoice, such as the type and reference number, and different values of the detail line elements.

Submit() Method

You use the proper `Submit()` method of a `Screen` object to submit data to Acumatica ERP. You select the needed `Submit()` method by using in the name of the method the prefix that contains the ID of the Acumatica ERP form with which the method works.

Syntax

```
public Content[] Submit(Command[] commands)
```

Parameter

- `commands`: You use this parameter to specify the data that you are going to submit. In this parameter, you pass to the web service an array of `Command` objects in which you can specify commands that do the following:
  - Set the values of elements on the form by using `Value` commands.
  - Click buttons on the form (for example, the **Save** button) by using `Action` commands.
  - Get the result of data processing by using `Field` commands.

Return value

The result of processing of the data submitted by using the `Submit()` method is returned as a `Content` object specific to the form to which the data has been submitted. This object contains the values of the elements that you specified by using `Field` commands in the

array of `Command` objects, which you pass to the `Submit()` method. The values of the elements that were not specified by using `Field` commands are `null`.

If you want only to submit data to an Acumatica ERP form and do not need to obtain any values of elements after submitting, do not specify any `Field` commands in the array of the `Command` object that you pass to a `Submit()` method. In this case, the `Submit()` method does not return any value.

Example 1: Submitting Data and Obtaining the Result of Processing

Suppose that you want to submit a customer record to the *Customers* (AR303000) form and obtain as a result of processing the values of the **Customer Name** and **Customer Class** elements. You pass the list of commands, which includes `Field` commands for the `CustomerName` and `CustomerClass` fields, to the `AR303000Submit()` method, as the following code shows. In this example, the `AR303000Submit()` method returns a `AR303000Content` object that has non-null values of the `CustomerName` and `CustomerClass` fields. The values of the other elements of the returned `AR303000Content` object are `null`.

```
AR303000Content custSchema = context.AR303000GetSchema();
var commands = new Command[]
{
    ...
    custSchema.Actions.Save,
    custSchema.CustomerSummary.CustomerName,
    custSchema.GeneralInfoFinancialSettings.CustomerClass
};
AR303000Content customer = context.AR303000Submit(commands)[0];
```

Example 2: Submitting Data without Obtaining the Result of Processing

Suppose that you want to create a customer record on the *Customers* form and do not need to get the values of any element on the form after the customer record is created. You pass the list of commands, which sets the needed values and saves the changes on the form (the list of commands does not include any `Field` commands), to the `AR303000Submit()` method, as the following code shows. In this example, the `AR303000Submit()` method does not return any value.

```
AR303000Content custSchema = context.AR303000GetSchema();
var commands = new Command[]
{
    new Value
    {
        ...
    },
    custSchema.Actions.Save
};
context.AR303000Submit(commands);
```

Import() Method

To import data to Acumatica ERP by using the web services API, you should use the proper `Import()` method of a `Screen` object. You select the needed `Import()` method by using in the name of the method a prefix that contains the ID of the Acumatica ERP form to which the method imports data.

Syntax

```
public ImportResults[] Import(
  Command[] commands,
  Filter[] filters,
  string[][] data,
  bool includedHeaders,
  bool breakOnError,
  bool breakOnIncorrectTarget
)
```

Parameters

- `commands`: In this parameter, you specify the UI elements of the Acumatica ERP form to which you need to import data by using `Field` commands. You can click buttons on the form (for example, the **Save** button) by using `Action` commands.
- `filters`: In this parameter, you specify any restrictions on the data to be imported. For example, you can configure the system to import only the records that have the *CUST* prefix in the customer ID field.
- `data`: In this parameter, you specify the data that should be imported in a two-dimensional string array. Each row of the array should contain the values of the fields in the order that you specified in the `commands` parameter.
- `includeHeaders`: In this parameter, you specify whether the imported data include column headers. If this parameter is set to `true`, this signals to the system that the data, which is imported, includes the names of the imported elements in the first row.
- `breakOnError`: In this parameter, you specify whether the system should stop the data import if an error occurs during this process. If this parameter is set to `true`, the system stops importing data when the first error occurs during the import.
- `breakOnIncorrectTarget`: In this parameter, you specify whether the system should stop the data import if the record does not meet the condition specified for the imported record. If this parameter is set to `true`, the system stops processing records and displays an error.

Return Value

The result of the processing of the data imported by using the `Import()` method is returned as a `ImportResults` object specific to the form to which the data has been imported. This object contains the result of the processing.

Clear() Method

You use the `Clear()` method to clear all changes on the form. The method works in the same way as the **Cancel** button on the toolbar of an Acumatica ERP form does. This method is specific to the particular Acumatica ERP form, and you should use the method with the ID of the needed form in the prefix of the method name.

Syntax

```
public void Clear()
```

GetProcessStatus() Method

You use the `GetProcessStatus()` method to monitor the status of a long-running operation (such as the release or confirmation operation).

Syntax

```
public ProcessResult GetProcessStatus()
```

Return Value

The method returns a `ProcessResult` object. You should use the `Status` property of this object to get the status of the processing operation. When the status of the operation is `Completed`, you can get the result of processing.

Contract-Based API Examples

In this chapter, you can find code examples that show how to implement the integration of Acumatica ERP with external systems through the contract-based REST and SOAP API. Each example includes a short description, a user scenario that you can implement by using this example, and code snippet that use REST or SOAP API to implement the scenario. You can reuse these examples in your application.

You can find the REST examples in the *Help-and-Training-Examples* repository on GitHub. To test the examples, you do the following:

1. Clone the *2019R2* branch of the repository or download it's contents.
2. Deploy a new Acumatica ERP instance with the *U100* dataset. For details on deploying an instance, see *To Deploy an Acumatica ERP Instance* in the Installation Guide.
3. Find the examples in the **IntegrationDevelopmentGuide.postman_collection.json** file (which is a Postman collection) of the **IntegrationDevelopment\Help** folder in the branch.
4. Specify the URL of the instance in the *BaseURL* variable of the collection.

In This Chapter

- *Integration of Acumatica ERP Projects with External Systems (REST API Examples)*
- *Integration of Acumatica ERP with POS systems (SOAP API Examples)*

Integration of Acumatica ERP Projects with External Systems (REST API Examples)

You can use contract-based REST APIs for the integration of Acumatica ERP projects with external systems. For details on projects and project accounting, see *Projects*.

The following topics present examples of the integration of Acumatica ERP projects with external systems:

- *Creation of a Pro Forma Invoice*
- *Management of Account Groups*
- *Running of Project Billing*
- *Creation of a General Ledger Transaction with a Project Code That Does Not Produce a Project Transaction*
- *Time Entry*

Creation of a Pro Forma Invoice

This example has been tested for Acumatica ERP 2019 R2 and may require code modifications for other versions of Acumatica ERP.

Through the contract-based REST and SOAP APIs, external systems can create pro forma invoices and send them by email. For details about pro forma invoices, see *Pro Forma Invoices*.

For a pro forma to be created from the project, the project must have `Customer`, `BillingRule`, `BillingPeriod`, and `NextBillingDate` specified, and must have the *Active* status. Because of data validation in the project, `NextBillingDate` cannot be specified for a project with the *On Hold* status, and you cannot change the customer in a project with the *Active* status. Therefore, these settings can be specified only in multiple API calls, as shown in the code examples below.

User Scenario

A project manager of the company through an external system needs to create a pro forma invoice in Acumatica ERP and send it to the client by email.

Preliminary Steps

A `ProFormaInvoice` entity cannot be created by using the `PUT` HTTP method; it can be created only through the invocation of the `RunProjectBilling` action of the `Project` entity. Because email settings are not mapped to any fields of the `Project` entity, you have to prepare a project template with the specified email settings on the *Project Templates* (PM208000) form and then use this template for the creation of the project through the API. The project template can also contain preconfigured project tasks, as is the case with the template used in this example. For details about project templates, see *Configuring Project Templates and Tasks*.

Testing of the Examples

To test the code below, you configure your client application and the Acumatica ERP instance to be used as follows:

- Deploy a new Acumatica ERP instance with the *U100* dataset. For details on deploying an instance, see *To Deploy an Acumatica ERP Instance* in the Installation Guide.
- On the *Enable/Disable Features* (CS100000) form, make sure the *Projects* feature is enabled.
- To sign in to the instance in the client application, use the company name (which you specified when you created the instance) and the *PRODWHOLE* branch.

Examples of REST API Requests

You can use the following sequence of examples of HTTP requests to create a pro forma invoice and send it by email through the contract-based REST API:

In the request examples below, `<Acumatica ERP instance name>` is the name of the Acumatica ERP instance (such as `MyInstance`), and `<host name>` is the name of the host where the instance is located (such as `my.acumatica.com`). `[/<Acumatica ERP instance name>]` may be omitted if the instance is installed in the root of the website.

1. Create a project from the project template and specify the `Customer`, `BillingRule`, and `BillingPeriod` settings of the project.

```
PUT [/<Acumatica ERP instance name>]/entity/Default/18.200.001/Project HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

{
  "ProjectID" : {"value" : "TESTPR3"},
  "ProjectTemplateID" : {"value" : "PROGRESS"},
  "Customer" : {"value" : "COFFEESHOP"},
  "BillingAndAllocationSettings" :
   {
      "BillingRule" : {"value" : "PROGRESS"},
      "BillingPeriod" : {"value" : "Month"},
    }
}

HTTP/1.1 200 OK
Content-Type: application/json
{ ... }
```

2. Make the project active.

```
PUT [/<Acumatica ERP instance name>]/entity/Default/18.200.001/Project HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

{
  "ProjectID" : {"value" : "TESTPR3"},
  "Hold" : {"value" : false}
}

HTTP/1.1 200 OK
Content-Type: application/json
{ ... }
```

3. Specify `NextBillingDate` for the project.

```
PUT [/<Acumatica ERP instance name>]/entity/Default/18.200.001/Project HTTP/1.1
Host: <host name>
Accept: application/json
```

```
Content-Type: application/json

{
  "ProjectID" : {"value" : "TESTPR3"},
    "BillingAndAllocationSettings" :
    {
      "NextBillingDate" : {"value" : "2019-08-15"},
    }
}

HTTP/1.1 200 OK
Content-Type: application/json
{ ... }
```

4. Activate a project task.

```
PUT [/<Acumatica ERP instance name>]/entity/Default/18.200.001/ProjectTask HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

{
  "ProjectTaskID" : {"value" : "PHASE1"},
  "ProjectID" : {"value" : "TESTPR3"},
  "Status" : {"value" : "Active"},
}

HTTP/1.1 200 OK
Content-Type: application/json
{ ... }
```

5. Create a project transaction and invoke the release of the transaction.

```
POST [/<Acumatica ERP instance name>]/entity/Default/18.200.001/ProjectTransaction/
ReleaseTransactions HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

{
  "entity" : {
    "Details" :
    [
      {
        "Project" : {"value" : "TESTPR3"},
        "ProjectTask" : {"value" : "PHASE1"},
        "AccountGroup" : {"value" : "LABOR"},
        "Qty" : {"value" : 10},
        "Amount" : {"value" : 10},
    },
  ]
```

```
}
}

HTTP/1.1 202 Accepted
Location: [/<Acumatica ERP instance name>]/entity/Default/18.200.001/
ProjectTransaction/ReleaseTransactions/status/ce6a7728-5f8e-416f-bbe5-617d2725465c
```

6. Use the URL from the `Location` header to obtain the status of the long-running operation. When the `GET` HTTP method with this URL returns *204 No Content*, the operation is completed. For details on the other response statuses, see *Execution of an Action*.

```
GET [/<Acumatica ERP instance name>]/entity/Default/18.200.001/ProjectTransaction/
ReleaseTransactions/status/ce6a7728-5f8e-416f-bbe5-617d2725465c HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

HTTP/1.1 204 No Content
```

7. Invoke project billing to create a pro forma invoice.

```
POST [/<Acumatica ERP instance name>]/entity/Default/18.200.001/Project/
RunProjectBilling HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

{
  "entity" : {
    "ProjectID": {
        "value": "TESTPR3"
    },
  }
}

HTTP/1.1 202 Accepted
Location: [/<Acumatica ERP instance name>]/entity/Default/18.200.001/Project/
RunProjectBilling/status/6952c6d1-04be-4330-a26e-c6b855ba332c
```

8. Monitor the status of the operation.

```
GET [/<Acumatica ERP instance name>]/entity/Default/18.200.001/Project/
RunProjectBilling/status/6952c6d1-04be-4330-a26e-c6b855ba332c HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

HTTP/1.1 204 No Content
```

9. Obtain the list of pro forma invoices of the project (which currently contains only one pro forma invoice) by adding `$expand=Invoices` to the endpoint address. For details about parameters, see *Parameters for Retrieving Records*.

To obtain the list of pro forma invoices of the project with the `TESTPR3` project ID, you can also use the `GET` request for the `ProFormaInvoice` entity with the `$filter=ProjectID eq 'TESTPR3` parameter.

```
GET [/<Acumatica ERP instance name>]/entity/Default/18.200.001/Project/TESTPR3?
$expand=Invoices HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

HTTP/1.1 200 OK
Content-Type: application/json
{ ... }
```

10. Send the pro forma invoice by email.

```
POST [/<Acumatica ERP instance name>]/entity/Default/18.200.001/ProFormaInvoice/
EmailProFormaInvoice HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

{
  "entity" : {
    "RefNbr": {
        "value": "000004"
    },
  }
}

HTTP/1.1 202 Accepted
Location: [/<Acumatica ERP instance name>]/entity/Default/18.200.001/ProFormaInvoice/
EmailProFormaInvoice/status/a4caa455-0eed-4c11-a5a9-2a8333e53db1
```

11. Monitor the status of the operation.

```
GET [/<Acumatica ERP instance name>]/entity/Default/18.200.001/ProFormaInvoice/
EmailProFormaInvoice/status/a4caa455-0eed-4c11-a5a9-2a8333e53db1 HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

HTTP/1.1 204 No Content
```

Management of Account Groups

This example has been tested for Acumatica ERP 2019 R2 and may require code modifications for other versions of Acumatica ERP.

Through the contract-based REST and SOAP APIs, external systems can create, modify, and remove account groups.

The `AccountGroup` entity supports the creation, retrieval, update, and removal of the entity itself; however, you cannot modify the list of accounts of a particular account group by using the `AccountGroup` entity. Instead, you have to use the `AccountGroup` property in the `Account` entity. You can use the `DefaultAccountID` property of the `AccountGroup` entity in order to specify the default account for the group.

The removal of the default account from the group does not cause the `DefaultAccountID` property to be updated automatically. If you remove the default account from the group, you have to update the `DefaultAccountID` property.

User Scenario

Through an external system, a project manager of a company needs to add and modify account groups in Acumatica ERP to set up Acumatica ERP for project accounting.

Testing of the Examples

To test the code below, you configure your client application and the Acumatica ERP instance to be used as follows:

- Deploy a new Acumatica ERP instance with the *U100* dataset. For details on deploying an instance, see *To Deploy an Acumatica ERP Instance* in the Installation Guide.
- On the *Enable/Disable Features* (CS100000) form, make sure the *Projects* feature is enabled.
- To sign in to the instance in the client application, use the company name (which you specified when you created the instance) and the *PRODWHOLE* branch.

Examples of REST API Requests

You can use the following sequence of examples of HTTP requests to create an account group and edit accounts in it through the contract-based REST API:

In the request examples below, `<Acumatica ERP instance name>` is the name of the Acumatica ERP instance (such as `MyInstance`), and `<host name>` is the name of the host where the instance is located (such as `my.acumatica.com`). `[/<Acumatica ERP instance name>]` may be omitted if the instance is installed in the root of the website.

1. Create an account group.

```
PUT [/<Acumatica ERP instance name>]/entity/Default/18.200.001/AccountGroup HTTP/1.1
Host: <host name>
```

```
Accept: application/json
Content-Type: application/json

{
  "AccountGroupID" : {"value" : "ACCG02"},
  "Description" : {value: "Test Account Group"}
}

HTTP/1.1 200 OK
Content-Type: application/json
{ ... }
```

2. Add accounts to the account group.

```
PUT [/<Acumatica ERP instance name>]/entity/Default/18.200.001/Account HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

{
  "AccountCD" : {"value" : "40000"},
  "AccountGroup" : {value: "ACCG02"}
}

HTTP/1.1 200 OK
Content-Type: application/json
{ ... }

PUT [/<Acumatica ERP instance name>]/entity/Default/18.200.001/Account HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

{
  "AccountCD" : {"value" : "40010"},
  "AccountGroup" : {value: "ACCG02"}
}

HTTP/1.1 200 OK
Content-Type: application/json
{ ... }
```

3. Specify the default account of the account group.

```
PUT [/<Acumatica ERP instance name>]/entity/Default/18.200.001/AccountGroup HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

{
  "DefaultAccountID" : {"value" : "40000"},
```

```
  "AccountGroupID" : {value: "ACCG02"}
}

HTTP/1.1 200 OK
Content-Type: application/json
{ ... }
```

4. Obtain the list of accounts of the group.

```
GET [/<Acumatica ERP instance name>]/entity/Default/18.200.001/Account?
$filter=AccountGroup%20eq%20'ACCG02'&$select=AccountCD HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

HTTP/1.1 200 OK
Content-Type: application/json
{ ... }
```

5. Remove an account from the group.

```
PUT [/<Acumatica ERP instance name>]/entity/Default/18.200.001/Account HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

{
  "AccountCD" : {"value" : "40010"},
  "AccountGroup" : {value: null}
}

HTTP/1.1 200 OK
Content-Type: application/json
{ ... }
```

Running of Project Billing

This example has been tested for Acumatica ERP 2019 R2 and may require code modifications for other versions of Acumatica ERP.

Through the contract-based REST and SOAP APIs, external systems can run project billing for one project as well as for multiple projects. This example shows how to run project billing for multiple projects in Acumatica ERP from an external system. To run project billing for one project, you can use the `RunProjectBilling` action of the `Project` entity. (You can find an example of the call of `RunProjectBilling` in *Creation of a Pro Forma Invoice*.)

You use the `ProjectBillingProcess` or `ProjectBillingProcessAll` action of the `ProjectBilling` entity to bill multiple projects at once. If you use the `ProjectBillingProcess` action, you select the projects entities to be billed by specifying the value of the `Selected` field of `ProjectBillingDetails`, as shown in the code example below. If you use the `ProjectBillingProcessAll` action, you can filter the projects to be

billed by using the `Customer`, `CustomerClass`, `ProjectTemplate`, and `StatementCycleThere` fields of the `ProjectBilling` entity or bill all projects. You can use the `InvoiceDate` and `PostPeriod` properties to specify the invoice date and fiscal period data for the billing operation.

User Scenario

Once a week, through an external system, a project manager of a company needs to run project billing for selected projects in Acumatica ERP.

Preliminary Step

You need to create the projects that will be billed either through the UI or through the contract-based API.

Testing of the Examples

To test the code below, you configure your client application and the Acumatica ERP instance to be used as follows:

- Deploy a new Acumatica ERP instance with the *U100* dataset. For details on deploying an instance, see *To Deploy an Acumatica ERP Instance* in the Installation Guide.
- On the *Enable/Disable Features* (CS100000) form, make sure the *Projects* feature is enabled.
- To sign in to the instance in the client application, use the company name (which you specified when you created the instance) and the *PRODWHOLE* branch.

Examples of REST API Requests

You can use the following request examples to bill multiple projects through the contract-based REST API:

In the request examples below, `<Acumatica ERP instance name>` is the name of the Acumatica ERP instance (such as `MyInstance`), and `<host name>` is the name of the host where the instance is located (such as `my.acumatica.com`). `[/<Acumatica ERP instance name>]` may be omitted if the instance is installed in the root of the website.

1. Retrieve the list of projects that can be billed by using the `PUT` HTTP method because the `ProjectBilling` entity retrieves data from an inquiry. For details about retrieving data from an inquiry, see *Retrieval of Data from an Inquiry Form*

```
PUT [/<Acumatica ERP instance name>]/entity/Default/18.200.001/ProjectBilling?
$expand=Details HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

{
  "InvoiceDate" : {"value" : "2019-08-15T00:00:00.000"}
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json
{ ... }
```

2. Use the returned object to select the projects for billing and invoke billing. For details on the IDs of the entities, see *Retrieval of a Record by ID*.

```
POST [/<Acumatica ERP instance name>]/entity/Default/18.200.001/ProjectBilling/
ProjectBillingProcess HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

{
  "entity" :
  {
    "id": "46f65edb-bdd1-473b-95c1-357aa37559f6",
    "Details": [
        {
            "id": "d6821308-4c58-48c2-9e1f-3e2daa40960b",
            "Selected": {
                "value": true
            }
        }
    ]
}
}

HTTP/1.1 202 Accepted
Location: [/<Acumatica ERP instance name>]/entity/Default/18.200.001/ProjectBilling/
ProjectBillingProcess/status/ce6a7728-5f8e-416f-bbe5-617d2725465c
```

3. Use the URL from the `Location` header to obtain the status of the long-running operation. When the `GET` HTTP method with this URL returns *204 No Content*, the operation is completed.

```
GET [/<Acumatica ERP instance name>]/entity/Default/18.200.001/ProjectBilling/
ProjectBillingProcess/status/ce6a7728-5f8e-416f-bbe5-617d2725465c HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

HTTP/1.1 204 No Content
```

4. Invoke project billing for all projects available for billing.

```
POST [/<Acumatica ERP instance name>]/entity/Default/18.200.001/ProjectBilling/
ProjectBillingProcessAll HTTP/1.1
Host: <host name>
Accept: application/json
```

```
Content-Type: application/json

{
  "entity" :
  {
  }
}

HTTP/1.1 202 Accepted
Location: [/<Acumatica ERP instance name>]/entity/Default/18.200.001/ProjectBilling/
ProjectBillingProcessAll/status/19ec8f98-6204-4758-b464-6e62d30b2973
```

5. Use the URL from the `Location` header to obtain the status of the long-running operation. When the `GET` HTTP method with this URL returns *204 No Content*, the operation is completed.

```
GET [/<Acumatica ERP instance name>]/entity/Default/18.200.001/ProjectBilling/
ProjectBillingProcessAll/status/19ec8f98-6204-4758-b464-6e62d30b2973 HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

HTTP/1.1 204 No Content
```

Creation of a General Ledger Transaction with a Project Code That Does Not Produce a Project Transaction

This example has been tested for Acumatica ERP 2019 R2 and may require code modifications for other versions of Acumatica ERP.

Through the contract-based REST and SOAP APIs, external systems can import to Acumatica ERP general ledger transactions with project codes that do not produce project transactions.

To create a general ledger transaction with a project code that does not produce a project transaction, you set the `IsNonPM` field of the `JournalTransaction` entity to `true`.

User Scenario

A construction company builds an integrated solution of Acumatica ERP with an external payroll system. The external payroll system calculates payoffs, including benefits, additions, deductions, and taxes. Once a week, the construction company needs to import general ledger transactions with project information from this payroll system to Acumatica ERP, where they are verified and released. The construction company doesn't want to update the project subledger in Acumatica ERP with the information from general ledger transactions (for example, if the standard labor costs have already been posted to the project subledger from time entries).

Testing of the Examples

To test the code below, you configure your client application and the Acumatica ERP instance to be used as follows:

- Deploy a new Acumatica ERP instance with the *U100* dataset. For details on deploying an instance, see *To Deploy an Acumatica ERP Instance* in the Installation Guide.
- On the *Enable/Disable Features* (CS100000) form, make sure the *Projects* feature is enabled.
- To sign in to the instance in the client application, use the company name (which you specified when you created the instance) and the *PRODWHOLE* branch.

Examples of REST API Requests

You can use the following request example to create a general ledger transaction through the contract-based REST API.

In the request example below, `<Acumatica ERP instance name>` is the name of the Acumatica ERP instance (such as `MyInstance`), and `<host name>` is the name of the host where the instance is located (such as `my.acumatica.com`). `[/<Acumatica ERP instance name>]` may be omitted if the instance is installed in the root of the website.

```
PUT [/<Acumatica ERP instance name>]/entity/Default/18.200.001/JournalTransaction HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

{
  "Module" : {"value" : "GL"},
  "TransactionDate" : {"value" : "2019-08-15T00:00:00"},
  "Description" : {"value" : "Transaction description"},
  "BranchID" : {"value" : "HEADOFFICE"},
  "Details" : [
   {
    "BranchID" : {"value" : "HEADOFFICE"},
    "Account" : {"value" : "10200"},
    "Subaccount" : {"value" : "000-000"},
    "CostCode" : {"value" : "0000"},
    "IsNonPM" : {"value" : true},
   }
  ]
}

HTTP/1.1 200 OK
Content-Type: application/json
{ ... }
```

Time Entry

This example has been tested for Acumatica ERP 2019 R2 and may require code modifications for other versions of Acumatica ERP.

Through the contract-based REST and SOAP APIs, the time spent on the tasks of projects can be imported to Acumatica ERP from an external system.

The `TimeSpent`, `BillableTime`, and `BillableOvertime` fields of the `TimeEntry` entity have the `StringValue` type. These fields accept values in the following format: `"hh:mm"`, where `hh` is the amount of hours, and `mm` is the amount of minutes.

The `TimeEntryID` field has the `GuidValue` type; however, its value is a sequentially generated string that looks like a GUID. Therefore, the global uniqueness of the values is not guaranteed.

User Scenario

Employees enter the time spent on the project tasks into an external payroll system. Once a week, the time entries are imported to Acumatica ERP from the payroll system.

Testing of the Examples

To test the code below, you configure your client application and the Acumatica ERP instance to be used as follows:

- Deploy a new Acumatica ERP instance with the *U100* dataset. For details on deploying an instance, see *To Deploy an Acumatica ERP Instance* in the Installation Guide.
- On the *Enable/Disable Features* (CS100000) form, make sure the *Projects* feature is enabled.
- To sign in to the instance in the client application, use the company name (which you specified when you created the instance) and the *PRODWHOLE* branch.

Examples of REST API Requests

You can use the following request example for time entry through the contract-based REST API.

In the request example below, `<Acumatica ERP instance name>` is the name of the Acumatica ERP instance (such as `MyInstance`), and `<host name>` is the name of the host where the instance is located (such as `my.acumatica.com`). `[/<Acumatica ERP instance name>]` may be omitted if the instance is installed in the root of the website.

```
PUT [/<Acumatica ERP instance name>]/entity/Default/18.200.001/TimeEntry HTTP/1.1
Host: <host name>
Accept: application/json
Content-Type: application/json

{
```

```
  "Summary" : {"value" : "Time entry summary"},
  "Date" : {"value" : "2019-08-17T05:50:43.233" },
  "Time" : {"value" : "2019-08-17T05:50:43.233" },
  "Employee" : {"value" : "EP00000002" },
  "ProjectID" : {"value" : "TOMYUM1" },
  "ProjectTaskID" : {"value" : "PHASE1" },
  "CostCode" : {"value" : "0000" },
  "EarningType" : {"value" : "RG" },
  "TimeSpent" : {"value" : "01:30" },
  "BillableTime" : {"value" : "00:30" },
}

HTTP/1.1 200 OK
Content-Type: application/json
{ ... }
```

Integration of Acumatica ERP with POS systems (SOAP API Examples)

A point-of-sale (POS) system is a system where the customer pays for the items or services that he or she wants to buy. The POS system can be operated by a cashier or can be a self-service terminal where customers perform all operations by themselves.

Examples of the integration of Acumatica ERP with POS systems include the following:

- *Entry of a Direct Sales Invoice*
- *Entry of a Direct Sales Invoice Along with a Return*
- *Entry of a Credit Memo with Positive and Negative Lines*
- *Entry of a Direct Sales Invoice in a Non-Default Currency*
- *Entry of a Direct Sales Invoice for a Shipped Order and Return*
- *Entry of a Direct Sales Invoice for an Unshipped Sales Order*
- *Entry of a Direct Sales Invoice for a Partially Shipped Sales Order*
- *Entry of a Credit Memo for an Unshipped Sales Order*

Entry of a Direct Sales Invoice

This example has been tested for Acumatica ERP 2018 R1 and may require code modifications for other versions of Acumatica ERP.

A point-of-sale (POS) system can create and process direct sales invoices—that is, invoices for which neither a sales order nor a shipment has been created. The POS system creates a direct sales invoice and a payment, applies this payment to the invoice, and releases the invoice.

User Scenario

A customer comes to the store, picks up a number of items (including a motherboard, which is a serialized item), and receives a consulting service from a store consultant. The customer would like to buy the items and pay for the service. In the POS system, one invoice is created for this operation.

Code Example

You can use the code below to create a payment and a sales invoice, which includes lines for serialized stock items, not-serialized stock items, and non-stock items.

To test this code example, configure your client application and an Acumatica ERP instance as follows:

- Deploy a new Acumatica ERP instance with the *SalesDemo* dataset inserted. For details on deploying an instance, see *To Deploy an Acumatica ERP Instance* in the Installation Guide.
- On the *Enable/Disable Features* (CS100000) form, enable the *Advanced SO Invoices* feature.
- In the client application, add a service reference to the *POS/17.200.01* endpoint, which is an extension of the *Default/17.200.001* endpoint. For details on how to add a service reference, see *To Configure the Client Application* in this guide.
- To sign in to the instance in the client application, use the company name (which you specified when you created the instance) and the *HQ* branch.

```
using System;
using System.Linq;
using System.IO;
using System.Threading;
using System.Collections.Generic;
//Add the namespace of the service reference here

public static void EnterDirectSalesInvoice(DefaultSoapClient soapClient)
{
    //Input data
    //The customer
    const string CustomerABARTENDE = "ABARTENDE";
    //The customer's tax zone
    const string TaxZoneCANADABC = "CANADABC";
    //The sold stock items
    const string InventoryAACOMPUT01 = "AACOMPUT01";
    const string InventoryAALEGO500 = "AALEGO500";
    //The sold non-stock items
    const string InventoryADMCHARGE = "ADMCHARGE";
    const string InventoryCASELABOR = "CASELABOR";
    //The sold serialized stock item
    const string InventoryAAMACHINE1 = "AAMACHINE1";
```

```
//The warehouse, which is the same for all items in this example
const string WarehouseWHOLESALE = "WHOLESALE";
//The location related to InventoryAACOMPUT01, InventoryAALEGO500,
//and InventoryAAMACHINE1
const string LocationR1S1 = "R1S1";
//The UOM related to InventoryCASELABOR
const string UOMMINUTE = "MINUTE";
//The payment amount
const decimal PaymentAmountDisc = 31539.05m;
//The billing address
const string AddressLine1 = "Fillmore Str";
const string City = "San Francisco";
const string State = "CA";
const string SerialNumberAAMACHINE1 = "SRF000007";

//Specify the tax zone for the customer
soapClient.Put(new Customer
{
    CustomerID = new StringSearch { Value = CustomerABARTENDE },
    TaxZone = new StringValue { Value = TaxZoneCANADABC },
    ReturnBehavior = ReturnBehavior.OnlySpecified
});

//Create a payment for the customer
Payment payment = (Payment)soapClient.Put(new Payment
{
    Type = new StringValue { Value = "Payment" },
    CustomerID = new StringValue { Value = CustomerABARTENDE },
    PaymentAmount = new DecimalValue { Value = PaymentAmountDisc },
    ReferenceNbr = new StringReturn(),
    ReturnBehavior = ReturnBehavior.OnlySpecified
});

//Create a sales invoice
SalesInvoice invoice = (SalesInvoice)soapClient.Put(new SalesInvoice
{
    ReturnBehavior = ReturnBehavior.OnlySpecified,
    Type = new StringValue { Value = "Invoice" },
    CustomerID = new StringValue { Value = CustomerABARTENDE },
    //Specify details of the sales invoice
    Details = new SalesInvoiceDetail[]
    {
        //a sold stock item
        new SalesInvoiceDetail()
        {
            ReturnBehavior = ReturnBehavior.OnlySpecified,
            InventoryID = new StringValue {Value = InventoryAACOMPUT01},
            WarehouseID = new StringValue {Value = WarehouseWHOLESALE},
            Location = new StringValue {Value = LocationR1S1},
            Qty = new DecimalValue {Value = 5}
        },
```

```
            //a sold stock item
            new SalesInvoiceDetail()
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryID = new StringValue {Value = InventoryAALEGO500},
                WarehouseID = new StringValue {Value = WarehouseWHOLESALE},
                Location = new StringValue {Value = LocationR1S1},
                Qty = new DecimalValue {Value = 4},
                UnitPrice = new DecimalValue {Value = 100}
            },
            //a sold non-stock item
            new SalesInvoiceDetail()
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryID = new StringValue {Value = InventoryADMCHARGE},
                Qty = new DecimalValue {Value = 3},
                UnitPrice = new DecimalValue {Value = 100}
            },
            //a sold non-stock item
            new SalesInvoiceDetail()
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryID = new StringValue {Value = InventoryCASELABOR},
                WarehouseID = new StringValue {Value = WarehouseWHOLESALE},
                Qty = new DecimalValue {Value = 2},
                UOM = new StringValue {Value = UOMMINUTE},
                UnitPrice = new DecimalValue {Value = 20}
            },
            //a sold serialized stock item
            new SalesInvoiceDetail()
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryID = new StringValue {Value = InventoryAAMACHINE1},
                WarehouseID = new StringValue {Value = WarehouseWHOLESALE},
                Location = new StringValue {Value = LocationR1S1},
                Qty = new DecimalValue {Value = 1},
                LotSerialNbr = new StringValue {Value = SerialNumberAAMACHINE1 }
            }
        },
        //Override the billing address of the invoice
        BillingSettings = new BillToSettings
        {
            BillToAddressOverride = new BooleanValue { Value = true },
            BillToAddress = new Address
            {
                AddressLine1 = new StringValue { Value = AddressLine1 },
                City = new StringValue { Value = City },
                State = new StringValue { Value = State }
            },
            ReturnBehavior = ReturnBehavior.All
        },
```

```
        //Specify additional data to be retrieved
        ApplicationsInvoice = new SalesInvoiceApplicationInvoice[]
        {
            new SalesInvoiceApplicationInvoice
            {
                ReturnBehavior = ReturnBehavior.All
            }
        },
        FinancialDetails = new SalesInvoiceFinancialDetails
        {
            ReturnBehavior = ReturnBehavior.OnlySpecified,
            CustomerTaxZone = new StringReturn()
        }
    });

    //Among the applications, select the payment created in this example
    //and specify the Amount Paid value
    SalesInvoiceApplicationInvoice application = invoice.ApplicationsInvoice
        .Where(applicationPayment =>
        applicationPayment.AdjustingDocReferenceNbr.Value ==
        payment.ReferenceNbr.Value).Single();
    application.AmountPaid.Value = PaymentAmountDisc;

    //Release the invoice.
    //(The Invoke method updates the invoice data (the AmountPaid value) first
    //and then releases the invoice.)
    InvokeResult invokeResult =
        soapClient.Invoke(invoice, new ReleaseSalesInvoice());
    //Monitor the status of the release operation.
    //(The GetProcessResult function is defined below.)
    ProcessResult processResult = GetProcessResult(soapClient,invokeResult);

    //Obtain the released invoice with the needed information
    invoice = (SalesInvoice)soapClient.Get(new SalesInvoice
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        ID = processResult.EntityId,
        Type = new StringReturn(),
        ReferenceNbr = new StringReturn(),
        Status = new StringReturn(),
        FinancialDetails = new SalesInvoiceFinancialDetails
        {
            ReturnBehavior = ReturnBehavior.OnlySpecified,
            BatchNbr = new StringReturn(),
        },
        TaxDetails = new SalesInvoiceTaxDetail[]
        {
            new SalesInvoiceTaxDetail {ReturnBehavior = ReturnBehavior.All }
        },
        ApplicationsInvoice = new SalesInvoiceApplicationInvoice[]
        {
```

```
            new SalesInvoiceApplicationInvoice
            {
                ReturnBehavior = ReturnBehavior.All
            }
        },
        Details = new SalesInvoiceDetail[]
        {
            new SalesInvoiceDetail
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryDocType = new StringReturn(),
                InventoryRefNbr = new StringReturn()
            }
        }
    });
}

//Long-running operation helper
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:
                //Insert the needed waiting time here
                if (processResult.Seconds > 30)
                    throw new TimeoutException();
                continue;
            default:
                throw new InvalidOperationException();
        }
    }
}
```

Entry of a Direct Sales Invoice Along with a Return

This example has been tested for Acumatica ERP 2018 R1 and may require code modifications for other versions of Acumatica ERP.

A point-of-sale (POS) system can create and process direct sales invoices (that is, invoices for which neither a sales order nor a shipment has been created) and include in these

invoices lines for a return of previously sold items. The POS system creates a direct sales invoice and releases the invoice.

For simplicity, this example does not include the creation of the payment or the application of the payment to the invoice. You can find an example showing how to create and apply a payment in *Entry of a Direct Sales Invoice*.

User Scenario

A customer comes to the store and picks up a motherboard (which is a serialized item) and a patch cord from store shelves. The customer would like to buy these items and to return the previously purchased computer mouse. In a POS system, one invoice is created for the whole operation. The customer pays the difference between the sale and the return.

Code Example

You can use the code below to create a sales invoice that includes lines for serialized stock items, not-serialized stock items, and return lines for previously sold items.

To test this code example, configure your client application and an Acumatica ERP instance as follows:

- Deploy a new Acumatica ERP instance with the *SalesDemo* dataset inserted. For details on deploying an instance, see *To Deploy an Acumatica ERP Instance* in the Installation Guide.
- On the *Enable/Disable Features* (CS100000) form, enable the *Advanced SO Invoices* feature.
- In the client application, add a service reference to the *POS/17.200.01* endpoint, which is an extension of the *Default/17.200.001* endpoint. For details on how to add a service reference, see *To Configure the Client Application* in this guide.
- To sign in to the instance in the client application, use the company name (which you specified when you created the instance) and the *HQ* branch.

You can use the data of the direct sales invoice created in *Entry of a Direct Sales Invoice* for the return lines for previously sold items.

```
using System;
using System.Linq;
using System.IO;
using System.Threading;
using System.Collections.Generic;
//Add the namespace of the service reference here

public static void EnterReturnAndDirectSalesInvoice(DefaultSoapClient soapClient)
{
    //Input data
    //The customer
    const string CustomerABARTENDE = "ABARTENDE";
    //The customer's tax zone
    const string TaxZoneCANADAAB = "CANADAAB";
```

```
//The sold serialized stock item
const string InventoryAAPOWERAID = "AAPOWERAID";
//The serial number of InventoryAAPOWERAID
const string LotSerialNbrLREX0 = "LREX0";
//The warehouse related to InventoryAAPOWERAID
const string WarehouseRETAIL = "RETAIL";
//The location related to InventoryAAPOWERAID
const string LocationSTORAGE = "STORAGE";
//The sold non-stock item
const string InventoryACCOMODATE = "ACCOMODATE";
//The reference number of the invoice whose line items are returned
const string CreatedSOInvoiceReferenceNbr = "AR005519";
//The number of the returned line in the invoice
//(in this example, the number of the line for InventoryAAMACHINE1)
const int CreatedSOInvoiceLineNbr = 5;
//The returned serialized stock item
const string InventoryAAMACHINE1 = "AAMACHINE1";
//The warehouse related to InventoryAAMACHINE1
const string WarehouseWHOLESALE = "WHOLESALE";
//The location related to InventoryAAMACHINE1
const string LocationR1S1 = "R1S1";
//The returned non-stock item
const string InventoryADMCHARGE = "ADMCHARGE";

//Create a sales invoice
var invoice = (SalesInvoice)soapClient.Put(new SalesInvoice
{
    ReturnBehavior = ReturnBehavior.OnlySpecified,
    Type = new StringValue { Value = "Invoice" },
    CustomerID = new StringValue { Value = CustomerABARTENDE },
    //Specify the customer's tax zone
    FinancialDetails = new SalesInvoiceFinancialDetails
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        CustomerTaxZone = new StringValue { Value = TaxZoneCANADAAB }
    },
    //Specify the details of the sales invoice
    Details = new SalesInvoiceDetail[]
    {
        //a sold serialized stock item
  new SalesInvoiceDetail()
        {
            ReturnBehavior = ReturnBehavior.OnlySpecified,
            InventoryID = new StringValue {Value = InventoryAAPOWERAID},
            WarehouseID = new StringValue {Value = WarehouseRETAIL},
            Location = new StringValue {Value = LocationSTORAGE},
            LotSerialNbr = new StringValue {Value = LotSerialNbrLREX0},
            Qty = new DecimalValue {Value = 100}
        },
        //a sold non-stock item
  new SalesInvoiceDetail()
```

```
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryID = new StringValue {Value = InventoryACCOMODATE},
                Qty = new DecimalValue {Value = 200},
                UnitPrice = new DecimalValue {Value = 200}
            },
            //a returned non-stock item
      new SalesInvoiceDetail()
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryID = new StringValue {Value = InventoryADMCHARGE},
                Qty = new DecimalValue {Value = -1},
                OrigInvType = new StringValue {Value = "INV"},
                OrigInvNbr = new StringValue {Value = CreatedSOInvoiceReferenceNbr},
                UnitPrice = new DecimalValue {Value = 100}
            },
            //a returned serialized stock item
      new SalesInvoiceDetail()
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryID = new StringValue {Value = InventoryAAMACHINE1},
                WarehouseID = new StringValue {Value = WarehouseWHOLESALE},
                Location = new StringValue {Value = LocationR1S1},
                Qty = new DecimalValue {Value = -1},
                OrigInvType = new StringValue {Value = "INV"},
                OrigInvNbr = new StringValue {Value = CreatedSOInvoiceReferenceNbr },
                OrigInvLineNbr = new IntValue {Value = CreatedSOInvoiceLineNbr }
            }
        },
        //Specify additional data to be retrieved
        TaxDetails = new SalesInvoiceTaxDetail[]
        {
            new SalesInvoiceTaxDetail
            {
                ReturnBehavior = ReturnBehavior.All
            }
        },
    });

    //Change the tax amount of the invoice to -100
    invoice.TaxDetails[0].TaxAmount.Value = -100m;

    //Release the invoice.
    //(The Invoke method updates the invoice data (the TaxAmount value) first
    //and then releases the invoice.)
    InvokeResult invokeResult = soapClient.Invoke(invoice, new ReleaseSalesInvoice());
    //Monitor the status of the release operation.
    //(The GetProcessResult function is defined below.)
    ProcessResult processResult = GetProcessResult(soapClient, invokeResult);

    //Obtain the released invoice with the needed information
```

```
    invoice = (SalesInvoice)soapClient.Get(new SalesInvoice
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        ID = processResult.EntityId,
        Type = new StringReturn(),
        ReferenceNbr = new StringReturn(),
        Status = new StringReturn(),
        FinancialDetails = new SalesInvoiceFinancialDetails
        {
            ReturnBehavior = ReturnBehavior.OnlySpecified,
            BatchNbr = new StringReturn(),
        },
        TaxDetails = new SalesInvoiceTaxDetail[]
        {
            new SalesInvoiceTaxDetail {ReturnBehavior = ReturnBehavior.All }
        },
        Details = new SalesInvoiceDetail[]
        {
            new SalesInvoiceDetail
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryDocType = new StringReturn(),
                InventoryRefNbr = new StringReturn()
            }
        }
    });
}

//Long-running operation helper
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:
                //Insert the needed waiting time here
                if (processResult.Seconds > 30)
                    throw new TimeoutException();
                continue;
            default:
                throw new InvalidOperationException();
```

```
            }
        }
}
```

Entry of a Credit Memo with Positive and Negative Lines

This example has been tested for Acumatica ERP 2018 R1 and may require code modifications for other versions of Acumatica ERP.

A point-of-sale (POS) system can create and process credit memos with positive lines (for the returned items) and negative lines (for the purchased items). (A credit memo is created instead of a direct sales invoice if the payment amount of the returned items is greater than the payment amount of the newly purchased items.) The POS system creates a credit memo, links the invoice that contains the returned lines to the credit memo, and releases the credit memo.

User Scenario

A customer comes to the store and picks up a motherboard (which is a serialized item) and a patch cord from the store shelves. The customer would like to buy these items and to return the previously purchased notebook computer. The price of the returned item is greater than the price of the purchased items. In a POS system, one invoice is created for the whole operation. The customer is given the difference between the return and the sale.

Code Example

You can use the code below to create a credit memo with positive and negative lines.

To test this code example, configure your client application and an Acumatica ERP instance as follows:

- Deploy a new Acumatica ERP instance with the *SalesDemo* dataset inserted. For details on deploying an instance, see *To Deploy an Acumatica ERP Instance* in the Installation Guide.
- On the *Enable/Disable Features* (CS100000) form, enable the *Advanced SO Invoices* feature.
- In the client application, add a service reference to the *POS/17.200.01* endpoint, which is an extension of the *Default/17.200.001* endpoint. For details on how to add a service reference, see *To Configure the Client Application* in this guide.
- To sign in to the instance in the client application, use the company name (which you specified when you created the instance) and the *HQ* branch.

You can use the data of the invoice created in *Entry of a Direct Sales Invoice Along with a Return* for the returned lines for previously sold items.

```
using System;
using System.Linq;
using System.IO;
using System.Threading;
```

```
using System.Collections.Generic;
//Add the namespace of the service reference here

public static void EnterCreditMemoWithPositiveAndNegativeLines(
       DefaultSoapClient soapClient)
{
    //Input data
    //The customer
    const string CustomerABARTENDE = "ABARTENDE";
    //The sold stock item
    const string InventoryAACOMPUT01 = "AACOMPUT01";
    //The warehouse related to InventoryAACOMPUT01
    const string WarehouseWHOLESALE = "WHOLESALE";
    //The location related to InventoryAACOMPUT01
    const string LocationR1S1 = "R1S1";
    //The tax category related to InventoryAACOMPUT01
    const string TaxCategoryTAXABLE = "TAXABLE";
    //The sold non-stock item
    const string InventoryCASELABORO = "CASELABORO";
    //The UOM related to InventoryCASELABORO
    const string UOMMINUTE = "MINUTE";
    //The reference number of the invoice whose line items are returned
    const string CreatedInvoiceNbr = "AR005520";
    //The returned serialized stock item
    const string InventoryAAPOWERAID = "AAPOWERAID";
    //The serial number of InventoryAAPOWERAID
    const string LotSerialNbrLREX000004 = "LREX000004";
    //The warehouse related to InventoryAAPOWERAID
    const string WarehouseRETAIL = "RETAIL";
    //The location related to InventoryAAPOWERAID
    const string LocationSTORAGE = "STORAGE";
    //The returned non-stock item
    const string InventoryCASELABOR = "CASELABOR";

    //Create a credit memo for the customer
    SalesInvoice creditMemo = (SalesInvoice)soapClient.Put(new SalesInvoice
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        Type = new StringValue { Value = "Credit Memo" },
        CustomerID = new StringValue { Value = CustomerABARTENDE },
        //Specify the details of the credit memo
        Details = new SalesInvoiceDetail[]
        {
            //a returned serialized stock item
            new SalesInvoiceDetail()
            {
                InventoryID = new StringValue {Value = InventoryAAPOWERAID},
                WarehouseID = new StringValue {Value = WarehouseRETAIL},
                Location = new StringValue {Value = LocationSTORAGE},
                Qty = new DecimalValue {Value = 10},
                LotSerialNbr = new StringValue {Value = LotSerialNbrLREX000004},
```

```
            ExpirationDate = new DateTimeValue
            {
                Value = new DateTime(2018, 10, 30)
            }
        },
        //a sold stock item
        new SalesInvoiceDetail()
        {
            InventoryID = new StringValue {Value = InventoryAACOMPUT01},
            Qty = new DecimalValue {Value = -1},
            WarehouseID = new StringValue {Value = WarehouseWHOLESALE},
            Location = new StringValue {Value = LocationR1S1},
            TaxCategory = new StringValue { Value = TaxCategoryTAXABLE }
        },
        //a returned non-stock item
        new SalesInvoiceDetail()
        {
            InventoryID = new StringValue {Value = InventoryCASELABOR},
            Qty = new DecimalValue {Value = 100}
        },
        //a sold non-stock item
        new SalesInvoiceDetail()
        {
            InventoryID = new StringValue {Value = InventoryCASELABORO},
            Qty = new DecimalValue {Value = -2},
            UOM = new StringValue {Value = UOMMINUTE}
        }
    },
    //Link the credit memo to the invoice
    ApplicationsCreditMemo = new SalesInvoiceApplicationCreditMemo[]
    {
        new SalesInvoiceApplicationCreditMemo()
        {
            DocType = new StringValue {Value = "Invoice"},
            ReferenceNbr = new StringValue {Value = CreatedInvoiceNbr},
            AmountPaid = new DecimalValue {Value = 2000}
        }
    },
    //Specify additional data to be retrieved
    TaxDetails = new SalesInvoiceTaxDetail[]
    {
        new SalesInvoiceTaxDetail
        {
            ReturnBehavior = ReturnBehavior.All
        }
    }
});

//Change the tax amount of the credit memo to -55
creditMemo.TaxDetails[0].TaxAmount.Value = -55m;
```

```
    //Release the credit memo.
    //(The Invoke method updates the data (the TaxAmount value) first
    //and then releases the credit memo.)
    InvokeResult invokeResult =
        soapClient.Invoke(creditMemo, new ReleaseSalesInvoice());
    //Monitor the status of the release operation.
    //(The GetProcessResult function is defined below.)
    ProcessResult processResult = GetProcessResult(soapClient, invokeResult);

    //Obtain the released credit memo with the needed information
    creditMemo = (SalesInvoice)soapClient.Get(new SalesInvoice
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        ID = processResult.EntityId,
        Type = new StringReturn(),
        ReferenceNbr = new StringReturn(),
        Status = new StringReturn(),
        FinancialDetails = new SalesInvoiceFinancialDetails
        {
            ReturnBehavior = ReturnBehavior.OnlySpecified,
            BatchNbr = new StringReturn(),
        },
        Details = new SalesInvoiceDetail[]
        {
            new SalesInvoiceDetail
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryDocType = new StringReturn(),
                InventoryRefNbr = new StringReturn()
            }
        },
        ApplicationsInvoice = new SalesInvoiceApplicationInvoice[]
        {
            new SalesInvoiceApplicationInvoice
            {
                ReturnBehavior = ReturnBehavior.All
            }
        }
    });
}

//Long-running operation helper
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:
                //Insert the needed waiting time here
                if (processResult.Seconds > 30)
                    throw new TimeoutException();
                continue;
            default:
                throw new InvalidOperationException();
        }
    }
}
```

Entry of a Direct Sales Invoice in a Non-Default Currency

This example has been tested for Acumatica ERP 2018 R1 and may require code modifications for other versions of Acumatica ERP.

A point-of-sale (POS) system can create and process direct sales invoices—that is, invoices for which neither a sales order nor a shipment has been created—in a currency that differs from the default currency of the customer account. The POS system creates a direct sales invoice in the needed currency and releases the invoice.

For simplicity, this example does not include the creation of the payment or the application of the payment to the invoice. You can find an example showing how to create and apply a payment in *Entry of a Direct Sales Invoice*.

User Scenario

A customer comes to the Canadian store and picks up a patch cord from store shelves. The customer would like to buy the item, return the previously purchased computer mouse, and pay for the service in United States dollars. In a POS system, one invoice is created for this operation. The customer pays the difference between the sale and return.

Code Example

You can use the code below to create a sales invoice for a direct sale in a currency that is different from the customer's default currency.

To test this code example, configure your client application and an Acumatica ERP instance as follows:

- Deploy a new Acumatica ERP instance with the *SalesDemo* dataset inserted. For details on deploying an instance, see *To Deploy an Acumatica ERP Instance* in the Installation Guide.
- On the *Enable/Disable Features* (CS100000) form, enable the *Advanced SO Invoices* feature.

- In the client application, add a service reference to the *POS/17.200.01* endpoint, which is an extension of the *Default/17.200.001* endpoint. For details on how to add a service reference, see *To Configure the Client Application* in this guide.
- To sign in to the instance in the client application, use the company name (which you specified when you created the instance) and the *HQ* branch.

```
using System;
using System.Linq;
using System.IO;
using System.Threading;
using System.Collections.Generic;
//Add the namespace of the service reference here

public static void EnterDirectSalesInvoiceInOtherCurrency(
    DefaultSoapClient soapClient)
{
    //Input data
    //The customer
    const string CustomerABARTENDE = "ABARTENDE";
    //The sold stock item
    const string InventoryAACOMPUT01 = "AACOMPUT01";
    //The returned stock item
    const string InventoryAALEGO500 = "AALEGO500";
    //The returned non-stock item
    const string InventoryADMCHARGE = "ADMCHARGE";
    //The warehouse related to InventoryAALEGO500 and InventoryAACOMPUT01
    const string WarehouseWHOLESALE = "WHOLESALE";
    //The location related to InventoryAALEGO500 and InventoryAACOMPUT01
    const string LocationR1S1 = "R1S1";

    //Create an invoice for the customer
    SalesInvoice invoice = (SalesInvoice)soapClient.Put(new SalesInvoice
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        Type = new StringValue { Value = "Invoice" },
        CustomerID = new StringValue { Value = CustomerABARTENDE },
        //Specify the currency of the invoice
        Currency = new StringValue {Value = "EUR" },
        //Specify the invoice details
        Details = new SalesInvoiceDetail[]
        {
            //a sold stock item
      new SalesInvoiceDetail()
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryID = new StringValue {Value = InventoryAACOMPUT01},
                WarehouseID = new StringValue {Value = WarehouseWHOLESALE},
                Location = new StringValue {Value = LocationR1S1},
                Qty = new DecimalValue {Value = 20}
            },
```

```
            //a returned stock item
      new SalesInvoiceDetail()
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryID = new StringValue {Value = InventoryAALEGO500},
                WarehouseID = new StringValue {Value = WarehouseWHOLESALE},
                Location = new StringValue {Value = LocationR1S1},
                Qty = new DecimalValue {Value = -5}
            },
            //a returned non-stock item
      new SalesInvoiceDetail()
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryID = new StringValue {Value = InventoryADMCHARGE},
                Qty = new DecimalValue {Value = -2},
                UnitPrice = new DecimalValue {Value = 100}
            }
        },
        //Specify additional values to be retrieved
        ReferenceNbr = new StringReturn()
    });

    //Release the invoice
    InvokeResult invokeResult = soapClient.Invoke(
        new SalesInvoice
        {
            Type = new StringSearch { Value = "Invoice" },
            ReferenceNbr = new StringSearch
            {
                Value = invoice.ReferenceNbr.Value
            },
        },
        new ReleaseSalesInvoice()
    );
    //Monitor the status of the release operation.
    //(The GetProcessResult function is defined below.)
    ProcessResult processResult = GetProcessResult(soapClient, invokeResult);

    //Obtain the released invoice with the needed information
    invoice = (SalesInvoice)soapClient.Get(new SalesInvoice
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        ID = processResult.EntityId,
        Type = new StringReturn(),
        ReferenceNbr = new StringReturn(),
        Status = new StringReturn(),
        Balance = new DecimalReturn(),
        VATTaxableTotal = new DecimalReturn(),
        FinancialDetails = new SalesInvoiceFinancialDetails
        {
            ReturnBehavior = ReturnBehavior.OnlySpecified,
```

```
            BatchNbr = new StringReturn(),
        },
        TaxDetails = new SalesInvoiceTaxDetail[]
        {
           new SalesInvoiceTaxDetail {ReturnBehavior = ReturnBehavior.All }
        },
        Details = new SalesInvoiceDetail[]
        {
            new SalesInvoiceDetail
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryDocType = new StringReturn(),
                InventoryRefNbr = new StringReturn()
            }
        },
        Commissions = new SalesInvoiceCommissions
        {
            ReturnBehavior = ReturnBehavior.All
        }
    });
}

//Long-running operation helper
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:
                //Insert the needed waiting time here
                if (processResult.Seconds > 30)
                    throw new TimeoutException();
                continue;
            default:
                throw new InvalidOperationException();
        }
    }
}
```

Entry of a Direct Sales Invoice for a Shipped Order and Return

This example has been tested for Acumatica ERP 2018 R1 and may require code modifications for other versions of Acumatica ERP.

A point-of-sale (POS) system can create and process sales invoices that contain both lines that are not linked to any sales order or shipment and lines for which a sales order and shipment have been created. Either type of line can include information about newly bought items or returned items.

To process these invoices, the POS system performs the following steps:

1. For the returned items that are linked to a sales order and shipment, creates a new sales order of *RM* type (a generic authorized return).
2. In the created *RM* order, adds the previously issued invoice that contains the returned items.
3. In the *RM* order, adds the newly ordered items.
4. Creates and confirms shipments of the newly ordered items.
5. Creates shipments with the *Receipt* operation for the returned items and confirms these shipments.
6. Creates a Sales Orders invoice and releases the invoice.

For simplicity, this example does not include the creation of the payment or the application of the payment to the invoice. You can find an example showing how to create and apply a payment in *Entry of a Direct Sales Invoice*.

User Scenario

In the online shop, a customer buys a computer mouse, but then the customer decides to return the mouse. In the online shop, the customer creates an order that includes both the returned item and a motherboard, then the customer decides to get the motherboard and to return the mouse in the store. The customer comes to the store and picks up a patch cord from the store shelves. Thus, the customer would like to buy the patch cord, to return the mouse, and to pay for the previously ordered motherboard. In a POS system, one invoice is created for the whole operation.

Code Example

You can use the code below to create a direct sales invoice that combines a shipped order and a return.

To test this code example, configure your client application and an Acumatica ERP instance as follows:

- Deploy a new Acumatica ERP instance with the *SalesDemo* dataset inserted. For details on deploying an instance, see *To Deploy an Acumatica ERP Instance* in the Installation Guide.
- On the *Enable/Disable Features* (CS100000) form, enable the *Advanced SO Invoices* feature.
- In the client application, add a service reference to the *POS/17.200.01* endpoint, which is an extension of the *Default/17.200.001* endpoint. For details on how to add a service reference, see *To Configure the Client Application* in this guide.
- To sign in to the instance in the client application, use the company name (which you specified when you created the instance) and the *HQ* branch.

You can insert in the code the data of the direct sales invoice created in *Entry of a Direct Sales Invoice in a Non-Default Currency* for the return lines for previously sold items.

```
using System;
using System.Linq;
using System.IO;
using System.Threading;
using System.Collections.Generic;
//Add the namespace of the service reference here

public static void EnterDirectSalesInvoiceWithShippedOrderAndReturn(
    DefaultSoapClient soapClient)
{
    //Input data
    //The customer
    const string CustomerABARTENDE = "ABARTENDE";
    //The sold stock item
    const string InventoryCONAIRT1 = "CONAIRT1";
    //The reference number of the customer's direct sales invoice
    //whose line is returned
    const string OriginalDirectSalesInvoiceReferenceNbr = "AR005522";
    //The number of the returned line in the invoice
    const int OriginalDirectSalesInvoiceLineNbr = 1;
    //The returned stock item from the direct sales invoice
    const string InventoryAACOMPUT01 = "AACOMPUT01";
    //The warehouse, which is the same for all items in this example
    const string WarehouseWHOLESALE = "WHOLESALE";
    //The location related to InventoryAACOMPUT01 and InventoryCONAIRT1
    const string LocationR1S1 = "R1S1";
    //A previously created customer's invoice that is linked to a sales order
    //and whose line is returned
    const string OriginalInvoiceWithSalesOrderReferenceNbr = "AR003117";
    //The returned stock item from the sales order
    const string InventoryWIDGET01 = "WIDGET01";
    //The sold stock item from the sales order
    const string InventoryAALEGO500 = "AALEGO500";
```

```
//Create a generic authorized return (that is, a sales order of the RM type)
SalesOrder order = (SalesOrder)soapClient.Put(new SalesOrder
{
    ReturnBehavior = ReturnBehavior.OnlySpecified,
    OrderType = new StringValue { Value = "RM" },
    CustomerID = new StringValue { Value = CustomerABARTENDE },
    OrderNbr = new StringReturn()
});

//Add an invoice with the returned line to the sales order
soapClient.Invoke(order, new SalesOrderAddInvoice
{
    DocumentType = new StringValue { Value = "Invoice" },
    ReferenceNbr = new StringValue
    {
        Value = OriginalInvoiceWithSalesOrderReferenceNbr
    }
});

//Obtain the sales order with the added invoice
order = (SalesOrder)soapClient.Get(new SalesOrder
{
    ReturnBehavior = ReturnBehavior.OnlySpecified,
    OrderType = new StringSearch { Value = "RM" },
    OrderNbr = new StringSearch { Value = order.OrderNbr.Value },
    Details = new SalesOrderDetail[]
    {
        new SalesOrderDetail { ReturnBehavior = ReturnBehavior.All }
    }
});

//To the sales order, add a detail line with a newly ordered item
var orderDetails = new SalesOrderDetail[]
{
    order.Details[0],
    new SalesOrderDetail
    {
        InventoryID = new StringValue {Value = InventoryAALEGO500},
        OrderQty = new DecimalValue {Value = 1},
        Operation = new StringValue {Value = "Issue"}
    }
};
order.Details = orderDetails;

//Create shipments for the created sales order
//(The Invoke method updates the sales order first and then
//creates shipments.)
InvokeResult invokeResult = soapClient.Invoke(
    order,
    new SalesOrderCreateShipment
    {
```

```
        WarehouseID = new StringValue { Value = WarehouseWHOLESALE }
    }
);
//Monitor the status of the long-running operation.
//(The GetProcessResult function is defined below.)
ProcessResult processResult = GetProcessResult(soapClient, invokeResult);

//Obtain the sales order
order = (SalesOrder)soapClient.Get(new SalesOrder
{
    ReturnBehavior = ReturnBehavior.OnlySpecified,
    ID = processResult.EntityId,
    Shipments = new SalesOrderShipment[]
    {
        new SalesOrderShipment { ReturnBehavior = ReturnBehavior.All }
    }
});

//Confirm each shipment
foreach (SalesOrderShipment salesOrderShipment in order.Shipments)
{
    if (salesOrderShipment.ShipmentType.Value == "Shipment")
    {
        Shipment shipment = (Shipment)soapClient.Get(new Shipment
        {
            ReturnBehavior = ReturnBehavior.OnlySpecified,
            ShipmentNbr = new StringSearch
            {
               Value = salesOrderShipment.ShipmentNbr.Value
            },
            Status = new StringReturn()
        });
        if (shipment.Status.Value != "Confirmed")
        {
            invokeResult =
               soapClient.Invoke(shipment, new ConfirmShipment());
            GetProcessResult(soapClient, invokeResult);
        }
    }
}

//Create a shipment with the Receipt operation for the returned items
invokeResult = soapClient.Invoke(
    new SalesOrder{ID = order.ID},
    new SalesOrderCreateReceipt()
);
processResult = GetProcessResult(soapClient, invokeResult);

//Obtain the sales order
order = (SalesOrder)soapClient.Get(new SalesOrder
{
```

```
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        ID = processResult.EntityId,
        OrderType = new StringReturn(),
        OrderNbr = new StringReturn(),
        Shipments = new SalesOrderShipment[]
        {
            new SalesOrderShipment { ReturnBehavior = ReturnBehavior.All }
        }
    });

    //Confirm each new shipment
    foreach (SalesOrderShipment salesOrderShipment in order.Shipments)
    {
        if (salesOrderShipment.ShipmentType.Value == "Shipment")
        {
            Shipment shipment = (Shipment)soapClient.Get(new Shipment
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                ShipmentNbr = new StringSearch
                {
                   Value = salesOrderShipment.ShipmentNbr.Value
                },
                Status = new StringReturn()
            });
            if (shipment.Status.Value != "Confirmed")
            {
                invokeResult =
                   soapClient.Invoke(shipment, new ConfirmShipment());
                processResult = GetProcessResult(soapClient, invokeResult);
            }
        }
    }

    //Create a sales invoice
    SalesInvoice invoice = (SalesInvoice)soapClient.Put(new SalesInvoice
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        Type = new StringValue { Value = "Invoice" },
        CustomerID = new StringValue { Value = CustomerABARTENDE },
        Details = new SalesInvoiceDetail[]
        {
            //a sold stock item from sales order
            new SalesInvoiceDetail()
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                OrderType = new StringValue {Value = "RM"},
                OrderNbr = order.OrderNbr,
                ShipmentNbr = order.Shipments[0].ShipmentNbr,
                InventoryID = new StringValue {Value = InventoryAALEGO500},
                WarehouseID = new StringValue {Value = WarehouseWHOLESALE},
                Qty = new DecimalValue {Value = 1}
```

```
            },
            //a returned stock item from sales order
            new SalesInvoiceDetail()
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                OrderType = new StringValue {Value = "RM"},
                OrderNbr = order.OrderNbr,
                ShipmentNbr = order.Shipments[1].ShipmentNbr,
                InventoryID = new StringValue {Value = InventoryWIDGET01},
                WarehouseID = new StringValue {Value = WarehouseWHOLESALE},
                Qty = new DecimalValue {Value = -20}
            },
            //a returned stock item from the invoice
            new SalesInvoiceDetail()
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                OrigInvType = new StringValue {Value = "INV"},
                OrigInvNbr = new StringValue
                {
                   Value = OriginalDirectSalesInvoiceReferenceNbr
                },
                OrigInvLineNbr = new IntValue
                {
                   Value = OriginalDirectSalesInvoiceLineNbr
                },
                InventoryID = new StringValue {Value = InventoryAACOMPUT01},
                WarehouseID = new StringValue {Value = WarehouseWHOLESALE},
                Location = new StringValue {Value = LocationR1S1},
                Qty = new DecimalValue {Value = -3}
            },
            //a sold stock item
            new SalesInvoiceDetail()
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryID = new StringValue {Value = InventoryCONAIRT1},
                WarehouseID = new StringValue {Value = WarehouseWHOLESALE},
                Location = new StringValue {Value = LocationR1S1},
                Qty = new DecimalValue {Value = 20}
            }
        },
        //Specify additional data to be retrieved
        ReferenceNbr = new StringReturn(),
    });

    //Release the invoice
    invokeResult = soapClient.Invoke(
        new SalesInvoice
        {
            Type = new StringSearch { Value = "Invoice" },
            ReferenceNbr = new StringSearch
            {
```

```
                Value = invoice.ReferenceNbr.Value
            },
        },
        new ReleaseSalesInvoice()
    );
    processResult = GetProcessResult(soapClient, invokeResult);

    //Obtain the released invoice with the needed information
    invoice = (SalesInvoice)soapClient.Get(new SalesInvoice
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        ID = processResult.EntityId,
        Type = new StringReturn(),
        ReferenceNbr = new StringReturn(),
        Status = new StringReturn(),
        Balance = new DecimalReturn(),
        FinancialDetails = new SalesInvoiceFinancialDetails
        {
            ReturnBehavior = ReturnBehavior.OnlySpecified,
            BatchNbr = new StringReturn(),
        },
        Details = new SalesInvoiceDetail[]
        {
            new SalesInvoiceDetail
            {
                ReturnBehavior = ReturnBehavior.All
            }
        }
    });
}

//Long-running operation helper
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:
                //Insert the needed waiting time here
                if (processResult.Seconds > 30)
                    throw new TimeoutException();
```

```
                continue;
            default:
                throw new InvalidOperationException();
        }
    }
}
```

Entry of a Direct Sales Invoice for an Unshipped Sales Order

This example has been tested for Acumatica ERP 2018 R1 and may require code modifications for other versions of Acumatica ERP.

A point-of-sale (POS) system can create and process sales invoices that contain both lines that have not been linked to any sales order or shipment and lines for which a sales order has been created and a shipment has not been created. Either type of line can include information about newly bought items or returned items.

To process these invoices, the POS system performs the following steps:

1. Creates a new sales order of the *RM* type (a generic authorized return) with the new items and returned items.
2. Creates a payment.
3. Creates a Sales Orders (SO) invoice and adds detail lines for both the items from the sales order and the new items.
4. Applies the payment to the invoice.
5. Releases the invoice. As a result of this operation, the sales order gets the *Completed* status in Acumatica ERP. The SO invoice is added to the **Shipments** tab of the *Sales Orders* (SO301000) form and is treated by the system as a shipment (that is, the invoice updates shipped quantity in the sales order lines and updates inventory).

   The sales order can be not completed if it was closed by the SO invoice partially (that is, if some lines of the sales order are not shipped or invoiced).

User Scenario

A customer comes to the store and through a self-service terminal creates a sales order to buy a computer and to return a computer mouse that the customer previously purchased in the store. Then the customer picks up a patch cord from the store shelves. The customer would like to buy the computer (specified in the created sales order) and the patch cord and to return the mouse (specified in the sales order). In a POS system, one invoice is created for the whole operation.

Code Example

You can use the code below to create a direct sales invoice and include in it the items that were previously ordered but not shipped.

To test this code example, configure your client application and an Acumatica ERP instance as follows:

- Deploy a new Acumatica ERP instance with the *SalesDemo* dataset inserted. For details on deploying an instance, see *To Deploy an Acumatica ERP Instance* in the Installation Guide.
- On the *Enable/Disable Features* (CS100000) form, enable the *Advanced SO Invoices* feature.
- In the client application, add a service reference to the *POS/17.200.01* endpoint, which is an extension of the *Default/17.200.001* endpoint. For details on how to add a service reference, see *To Configure the Client Application* in this guide.
- To sign in to the instance in the client application, use the company name (which you specified when you created the instance) and the *HQ* branch.

```
using System;
using System.Linq;
using System.IO;
using System.Threading;
using System.Collections.Generic;
//Add the namespace of the service reference here

public static void EnterDirectSalesInvoiceWithNotShippedSalesOrder(
    DefaultSoapClient soapClient)
{
    //Input data
    //The customer
    const string CustomerABARTENDE = "ABARTENDE";
    //The sold stock item from a new sales order
    const string InventoryAACOMPUT01 = "AACOMPUT01";
    //The returned stock item from the sales order
    const string InventoryAALEGO500 = "AALEGO500";
    //The sold serialized stock item
    const string InventoryAAPOWERAID = "AAPOWERAID";
    //The warehouse related to InventoryAAPOWERAID
    const string WarehouseRETAIL = "RETAIL";
    //The warehouse related to InventoryAAPOWERAID
    const string LocationSTORAGE = "STORAGE";
    //The sold non-stock item
    const string InventoryACCOMODATE = "ACCOMODATE";
    //The serial number related to InventoryAAPOWERAID
    const string SerialNbrAAPOWERAID = "LREX0";
    //The payment amount
    const decimal PaymentAmount = 2863.6m;

    //Create a sales order of RM type
    SalesOrder order = (SalesOrder)soapClient.Put(new SalesOrder
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
```

```
        OrderType = new StringValue { Value = "RM" },
        CustomerID = new StringValue { Value = CustomerABARTENDE },
        Details = new SalesOrderDetail[]
        {
            new SalesOrderDetail
            {
                ReturnBehavior = ReturnBehavior.All,
                InventoryID = new StringValue {Value = InventoryAACOMPUT01},
                OrderQty = new DecimalValue {Value = 2m},
                Operation = new StringValue {Value = "Issue"}
            },
            new SalesOrderDetail
            {
                ReturnBehavior = ReturnBehavior.All,
                InventoryID = new StringValue {Value = InventoryAALEGO500},
                OrderQty = new DecimalValue {Value = 2m},
                Operation = new StringValue {Value = "Receipt"}
            }
        },
        //Specify additional values to be retrieved
        OrderNbr = new StringReturn()
    });

    //Create a payment
    Payment payment = (Payment)soapClient.Put(new Payment
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        Type = new StringValue { Value = "Payment" },
        CustomerID = new StringValue { Value = CustomerABARTENDE },
        PaymentAmount = new DecimalValue { Value = 10000m },
        ReferenceNbr = new StringReturn()
    });

    //Create a sales invoice
    SalesInvoice invoice = (SalesInvoice)soapClient.Put(new SalesInvoice
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        Type = new StringValue { Value = "Invoice" },
        CustomerID = new StringValue { Value = CustomerABARTENDE },
        //Specify the details of the invoice
        Details = new SalesInvoiceDetail[]
        {
            //a sold stock item from the sales order
            new SalesInvoiceDetail()
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryID = new StringValue {Value = InventoryAACOMPUT01},
                OrderType = new StringValue {Value = "RM"},
                OrderNbr = new StringValue {Value = order.OrderNbr.Value},
                OrderLineNbr = order.Details[0].LineNbr,
                Qty = new DecimalValue {Value = 2}
```

```
            },
            //a returned stock item from the sales order
            new SalesInvoiceDetail()
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryID = new StringValue {Value = InventoryAALEGO500},
                OrderType = new StringValue {Value = "RM"},
                OrderNbr = new StringValue {Value = order.OrderNbr.Value},
                OrderLineNbr = order.Details[1].LineNbr,
                Qty = new DecimalValue {Value = -2},
            },
            //a sold serialized stock item
            new SalesInvoiceDetail()
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryID = new StringValue {Value = InventoryAAPOWERAID},
                WarehouseID = new StringValue {Value = WarehouseRETAIL},
                Location = new StringValue {Value = LocationSTORAGE},
                LotSerialNbr = new StringValue {Value = SerialNbrAAPOWERAID },
                Qty = new DecimalValue {Value = 100}
            },
            //a sold non-stock item
            new SalesInvoiceDetail()
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                InventoryID = new StringValue {Value = InventoryACCOMODATE},
                Qty = new DecimalValue {Value = 10},
                UnitPrice = new DecimalValue {Value = 200}
            },
        },
        //Specify additional values to be retrieved
        Balance = new DecimalReturn(),
        ApplicationsInvoice = new[]
        {
            new SalesInvoiceApplicationInvoice
            {
                ReturnBehavior = ReturnBehavior.All
            }
        }
    });

    //Select the application of the payment created in this example
    //and specify the Amount Paid value
    SalesInvoiceApplicationInvoice application = invoice.ApplicationsInvoice
        .Where(applicationPayment =>
        applicationPayment.AdjustingDocReferenceNbr.Value ==
        payment.ReferenceNbr.Value).Single();
    application.AmountPaid.Value = PaymentAmount;

    //Release the invoice.
    //(The Invoke method updates the invoice data (the AmountPaid value) first
```

```
    //and then releases the invoice.)
    InvokeResult invokeResult =
        soapClient.Invoke(invoice, new ReleaseSalesInvoice());
    //Monitor the status of the release operation.
    //(The GetProcessResult function is defined below.)
    ProcessResult processResult = GetProcessResult(soapClient, invokeResult);

    //Obtain the released invoice with the needed information
    invoice = (SalesInvoice)soapClient.Get(new SalesInvoice
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        ID = processResult.EntityId,
        Type = new StringReturn(),
        ReferenceNbr = new StringReturn(),
        Status = new StringReturn(),
        FinancialDetails = new SalesInvoiceFinancialDetails
        {
            ReturnBehavior = ReturnBehavior.OnlySpecified,
            BatchNbr = new StringReturn(),
        },
        TaxDetails = new SalesInvoiceTaxDetail[]
        {
            new SalesInvoiceTaxDetail {ReturnBehavior = ReturnBehavior.All }
        },
        ApplicationsInvoice = new SalesInvoiceApplicationInvoice[]
        {
            new SalesInvoiceApplicationInvoice
            {
                ReturnBehavior = ReturnBehavior.All
            }
        },
        Details = new SalesInvoiceDetail[]
        {
            new SalesInvoiceDetail
            {
                ReturnBehavior = ReturnBehavior.All
            }
        }
    });
}

//Long-running operation helper
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:
                //Insert the needed waiting time here
                if (processResult.Seconds > 30)
                    throw new TimeoutException();
                continue;
            default:
                throw new InvalidOperationException();
        }
    }
}
```

Entry of a Direct Sales Invoice for a Partially Shipped Sales Order

This example has been tested for Acumatica ERP 2018 R1 and may require code modifications for other versions of Acumatica ERP.

A point-of-sale (POS) system can create and process a sales invoice that contains both of the following: lines that have not been linked to any sales order or shipment, and lines for which a sales order has been created and a shipment has been created (but the shipment does not include all items of the sales order).

To process an invoice of this type, the POS system performs the following steps:

1. Creates a new sales order of the *SO* type with multiple items.
2. Creates a payment.
3. Creates a Sales Orders invoice and adds detail lines for some (but not all) items from the sales order and the new items.
4. Applies the payment to the invoice.
5. Releases the invoice. At this moment, not all lines of the sales order are shipped. The sales order has the *Open* status in Acumatica ERP. On the **Shipments** tab of the *Sales Orders* (SO301000) form, for the sales order lines added to the invoice, a dummy shipment is created with the link to the invoice.
6. Creates the second payment.
7. Applies the second payment to the sales order.
8. Creates a shipment or multiple shipments for the remaining (not shipped) lines of the sales order and confirms each shipment.

9. Creates a Sales Orders invoice for each shipment and releases each invoice. As a result of the release of the invoice or invoices the sales order gets the *Completed* status in Acumatica ERP.

The sales order can be not completed if it was closed by the SO invoice partially (that is, if some lines of the sales order are not shipped or invoiced).

User Scenario
A customer comes to the store and through a self-service terminal creates a sales order to buy a computer and a computer mouse. Then the customer picks up a patch cord from the store shelves. The customer would like to buy the mouse (from the sales order) and the grabbed patch cord, and to have the computer shipped to the customer's home. In a POS system, one invoice is created for the purchase in the store and another invoice is created for the computer to be shipped.

Code Example

You can use the code below to create a direct sales invoice and include in it the previously ordered items that aren't yet shipped, and to configure the shipment and the invoice for the shipment.

To test this code example, configure your client application and an Acumatica ERP instance as follows:

- Deploy a new Acumatica ERP instance with the *SalesDemo* dataset inserted. For details on deploying an instance, see *To Deploy an Acumatica ERP Instance* in the Installation Guide.
- On the *Enable/Disable Features* (CS100000) form, enable the *Advanced SO Invoices* feature.
- In the client application, add a service reference to the *POS/17.200.01* endpoint, which is an extension of the *Default/17.200.001* endpoint. For details on how to add a service reference, see *To Configure the Client Application* in this guide.
- To sign in to the instance in the client application, use the company name (which you specified when you created the instance) and the *HQ* branch.

```
using System;
using System.Linq;
using System.IO;
using System.Threading;
using System.Collections.Generic;
//Add the namespace of the service reference here

public static void EnterDirectSalesInvoiceWithPartiallyShippedOrder(
    DefaultSoapClient soapClient)
{
    //Input data
    //The customer
    const string CustomerABARTENDE = "ABARTENDE";
```

```
//The sold stock items from a new sales order
const string InventoryAACOMPUT01 = "AACOMPUT01";
const string InventoryCONAIRT1 = "CONAIRT1";
//The stock item (from the sales order) that is shipped later
const string InventoryAALEGO500 = "AALEGO500";
//The warehouse related to InventoryAALEGO500
const string WarehouseWHOLESALE = "WHOLESALE";
//The sold serialized stock item
const string InventoryAAPOWERAID = "AAPOWERAID";
//The warehouse related to InventoryAAPOWERAID
const string WarehouseRETAIL = "RETAIL";
//The location related to InventoryAAPOWERAID
const string LocationSTORAGE = "STORAGE";
//The serial number of InventoryAAPOWERAID
const string LotSerialNbrLREX000004 = "LREX000004";
//The first payment amount
const decimal PaymentAmount = 922m;
//The second payment amount
const decimal PaymentAmount2 = 560m;

//Create a sales order of the SO type
SalesOrder order = (SalesOrder)soapClient.Put(new SalesOrder
{
    ReturnBehavior = ReturnBehavior.OnlySpecified,
    OrderType = new StringValue { Value = "SO" },
    CustomerID = new StringValue { Value = CustomerABARTENDE },
    Details = new SalesOrderDetail[]
    {
        new SalesOrderDetail
        {
            ReturnBehavior = ReturnBehavior.All,
            InventoryID = new StringValue {Value = InventoryAACOMPUT01},
            OrderQty = new DecimalValue {Value = 4m},
            UnitPrice = new DecimalValue { Value = 200}
        },
        new SalesOrderDetail
        {
            ReturnBehavior = ReturnBehavior.All,
            InventoryID = new StringValue {Value = InventoryAALEGO500},
            OrderQty = new DecimalValue {Value = 4m},
            UnitPrice = new DecimalValue { Value = 100}
        },
        new SalesOrderDetail
        {
            ReturnBehavior = ReturnBehavior.All,
            InventoryID = new StringValue {Value = InventoryCONAIRT1},
            OrderQty = new DecimalValue {Value = 4m},
            UnitPrice = new DecimalValue { Value = 50}
        }
    },
    OrderNbr = new StringReturn()
```

```
});

//Create a payment
Payment payment = (Payment)soapClient.Put(new Payment
{
    Type = new StringValue { Value = "Payment" },
    CustomerID = new StringValue { Value = CustomerABARTENDE },
    PaymentAmount = new DecimalValue { Value = PaymentAmount }
});

//Create a sales invoice
SalesInvoice invoice = (SalesInvoice)soapClient.Put(new SalesInvoice
{
    Type = new StringValue { Value = "Invoice" },
    CustomerID = new StringValue { Value = CustomerABARTENDE },
    Details = new SalesInvoiceDetail[]
    {
        //a sold serialized stock item
        new SalesInvoiceDetail()
        {
            InventoryID = new StringValue {Value = InventoryAAPOWERAID},
            WarehouseID = new StringValue {Value = WarehouseRETAIL},
            Location = new StringValue {Value = LocationSTORAGE},
            Qty = new DecimalValue {Value = 2},
            LotSerialNbr = new StringValue {Value = LotSerialNbrLREX000004},
            ExpirationDate = new DateTimeValue
            {
                Value = new DateTime(2018, 10, 30)
            },
            UnitPrice = new DecimalValue { Value = 5m}
        },
        //a sold stock item from the sales order
        new SalesInvoiceDetail()
        {
            InventoryID = new StringValue {Value = InventoryAACOMPUT01},
            OrderType = new StringValue {Value = "SO"},
            OrderNbr = new StringValue {Value = order.OrderNbr.Value},
            OrderLineNbr = order.Details[0].LineNbr,
            Qty = new DecimalValue {Value = 4}
        },
        //a sold stock item from the sales order
        new SalesInvoiceDetail()
        {
            InventoryID = new StringValue {Value = InventoryCONAIRT1},
            OrderType = new StringValue {Value = "SO"},
            OrderNbr = new StringValue {Value = order.OrderNbr.Value},
            OrderLineNbr = order.Details[2].LineNbr,
            Qty = new DecimalValue {Value = 2},
        }
    },
    //Specify additional values to be retrieved
```

```
        ApplicationsInvoice = new[]
        {
            new SalesInvoiceApplicationInvoice
            {
                ReturnBehavior = ReturnBehavior.All
            }
        }
    });

    //Select the application of the payment created in this example
    //and specify the Amount Paid value
    SalesInvoiceApplicationInvoice application = invoice.ApplicationsInvoice
        .Where(applicationPayment =>
        applicationPayment.AdjustingDocReferenceNbr.Value ==
        payment.ReferenceNbr.Value).Single();
    application.AmountPaid.Value = PaymentAmount;

    //Release the invoice.
    //(The Invoke method updates the invoice data (the AmountPaid value) first
    //and then releases the invoice.)
    InvokeResult invokeResult =
        soapClient.Invoke(invoice, new ReleaseSalesInvoice());
    //Monitor the status of the release operation.
    //(The GetProcessResult function is defined below.)
    ProcessResult processResult = GetProcessResult(soapClient, invokeResult);

    //Obtain the released invoice with the needed information
    invoice = (SalesInvoice)soapClient.Get(new SalesInvoice
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        ID = processResult.EntityId,
        Type = new StringReturn(),
        ReferenceNbr = new StringReturn(),
        Status = new StringReturn(),
        FinancialDetails = new SalesInvoiceFinancialDetails
        {
            ReturnBehavior = ReturnBehavior.OnlySpecified,
            BatchNbr = new StringReturn(),
        },
        TaxDetails = new SalesInvoiceTaxDetail[]
        {
            new SalesInvoiceTaxDetail {ReturnBehavior = ReturnBehavior.All }
        },
        ApplicationsInvoice = new SalesInvoiceApplicationInvoice[]
        {
            new SalesInvoiceApplicationInvoice
            {
                ReturnBehavior = ReturnBehavior.All
            }
        },
        Details = new SalesInvoiceDetail[]
```

```
        {
            new SalesInvoiceDetail
            {
                ReturnBehavior = ReturnBehavior.All
            }
        }
    });

    //Obtain the sales order with the needed data
    order = (SalesOrder)soapClient.Get(new SalesOrder
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        OrderType = new StringSearch { Value = order.OrderType.Value},
        OrderNbr = new StringSearch { Value = order.OrderNbr.Value},
        Status = new StringReturn(),
        Shipments = new SalesOrderShipment[]
        {
            new SalesOrderShipment
            {
                ReturnBehavior = ReturnBehavior.All
            }
        },
        Details = new SalesOrderDetail[]
        {
            new SalesOrderDetail
            {
                ReturnBehavior = ReturnBehavior.All
            }
        }
    });

    //Create the second payment
    Payment payment2 = (Payment)soapClient.Put(new Payment
    {
        Type = new StringValue { Value = "Payment" },
        CustomerID = new StringValue { Value = CustomerABARTENDE },
        PaymentAmount = new DecimalValue { Value = PaymentAmount2 }
    });

    //Apply the second payment to the sales order
    order.Payments = new Payments[]
    {
        new Payments()
        {
            ReferenceNbr = payment2.ReferenceNbr,
            AppliedToOrder = new DecimalValue {Value = PaymentAmount2}
        }
    };

    //Create a shipment for the not-yet-shipped items of the sales order
    //(The Invoke method updates the sales order first
```

```
//and then creates shipments.)
invokeResult = soapClient.Invoke(
    order,
    new SalesOrderCreateShipment
    {
        WarehouseID = new StringValue { Value = WarehouseWHOLESALE }
    }
);
//Monitor the status of the operation
processResult = GetProcessResult(soapClient, invokeResult);

//Obtain the sales order with the created shipment or shipments
order = (SalesOrder)soapClient.Get(new SalesOrder
{
    ReturnBehavior = ReturnBehavior.OnlySpecified,
    ID = processResult.EntityId,
    OrderType = new StringReturn(),
    OrderNbr = new StringReturn(),
    Status = new StringReturn(),
    Shipments = new SalesOrderShipment[]
    {
        new SalesOrderShipment { ReturnBehavior = ReturnBehavior.All }
    }
});

//Confirm each shipment and create an invoice from it
List<Shipment> invoicedShipments = new List<Shipment>();
foreach (SalesOrderShipment salesOrderShipment in order.Shipments)
{
    //Confirm each shipment
    if (salesOrderShipment.ShipmentType.Value == "Shipment")
    {
        Shipment shipment = (Shipment)soapClient.Get(new Shipment
        {
            ReturnBehavior = ReturnBehavior.OnlySpecified,
            ShipmentNbr = new StringSearch
            {
                Value = salesOrderShipment.ShipmentNbr.Value
            },
            Status = new StringReturn()
        });
        if (shipment.Status.Value != "Confirmed")
        {
            invokeResult =
                soapClient.Invoke(shipment, new ConfirmShipment());
            processResult = GetProcessResult(soapClient, invokeResult);
        }
    }
    //Create an invoice for each shipment
    if (salesOrderShipment.ShipmentType.Value == "Shipment" &&
        salesOrderShipment.InvoiceNbr.Value == null)
```

```
        {
            Shipment shipment = (Shipment)soapClient.Get(new Shipment
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                ShipmentNbr = new StringSearch
                {
                    Value = salesOrderShipment.ShipmentNbr.Value
                },
                Status = new StringReturn()
            });

            invokeResult = soapClient.Invoke(shipment, new PrepareInvoice());
            processResult = GetProcessResult(soapClient, invokeResult);

            invoicedShipments.Add((Shipment)soapClient.Get(new Shipment
            {
                ReturnBehavior = ReturnBehavior.OnlySpecified,
                ID = processResult.EntityId,
                ShipmentNbr = new StringReturn()
            }));
        }
    }

    //Obtain the sales order with the updated shipment or shipments
    order = (SalesOrder)soapClient.Get(order);

    //Release the invoices for each shipment
    foreach (SalesOrderShipment salesOrderShipment in order.Shipments)
    {
        if (invoicedShipments.FindIndex(x =>
            x.ShipmentNbr.Value == salesOrderShipment.ShipmentNbr.Value) > -1)
        {
            SalesInvoice invoice2 = (SalesInvoice)soapClient.Get(
                new SalesInvoice()
                {
                    Type = new StringSearch
                    {
                        Value = salesOrderShipment.InvoiceType.Value
                    },
                    ReferenceNbr = new StringSearch
                    {
                        Value = salesOrderShipment.InvoiceNbr.Value
                    }
                }
            );

            //Release the invoice
            invokeResult =
                soapClient.Invoke(invoice2, new ReleaseSalesInvoice());
            //Monitor the status of the release operation
            processResult = GetProcessResult(soapClient, invokeResult);
```

```
        }
    }
}

//Long-running operation helper
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:
                //Insert the needed waiting time here
                if (processResult.Seconds > 30)
                    throw new TimeoutException();
                continue;
            default:
                throw new InvalidOperationException();
        }
    }
}
```

Entry of a Credit Memo for an Unshipped Sales Order

This example has been tested for Acumatica ERP 2018 R1 and may require code modifications for other versions of Acumatica ERP.

A point-of-sale (POS) system can create and process credit memos that contain both lines that have been linked to a sales order of the *RM* type (a generic authorized return) and lines that have not been linked to any sales order. The POS system creates a sales order of the *RM* type, creates a credit memo, adds to the credit memo the lines from the sales order and other lines, and releases the credit memo.

User Scenario

In the online shop, a customer creates a sales order to buy a computer mouse and to return a computer that the customer previously bought in the store; the customer then decides to perform these operations in the store. After that, the customer decides to also return the patch cord (also bought in the store) and goes to the store. In the POS system, one invoice is created for the whole operation. The customer is given the difference between the return and sale.

Code Example

You can use the code below to create a credit memo with some lines linked to a sales order and other lines not linked to any sales order.

To test this code example, configure your client application and an Acumatica ERP instance as follows:

- Deploy a new Acumatica ERP instance with the *SalesDemo* dataset inserted. For details on deploying an instance, see *To Deploy an Acumatica ERP Instance* in the Installation Guide.
- On the *Enable/Disable Features* (CS100000) form, enable the *Advanced SO Invoices* feature.
- In the client application, add a service reference to the *POS/17.200.01* endpoint, which is an extension of the *Default/17.200.001* endpoint. For details on how to add a service reference, see *To Configure the Client Application* in this guide.
- To sign in to the instance in the client application, use the company name (which you specified when you created the instance) and the *HQ* branch.

```
using System;
using System.Linq;
using System.IO;
using System.Threading;
using System.Collections.Generic;
//Add the namespace of the service reference here

public static void EnterCreditMemoWithSalesOrder(DefaultSoapClient soapClient)
{
    //Input data
    //The customer
    const string CustomerABARTENDE = "ABARTENDE";
    //The sold stock item from a new sales order
    const string InventoryAACOMPUT01 = "AACOMPUT01";
    //The returned stock item from the sales order
    const string InventoryAALEGO500 = "AALEGO500";
    //The returned stock item
    const string InventoryCONAIRT1 = "CONAIRT1";
    //The warehouse related to InventoryCONAIRT1
    const string WarehouseWHOLESALE = "WHOLESALE";
    //The location related to InventoryCONAIRT1
    const string LocationR1S1 = "R1S1";

    //Create a sales order of the RM type
    SalesOrder order = (SalesOrder)soapClient.Put(new SalesOrder
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        OrderType = new StringValue { Value = "RM" },
        CustomerID = new StringValue { Value = CustomerABARTENDE },
```

```
        Details = new SalesOrderDetail[]
        {
            new SalesOrderDetail
            {
                ReturnBehavior = ReturnBehavior.All,
                InventoryID = new StringValue {Value = InventoryAACOMPUT01},
                OrderQty = new DecimalValue { Value =  3m},
                Operation = new StringValue { Value = "Issue"},
                UnitPrice = new DecimalValue { Value = 100m}
            },
            new SalesOrderDetail
            {
                ReturnBehavior = ReturnBehavior.All,
                InventoryID = new StringValue {Value = InventoryAALEGO500},
                OrderQty = new DecimalValue { Value =  3m},
                Operation = new StringValue { Value = "Receipt"},
                UnitPrice = new DecimalValue { Value = 50m}
            }
        },
        OrderNbr = new StringReturn()
    });

    //Create a credit memo
    SalesInvoice invoice = (SalesInvoice)soapClient.Put(new SalesInvoice
    {
        ReturnBehavior = ReturnBehavior.OnlySystem,
        Type = new StringValue { Value = "Credit Memo" },
        CustomerID = new StringValue { Value = CustomerABARTENDE },
        Details = new SalesInvoiceDetail[]
        {
            //a sold stock item from the sales order
            new SalesInvoiceDetail()
            {
                InventoryID = new StringValue {Value = InventoryAACOMPUT01},
                OrderType = new StringValue {Value = "RM"},
                OrderNbr = new StringValue {Value = order.OrderNbr.Value},
                OrderLineNbr = order.Details[0].LineNbr,
                Qty = new DecimalValue {Value = -3}
            },
            //a returned stock item from the sales order
            new SalesInvoiceDetail()
            {
                InventoryID = new StringValue {Value = InventoryAALEGO500},
                OrderType = new StringValue {Value = "RM"},
                OrderNbr = new StringValue {Value = order.OrderNbr.Value},
                OrderLineNbr = order.Details[1].LineNbr,
                Qty = new DecimalValue {Value = 3},
            },
            //a returned stock item
            new SalesInvoiceDetail()
            {
```

```
                InventoryID = new StringValue {Value = InventoryCONAIRT1},
                WarehouseID = new StringValue {Value = WarehouseWHOLESALE},
                Location = new StringValue {Value = LocationR1S1},
                Qty = new DecimalValue {Value = 3},
                UnitPrice = new DecimalValue { Value = 200m}
            }
        },
    });

    //Release the credit memo
    InvokeResult invokeResult =
        soapClient.Invoke(
            new SalesInvoice { ID = invoice.ID},
            new ReleaseSalesInvoice()
        );
    //Monitor the status of the release operation.
    //(The GetProcessResult function is defined below.)
    ProcessResult processResult = GetProcessResult(soapClient, invokeResult);

    //Obtain the released invoice with the needed information
    invoice = (SalesInvoice)soapClient.Get(new SalesInvoice
    {
        ReturnBehavior = ReturnBehavior.OnlySpecified,
        ID = processResult.EntityId,
        Type = new StringReturn(),
        ReferenceNbr = new StringReturn(),
        Status = new StringReturn(),
        Amount = new DecimalReturn(),
        Details = new SalesInvoiceDetail[]
        {
            new SalesInvoiceDetail
            {
                ReturnBehavior = ReturnBehavior.All
            }
        }
    });
}

//Long-running operation helper
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:
                //Insert the needed waiting time here
                if (processResult.Seconds > 30)
                    throw new TimeoutException();
                continue;
            default:
                throw new InvalidOperationException();
        }
    }
}
```