Infor Mongoose IDO Development Guide

Release 10.x
Important Notices

The material contained in this publication (including any supplementary information) constitutes and contains confidential and proprietary information of Infor.

By gaining access to the attached, you acknowledge and agree that the material (including any modification, translation or adaptation of the material) and all copyright, trade secrets and all other right, title and interest therein, are the sole property of Infor and that you shall not gain right, title or interest in the material (including any modification, translation or adaptation of the material) by virtue of your review thereof other than the non-exclusive right to use the material solely in connection with and the furtherance of your license and use of software made available to your company from Infor pursuant to a separate agreement, the terms of which separate agreement shall govern your use of this material and all supplemental related materials ("Purpose").

In addition, by accessing the enclosed material, you acknowledge and agree that you are required to maintain such material in strict confidence and that your use of such material is limited to the Purpose described above. Although Infor has taken due care to ensure that the material included in this publication is accurate and complete, Infor cannot warrant that the information contained in this publication is complete, does not contain typographical or other errors, or will meet your specific requirements. As such, Infor does not assume and hereby disclaims all liability, consequential or otherwise, for any loss or damage to any person or entity which is caused by or relates to errors or omissions in this publication (including any supplementary information), whether such errors or omissions result from negligence, accident or any other cause.

Without limitation, U.S. export control laws and other applicable export and import laws govern your use of this material and you will neither export or re-export, directly or indirectly, this material nor any related materials or supplemental information in violation of such laws, or use such materials for any purpose prohibited by such laws.

Trademark Acknowledgements

The word and design marks set forth herein are trademarks and/or registered trademarks of Infor and/or related affiliates and subsidiaries. All rights reserved. All other company, product, trade or service names referenced may be registered trademarks or trademarks of their respective owners.

Publication Information

Release: Infor Mongoose 10.x
Publication Date: December 5, 2018
Document code: mg_10.x_mgidg__en-us
## Contents

About this guide ................................................................................................................................6

  Intended audience..........................................................................................................................6
  Related documents.......................................................................................................................6
  Contacting Infor..........................................................................................................................6

Chapter 1: IDO .NET API and class library.....................................................................................7

  IDOREquestClient .NET class library ..........................................................................................7
  IDOProtocol .NET class library ..................................................................................................8
  IIDOCommands interface ..........................................................................................................9
    IIDOCommands interface (C#) .................................................................................................9
    IIDOCommands interface (Visual Basic) ..................................................................................10

  APIs and related IDO protocol classes......................................................................................10
    GetPropertyInfo ......................................................................................................................11
    LoadCollection .......................................................................................................................11
    UpdateCollection ...................................................................................................................14
    Invoke ........................................................................................................................................16

Assigning and checking null values in IDO requests ....................................................................18

  Assigning null values ................................................................................................................18
  Checking null values ................................................................................................................18

Translation/localization considerations .........................................................................................19

  Internal format for numbers ....................................................................................................19
  Internal format for dates ............................................................................................................19
  Type conversions .....................................................................................................................19

Setting property and parameter values .........................................................................................20

Getting property and parameter values .......................................................................................21

Creating filter strings ..................................................................................................................22

Supported .NET CLR types ..........................................................................................................22

Transactions ....................................................................................................................................23
Diagnostics and debugging ............................................................................................................ 24
  Logging diagnostic messages .................................................................................................................. 24
  Wire Tap ........................................................................................................................................... 25

Chapter 2: IDO extension classes ........................................................................................................ 26
  About IDO extension classes .................................................................................................................. 26
  Creating an IDO extension class project .................................................................................................. 26
  Importing an IDO extension class assembly .......................................................................................... 27
  Creating, extending, and replacing an IDO extension class assembly .................................................. 28
  Implementing an IDO extension class ...................................................................................................... 29
    Mongoose.IDO.IIDOExtensionClass interface ....................................................................................... 29
    Mongoose.IDO.IIDOExtensionClassContext interface ......................................................................... 29
    Declaration of an IDO extension class for an IDO .................................................................................. 30
    IDO extension class starter template .................................................................................................... 30
  Adding methods ................................................................................................................................... 30
    IDOMethod parameters .......................................................................................................................... 31
    Including filters in custom load methods to load collections ................................................................. 32
    Using variables and a bookmarking algorithm to get more rows in a custom load method ........... 32
  IDO events ........................................................................................................................................ 33
    Adding an IDO event handler .................................................................................................................. 34
    Event handler parameters ......................................................................................................................... 35
    Special cases for UpdateCollection events ............................................................................................. 37
    Replacing standard IDO processing with event handlers ....................................................................... 38
    Generating application events ................................................................................................................. 38
  Code samples ..................................................................................................................................... 40
  ApplicationDB class .............................................................................................................................. 40
    Application database connections .......................................................................................................... 40
    Application messages ............................................................................................................................... 41
    Session variables ................................................................................................................................... 42
    Process variables ................................................................................................................................... 42
    Using ApplicationDB in an extension class method ............................................................................... 43
  Disposing of IDO extension classes ....................................................................................................... 44
  Translation/localization considerations .................................................................................................. 44
    Type conversions ................................................................................................................................... 44
  Testing extension class assemblies locally ............................................................................................... 45
  Debugging extension class es ............................................................................................................... 45
Contents

Logging diagnostic messages ...........................................................................................................46
Appendix A: Code samples ...............................................................................................................48
  LoadCollection examples ..............................................................................................................48
  UpdateCollection examples .........................................................................................................49
  Invoke example ............................................................................................................................50
About this guide

This guide provides information on how to develop your own Intelligent Data Objects (IDO)s based on the Infor Mongoose framework and how to extend the functionality of an existing IDO.

This guide describes the classes and libraries available to you and provides some examples of their use.

Intended audience

This guide is intended for form and application developers who develops their own IDO$s for Mongoose-based applications. This guide is also intended for training, support, marketing, and other personnel.

Related documents

You can find the Infor Mongoose Integrating IDO$s with the External Applications guide in the Infor Support portal and the Mongoose portal.

You can find the information about the Application Event System in the Infor Mongoose Online Help.

Contacting Infor

If you have questions about Infor products, go to Infor Concierge at https://concierge.infor.com/ and create a support incident.

If we update this document after the product release, we will post the new version on the Infor Support Portal. To access documentation, select Search > Browse Documentation. We recommend that you check this portal periodically for updated documentation.

If you have comments about Infor documentation, contact documentation@infor.com.
Chapter 1: IDO .NET API and class library

Developers can use the IDO .NET class library to programmatically access the Mongoose business objects, which are called IDOs.

Interactions with an IDO are based on requests and responses. The caller constructs an IDO request and sends it to the IDO runtime service to be executed. The IDO runtime service constructs and returns a response to the caller containing the results of the action requested.

All client access to the application database is through the IDOs. The primary responsibilities of the IDO runtime service are to query data sets (LoadCollection), save data (UpdateCollection), and call methods (Invoke).

Each IDO request has a request type, such as OpenSession, LoadCollection, UpdateCollection, or Invoke. Each IDO request also contains an optional request payload that is dependent on the request type. For example, the payload for an OpenSession request contains logon information for a user, such as user ID, password, and configuration.

The Application Program Interface (API) is used by stand-alone client applications, WinStudio scripts, user controls, and within IDO extension classes. The same programming interface is available on the client using the Mongoose.IDO.Client class and from an IDO extension class using the Me.Context.Commands property. This chapter focuses on use of the class library. The extension classes are described in IDO extension classes on page 26.

IDORequestClient .NET class library

The IDORequestClient .NET class library contains the Client class. This class is in the Mongoose.IDO.Client namespace.

The Client class implements the IIDOCommands interface, such as LoadCollection, UpdateCollection, and Invoke, and additional commands, such as OpenSession, GetConfigurations, CloseSession, that are used only for external application communications.
The IDOProtocol .NET class library contains classes that correspond to each level in the hierarchy of an IDO request or response. These classes are used to construct IDO requests and responses. All IDOProtocol classes are in the Mongoose.IDO.Protocol namespace.

The top level of the IDOProtocol class hierarchy is implemented by the IDORequestEnvelope class, and the IDO request or RequestHeader in XML, is implemented by the IDORequest class. The request payload implementation varies depending on the request type.

- For an OpenSession request, the payload is implemented by the OpenSessionRequestData class.
- For a LoadCollection request, the payload is implemented by the LoadCollectionRequestData class.

There are other similar classes for each request type that carries a payload.

The IDO responses are implemented in classes that mirror the requests, with the IDOResponseEnvelope at the top of the hierarchy.

Executing an IDO runtime Invoke call using IDOProtocol classes

As an example, these steps are required to execute an IDO runtime Invoke call using IDOProtocol classes:

1. Create a new IDORequestEnvelope instance.
2. Set the IDORequestEnvelope.SessionID property.
3. Create a new instance of IDORequest.
4. Set the IDORequest.Type property to Invoke.
5. Create a new instance of InvokeRequestData.
6. Set the InvokeRequestData.IDOName property.
7. Set the InvokeRequestData.MethodName property.
8. Add the parameters to the InvokeRequestData.Parameters collection.
9. Add the IDORequest to the IDORequestEnvelope.Requests collection.
10. Set the payload for the IDORequest using the IDORequest.SetPayload method.
11. Send your completed IDORequestEnvelope to the IDO runtime and receive the IDOResponseEnvelope in return.

Using the IIDOCommands interface

Whether you write the code on the client side or in an IDO extension class, the framework also provides wrapper methods through the IIDOCommands interface. These wrappers typically reduce IDO interactions to a single line of code.

When you use the IIDOCommands interface, you can only specify one IDO request per command and receive one response per command. But when you use the IDOProtocols classes instead, you can build one IDORequestEnvelope that includes multiple requests, and then receive one response that includes information for all the requests.

Using the ApplicationDB class

The ApplicationDB class, part of the Mongoose.IDO.DataAccess namespace, is available only to extension classes. This class provides direct access to the application database and access to common
framework features, such as application messages and session variables. See ApplicationDB class on page 40.

**Note:** The ApplicationDB class replaces the AppDB class. Existing code that uses the AppDB class will continue to work, but will only work on Microsoft SQL Server databases. The ApplicationDB class will work with any framework-supported database.

## IIDOCommands interface

This basic IDO runtime API is defined by the IIDOCommands interface in both C# and VB formats.

The IIDOCommands interface provides a simple alternative to constructing IDO requests when interacting with IDOs. The process of creating an IDOResponseEnvelope, IDORequest, and payload class instance for each interaction is replaced by a single method call, which constructs the request for you using the parameter list.

### IIDOCommands interface (C#)

```csharp
public interface IIDOCommands
{
    GetPropertyInfoResponseData GetPropertyInfo( string idoName );
    LoadCollectionResponseData LoadCollection(
        LoadCollectionRequestData requestData );
    UpdateCollectionResponseData UpdateCollection(
        UpdateCollectionRequestData requestData );
    InvokeResponseData Invoke(
        InvokeRequestData requestData );
    string[] GetIDONames();
    LoadCollectionResponseData LoadCollection(
        string idoName, string propertyList, string filter, string orderBy, int recordCap );
    LoadCollectionResponseData LoadCollection(
        string idoName, PropertyList propertyList, string filter, string orderBy, int recordCap );
    InvokeResponseData Invoke(
        string idoName, string methodName, params object[] parameters );
}
```
IIDOCOMmands interface (Visual Basic)

Public Interface IIDOCOMmands
    Function GetIDONames() As String()
    Function GetPropertyInfo(ByVal idoName As String) As _
        GetPropertyInfoResponseData
    Function GetPropertyInfo(ByVal idoName As String, _
        ByVal includeClassNotesFlag As Boolean) As _
        GetPropertyInfoResponseData
    Function Invoke(ByVal requestData As InvokeRequestData) As _
        InvokeResponseData
    Function Invoke(ByVal idoName As String, ByVal methodName As _
        String, ByVal ParamArray parameters() As Object) As _
        InvokeResponseData
    Function LoadCollection(ByVal requestData As _
        LoadCollectionRequestData) As LoadCollectionResponseData
    Function LoadCollection(ByVal idoName As String, _
        ByVal propertyList As PropertyList, _
        ByVal filter As String, _
        ByVal orderBy As String, _
        ByVal recordCap As Integer) As LoadCollectionResponseData
    Function UpdateCollection(ByVal requestData As _
        UpdateCollectionRequestData) As UpdateCollectionResponseData
End Interface

APIs and related IDO protocol classes

This section describes each API and associated protocol classes, and provides examples of each.

The examples in this section use the syntax of extension class methods, for example:

- **VB.NET**: `Me.Context.Commands.LoadCollection(request)`
- **C#**: `this.Context.Commands.LoadCollection(request)`

The syntax for using the IIDOCOMmands interface is different in other contexts, such as a form script or a stand-alone client application.

**Note**: The code examples are given for C# unless otherwise indicated.
GetPropertyInfo

Use GetPropertyInfo to retrieve a list of properties and their attributes for an IDO. GetPropertyInfo requests are constructed using the LoadCollectionRequestData IDO protocol class.

Properties

These properties are available on the LoadCollectionRequestData class:

<table>
<thead>
<tr>
<th>Property</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDOName</td>
<td>System.String</td>
<td>Identifies the IDO used to execute the request</td>
</tr>
<tr>
<td>IncludeClassNotesFlag</td>
<td>System.Boolean</td>
<td>If set to True, the ClassNotesExist property in the response is set to indicate if any class notes exist for this IDO. This property is only available if you use the GetResponse client class method.</td>
</tr>
</tbody>
</table>

GetPropertyInfo example

```
GetPropertyInfoResponseData  response;
response = this.Context.Commands.GetPropertyInfo( "SLItems" );
foreach ( PropertyInfo idoProp in response.Properties )
{
    String propertyName = idoProp.Name;
    // etc.
}
```

LoadCollection

Use LoadCollection to perform queries. LoadCollection requests are constructed using the LoadCollectionRequestData IDO protocol class.

Properties

These properties are available on the LoadCollectionRequestData class:

<table>
<thead>
<tr>
<th>Property</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDOName</td>
<td>System.String</td>
<td>Identifies the IDO used to execute the request.</td>
</tr>
<tr>
<td>Property</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ReadMode</td>
<td>Mongoose.IDO.Protocol.ReadMode</td>
<td>Specifies the collection read mode, which controls the isolation level used when executing queries. Options are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ReadCommitted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ReadUncommitted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Default</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This property does not apply to custom load methods.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See the Infor Mongoose Online Help for the list of process defaults.</td>
</tr>
<tr>
<td>PropertyList</td>
<td>Mongoose.IDO.Protocol.PropertyList</td>
<td>Specifies a subset of properties published by the IDO to be included in the response</td>
</tr>
<tr>
<td>Filter</td>
<td>System.String</td>
<td>Specifies a filter or WHERE clause to be used for the query</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: Empty</td>
</tr>
<tr>
<td>RecordCap</td>
<td>System.Int32</td>
<td>Specifies the maximum number of items to return in the response</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If set to 0, all items are returned. If set to -1, the default cap of 200 items is used.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: -1</td>
</tr>
<tr>
<td>OrderBy</td>
<td>System.String</td>
<td>Specifies a list of property names used to override the default sort order</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: Empty</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Only bound properties can be included in OrderBy.</td>
</tr>
<tr>
<td>Distinct</td>
<td>System.Boolean</td>
<td>If set to True, the request returns a set of items containing only the unique combinations of properties named in the PropertyList property</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: False</td>
</tr>
<tr>
<td>CustomLoadMethod</td>
<td>Mongoose.IDO.Protocol.</td>
<td>Allows the caller to specify a method to perform the query in place of the standard query built by the IDO runtime</td>
</tr>
<tr>
<td></td>
<td>CustomLoadMethod</td>
<td></td>
</tr>
<tr>
<td>PostQueryCommand</td>
<td>System.String</td>
<td>Allows the caller to specify a method to be executed for each item returned by the query</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The method must be a member of the IDO specified by the IDOName property.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: Empty</td>
</tr>
<tr>
<td>Property</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LinkBy</td>
<td>Mongoose.IDO.Protocol.PropertyPair array</td>
<td>Applies to inner nested LoadCollection requests only and specifies the relationship between a parent and child IDO in terms of property pairs. Use the SetLinkBy method to set this property. Default: Empty</td>
</tr>
<tr>
<td>NestedRequests</td>
<td>System.Collections.ICollection</td>
<td>Specifies a collection of additional LoadCollectionRequestData instances used to load child collections.</td>
</tr>
<tr>
<td>LoadCap</td>
<td>System.Int32</td>
<td>Specifies the maximum number of nested LoadCollection requests to process, for example, if this property is set to 1, and 200 items are returned by the parent LoadCollection request, only the first parent item in the response contains nested child items.</td>
</tr>
</tbody>
</table>

**LoadCollection example 1**

To execute a LoadCollection request, first construct an instance of the LoadCollectionRequestData class, and pass it to the LoadCollection method on the IIDOCommands interface.

```csharp
LoadCollectionRequestData request = new LoadCollectionRequestData();
LoadCollectionResponseData response;
request.IDOName = "SLItems";
request.PropertyList.SetProperties( "Item, Description, QtyOnHand" );
request.Filter = "Item LIKE N'AB%'";
request.OrderBy = "QtyOnHand";
request.RecordCap = 0;
response = this.Context.Commands.LoadCollection( request );
```

**LoadCollection example 2**

In many cases, it is more convenient to call the overloaded version of the LoadCollection method that accepts parameters for the required and most common LoadCollectionRequestData properties.

```csharp
LoadCollectionResponseData response = LoadCollectionResponseData;
// This is equivalent to example 1.
response = this.Context.Commands.LoadCollection( "SLItems",
                                                "Item, Description, QtyOnHand",
                                                "Item LIKE N'AB%'",
                                                "QtyOnHand",
                                                0);
```
LoadCollection example 3: accessing item properties

The LoadCollection methods return an instance of the LoadCollectionResponseData class, which is populated with the same values that were sent in the request plus the results from the query. These results are accessed through the Items property, which is a collection of IDOItem instances. Properties can be accessed through the IDOItem instances or directly using the indexer on the LoadCollectionResponseData class.

```csharp
int x = response.PropertyList.IndexOf("QtyOnHand");
int qty = 0;
foreach (IDOItem item in response.Items)
{
    qty += item.PropertyValues[x].GetValue<int>();
}
qty = 0;
for (int row = 0; row < response.Items.Count; row++)
{
    qty += response[row, "QtyOnHand"].GetValue<int>();
}
```

UpdateCollection

Use UpdateCollection to perform updates on data, which includes insertion, modification, and deletion of data. UpdateCollection requests are constructed using the UpdateCollectionRequestData IDO protocol class.

Properties

These properties are available on the UpdateCollectionRequestData class:

<table>
<thead>
<tr>
<th>Property</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDOName</td>
<td>System.String</td>
<td>Identifies the IDO used to execute the request</td>
</tr>
<tr>
<td>CollectionID</td>
<td>System.String</td>
<td>Specifies an identifier associated with this UpdateCollection request</td>
</tr>
<tr>
<td>RefreshAfterUpdate</td>
<td>System.Boolean</td>
<td>Specifies a boolean value used to indicate if the caller wants the response to contain the updated items refreshed after they were saved</td>
</tr>
<tr>
<td>CustomInsert</td>
<td>System.String</td>
<td>Specifies the properties used for custom actions that are used to save items</td>
</tr>
<tr>
<td>CustomUpdate</td>
<td>System.String</td>
<td></td>
</tr>
<tr>
<td>CustomDelete</td>
<td>System.String</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** This property is the same as a custom INS/UPD/DEL in WinStudio.
<table>
<thead>
<tr>
<th>Property</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkBy</td>
<td>Mongoose.IDO.Protocol.PropertyPair array</td>
<td>Applies to inner nested UpdateCollection requests only and specifies the relationship between a parent and child IDO in terms of property pairs. Use the SetLinkBy method to set this property. Default: Empty.</td>
</tr>
<tr>
<td>Items</td>
<td>Mongoose.IDO.Protocol.IDOUpdateItem array</td>
<td>Specifies an instance of the IDOUpdateItem class that contains zero or more IDOUpdateItem instances. The IDOUpdateItem class contains the update information for a single item.</td>
</tr>
<tr>
<td>TxnScope</td>
<td>Mongoose.IDO.Protocol.TxnScope</td>
<td>If set to Item, each individual item in the UpdateCollection request is saved in a separate transaction. For a hierarchical (nested) UpdateCollection request, the value of the TxnScope attribute at the root level determines the behavior for the entire UpdateCollection request. This attribute has no effect when it is included in an inner UpdateCollection within hierarchical requests.</td>
</tr>
</tbody>
</table>

**UpdateCollection example**

To execute an UpdateCollection request, first construct an instance of the UpdateCollectionRequestData class, and pass it to the UpdateCollection method on the IIDOCommands interface.

```csharp
UpdateCollectionRequestData request = new
UpdateCollectionRequestData();
UpdateCollectionResponseData response;
IDOUpdateItem newCust = new IDOUpdateItem();

request.IDOName = "SLCustomers";
request.RefreshAfterUpdate = true;

newCust.Action = UpdateAction.Insert;
newCust.ItemNumber = 1;
// used for error reporting
newCust.Properties.Add( "CustNum", "C000100" );
newCust.Properties.Add( "Name", "Boxmart" );
newCust.Properties.Add( "CurrCode", "USD" );
newCust.Properties.Add( "BankCode", "BK1" );

request.Items.Add( newCust );

response = this.Context.Commands.UpdateCollection( request );
```
LoadCollection/UpdateCollection example: saving changes to existing records

This example loads data from an IDO, updates the data and saves the collection.

```csharp
public void DoUpdate()
{
    LoadCollectionResponseData loadresponse;
    string sFilter = "CoNum = 'C0000567'";

    loadresponse = this.Context.Commands.LoadCollection("SLCos", "Charfld1, CustNum", sFilter, "", -1);
    if ( loadresponse.Items.Count > 0 )
    {
        UpdateCollectionRequestData updateRequest;
        IDOUpdateItem updateItem;
        // Create a new UpdateCollection request:
        updateRequest = new UpdateCollectionRequestData("SLCos");
        // Create a new update item for the row we loaded:
        updateItem = new IDOUpdateItem(UpdateAction.Update, loadresponse.Items[0].ItemID);
        // Add the CustNum property from LoadResposne, not modified:
        updateItem.Properties.Add("CustNum", loadresponse[0,"CustNum"].Value, false);
        // Add the Charfld1 property using a new value, modified:
        updateItem.Properties.Add("Charfld1", "Owzat?", true);
        // Add the update item to update the request:
        updateRequest.Items.Add(updateItem);
        // Save the changes:
        this.Context.Commands.UpdateCollection(updateRequest);
    }
}
```

Invoke

Use Invoke to call IDO methods, not including the custom load methods. Invoke requests are constructed using the InvokeRequestData IDO protocol class.

Properties

These properties are available on the InvokeRequestData class:

<table>
<thead>
<tr>
<th>Property</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDOName</td>
<td>System.String</td>
<td>Identifies the IDO used to execute the request</td>
</tr>
<tr>
<td>MethodName</td>
<td>System.String</td>
<td>Identifies the IDO method to be executed</td>
</tr>
<tr>
<td>Parameters</td>
<td>Mongoose.IDO.Protocol.InvokeParameterList</td>
<td>Specifies a collection that contains instances of the InvokeParameter class corresponding to the IDO method parameters</td>
</tr>
</tbody>
</table>
Invoke example 1

To execute an Invoke request, first construct an instance of the InvokeRequestData class, and pass it to the Invoke method on the IIDOCommands interface.

```csharp
InvokeRequestData request = new InvokeRequestData();
InvokeResponseData response;
request.IDOName = "UserNames";
request.MethodName = "UserValidSp";
request.Parameters.Add( "ajones" );
  // user name, input
request.Parameters.Add( IDONull.Value );
  // user ID, output
request.Parameters.Add( IDONull.Value );
  // description, output
request.Parameters.Add( IDONull.Value );
  // infobar, output
response = this.Context.Commands.Invoke( request );

if ( response.IsReturnValueStdError() )
{
  // get infobar output parameter
  string errorMsg = null;
  errorMsg = response.Parameters[3].Value;
}
else
{
  int userID = 0;
  string desc = null;

  userID = response.Parameters[1].GetValue<int>();
  if ( !response.Parameters[2].IsNull )
  {
    desc = response.Parameters[2].Value;
  }
}
```

Invoke example 2

In many cases it is more convenient to call the overloaded version of the Invoke method that accepts parameters for the IDONAME, MethodNAME, and method parameters InvokeRequestData properties.

```csharp
InvokeResponseData response;
if ( response.IsReturnValueStdError() )
```
Assigning and checking null values in IDO requests

You can use several IDO properties and methods to assign or check null values in IDO requests.

Assigning null values

When assigning a null value to a property or parameter in the IDO protocol classes, the best practice is to use the `IDONull.Value` static property.

```csharp
InvokeResponseData myMethodResponse;
UpdateCollectionRequestData updateRequest;

// pass NULL for the 3rd parameter
myMethodResponse = this.Invoke("MyMethod", "test", 100, IDONull.Value);

// set the Note property to NULL
updateRequest.Items[0].Properties.Add("Note", IDONull.Value);
```

Checking null values

When checking for null values, the best practice is to use the `IDONull.IsNull` static method. For parameter or property values in an IDO protocol class, you may also use the `IsNull` property to check for null values.

```csharp
// Check for null value in a LoadCollectionResponseData:
{
```
if (!loadResponse(0, "UserId").IsNull)
{
    userId = loadResponse(0, "UserId").GetValue<long>(-1);
}

// Check for a null parameter value:
public int MyFunc(Nullable<int> param)
{
    // Test for a null value:
    if (IDONull.IsNull(param))
    {
        // Handle the null parameter.
    }
}

Translation/localization considerations

All data values in IDO requests and responses are stored as string values in an "internal" format that is independent of any regional settings for the application or local machine where the code is executing.

Internal format for numbers

Numeric values are always stored internally using a period for the decimal separator (for non-integer values) and without any digit-grouping characters.

Internal format for dates

Date values are always stored internally using the format YYYYMMDD HH:MM:SS.mmm for date and time, YYMMDD for date only, and HH:MM:SS.mmm for time only (milliseconds are optional).

Type conversions

The IDO protocol classes store all property and parameter values internally as strings in a non-localized internal format. However, the framework also provides methods that allow you to transparently convert between supported native .NET CLR data types and the internal formatted string values. You should always use strongly typed variables when getting values from, or putting values into, any of the IDO protocol classes in order to take advantage of this feature and avoid problems related to systems running with different regional settings. This is especially important when working with date and numeric values.
The built-in VB functions for data type conversions (CStr, CDate, CInt, etc.) are sensitive to the regional settings for the application or local machine. Do not use these functions for converting to and from internal IDO strings.

Setting property and parameter values

Use the SetValue or Add methods to set parameter and property values into the IDO protocol classes.

Add/SetValue with InvokeRequestData

This example shows how to do this with the InvokeRequestData IDO protocol class:

```csharp
InvokeRequestData invokeRequest = new InvokeRequestData();
int version = 600;
DateTime recordDate = DateTime.Now;

invokeRequest.IDOName = "MyIDO";
invokeRequest.MethodName = "MyMethod";
invokeRequest.Parameters.Add("Mongoose");
invokeRequest.Parameters.Add(version);
invokeRequest.Parameters.Add(Guid.NewGuid());
invokeRequest.Parameters.Add(recordDate);
invokeRequest.Parameters.Add(IDONull.Value);  // Set later.

// Set the fifth parameter after it has been added:
invokeRequest.Parameters[4].SetValue(100);
```

Both the Add and SetValue methods accept a value of any supported .NET data type and automatically handle the conversion to internal format, so that the developer does not need to think about conversion issues.

Add/SetValue with UpdateCollectionRequestData

The code to set property values into the UpdateCollectionRequestData IDO protocol class is very similar to the previous example.

```csharp
UpdateCollectionRequestData updateRequest = new UpdateCollectionRequestData();
IDOUpdateItem updateItem = new IDOUpdateItem();
updateItem.Action = UpdateAction.Insert;
updateItem.Properties.Add("UserId", IDONull.Value);

// Set later.
updateItem.Properties.Add("UserName", "MGUser");
updateItem.Properties.Add("RecordDate", DateTime.Now);
updateItem.Properties["UserId"].SetValue(100L);
updateRequest.IDOName = "MyIDO";
updateRequest.Items.Add(updateItem);
```
Getting property and parameter values

Use one of the overloaded GetValue methods to get parameter and property values from the IDO protocol classes. These examples show how to do this using the LoadCollectionResponseData IDO protocol class.

GetValue with LoadCollectionResponseData

In this example, the desired native .NET data type is specified as an additional parameter in the call to GetValue (using the .NET Framework's generics feature).

```csharp
LoadCollectionResponseData loadResponse;
long userId = 0;
string userName = null;
DateTime recordDate;
Guid rowPointer;

loadResponse = this.Context.Commands.LoadCollection("UserNames", "UserId, UserName, RecordDate, RowPointer", ",", ",", 0);
userId = loadResponse[0, "UserId"].GetValue<long>();
userName = loadResponse[0, "UserName"].GetValue<string>();
recordDate = loadResponse[0, "RecordDate"].GetValue<DateTime>();
rowPointer = loadResponse[0, "RowPointer"].GetValue<Guid>();
```

Handling nullable values

Be aware that if any of the values are null, this version of the GetValue method throws an exception. When working with nullable values, you have three options to avoid throwing an exception:

• Check the IsNull property before calling GetValue.
• Call the overloaded GetValue method with a default value to be returned in case of null.
• Use .NET nullable value types.

Those three techniques are illustrated in this example:

```csharp
LoadCollectionResponseData loadResponse;
Nullable<long> userId;

loadResponse = this.Context.Commands.LoadCollection("UserNames", "UserId, UserName, RecordDate, RowPointer", ",", ",", 0);

// Check for null before calling GetValue:
if (!loadResponse[0, "UserId"].IsNull) {
    userId = loadResponse[0, "UserId"].GetValue<long>();
} else {
    userId = -1;
}

// Call GetValue, return -1 if null:
userId = loadResponse[0, "UserId"].GetValue<long>(-1);

// Call GetValue, check for null return value:
```
userId = loadResponse[0, "UserId"].GetNullableValue<long>();
if (!userId.HasValue)
    userId = -1;

Creating filter strings

If you need to dynamically build a SQL filter (WHERE clause) to be used in a LoadCollection request, the framework provides the SqlLiteral class to help build filter strings that contain embedded literals.

SqlLiteral.Format method

The static SqlLiteral.Format method accepts any supported .NET CLR typed value and converts it into a literal that can be used when constructing a filter as shown in this example:

```csharp
string filter = null;
filter = string.Format(
    "SessionId = {0} AND trans_num = {1}"
, SQL Literal.Format(SessionID),
    SQL Literal.Format(TransNum));
```

The above example produces a filter with the appropriate syntax for the literal values, similar to this:

```
SessionId = N' 7A4930D6-AE9E-4F32-9687-17ABDBF4E818' and trans_num = 1234
```

You must always pass in a strongly typed value in order to get the correct result. For example, you should not pass a string containing a date value.

This table shows some examples of input types, values, and the SqlLiteral.Format result for each.

<table>
<thead>
<tr>
<th>.NET CLR type</th>
<th>Input value</th>
<th>SqlLiteral.Format result</th>
</tr>
</thead>
<tbody>
<tr>
<td>System.Int32</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>System.String</td>
<td>Joe’s Bar</td>
<td>N'Joe”s Bar’</td>
</tr>
<tr>
<td>System.Decimal</td>
<td>150.32</td>
<td>150.32</td>
</tr>
<tr>
<td>System.Guid</td>
<td>7A4930D6-AE9E-4F32-9687-17ABDBF4E818</td>
<td>N' 7A4930D6-AE9E-4F32-9687-17ABDBF4E818'</td>
</tr>
</tbody>
</table>

Supported .NET CLR types

These .NET data types are supported by the IDO protocol classes using the GetValue, Add, and SetValue methods.
## Transactions

The IDO runtime automatically executes all UpdateCollection requests in a transaction. It also executes IDO methods in a transaction if they are marked as “transactional” methods. All other IDO requests are executed without a transaction. Note that if a method is not marked “transactional” and is called from a method that is marked “transactional”, then it still executes in the caller’s transaction.

For transactional IDO methods, if the method returns an integer value less than 5, the transaction is committed. The exception is if the method was executed within an existing outer transaction. Then the transaction commits only if and when the outer transaction commits. If the IDO method returns an integer value greater than or equal to 5 the transaction is rolled back immediately. The Mongoose standard is to return 16 to indicate a method failed. The transaction also rolls back if the method throws an exception.

If you need more control over transaction scope, which is often necessary for long running processes, this can be achieved while still allowing the IDO runtime to manage transactions.

For example, consider a method that posts a large batch of records. We need a method that reads the batch, loops through each record, and attempts to post it. If the post is successful, the action commits, otherwise it rolls back. The best way to accomplish this is to create two methods: a top level entry point method that queries the records to be posted; and another method to do the posting. The top-level method should not be marked as transactional, while the posting is transactional. The entry point method queries the records to be posted and loops through each one, calling the posting method to post each record. This way, if a record fails to post, it does not roll back all posted records, only the failed record.
Diagnostics and debugging

Logging diagnostic messages

All applications and services in the Infor framework use a common message logging facility. This enables the framework to provide a single, consolidated view of all activity logged on the local machine. For developers, this log is available in the IDO Runtime Development Server (IDORuntimeHost.exe).

Message Logging API

```csharp
public static void LogUserMessage(
    string messageSource,
    UserDefinedMessageType messageType,
    string message)
{
}

public static void LogUserMessage(
    string messageSource,
    UserDefinedMessageType messageType,
    string format, params object[] args)
{
}
```

Parameters

- `messageSource` - A short, user-defined identifier of the source of a message (for example, class or project name)
- `messageType` - One of the UserDefinedMessageType enumeration values; used for searching and filtering log messages
- `message` - A string containing the log message text
- `format` - A log message format string (see “String.Format()” in the .NET Framework documentation for more information about formatting strings)
- `args` - A variable-length list of format-string substitution parameters

Examples

```csharp
IDORuntime.LogUserMessage(
    "SLCustomers",
    UserDefinedMessageType.UserDefined0,
    "Method A() called.");

IDORuntime.LogUserMessage(
    "SLCustomers",
    UserDefinedMessageType.UserDefined0,
```
"Parameter 1 is {0}.",
pl); 

Wire Tap

You can use the Wire Tap feature of the IDO Runtime Development Server to monitor all the IDO request and response documents generated by a specific user session. See the online help for the IDO Runtime Development Server.
Chapter 2: IDO extension classes

About IDO extension classes

An IDO extension class is a .NET class that allows developers to extend the functionality of an existing IDO by adding methods and event handlers. IDO extension classes are compiled into a .NET class library assembly and stored in the IDO metadata database. The IDO runtime engine loads these assemblies on demand and calls methods and event handlers in the extension classes in response to IDO requests.

An extension class is short-lived; it is created at the start of a request and disposed of immediately when the response is completed. Therefore, no state should be stored in an extension class.

Any public class in an IDO extension class assembly can be identified as the extension class for an IDO in the IDO metadata database.

If the entry point IDO method is marked as Transactional, then all IDO calls and direct database calls are wrapped in the same transaction within the context of that method. UpdateCollection and LoadCollection requests are always executed in a transaction, with the exception that a caller may specify on UpdateCollection that each row be executed in its own transaction. There is no way to submit multiple IDO requests from outside the IDO runtime and combine those into a single transaction, but within a single request the developer has full control over transaction scope.

IDO extension class assemblies are .NET Framework class libraries that are created using any of the .NET languages.

Creating an IDO extension class project

To create an IDO extension class project:

2. From the File menu, select New > Project.
3. In the New Project dialog box, perform these steps:
   a. Select the desired language.
   b. On the Templates > Visual Basic tab, select the Class Library template.
   c. Specify a name and location for your project.
   d. Click OK.
   A Class1.cs file is created.
4 In the Solution Explorer window, right-click on the file and select Remove.
5 From the Project menu, select Add Reference.
6 In the Reference Manager dialog box, perform these steps:
   a Click the Browse tab and navigate to your application’s installation directory.
   b Add references to these assemblies:
      • IDOCore.dll
      • IDOBase.dll
      • IDOProtocol.dll
      • MGShared.dll
      • WSEnums.dll
   c Click OK.
7 From the Project menu, select Add New Item.
8 In the Add New Item dialog box, perform these steps:
   a In the Installed > Common Items tab, select Code File.
   b In the Name field, specify AssemblyInfo.cs.
   c Click Add.
9 Open the new AssemblyInfo.cs source file and add this assembly attribute:

   [Assembly: Mongoose.IDO.IDOExtensionClassAssembly("assembly-name")]

Substitute a meaningful name for the assembly-name parameter. You can specify any name, but typically, the name of the IDO project associated with the classes in this assembly is used. When this class library is imported to the IDO Extension Class Assemblies form, the name you specify here becomes the name of the assembly.

Caution: If you add a reference to another assembly from your IDO extension class assembly, the referenced assembly must also be imported as another custom assembly. When .NET tries to locate the referenced assembly, the IDO runtime service searches the database for a matching assembly and loads it on demand.

Importing an IDO extension class assembly

To import an IDO extension class assembly:

1 Open the IDO Extension Class Assemblies form.
2 Specify the assembly name.
3 Click Import Assembly.
4 Specify the directory path of the assembly.
   Note: The assembly must be in DLL format.
5 Click Save.
6 Optionally, test the assembly.
7 Click **Check In**.

To link the custom assembly to the IDO, use the **IDO**s form. To link the IDO to the methods in the extension class, use the **IDO Methods** form.

Creating, extending, and replacing an IDO extension class assembly

To create or extend and replace an IDO extension class assembly:

1 Open the **IDO Extension Class Assemblies** form.
2 Click **New**.
3 Specify this information:
   - **Assembly Name**
     Specify the name of the assembly.
   - **Assembly Type**
     Select **Build From Source**.
   - **Language**
     Select **C#** or **VB.NET**.
   - **References**
     Specify a comma delimited list of assembly references that is required to build the assembly.
     **Note**: This information can be either the names of other IDO extension class assemblies or the .NET assembly file name in DLL format. This field defaults to the minimum set of required assembly references.

4 Click **Save**.
5 Click **Source Code**.
6 On the **IDO Extension Class Source Code** form, perform these steps:
   a Click **New**.
   b Optionally, specify a file name.
      **Note**: When you leave the **Filename** field as blank, the source code edit will default to a skeleton IDO extension class. This information is only a suggested value and can be modified or replaced as desired.
   c Click **Code Editor**.
      **Note**: This form is an enhanced editor that includes color syntax highlighting and automatic indenting.
   d On the **Code Editor** form, update the source code.
      **Note**: The source code can be either **C#** or **VB.NET** depending on the **Language** that you defined on the **IDO Extension Class Assemblies** form.
   e Click **OK**.
7 On the **IDO Extension Class Assemblies** form, click **Build Assembly**. A success message is displayed.

8 On the **IDO Extension Class Source Code** form and the **IDO Extension Class Assemblies** form, click **Check In**.

To link the custom assembly to the IDO, use the **IDOs** form. To link the IDO to the methods in the extension class, use the **IDO Methods** form.

Implementing an IDO extension class

**Mongoose.IDO.IIDOExtensionClass interface**

In order for an extension class to access the calling user's context (session information, databases, and so forth), the extension class should implement the IIDOExtensionClass interface in the Mongoose.IDO namespace.

```csharp
public interface IIDOExtensionClass
{
    void SetContext(Mongoose.IDO.IIDOExtensionClassContext context);
}
```

**Mongoose.IDO.IIDOExtensionClassContext interface**

The SetContext method is called once when the extension class is created, passing a reference to the IIDOExtensionClassContext interface for accessing the calling user's context.

```csharp
public interface IIDOExtensionClassContext
{
    Mongoose.IDO.IVirtualIDO IDO { get; }
    Mongoose.IDO.IIDOCommandHelper Commands { get; }
}
```

The framework provides a class named Mongoose.IDO.ExtensionClass that is intended to be used as the base class for most typical IDO extension classes. This class provides a default implementation of the IIDOExtensionClass interface and provides a number of methods to facilitate calling methods, loading and updating collections, and other common tasks, which will be covered later.
Declaration of an IDO extension class for an IDO

When declaring an IDO extension class, you can add the optional `IDOExtensionClass` attribute in order to facilitate adding the necessary IDO metadata to associate an extension class with an IDO. This sample code shows how to associate an extension class with the SLJobMats IDO.

```csharp
[IDOExtensionClass("SLJobMats")]
public class SLJobMats : IDOExtensionClass
{
    // ...Implementation goes here...
}
```

It is generally a good idea to import some common framework namespaces to make your code less verbose and more clear, and to enable the `Explicit` and `Strict` VB compiler options.

IDO extension class starter template

This sample code can be used as a template for creating new IDO extension classes.

```csharp
using Mongoose.IDO;
using Mongoose.IDO.Protocol;
using Mongoose.IDO.DataAccess;
using Mongoose.Core.Common;

[IDOExtensionClass("class-name")]
public class class-name : IDOExtensionClass
{
    // optionally override the SetContext method
    // public override void SetContext(
    //     IIDOExtensionClassContext context )
    // {
    //     // call base class implementation
    //     base.SetContext(context);
    //     // Add event handlers here, for example:
    //     context.IDO.PostLoadCollection +=
    //     new IDOEventHandler( IDO_PostLoadCollection );
    // }
    // ...Implementation goes here...
}
```

Adding methods

To call a method on an IDO extension class, the method must be declared as public and it must also be defined in the IDO metadata as a hand-coded method.
A hand-coded method can be either a standard method, which is a **Function** with zero or more input/output parameters and returns an integer; or a custom load method, which is a **Function** with zero or more parameters and returns an instance of either an IDataReader or a DataTable (see the .NET Framework documentation for information about these).

When implementing a hand-coded method that will be callable through the IDO runtime, you could add the optional **IDOMethod** attribute, which can be used in the future to facilitate adding the necessary IDO metadata to make the method accessible.

### Sample standard method call

For example, the declaration of a standard method might look like this Function declaration:

```csharp
[IDOMethod(MethodFlags.None, "Infobar")]
public int DoWork(string SessionID, long TransNum, ref string Infobar)
{
    // ...Implementation goes here...
}
```

### IDOMethod parameters

The IDOMethod attribute takes two optional parameters: `flags` and `messageParm`.

#### Flags

The `flags` parameter is one or more of the values defined by the **MethodFlags** enumeration. This table shows the **MethodFlags** values:

<table>
<thead>
<tr>
<th>MethodFlags value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Standard method call.</td>
</tr>
<tr>
<td>CustomLoad</td>
<td>Custom load method.</td>
</tr>
<tr>
<td>RequiresTransaction</td>
<td>Method runs in the context of a transaction.</td>
</tr>
</tbody>
</table>

#### MessageParm

The `messageParm` parameter is a string and is the name of the parameter that will be used to return messages to the caller (Infobar). The parameter that is identified should be declared as a **String** and passed by reference.

Remember that these attributes are not used at run time. They might be used in the future to facilitate adding and synchronizing the IDO metadata for an extension class and its methods when an assembly is imported.
Including filters in custom load methods to load collections

When a custom load method is used to load a collection, the filters defined on an IDO are not applied automatically by the system, so you must explicitly implement the filtering in the custom method. (Before you implement a custom load method, you should be very familiar with the IDO definition.)

The system passes the currently active filter to the method if the method defines a specifically-named parameter. For methods that are hand-coded in .NET, this parameter must be named IdoFilter and it must be of type string. For stored procedure methods, this parameter must be named @IdoFilter and it must be of type nvarchar(max). This must be the last parameter listed for the method.

However, when you specify method parameters in the IDO metadata (that is, by using the development forms IDOs, IDO Methods, and IDO Method Parameters) or in WinStudio Design mode, you must not provide this filter parameter. It is an implicit parameter that the system satisfies if it exists; however, if the parameter does not exist, the method still works. There is no logical value that can be specified in the IDO metadata or design mode for the filter parameter, since it varies depending on who is logged in, which groups they are members of, and so on.

The filter that is passed is a fully-parsed, valid SQL filter. The filter can be appended with a conjunction to the WHERE clause of a query, or the filter by itself can be the entire clause for the query. However, since the filter can refer to any table referenced in the IDO definition, you must make sure that any tables referenced in the filter are present in the query.

Using variables and a bookmarking algorithm to get more rows in a custom load method

In order to include the “Get More Rows” functionality in a custom load method, you must create a bookmarking algorithm. The framework provides and receives relevant information for bookmarking through these variables:

- **EnableBookmark**: This is set to either True or False, depending on whether the load request wants to apply bunching. If the variable is set to False, the custom load method is expected to ignore bunching.
- **Bookmark**: This is the bookmark value. It is blank for an initial load type, and contains values for bookmarks (which the algorithm must create) for subsequent load calls. A bookmark must uniquely identify a set of rows, referred to as a “bunch.”
- **LoadType**: This is set to First, Last, Next, or Prev. First and Last are initial load types: when a collection initially loaded, a bookmark is not needed and the framework expects either the first or the last bunch to be returned. Next and Prev are used on subsequent loads with a bookmark, and the framework expects the bunch either after or before the bunch identified by the bookmark. For the Last and Prev load types (essentially, “browsing backward” through the collection), the returned rows are expected to be sorted in reverse order.
- **RecordCap**: This is the maximum number of rows the framework expects to be returned for a single bunch.

**Note**: The method is expected to return one extra row (RecordCap+1 rows total), if it is available. The framework uses this information to determine whether there are more rows available to be
queried for the next bunch. The bookmark is still expected to identify uniquely the set of rows that will be returned to the user; the extra row is not considered part of the set.

After the custom load method constructs the set of rows it is going to return, it is expected to update the Bookmark variable with a unique value that represents the set of rows it is returning.

Custom load methods that are implemented as stored procedures receive these variables through the use of process variables (see Process variables on page 42). Before it calls into a custom load method, the framework creates these variables from the load request that generated the call. Upon return, the framework checks the process variable called “Bookmark” for the new bookmark. Custom load methods that are implemented as hand-coded methods have access to the load request structure, which contains these variables as properties of the request.

**Bookmarking algorithms**

Although you must create the bookmarking algorithm for your custom load method, Infor provides some examples.

Example 1: If you have a collection with a unique, consecutive integer row number property, your bookmark could be a single number, and the custom load method sorts the collection by that property and returns the top RecordCap+1 rows for a load type of **First**, setting the bookmark to the second-to-last row's number. Loads of type **Next** return RecordCap+1 rows, where RowNumber > bookmark. The method performs similarly for load types **Last** and **Prev**, except that it sorts in reverse order and uses RowNumber < bookmark. The algorithm must have special cases for an empty result set, and for a single row being returned.

Example 2: The algorithm the framework uses for standard loads is more complicated, since it must work with any possible collection, combination of properties being loaded, etc. The bookmark is an XML fragment with this schema:

```
<B>
  <P><p>ColumnName1</p><p>ColumnName2</p>…</P>
  <D><f>DescendingFlag1(true or false)</f><f>Flag2</f>…</D>
  <F><v>FirstRowValue1</v><v>FirstRowValue2</v>…</F>
  <L><v>LastRowValue1</v><v>LastRowValue2</v>…</L>
</B>
```

This XML fragment is combined with the `<RecordCap>` and `<LoadType>` elements in the request. The framework fills the bookmark with the data necessary to uniquely describe the set of rows that it just retrieved. It uses either the RowPointer column, or a set of columns that make up a unique key, together with a descending flag for each column, and the values of those columns in the first and last rows of the set.

**IDO events**

When an IDO is processing a LoadCollection or UpdateCollection request, it fires several events. An IDO extension class can add its own event handlers to these events to get control when these events are fired.
The available events are:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PreLoadCollection</td>
<td>Fires when a LoadCollection request is received.</td>
</tr>
<tr>
<td>PostLoadCollection</td>
<td>Fires after a LoadCollection request has been processed.</td>
</tr>
<tr>
<td>PreUpdateCollection</td>
<td>Fires when an UpdateCollection request is received.</td>
</tr>
<tr>
<td>PostUpdateCollection</td>
<td>Fires after an UpdateCollection request has been processed.</td>
</tr>
<tr>
<td>PreItemInsert</td>
<td>Fires when an insert item request is received.</td>
</tr>
<tr>
<td>PostItemInsert</td>
<td>Fires after an insert item request has been processed.</td>
</tr>
<tr>
<td>PreItemUpdate</td>
<td>Fires when an update item request is received.</td>
</tr>
<tr>
<td>PostItemUpdate</td>
<td>Fires after an update item request has been processed.</td>
</tr>
<tr>
<td>PreItemDelete</td>
<td>Fires when a delete item request is received.</td>
</tr>
<tr>
<td>PostItemDelete</td>
<td>Fires after a delete item request has been processed.</td>
</tr>
<tr>
<td>PreInvokeMethod</td>
<td>Fires when an invoke IDO method request is received.</td>
</tr>
<tr>
<td>PostInvokeMethod</td>
<td>Fires after an invoke IDO method request has been processed.</td>
</tr>
<tr>
<td>Initialized</td>
<td>Fires after a create an IDO instance request has been processed.</td>
</tr>
<tr>
<td>Disposing</td>
<td>Fires when a dispose an IDO instance request is received.</td>
</tr>
</tbody>
</table>

Adding an IDO event handler

IDO event handler methods should be declared as private methods in your IDO extension class. These methods should use the same signature as this example:

```csharp
private void HandlePreLoadCollection(object sender, IDOEventArgs args)
```

These IDO events are available on the IDO property of the IIDOEExtensionClassContext interface. To add your handler to an IDO event, you must override the IIDOEExtensionClass.SetContext method in your IDO extension class and use the AddHandler keyword as shown in this example:

```csharp
public override void SetContext(IIDOExtensionClassContext context)
{
    // Call the base class implementation:
    base.SetContext(context);

    // Add event handlers here, for example:
    this.Context.IDO.PreUpdateCollection += this.PreUpdateCollection;
}
```
When you create an extension class assembly from source, or you extend and replace one, you must not override SetContext. Instead, you must use attributes on the methods to add event handlers, as shown in this example:

```csharp
[IDOAddEventHandler( IDOEvent.PostInvokeMethod )]
private void ExtPostInvokeMethod( object sender, IDOInvokeEventArgs args )
{
    if ( args.InvokeRequest.MethodName == "MyTestMethod" )
    {
    }
}
```

### Event handler parameters

The first parameter on an IDO event handler method is the `sender` parameter, which is simply a reference to the IDO that initiated the event. This is equivalent to the `IIDOExtensionClassContext.IDO` property.

The second parameter is an instance of the `IDOEventArgs` class which allows you to access the original IDO request that initiated the event, and for the Post events, the IDO response also.

```csharp
public class IDOEventArgs : EventArgs
{
    public PayloadBase RequestPayload { get; }
    public PayloadBase ResponsePayload;
}
```

This table describes the contents of the `IDOEventArgs` parameter for each IDO event.

<table>
<thead>
<tr>
<th>Event</th>
<th>RequestPayload property</th>
<th>ResponsePayload field</th>
</tr>
</thead>
<tbody>
<tr>
<td>PreLoadCollection</td>
<td>Contains an instance of the LoadCollectionRequestData class that the IDO is processing</td>
<td>Nothing</td>
</tr>
<tr>
<td>PostLoadCollection</td>
<td>Contains an instance of the LoadCollectionRequestData class that the IDO processed</td>
<td>Contains an instance of the LoadCollectionResponseData class that will be returned to the caller</td>
</tr>
<tr>
<td>PreUpdateCollection</td>
<td>Contains an instance of the UpdateCollectionRequestData class that the IDO is processing</td>
<td>Nothing</td>
</tr>
<tr>
<td>PostUpdateCollection</td>
<td>Contains an instance of the UpdateCollectionRequestData class that the IDO processed</td>
<td>Contains an instance of the UpdateCollectionResponseData class that will be returned to the caller</td>
</tr>
</tbody>
</table>
An event handler method can access the IDO request and response payloads as shown in this examples.

```csharp
private void HandlePreLoadCollection(object sender, IDOEventArgs args)
{
    // Get the original request:
    LoadCollectionRequestData loadRequest;
    loadRequest = (LoadCollectionRequestData)args.RequestPayload;
    // ...Additional logic based on loadRequest.
}

private void HandlePostLoadCollection(object sender, IDOEventArgs args)
{
    // Get the original request:
    LoadCollectionRequestData loadRequest;
    LoadCollectionResponseData loadResponse;
    loadRequest = (LoadCollectionRequestData)args.RequestPayload;
    loadResponse = (LoadCollectionResponseData)args.ResponsePayload;
    // ...Additional logic based on loadRequest or loadResponse.
}

private void HandlePreUpdateCollection(object sender, IDOEventArgs args)
{
    // Get the original request:
    UpdateCollectionRequestData updateRequest;
    updateRequest = (UpdateCollectionRequestData)args.RequestPayload;
    // ...Additional logic based on updateRequest.
}

private void HandlePostUpdateCollection(object sender, IDOEventArgs args)
{
    // Get the original request:
    UpdateCollectionRequestData updateRequest;
    UpdateCollectionResponseData updateResponse;
    updateRequest = (UpdateCollectionRequestData)args.RequestPayload;
    updateResponse = (UpdateCollectionResponseData)args.ResponsePayload;
    // ...Additional logic based on loadRequest or loadResponse.
}
```
Special cases for UpdateCollection events

There are special cases when an IDO UpdateCollection request might not process all items in the update. For example, this might happen when a caller is saving hierarchical data, which is represented as an UpdateCollection request with nested UpdateCollection child requests. The framework first processes only the deleted items in the child requests, then all items in the parent, followed by the inserted or updated items in the child requests. This example shows how an event handler can determine if the framework is processing inserted, update and/or deleted items.

```csharp
private void HandlePostUpdateCollection(object sender, IDOEventArgs args)
{
    // Get the original request:
    UpdateCollectionRequestData updateRequest;
    UpdateCollectionResponseData updateResponse;
    IDOUpdateEventArgs updateArgs;

    updateArgs = (IDOUpdateEventArgs)args;
    updateRequest = (UpdateCollectionRequestData)args.RequestPayload;
    updateResponse = (UpdateCollectionResponseData)args.ResponsePayload;

    {
        // ...Only perform this logic if processing deleted items:
    }

    {
        // ...Only perform this logic if processing inserted items:
    }

    {
        // ...Only perform this logic if processing updated items:
    }
    // ...Additional logic based on loadRequest or loadResponse.
}
```

Using the ActionMask property

In the example above, the IDOUpdateEventArgs class inherits from IDOEventArgs, but has one additional property: ActionMask. The ActionMask is declared as an UpdateAction, which is an enum (shown below).

```csharp
public enum UpdateAction : ushort
{
    None = (ushort)0x0000,
    Insert = (ushort)0x0001,
    Update = (ushort)0x0002,
}
```
The ActionMask is usually a combination of Insert, Update and Delete (All), meaning the UpdateCollection that was just performed executed all types of actions.

However, there are certain circumstances when this can be a subset of Insert, Update and Delete, which means only the items that match the mask were actually saved. There are times when the IDO runtime must save items in a particular order (with nested updates), and it might first process only deleted items; then in another pass it executes all inserts and updates.

ActionMask is valid for both Pre- and PostUpdateCollection. The PreUpdateCollection parameters are the same as the PostUpdateCollection parameters – but the action indicates which items are about to be saved, rather than which items were just saved.

Replacing standard IDO processing with event handlers

By adding event handlers, not only can you perform additional logic when a particular event fires, but in the case of PreUpdateCollection and PreLoadCollection you can also completely replace the default framework implementation by constructing your own response and setting that in the IDOEventArg.ResponsePayload field. This example shows how this is done for an UpdateCollection request.

```csharp
private void HandlePreUpdateCollection(object sender, IDOEventArgs args)
{
    // Get the original request:
    UpdateCollectionRequestData updateRequest;
    UpdateCollectionResponseData updateResponse;
    updateRequest = (UpdateCollectionRequestData)args.RequestPayload;
    updateResponse = new UpdateCollectionResponseData(updateRequest);
    // Process items in the request, filling in the response...
    // Override the standard processing, returning my
    // custom response instead:
    args.ResponsePayload = updateResponse;
}
```

Generating application events

The ExtensionClassBase class provides a method named FireApplicationEvent that can be used to generate an application event.

```csharp
protected bool FireApplicationEvent(string eventName,
    bool synchronous,
    bool transactional,
    ref string result,
    ref ApplicationEventParameter[] parameters)
```
Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eventName</td>
<td>The name of the event to generate</td>
</tr>
<tr>
<td>synchronous</td>
<td>Determines if the event handlers should be executed synchronously (True) or asynchronously (False).</td>
</tr>
<tr>
<td>transactional</td>
<td>Determines if the event handlers should be executed in a transaction. If passed as False and one of the synchronous handlers fails, any database activity performed by prior handlers remains committed.</td>
</tr>
<tr>
<td>result</td>
<td>The value of the last RESULT() keyword executed on a Finish or Fail action on any handler.</td>
</tr>
<tr>
<td>parameters</td>
<td>An array of named parameters passed to the event handlers.</td>
</tr>
</tbody>
</table>

Example

```csharp
using Mongoose.IDO;
using Mongoose.IDO.Protocol;
using Mongoose.IDO.DataAccess;
using Mongoose.Core.Common;

[IDOExtensionClass("MyIDO")]
public class TestFiringEvent : IDOExtensionClass
{
    [IDOMethod(MethodFlags.None, "Infobar")]
    public void FireMyEvent(string Parm1, long Parm2, ref string Infobar)
    {
        ApplicationEventParameter[] ParmList = new ApplicationEventParameter[2];
        string result = null;
        ParmList[0] = new ApplicationEventParameter();
        ParmList[0].Name = "PString";
        ParmList[0].Value = Parm1;
        ParmList[1] = new ApplicationEventParameter();
        ParmList[1].Name = "PNumeric";
        ParmList[1].Value = Parm2.ToString();
        if (!FireApplicationEvent("MyEvent", true, true, out result, ref ParmList))
        {
            Infobar = result;
        }
    }
}
```
Code samples

For code samples, see Code samples on page 48.

ApplicationDB class

The ApplicationDB class, part of the Mongoose.IDO.DataAccess namespace, provides direct access to the application database, plus access to common framework features such as application messages and session variables. You can create a new instance of this class by calling the CreateApplicationDB method, which is implemented by the IDOExtensionClass class.

```csharp
AppDB db = Mongoose.IDO.IDORuntime.Context.CreateApplicationDB();
```

Application database connections

In general, the best practice to access the database to execute queries, updates or stored procedure calls is to go through the Infor framework using IDO requests. However, there might be times when it is necessary or more efficient to access the database directly (for example, building and executing dynamic SQL or iterating through records from an extremely large query result set). Direct database access is accomplished using the standard .NET Framework database interfaces: IDbConnection and IDbCommand. See the Microsoft .NET Framework documentation for more information on these classes.

Opening a connection to the application database

The ApplicationDB.Connection property provides an opened SqlConnection instance pointing to the application database. By going through the ApplicationDB.Connection property, the framework is able to open and initialize the connection based on the calling user's session information.

Executing database commands

The IDbCommand class is used to execute stored procedures, queries, updates, or any other T-SQL command. The IDbCommand must be created by using the ApplicationDB.CreateCommand() method, setting the CommandType and CommandText properties, and adding any parameters to the Parameters property. The parameters are added by using the ApplicationDB.AddCommandParameterWithValue or ApplicationDB.AddCommandParameter methods. When the command is executed, you should use the ApplicationDB.ExecuteScalar, ApplicationDB.ExecuteNonQuery, or ApplicationDB.ExecuteReader methods. Using any of these three methods allows you to leverage the framework infrastructure for executing SQL commands including logging and exception-message translation.
Application messages

Access to translatable strings and messages stored in the application database message tables is implemented by the ApplicationDB class. The strings and messages that are returned are queried and constructed according to the calling session’s regional settings. Access is provided through the IMessageProvider interface and makes use of application database messaging stored procedures such as MsgAppSp.

These methods are published by the IMessageProvider interface:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public string GetMessage(string msgID)</td>
<td>Get a string or message by message ID.</td>
</tr>
<tr>
<td>public string GetMessage(string msgID, params object[] args)</td>
<td>Get a string or message by message ID with substitution parameters.</td>
</tr>
<tr>
<td>public string AppendMessage(string message, string msgID)</td>
<td>Append a string or message by message ID.</td>
</tr>
<tr>
<td>public string AppendMessage(string message, string msgID, params object[] args)</td>
<td>Append a string or message by message ID with substitution parameters.</td>
</tr>
<tr>
<td>public string GetErrorMessage(string objectName, AppMessageType messageType)</td>
<td>Get a constraint error message by object name and type.</td>
</tr>
<tr>
<td>public string GetErrorMessage(string objectName, AppMessageType messageType, params object[] args)</td>
<td>Get a constraint error message by object name and type with substitution parameters.</td>
</tr>
</tbody>
</table>

Message lookup using the ApplicationDB class

This example shows how to use the ApplicationDB class to retrieve messages.

```csharp
IMessageProvider messageProvider = null;
using (ApplicationDB appDB = this.CreateApplicationDB())
{
    messageProvider = appDB.GetMessageProvider();
    InfoBar = messageProvider.AppendMessage(InfoBar, "I=CmMustPerform", "@:PostPendingMaterialTransactions");
}
```

If you are writing code in an extension class method for a class that derives from IDOExtensionClass, you can simply call the IDOExtensionClass.GetMessageProvider method as a shortcut to getting a reference to the IMessageProvider interface, as shown in this example:

```csharp
IMessageProvider messageProvider = null;
messageProvider = this.GetMessageProvider();
```
Session variables

The AppDB class publishes these methods for manipulating session variables for the caller's session.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void SetSessionVariable(string name, string varValue)</td>
<td>Set the value of a session variable.</td>
</tr>
<tr>
<td>public string GetSessionVariable(string name)</td>
<td>Get the value of a session variable.</td>
</tr>
<tr>
<td>public string GetSessionVariable(string name, string defaultValue)</td>
<td>Get the value of a session variable, returning the specified default value if it does not exist.</td>
</tr>
<tr>
<td>public string GetSessionVariable(string name, string defaultValue, bool delete)</td>
<td>Get the value of a session variable, returning the specified default value if it does not exist, and delete it if it does exist.</td>
</tr>
<tr>
<td>public void DeleteSessionVariable(string name)</td>
<td>Delete a session variable.</td>
</tr>
</tbody>
</table>

Process variables

Process variables are similar to session variables, except the scope is tied to a single database connection. This is unlike sessions, which span any number of database connections. A process
variable declared before one method call could be in a different scope than a later method call which was run on a different database connection.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void SetProcessVariable(string name, string varValue)</td>
<td>Create or update a process variable in the database.</td>
</tr>
<tr>
<td>public void SetProcessVariable(string name, string varValue, bool ignoreError)</td>
<td>Get the value of a process variable.</td>
</tr>
<tr>
<td>public string GetProcessVariable(string name)</td>
<td>Get the value of a process variable.</td>
</tr>
<tr>
<td>public string GetProcessVariable(string name, string defaultValue)</td>
<td>Get the value of a process variable, using default-Value if it does not exist.</td>
</tr>
<tr>
<td>public string GetProcessVariable(string name, string defaultValue, bool deleteVariable)</td>
<td>Get the value of a process variable, using default-Value if it does not exist. Delete the process variable after retrieving the value.</td>
</tr>
<tr>
<td>public void DeleteProcessVariable(string name)</td>
<td>Delete the process variable.</td>
</tr>
</tbody>
</table>

Using ApplicationDB in an extension class method

This example shows how to use ApplicationDB in an extension class.

```csharp
public int MyMethod()
{
    using (ApplicationDB appDB = this.CreateApplicationDB())
    {
        // Use appDB here...
    }
}
```

**Note:** The ApplicationDB instance is created as part of the Using...End Using block. This ensures that the ApplicationDB instance will be disposed and any resources it is holding will be released as soon as the instance goes out of scope. See the .NET Framework documentation for more information on the Using statement.
Disposing of IDO extension classes

If an IDO extension class implements the IDisposable interface, the Dispose method will be called by the IDO runtime when it is finished with each instance. The IDOExtensionClass class implements IDisposable and provides a virtual method which you can override in a derived class to get control when an instance is disposed.

```csharp
protected virtual void Dispose( bool disposing );
```

See the .NET Framework documentation for more information on the IDisposable interface.

Translation/localization considerations

Type conversions

The built-in VB functions for data type conversions (CStr, CDate, CInt, etc.) are sensitive to the regional settings for the application or local machine, and therefore should not be used for type conversions in IDO extension class methods.

The framework stores all property values internally as strings regardless of their data type. In WinStudio, this is also true for values stored in components, variables and IDO collections. The internal string format is culture independent so the built-in string conversion functions may not work, or may work inconsistently depending on the data or the current culture settings. The internal formatted string value is always available, but when accessing numeric, date or Guid property values programmatically, you should use the APIs provided by the framework that will convert the values from internal string format to the proper “strong” data types.

IDOREquest/Response Property Values (IDOProtocol classes)

Property values in IDO Request and Response classes are represented using the Mongoose.IDO.Protocol.IDOValueType class, or a class that inherits from this class.

This example accesses properties in a LoadCollectionResponseData instance:

```csharp
LoadCollectionResponseData responseData;
string name = null;
DateTime recordDate;
decimal cost;
decimal newCost = Convert.ToDecimal(100.0);
Guid id;

// Perform LoadCollection:
// Access internal value for strings:
name = responseData[0, "Name"].Value;
```
// Convert internal date to DateTime:
recordDate = responseData[0, "RecordDate"].GetValue<DateTime>();

// Convert to Decimal, specifying a default if the property is null:
cost = responseData[0, "Cost"];GetValue<decimal>(Convert.ToDecimal(0.0));
id = responseData[0, "ID"];GetValue<Guid>();

// Assign a new value to cost:
responseData[0, "Cost"];SetValue(newCost);

Note that the Name property simply gets the internal format, since string values do not need conversion. Also note that the GetValue call for the cost property specifies an optional default value to return if the property is null. You can also check for null values using the IsNull property.
The last line assigns a new value to the cost property. Passing in a decimal value to the SetValue method allows the framework to perform the conversion to the internal string format automatically.

Testing extension class assemblies locally

To test a new version of an extension class assembly locally before checking it in, copy the new version of the extension class assembly to a folder named:

appName\idoxca\configName

where:

• appName is the application root folder, usually C:\Program Files\Infor\MongooseApp.
• idoxca is a directory you must set up specifically for testing your application and configurations.
• configName is the name of the configuration you will be using to test the DLL.

The IDO runtime checks this location first when it loads an assembly.
This might also be useful to developers working on different classes in the same assembly, since only one developer can have an assembly checked out at a time.
You might need to discard the IDO metadata cache to force the IDO runtime to load the new version.
This section applies only if you have a Visual Studio version other than the “Express” edition.

Debugging extension classes

IDO extension class code can be debugged using the Visual Studio debugger. In order to debug an extension class, the developer must:

1 Build a debug version of the extension class assembly.
2 Import the assembly (DLL) and the symbols (PDB) into the objects database using the **IDO Custom Assemblies** form (first checking out the assembly, if it has been checked in).

3 Run the IDO Runtime Development Server (IDORuntimeHost.exe) on the developer’s local machine. Alternatively, you can use the technique described in **Testing extension class assemblies locally** on page 45.

4 Run WinStudio configured to access the IDO runtime on the local machine.

5 Open Visual Studio and select **Attach to Process** from the **Debug** menu.

6 Select IDORuntimeHost.exe in the list of available processes and click **Attach**.

You should now be able to set breakpoints in the Visual Studio debugger and step through code.

Logging diagnostic messages

IDO extension class methods can send messages to the common logging facility IDO Runtime Development Server (IDORuntimeHost.exe) by calling one of the overloaded IDORuntime.LogUserMessage methods.

Example

This sample sends a message to the log each time the ValidateCreditCard method is called.

- The first parameter is **messageSource** (String). This is the source identifier that displays in the log viewer.
- The second parameter is **messageType** (Mongoose.IDO.UserDefinedMessageType). This is the type identifier that displays in the log viewer.
• The third parameter is message (String). This is the text of the message to be logged.

```csharp
[IDOMethod(MethodFlags.None, "infobar")]
public int ValidateCreditCard(
    string cardNumber,
    DateTime expires,
    decimal amount,
    string infobar)
{
    int result = 0;

    // Call a Web service to validate the CC info:
    IDORuntime.LogUserMessage(
        "MyExtensionClass",
        UserDefinedMessageType.UserDefined0,
        "ValidateCreditCard called by user " + IDORuntime.Context.User►
        Name);

    return result;
}
```
Appendix A: Code samples

**Note:** These examples assume you are writing code in an IDO extension class that inherits from the Mongoose.IDO.ExtensionClassBase class.

### LoadCollection examples

This sample code demonstrates executing a LoadCollection for the SLTtJobtMatPosts IDO, bringing back the TransNum and TransSeq properties.

```csharp
LoadCollectionResponseData loadResponse;
loadResponse =     .Context.Commands.LoadCollection(
    "SLTtJobtMatPosts",
    "TransNum,TransSeq",
    [filter],
    .Empty, 0);
```

This sample demonstrates executing a LoadCollection for the IDO that the extension class belongs to (the current instance of the executing IDO).

```csharp
LoadCollectionResponseData loadResponse;
loadResponse =     .LoadCollection(
    "TransNum, EmpNum, EmpEmpType, EmpDept, DerJobPWcDept ",
    [filter],
    "Posted, LowLevel DESC, Job, Suffix, CloseJob, TransNum", 0);
```

This sample demonstrates one way to iterate through items in a LoadCollectionResponseData instance returned from a LoadCollection request, and how to access property values.

```csharp
empNum = null;

( index = 0; index <= responseData.Items.Count; index++)
{{ (!responseData[index, "EmpNum"].IsNull)
    { empNum = responseData[index, "EmpNum"].Value;
    }
}
```
UpdateCollection examples

Inserting a new item

This sample demonstrates inserting a new item.

```csharp
UpdateCollectionRequestData request = UpdateCollectionRequestData();
UpdateCollectionResponseData response;
IDOUpdateItem customerItem = IDOUpdateItem();
request.IDOName = "SLCustomers";
request.RefreshAfterUpdate = false;
customerItem.Action = UpdateAction.Insert;
customerItem.Properties.Add("CoNum", "C000100");
customerItem.Properties.Add("Name", "New Company");
customerItem.Properties.Add("CreditHold", 1);
request.Items.Add(customerItem);
response = .Context.Commands.UpdateCollection(request);
```

Updating an existing item

This sample demonstrates updating an existing item queried through a LoadCollection request.

```csharp
LoadCollectionResponseData loadResponse;
UpdateCollectionRequestData request;
UpdateCollectionResponseData response;
IDOUpdateItem customerItem = IDOUpdateItem();
loadResponse = .Context.Commands.LoadCollection("SLCustomers", "CoNum, Name, CreditHold", "CoNum = N'C000100'", "", 1);
if (loadResponse.Items.Count == 1)
{
    request.IDOName = "SLCustomers";
    request.RefreshAfterUpdate = false;
    customerItem.ItemID = loadResponse.Items[0].ItemID;
    customerItem.Action = UpdateAction.Update;
    customerItem.Properties.Add("CreditHold", 0);
    request.Items.Add(customerItem);
    response = .Context.Commands.UpdateCollection(request);
}
```
Invoke example

This sample demonstrates invoking the JobtClsLogErrorSp method on the SLJobtCls IDO, passing two parameters and retrieving the output value for the second parameter.

```csharp
InvokeResponseData invokeResponse = InvokeResponseData();
string message = ;
message = invokeResponse.Parameters[1].Value;
```