Infor VISUAL API Toolkit Development Guide

Publication Information

Release: Infor VISUAL API Toolkit

Publication date: December 6, 2017

Contents

About this guide

This guide provides an overview of the Infor VISUAL API Toolkit. It generally describes the documents, collections, transactions, and services available in the API Toolkit. It also describes the common methods used to interact with objects in the API Toolkit.

For more detailed information about the objects available in each class library in the API Toolkit, see the reference guide for the class library.

Intended audience

The intended audience of this guide is developers who are using the API Toolkit to extend the VISUAL solution.

Related documents

You can find the documents in the product documentation section of the Infor Xtreme Support portal, as described in "Contacting Infor."

Infor VISUAL API Toolkit Database Registration Guide

Infor VISUAL API Toolkit Financial Class Library Reference

Infor VISUAL API Toolkit Inventory Class Library Reference

Infor VISUAL API Toolkit Purchasing Class Library Reference

Infor VISUAL API Toolkit Sales Class Library Reference

Infor VISUAL API Toolkit Shared Class Library Reference

Infor VISUAL API Toolkit Shop Floor Class Library Reference

Infor VISUAL API Toolkit Trace Class Library Reference

Contacting Infor

If you have questions about Infor products, go to the Infor Xtreme Support portal at www.infor.com/inforxtreme.

If we update this document after the product release, we will post the new version on this Web site. We recommend that you check this Web site periodically for updated documentation.

If you have comments about Infor documentation, contact documentation@infor.com.

Support information

The API Toolkit will be updated regularly as more class members are added to each assembly, schema changes are made, and any reported issues are resolved. Infor Support cannot assist you with developing customized code using the API Toolkit. For assistance with customizations, contact Infor Consulting Services or your channel partner.

The functionality provided within the API Toolkit will not be extended beyond the standard functionality experienced in the VISUAL application itself. Enhancement requests with compelling business cases detailing how suggested alternatives are not viable will be evaluated and considered.

Only the public members that are described in the API toolkit documentation are supported for use. No other private or public members are supported.

Infor is not responsible for data incorrectly entered to the database through the use of the API Toolkit. Customers must establish a full test environment to ensure that data created by APIs functions in the same manner as data created through the VISUAL interface.

Chapter 1 Introduction

1

The Infor VISUAL API Toolkit is an Application Program Interface (API) designed to allow software developers the ability to interface with Infor VISUAL with their own custom code.

The toolkit's design consists of three layers: Data Management, Data Logic, and Business Logic. The data management layer provides portable data handling to either an SQL Server or Oracle database engine. The data logic layer provides document, transaction, and services handling on the server side. The business logic layer provides document, transaction, and services handling on the client side.

Compatibility

The toolkit is written in the Microsoft .NET framework. The toolkit is compatible with both version 4.0 & 4.5 of the .Net Framework.

Infor VISUAL 8.0.0 or higher is supported when using this toolkit. No other versions of VISUAL or other VISUAL products are compatible with the toolkit.

These files are required components of the toolkit:

- Microsoft.Scripting.dll
- Microsoft.Scripting.Metadata.dll

If you use Oracle, then these files are also required:

- Oracle.ManagedDataAccess.dll
- Oracle.ManagedDataAccessDTC.dll

Supported languages

These languages are supported for use with the toolkit:

- Visual Basic
- C#

While it is possible to use any .NET-aware programming language with the toolkit, other languages are not officially supported. For more information, see Support information.

Development environment

To interface with the API, developers can write their code in the C# or Visual Basic programming languages. All code samples in this document are provided in C#. Developers can interpolate to Visual Basic as necessary.

You must use a version of Microsoft Visual Studio that is capable of building to version 4 or higher of the Microsoft .NET framework. The minimum version of Visual Studio supported is version 2010 Service Pack 1. Microsoft offers several editions of Visual Studio such as Express, Professional, Premium, etc. All editions work with the API.

All DLLs in the toolkit are built as 32 bit (x86) assemblies. Any programs you develop to interface with these assemblies must also be built as 32 bit. 64 bit or “Any CPU” builds will not work. To configure Visual Studio for x86 builds, select the solution in the Solution Explorer, right click and select “Configuration Manager”. In the configuration manager dialog, create two x86 configurations: one for debug and one for release.

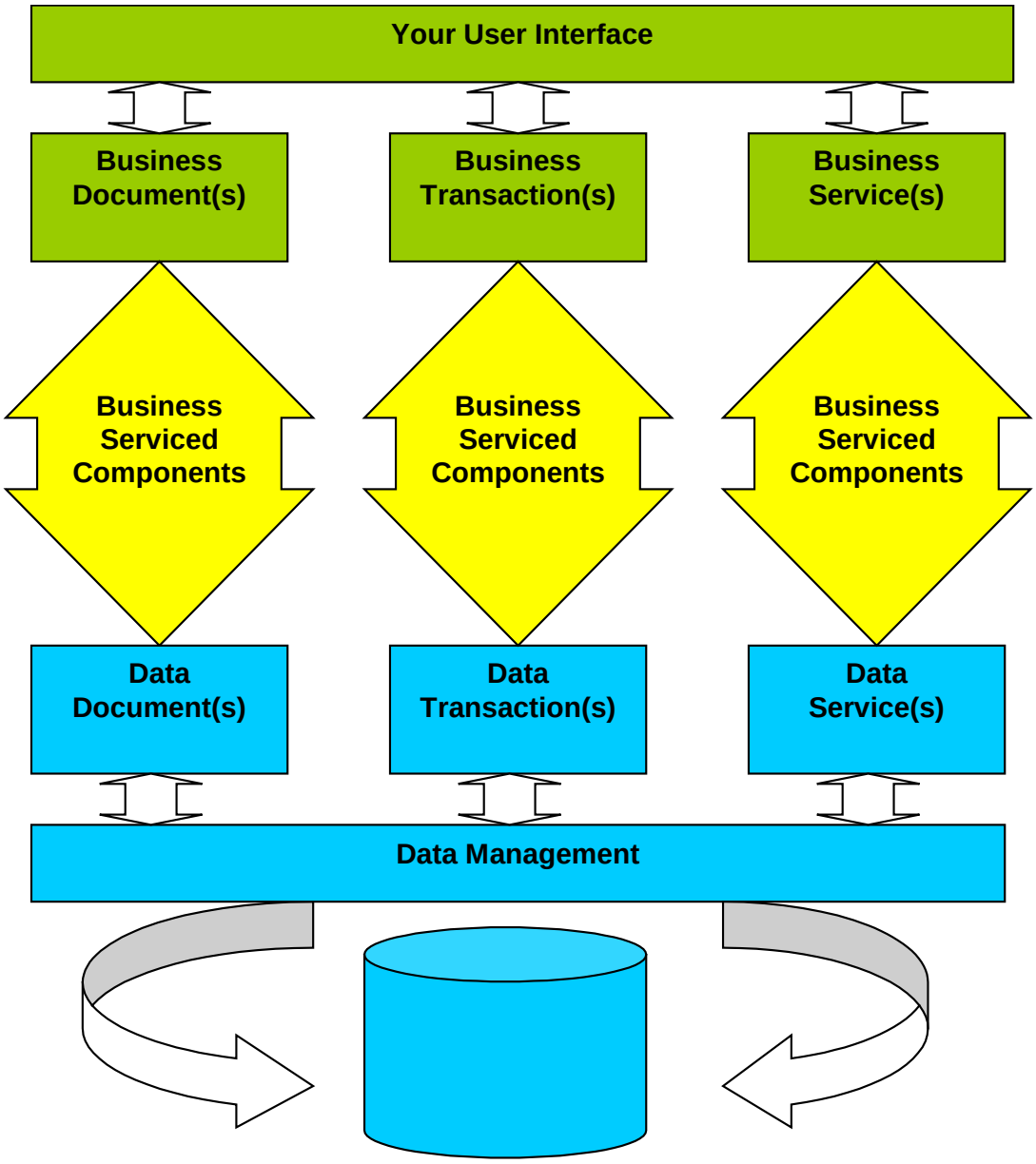
Class library reference documents

The class library reference documents list the documents, collections, transactions, and services used in each library. The documents provide sample code in C# and Visual Basic.

Toolkit design

The toolkit has four distinct layers: your user interface, the business logic layer, the data logic layer, and the data management layer.

This diagram shows the design of the toolkit:



Chapter 2 Coding with the Toolkit

2

This section describes the elements that are required in your custom code. The section also presents an overview of the business objects and calling methods you can use.

Calling conventions for business objects

This section describes how your user interface should call the business objects.

The **Data Management** layer of the API is stored in a single library called **LsaCore.DLL**.

LsaCore houses these namespaces:

- Lsa.Data
- Lsa.DataLogic
- Lsa.BusinessLogic

Common Helper methods of the API are stored in a single library called **LsaShared.DLL**.

LsaShared houses this namespace:

- Lsa.Shared

Since LsaCore's and LsaShared's classes serve as the base from which all VISUAL Business Logic Libraries are derived, you must reference all four in each of your programs. In addition, you must set up "references" in your project(s) to LsaCore.DLL and LsaShared.DLL. This C# code snippet illustrates integration of the LsaCore and LsaShared namespaces:

```
namespace TestInventory
{
            using System;
            using System.Diagnostics;
            using System.EnterpriseServices;
            using System.Text;
            using System.IO;
            using Lsa.Data;
```

```
using Lsa.BusinessLogic;
    using Lsa.DataLogic;
    using Lsa.Shared;
```

Note the references to the namespaces prefaced by “System”. These are also required, but you typically do not need to manually add these references. These references are built into Microsoft’s Visual Studio.

To add references to your project for Lsa.Data, Lsa.BusinessLogic, Lsa.DataLogic, and Lsa.Shared:

1. Locate the references header for your project in the Solution Explorer.
2. Right click the header and select “add reference.”
3. Browse for the locations of LsaCore.DLL and LsaShared.DLL and select LsaCore.DLL and LsaShared.DLL
4. Click OK.

The **Business Logic** layer of the toolkit is stored in multiple libraries, one library for each general business function group within VISUAL.

Library Name	Description
VmfgSales.dll	Customers, Customer Orders, Shipping, etc.
VmfgPurchasing.dll	Vendors, Purchase Orders, Purchase Order Receipts, etc.
VmfgInventory.dll	Parts, Inventory Adjustments and Transfers, etc.
VmfgShopFloor.dll	Labor Tickets, etc.
VmfgFinancials.dll	Payables, Receivables, etc.
VmfgTrace.dll	Trace and Trace Profile
VmfgShared.dll	Common functionality shared by all of the above libraries.

In addition to the mandatory references to **LsaCore, LsaShared** described above, add references to **VmfgShared** and one or more of the other Business Logic Libraries, depending on what type of business logic your program will perform. This table shows which libraries are needed based on business logic to be performed. When in doubt, add references to all **Vmfg** libraries.

Functionality	Required References
Sales	VmfgSales, VmfgInventory, VmfgShared.
Purchasing	VmfgPurchasing, VmfgInventory, VmfgShared.
Inventory	VmfgInventory, VmfgShared
Shop Floor	VmfgShopFloor, VmfgInventory, VmfgShared
Financials	VmfgFinancials, VmfgShared
All	VmfgShared

The API makes extensive use of Microsoft's ADO.NET. If you are not familiar with ADO.NET datasets, you should obtain a reference guide to ADO.NET and the use of the dataset object.

Many entry points, or methods, return ADO.NET Datasets. Datasets provide a table/row/column metaphor to manipulating data in a database. All of the methods defined in VISUAL business objects that return datasets, return them in a disconnected state. This means you may use the data and even change the content of the dataset, but you must return the dataset to the business object for storage and handling. You cannot update the database directly; i.e., all methods on the dataset that imply re-reading the data or writing to the database will return errors.

This C# snippet illustrates the calling syntax for developing a simple Part Maintenance routine that adds a part to the database.

```
Part part = new Part(instanceName);

DataRow dr = null;

part.Load("");

dr = part.NewPartRow(partID);

dr["DESCRIPTION"] = "DOTNET PART #1 Description";

dr["STOCK_UM"] = "EA";

dr["PRIMARY_WHS_ID"] = "MMC-MAIN";

dr["PRIMARY_LOC_ID"] = "AREA1";

part.Save();
```

In this example, we load a Part with null ID. This gives us a dataset with a blank schema for a Part data object.

Since we want to add a row to the Part data object, we initialize a new DataRow by calling the NewPartRow method of Part and passing the ID of the new Part. We then set the values of the various data row columns, and then we execute the Save method to place the data in the database.

Database connections

Before your customized code can interact with any of the API Toolkit's business logic libraries, you must first have a valid connection to a VISUAL Enterprise database. All database connectivity information is stored in an XML file named DATABASE.CONFIG. You can store an unlimited number of licensed VISUAL Enterprise databases in the configuration file.

Use the Infor VISUAL API Toolkit Database Registration applet to register your VISUAL database. The applet is located in the same directory as your other VISUAL executables. See the *Infor VISUAL API Toolkit Database Registration Guide.*

The toolkit supports only one Instance Group, and the Instance Group name must be set to "_default" as in the example. If you use the applet to register databases, the instance name is equal to the database name. If you prefer to use a different instance name, you can change the instance

name in the DATABASE.CONFIG file. In the sample, the instance name is “VMFG” and the database name is “VMFG”. The user name for all databases should be set to “SYSADM”, the default database owner for all VISUAL databases. Also note that the SYSADM password is encrypted when stored in the configuration file. It will be decrypted by the API data layer at connection time.

The DATABASE.CONFIG file created by the VmfgConfigForms.exe applet also stores the path to the executable directory. This path is used if your database uses average costing or if your database generates BODs for integration to other products.

To execute a connection to the database, you execute the **OpenLocal** method of **Dbms**. Since **Dbms** is a static object, you do not create (instantiate) it. You execute its methods directly.

For example, to connect to the Sql Server database using the configuration in the sample, you would call **OpenLocal** with these parameters:

```
Dbms.OpenLocal(“VMFG”, “SYSADM”, “the sysadm password”)
```

You should open the database connection early in your program’s execution, and hold on to the connection until the program terminates.

Single sign-on

The API Toolkit supports single sign-on. In single sign-on, Windows login credentials are used to connect to the VISUAL database. Before you can use single sign-on with the API toolkit, complete these tasks:

- Enable single sign-on in VISUAL.
- Add Windows users as VISUAL users.

After you complete these tasks, you can execute a connection to the database by using the **OpenLocalSSO** method of **Dbms**. This method requires that the Windows user information be passed in as parameters. The user information can be obtained using the **Lsa.Shared.GetSingleSignOnData** class. Once instantiated, this class will collect the required data for the Windows user that is currently logged in. This data can then be passed to the **OpenLocalSSO** method to execute the database connection.

For example, you would use this code to connect to the database using Single Sign On in the API:

```
Lsa.Shared.GetSingleSignOnData sso = new Lsa.Shared.GetSingleSignOnData();


string domain = sso.DomainName;

string domainSID = sso.DomainSID;

string userName = sso.UserName;

string userSID = sso.UserSID;


Dbms.OpenLocalSSO(“VMFG”, username, userSID, domain, domainSID)
```

Saving your code

In order for your code to function properly, your code, the API DLLs, and the DATABASE.CONFIG file must be located in the same directory. If you develop code using Open Text Team Developer, an additional copy of the DATABASE.CONFIG file must also be located in the Team Developer development folder.

Managing objects

Each type of object in the database is represented in one of four types of objects in the business logic libraries.

The database uses these types of objects:

Documents

A header and one or more subordinate records, which may have their own subordinate records. Example: A Customer Order.

Collections

Multiple rows of the same type of record generally having no subordinate records. Collections are a subset of Documents (i.e. there is no separate Collections base class). Example: Product Codes.

Transactions

A typically audited change to the database that usually affects multiple tables and often multiple data spaces. Example: A Labor Ticket.

Services

Information is returned to the caller, with no changes to the database having occurred. Example: A Unit of Measure Conversion.

Each of the objects is managed in a different way. Documents are handled one at a time. Collections typically work on all rows of a table at once. Transactions cause multiple changes and/or additions to the database.

Generally, all business objects have a Load command and a Save command. These two methods are used to access and modify the data. Transactions and services are available by calling the Prepare command. Documents and Collections are available by calling the Load command. One difference between a Document and a Collection is the signature of the Load command. Documents require you to specify a primary key. Collections return the entire result set. Services are initiated by calling Prepare, just as Transactions are. The Execute command (rather than Save as in Transactions) execute them, and the result set returned to the caller contains the requested information. This table shows all standard entry points exposed by the business logic libraries, which object types they apply to, and a brief description.

Method	Used By	Description / Comments
Load	Documents, Collections	Load retrieves an existing document or collection data from the database. For documents, specify the primary key of the document to process. If you pass a null or nonexistent key value to Load, a blank schema is returned. Use this blank schema as the starting point for inserting a new document. For collections, keys are optional. Providing no key will cause the loading of all rows of the table. If a key is provided, only the matching single row is returned.
Save	Documents, Collections, Transactions	Save provides the entry point designed to record changes to the database. All data changes are requested through this method. The call to save takes no arguments.
Prepare	Transactions, Services	Prepare returns an empty dataset corresponding to the particular transaction or service being used. The definition of this dataset is determined by the data object. Refer to the written documentation for this definition as there is no way to obtain this information via a browsing tool. Your program will populate a row or rows in this dataset, which in turn will be processed when the transaction is saved or the service is executed.
Execute	Services	Execute causes the requested service to be run. It is conceptually similar to Save; however, services never update the database. Services return the requested information.
Exists	Documents, Collections	Exists provides a way of determining if a row with the provided logical key exists in the database.
Find	Documents, Collections	Find is conceptually similar to Load. Unlike Load, which loads the entire data hierarchy of a document, Find only loads the top level table of the hierarchy.
Browse	Documents	Browse is method of retrieving a set of top level rows. Ideal for providing lists for the user. It returns a dataset containing a single table having the same name as the document's top level table. Similar to Find, except it returns multiple rows. All Browses have two modes, parameter-less (returning all rows) and limited to a specified number of rows.

Method	Used By	Description / Comments
New…Row	Documents, Transactions, Services	Adds a new row to an in memory dataset's data table that was previously initialized via a Load or Prepare.

General query

Because there may be times where your application needs to obtain data from the database for which there is no appropriate entry point available through the existing business logic library methods (i.e. Load, Browse, Find, etc.), a General Query interface is provided through the **VmfgShared** library. It is not possible to perform updates to the data through this interface.

General Query will return an ADO.NET Data Table containing all of the resultant rows from the query.

This code snippet illustrates usage of General Query:

```
GeneralQuery query = null;

query = new GeneralQuery(databaseInstanceName);

query.Prepare("INVENTORY", "select QTY from VMFG.PART_LOCATION where PART_ID
= ?  and WAREHOUSE_ID = 'MMC-MAIN'  and LOCATION_ID = 'AREA1' ");

query.Parameters[0] = partID;

query.Execute();


foreach (DataRow tempRow in query.Tables["INVENTORY"].Rows) {

     onHandQty += tempRow.ToDecimal("QTY");

}

query.Dispose();
```

An instance of the GeneralQuery class is initialized and set to the variable named query.

The Prepare method of query is called, passing a table name, and a string representing the query to be executed. Note the table name in the first parameter is the name of the results table, and can be set to any value you desire.

The query's parameters are set, in this case the part ID is the only parameter.

The query is executed by executing query's Execute method.

A foreach loop is entered for the purposes of processing each row of the results table.

Finally, the Dispose method is called to free all resources associated with this query. This is mandatory, since only one GeneralQuery may be active at any given time.

Sample solutions

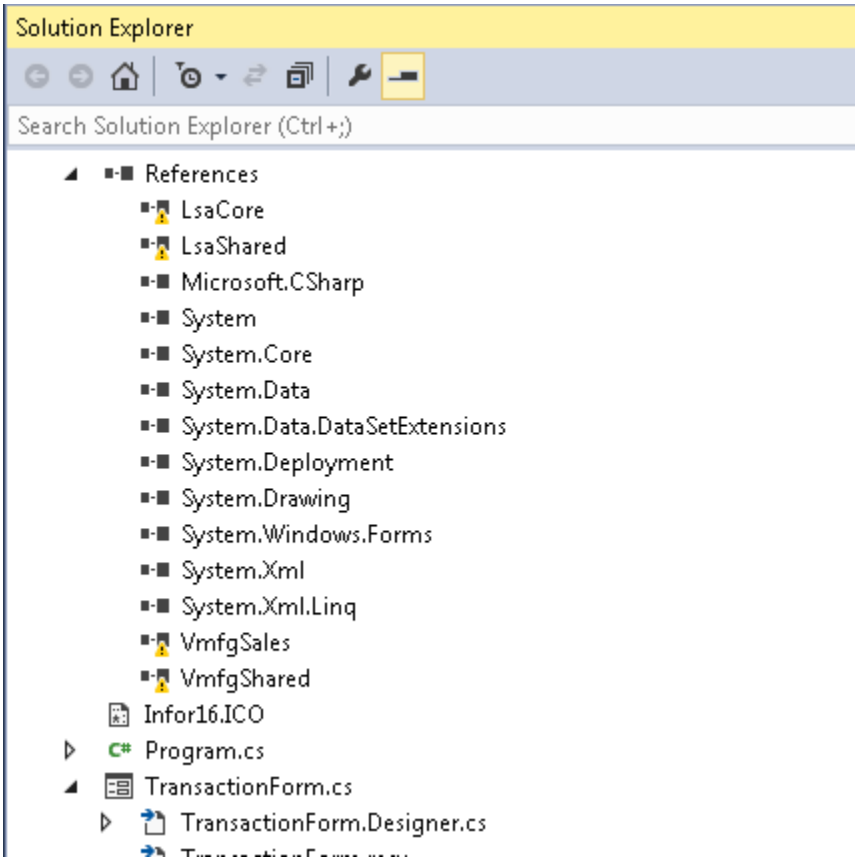
Most samples are provided in both C# and Visual Basic. You can find the samples in the VMFGSAMPLESAPITOOLKIT.zip file located in the patch release download. You can find the patch release download in the Downloads section of Infor Xtreme.

Note: By default, the API toolkit DLLs are located in the same folder as your VISUAL executables.

To use the sample code in a test environment, configure a valid connection to a VISUAL database using the configuration applet (VmfgConfigForms.EXE). You can use the applet to configure and test a database connection. We recommend you configure a connection before using the samples.

In order for the samples to function properly, the samples, the DLLs, and the DATABASE.CONFIG file must be located in the same directory. If you use Open Text Team Developer as the user interface with the API toolkit, you must have an additional copy of the DATABASE.CONFIG in the Team Developer development folder.

The first time you open any of the sample solutions, you will need to reset the project references to point to the exact location of the APIs on your system. If you open the Solution Explorer, select the project, then expand the References node, you should see a list similar to this:



Note the four entries with the warning icon next to them. These will have to be reapplied. For example, for LsaCore, right click LsaCore, then click **Remove**. Then right click References, and select **Add Reference…**. Browse to the location of “LsaCore.DLL” and select it to be added.

Visual Studio 2013 samples

Samples are provided in C# and Visual Basic. If you are using a version of Visual Studio newer than 2013, you may need to upgrade your samples to the newer Visual Studio format. You are informed of the need to upgrade the sample solution the first time you load it in Visual Studio. Follow the upgrade wizard, accepting all defaults, and your code will be converted and ready to use in the upgraded Visual Studio format.

These samples are provided in the Sales folder:

- **TestDocuments**. Use this sample to review the use of toolkit Business Documents. With this sample, users can create customers and customer orders. Note that the sample shows a minimal number of columns being populated. Also note that there is a one-to-one relationship between the tables and columns in the Business Document Dataset and the actual database schema. For example, to set the desired ship date of an order, you would use the column name “DESIRED_SHIP_DATE” in your program, since this is the name in the database.
- **TestTransactions**. Use this sample to review the use of toolkit Business Transactions. With this sample, users can ship a customer order and receive a return of a customer order. Unlike Business Documents where the schema in the program matches the database schema, the schema for the Transaction input must be obtained from the written documentation. Please refer to the class library reference documents. These documents list all supported Transactions in a particular class library, their columns, and whether the columns are required or optional.

These samples are provided in the ShopFloor folder:

- **LaborEntryApp**. Use this sample to review using the API Toolkit in conjunction with a web interface to create a labor ticket. **Note:** This sample is provided to prove the concept that a Web UI can be used with the toolkit. Infor cannot assist you with developing a Web UI. For more information, see Supported languages.
- **TestWorkOrders**. Use this sample to review a more complex use of the API toolkit. With this sample, users can add an operation and a leg to a work order.

Visual Studio 2015 sample

The VmfgAddIn sample shows how to use the APIs to load data from VISUAL into a Microsoft Excel spreadsheet and how to add data from the spreadsheet into VISUAL. This is intended for the loading of static data tables not requiring subordinate table data. This sample was created using Visual Studio 2015 and can interact with current versions of Excel and Microsoft Office.

Team Developer samples

Note: Infor does not officially support the use of Team Developer with the toolkit. The samples are provided to prove the concept that any .NET-aware language can be used with the toolkit. For more information, see Supported languages.

These samples are provided in Team Developer 6.2 and 7.0:

- **Test API**. This sample is the TestDocuments sample written in Team Developer.
- **Customer Orders.** This sample shows how to use the APIs to create an application with a similar look and feel as VISUAL applications. It includes a main form for entering in the customer document information such as entity, site, currency IDs, as well as a sold to address and customer contact information. A sales order form is available from the toolbar menu that allows users to create customer orders.

Chapter 3 Documents

3

Access a document by calling Load on the object that manages the document. Load will accept the logical key of the document and will return a dataset containing the entire data hierarchy for the matching logical key. If a document matching the provided key is not found, a dataset is returned having no tables or rows, but containing the complete logical definition (schema) for the data hierarchy.

The dataset returned is an ADO.NET dataset, which is essentially an in-memory database containing all of the elements of a traditional database such as tables, rows, and columns.

Standard entry methods

You can use these standard entry methods with documents:

- Load
- Save
- Exists
- Find
- Browse
- New…Row

Sample

This sample code loads a part by part ID. If the part ID is not in the database, then the part is added to the database. If the part does exist, then the information for the part in the database is updated.

In the sample code, txtPartID is a screen-hosted variable.

The sample uses the Load, New…Row, and Save methods.

```
Part part = new Part(databaseInstanceName);

DataRow dr;
```

```
part.Load(txtPartID);
if (part.Rows.Count = 0) {
     dr = part.NewPartRow(txtPartID);
}
else {
     dr = part.Row;
}

// set the part row values here

part.Save();
```

This loads the part identified by the txtPartID screen variable.

If this part exists, part.Rows.Count will be greater than 0. The part is returned along with several sub-tables, such as the PART_ALIAS table, the PART_SUBSTITUTE table, the PART_CROSS_SELLING table, and other related tables.

If the part does not exist (i.e. the **part.Rows.Count** equals 0), then a call to **NewPartRow** is used to add a new part to the database.

Regardless of whether you are adding a new part or updating an existing one, you finalize the document by executing **part.Save().**

To delete a row, you must execute the Delete method on the data row prior to executing Save. For example:

```
dr.Delete();
```

Binary data

Many VISUAL documents allow you to store long text data that coincides with that document. There is virtually no limit to the length of this data. For example, with Customer Orders, you can store long text at both the header and line item levels. This data is stored in the database in binary format, thus it is not human-readable as stored. In order to store human-readable text in these binary columns, your program must first convert this data to a Unicode byte array.

We will be expanding the previous example. This code snippet illustrates how you would convert your long text prior to saving it to the database:

```
dr = part.NewPartBinaryRow(partID, "D");
           dr["BITS"] = System.Text.Encoding.Unicode.GetBytes("LONG PART BINARY TEXT FOR
" + partID);
```

Note the second parameter to “NewPartBinaryRow()”. It is set to “D” to indicate “Data”. Although the VISUAL supports other types, the Toolkit only supports type “D”.

Your program treats the text to be inserted as a string. A system function is called to convert this string to a byte array. The value of the byte array data column (“BITS”) is set to this converted value.

Auto numbering of documents:

VISUAL uses the concept of “auto numbering” the IDs of many types of documents. In the standard VISUAL user interface, the user supplies as much information on the window as needed to save the document and leaves the ID field blank. When save is executed, the next available ID in the series for that particular document type is assigned.

The toolkit also supports auto numbering. Because ADO.NET does not allow a null primary key, you must use a placeholder ID to prompt auto-numbering. Any ID supplied in the format of “<n>” (where n is any integer value) is treated as a request to auto number. When a document is auto numbered, the toolkit posts the newly created number into the data row after the document is saved. This example illustrates adding a new Customer Order using auto-numbering and retrieving the newly created number after the save:

```
CustomerOrder custOrder = new  CustomerOrder(databaseInstanceName);

custOrder.Load(“”);

DataRow drHeader = custOrder.NewOrderRow(“<1>”);

//set drHeader columns to valid values here

custOrder.Save();

//obtain the auto generated number

string newOrderID = custOrder.Row.ToString (“ID”)
```

Chapter 4 Collections

4

Since Collections are similar to Documents, this section only describes the differences between Collections and Documents.

Collections almost never have transactions and are not automatically numbered. In some cases, Collections may even be read-only. The methods defined for a Collection reflect their design.

The most notable difference in the way Collections work is the Load method. When you Load a Collection, you can load all of the rows (the entire table), or just a specific row. Since most collections are designed to have a somewhat limited number of rows, and usually you would like to present all of these rows to the user in some sort of list or data grid, you typically load the entire collection at once. A deciding factor in whether something becomes a Collection in the business objects is how it is typically maintained in a user interface. If it is common to work on multiple rows at a time, the object will most likely be a Collection.

Standard entry methods

You can use these standard entry methods with collections:

- Load
- Save
- Exists
- Find

Sample

This sample code loads the entire contents of the Product Code data collection.

The sample uses the Load method.

```
ProductCodes productCodes = new ProductCodes(databaseInstanceName);

ProductCode.Load();
```

```
foreach(DataRow dataRow in productCodes.Rows) {
     //do something with the data
     //in this example we are just writing each code to the screen
     Debug.WriteLine(dataRow[“CODE”]);
}
```

At this point, the dataset has all of the product codes in the table. You would most likely present them in the user interface in a grid.

Chapter 5 Transactions

5

Transactions are defined by business objects. If a business object possesses the Prepare and Save methods, it has transactions.

You cannot identify supported transactions by examining the VISUAL software. Definitions of all transactions must be obtained from the reference documents. Transactions belong to particular classes in each library. At this time, you can find descriptions of transaction in these reference documents:

- *Infor VISUAL API Toolkit Inventory Class Library Reference*
- *Infor VISUAL API Toolkit Purchasing Class Library Reference*
- *Infor VISUAL API Toolkit Sales Class Library Reference*
- *Infor VISUAL API Toolkit Shop Floor Class Library Reference*

You use Prepare to obtain a dataset that contains a table or tables with column definitions pertaining to a particular type of transaction. The transactions available to you in the client program are strictly controlled by the design of each business object.

When the prepared dataset is returned, it is empty. After you add rows, you use the Save method to process the rows.

Standard entry methods

You can use these standard entry methods with transactions:

- Prepare
- Save
- New…Row

Sample

This sample code adds Adjust In inventory transactions.

This sample uses the Prepare, New…Row, and Save methods, where multiple transactions are generated, and each transaction is saved individually.

```
InventoryTransaction inv = new InventoryTransaction(instancename);

Inv.Prepare();

foreach (GridRow gridrow in myDataGrid)

{

DataRow dataRow = inv.NewInputRow();

dataRow[“ENTRY_NO”] = 1;

dataRow[“TRANSACTION_TYPE”] = “ADJUST_IN”;

dataRow[“QTY”] = gridRowQty;

dataRow[“PART_ID”] = gridRowPartID;

dataRow[“TO_WAREHOUSE_ID”] = gridRowWarehouseID;

dataRow[“TO_LOCATION_ID”] = gridRowLocation;

inv.Save();

dataRow.Tables[0].Clear();

}
```

Note the setting of the column “ENTRY_NO” to 1 in the example. For most transactions, ENTRY_NO is the primary key to the transaction’s data table. Since it is possible to submit multiple transaction rows to the Save method, each data row must have a unique primary key. For example, if you were to send two rows to be saved, the first row would have ENTRY_NO = 1 and the second row would have ENTRY_NO = 2. It is your program’s responsibility to set unique values for ENTRY_NO. Since some transactions may consist of multiple data tables, the ENTRY_NO also provides a convenient way to link top level tables with their subordinates.

In the above example, since each row in the grid is producing an inventory transaction via a loop, the input data row must be cleared after each save. Otherwise, the number of rows in the input table will continue to increase on each pass through the loop, resulting in too many inventory transactions being created when Save() is executed.

It is possible to pass multiple transaction rows to a single execution of Save. However, passing too many rows simultaneously may cause performance problems. Each Save is treated as a logic unit of work. Thus, the entire “batch” is treated as one transaction on the database engine. A transaction consisting of a large number of input rows could tie up the database manager’s system resources and potentially cause other user’s transactions to wait. If you decide to process multiple rows simultaneously, you may need to experiment on your system to determine the optimal number of rows per batch.

This example shows another method for inputting multiple rows into a transaction input table and then executing a single save. **Note:** The Save() method is executed only once, outside of the loop and there is no clearing of the input table. In this example, the input table has one row for each grid row in the loop.

```
InventoryTransaction inv = new InventoryTransaction(instancename);
Inv.Prepare();
int entryNo = 0;
foreach (GridRow gridrow in myDataGrid)
{
DataRow dataRow = inv.NewInputRow();
dataRow[“ENTRY_NO”] = ++entryNo;
dataRow[“TRANSACTION_TYPE”] = “ADJUST_IN”;
dataRow[“QTY”] = gridRowQty;
dataRow[“PART_ID”] = gridRowPartID;
dataRow[“TO_WAREHOUSE_ID”] = gridRowWarehouseID;
dataRow[“TO_LOCATION_ID”] = gridRowLocation;
}
inv.Save();
```

Chapter 6 Services

6

Services are defined by business objects. If a business object possesses both the Prepare and Execute methods, it is capable of executing services.

The behavior of Services is completely defined by the business logic implementation. They are provided as a tool for retrieving useful information from the database that otherwise may be difficult or cumbersome to obtain using straight calls to the Data Manager, since they often require fetching data from numerous sources and/or performing calculations. Services are similar to Transactions in that they both are initialized via the Prepare method. Services, unlike Transactions, are executed by the Execute method rather than the Save method, and they **do not** perform any database updates.

You cannot identify supported services by examining the VISUAL software. Definitions of all supported services must be obtained from the reference guides. At this time, all services are documented in the *Infor VISUAL API Shared Class Library Reference*.

You use Prepare to obtain a dataset with column definitions pertaining to the service you are executing. The services available to you in the client program are strictly controlled by the design of each business object.

The dataset returned after the Prepare method has two tables: a header table and a results table. Populate the header table with the parameters to pass to the service for execution. The results table is populated by the execution of the service and returned to you. Using the question and answer metaphor, the header table contains the “question” and the results table contains the “answer.”

Standard entry methods

You can use these standard entry methods with services:

- Prepare
- Execute
- New…Row

Sample

This sample converts a quantity measured in one unit to a quantity measured in a second unit. The Prepare, New…Row, and Execute methods are used.

```
UnitOfMeasureConversion um = new
UnitofMeasureConversion(databaseInstanceName);

um.Prepare();

DataRow inputRow = um.NewInputRow(1);

inputRow["PART_ID"] = partID;

inputRow["FROM_UM"] = dataRow["SELLING_UM"];

inputRow["TO_UM"] = dataRow["VMFG:PART:STOCK_UM"];

inputRow["FROM_QTY"] = dataRow["USER_ORDER_QTY"];

um.Execute();

DataTable conversions =(DataTable)um.Tables["CONVERT_UM_RESULT"];


DataRow conversionRow = (DataRow)conversions.Rows[0];

orderQty = conversionRow.ToDecimal("TO_QTY");
```

An Instance of UnitofMeasure is initialized and stored in a variable named “um”. Prepare() is called to access the dataset containing the header table. A new row is inserted into the header table via a call to NewInputRow(). The new row’s column values are populated.

The service is Executed.

The results table is extracted from the um dataset, and the first row from this table is accessed. The variable orderQty is set to a value from the results row.