

*QuickBooks® Point of Sale SDK*

# *Programmer's Guide*

Version 2.0

QBPOS SDK version 2.0, released April 2007. (c) 2007 Intuit Inc. All rights reserved.

QBPOS, QBPOS Point of Sale, and Intuit are registered trademarks of Intuit Inc. All other trademarks are the property of their respective owners and should be treated as such.

Acknowledgement: This product includes software developed by the Apache Software Foundation (<<http://www.apache.org>>) (c) 1999-2004 The Apache Software Foundation. All rights reserved.

Intuit Inc.  
P.O. Box 7850  
Mountain View, CA 94039-7850

For more information about the QBPOS POS SDK and the SDK documentation, visit <http://developer.intuit.com>.

# CONTENTS

## About This Guide

Who Should Read This Guide? . . . . .	ix
Before You Begin Programming . . . . .	ix
Common Abbreviations and Acronyms . . . . .	ix
What's New in This Guide . . . . .	ix

## Chapter 1: Overview

What is the QBPOS SDK? . . . . .	1
Architecture: How Applications Integrate with QBPOS . . . . .	1
Supported Configurations . . . . .	2
Compatibility and QBPOS Version Issues . . . . .	3
Requirements . . . . .	4
Integrating Applications with Both QBPOS and QuickBooks . . . . .	4
Multi-User vs Single User . . . . .	4
The QBPOS UI and SDK Applications: Behavioral Notes . . . . .	4
QBPOS Objects and the QBPOS Transaction Flow . . . . .	5
QBPOS Setup . . . . .	5
QBPOS Workflow . . . . .	6
Two Ways to Access QBPOS Via the SDK . . . . .	7
A Note About Programming with qbposFC. . . . .	8
Which QBPOS Edition Supports Which Features? . . . . .	8
Can My Application Access QB POS Merchant Services? . . . . .	8

## Chapter 2: Communicating with QBPOS

Call Sequence for QBPOS Communication . . . . .	9
My Language isn't VB: Where Do I Find Communication Syntax Information? . . . . .	12
How End Users Authorize Application-QBPOS Communication . . . . .	12
What Happens When a QBPOS Company has Login Protection? . . . . .	13
Security and SDK Applications . . . . .	14
Can I Access a QBPOS Company from a Remote Computer? . . . . .	14
Request Processor API and qbposFC Object Methods . . . . .	14
Communicating with QBPOS from Web Services . . . . .	16

## Chapter 3: Running, Deploying, and Distributing Your Application

Required Files . . . . .	19
Distributing Your Application . . . . .	19
Using the Stand-Alone Installers . . . . .	20
Using the Merge Modules . . . . .	20

## Chapter 4: Using the qbposFC Convenience Library

Understanding the Context and Flow of qbposFC Objects . . . . .	23
Objects, Objects Everywhere: Where Do I Start? . . . . .	23
Which Objects Do I Need to Create a Request? . . . . .	24
How Do I Use the OSR to Construct the Request? . . . . .	25
Other Useful IMessageSetRequest Methods . . . . .	27
Which Objects Do I Need to Process a Response? . . . . .	28
Getting Data from the Ret Object . . . . .	29
Objects and Methods Used in Processing Response Data . . . . .	30

## Chapter 5: Building Requests and Processing Responses

Building a qbposXML Request . . . . .	33
Building a Request Message Set: Pseudocode . . . . .	34
Building a Request Message Set: Sample Code . . . . .	34
Building a Request using qbposFC . . . . .	35
IMessageSetRequest: the Central Object in Sending QBPOS Requests. . . . .	35
Understanding the Request Message Set Structure in Detail . . . . .	37
How Do I build a Fully Constructed Message Set? . . . . .	39
Processing a qbposXML Response . . . . .	41
Processing a Response Message Set: Pseudocode . . . . .	41
Processing a Response Message Set: Sample Code. . . . .	42
Processing a Response Using qbposFC . . . . .	43
IMessageSetResponse: the Central Object in Processing Responses . . . . .	43
How Do I Get Data from the Response Message Set? . . . . .	44

## Chapter 6: Macros, Data Extensions, Object Deletion

What are Macros and How Do You Use Them? . . . . .	49
What are Data Extensions and How Do I Add Them? . . . . .	50
What are DataExt and DataExtDef? . . . . .	51
How Do I Read and Write Data to a Custom Field? . . . . .	51
How Do I Read and Write Data to Private Data Extensions? . . . . .	53
Why Aren't Data Extensions Returned in My Queries? . . . . .	53
How Do You Delete an Object? . . . . .	54
Why Can't I Delete (or Mod) Some QBPOS Objects? . . . . .	54

## Chapter 7: Creating Queries

About Query Requests . . . . .	55
Query Filter Groups and How They Work . . . . .	55
Specifying Multiple Filters . . . . .	57
Building a Query . . . . .	57
Sample qbposXML Query with Multiple Filters . . . . .	58
Sample qbposFC Query with Multiple Filters. . . . .	58
The Importance of Limiting the Number of Objects Returned . . . . .	59
Using Iterators to Walk Through Large Query Returns . . . . .	59

Miscellaneous Details on Query Filters. . . . .	61
Match Criterion for Names . . . . .	62
Ranges for Names . . . . .	62
Understanding the Time Modified Field . . . . .	62

## **Chapter 8: Moving Items Into and Out of Inventory: SalesOrders PurchaseOrders, Vouchers and SalesReceipts**

Process Overview . . . . .	63
Vouchers Update PurchaseOrder, SalesReceipts Update SalesOrder . . . . .	64
Creating Sales Orders . . . . .	64
Creating a SalesOrder . . . . .	64
Creating Purchase Orders . . . . .	69
Creating a PurchaseOrder. . . . .	70
Creating Receiving Vouchers . . . . .	73
The Receiving Voucher Request and the New Voucher Form . . . . .	74
Creating Return Vouchers . . . . .	77
Creating SalesReceipts . . . . .	78
Creating a SalesReceipt . . . . .	78

## **Chapter 9: Adding, Modifying, and Querying Time Entries**

About Time Entries in QBPOS and SDK Support of Them. . . . .	83
Adding a Time Entry in QBPOS SDK . . . . .	83
Adding a Time Entry in qbposXML . . . . .	84
Adding a Time Entry in qbposFC . . . . .	84
Modifying a Time Entry in QBPOS SDK . . . . .	85
Modifying a Time Entry in qbposXML . . . . .	85
Modifying a Time Entry in qbposFC . . . . .	85
Querying For Time Entry Records in QBPOS SDK . . . . .	86
Using qbposXML . . . . .	86
Using qbposFC . . . . .	86

## **Chapter 10: Setting and Modifying Sales Tax Preferences**

About Sales Tax Preferences in the QBPOS UI . . . . .	89
Tax Records: an SDK Concept. . . . .	91
How TaxCategories/TaxCodes are Applied in Transactions. . . . .	92
Creating/Editing a Tax Category (Location) in the UI. . . . .	92
Adding a TaxCategory via the SDK . . . . .	93
Adding a TaxCategory using qbposXML . . . . .	93
Adding a TaxCategory using qbposFC. . . . .	94
Modifying a TaxCategory using the SDK . . . . .	94
Querying for TaxCategories using the SDK . . . . .	94
Creating/Editing a Tax Code in the UI . . . . .	95
Adding a TaxCode via the SDK . . . . .	96
Adding a TaxCode using qbposXML . . . . .	97
Adding a TaxCode using qbposFC . . . . .	97

Modifying a TaxCode using the SDK . . . . .	97
Querying for TaxCodes using the SDK . . . . .	98
Setting TaxCode Rates in QBPOS . . . . .	98
Setting a Single Rate Tax (UI) . . . . .	98
Setting a Single Rate Tax (SDK) . . . . .	99
Setting a Price-Dependant Single Rate Tax (UI) . . . . .	100
Setting a Price-Dependant Single Rate Tax (SDK) . . . . .	101
Setting a Multi-Rate Tax (UI) . . . . .	102
Setting a Multi-Rate Tax (SDK) . . . . .	104

## Chapter 11: Using Units of measure (UOM)

About the QBPOS UOM Feature . . . . .	107
Multiple UOM Features in the QBPOS UI . . . . .	108
Specifying UOMs for Inventory Items (SDK) . . . . .	111
Specifying UOMs for an Item in qbposFC . . . . .	111
Specifying UOMs for an Item in qbposXML . . . . .	112

## Appendix A: Status Codes for qbposXML Responses

## Appendix B: Request Processor API Reference

BeginSession . . . . .	120
CloseConnection . . . . .	121
EndSession . . . . .	122
GetCurrentCompanyFileName . . . . .	123
MajorVersion . . . . .	124
MinorVersion . . . . .	125
OpenConnection . . . . .	126
POSServers . . . . .	127
POSVersions . . . . .	129
ProcessRequest . . . . .	130
QBPOSXMLVersionsForSession . . . . .	131
ReleaseLevel . . . . .	132
ReleaseNumber . . . . .	133

## Appendix C: qbposFC Language Reference

QBPOSSessionManager Object and Methods . . . . .	135
QBPOSSessionManager.BeginSession . . . . .	136
QBPOSSessionManager.CloseConnection . . . . .	137
QBPOSSessionManager.CreateMsgSetRequest . . . . .	138
QBPOSSessionManager.DoRequests . . . . .	139
QBPOSSessionManager.DoRequestsFromXMLString . . . . .	140
QBPOSSessionManager.EndSession . . . . .	141
QBPOSSessionManager.GetCurrentCompanyFileName . . . . .	142

QBPOSSessionManager.GetVersion . . . . .	143
QBPOSSessionManager.OpenConnection . . . . .	144
QBPOSSessionManager.POSServers . . . . .	145
QBPOSSessionManager.POSVersions . . . . .	146
QBPOSSessionManager.QBXMLVersionsForSession . . . . .	147
QBPOSSessionManager.ToMsgSetRequest . . . . .	148
QBPOSSessionManager.ToMsgSetResponse . . . . .	149





# ABOUT THIS GUIDE

This guide focuses on the structure and content of the request messages sent by your application to QBPOS, as well as the response messages returned by QBPOS. It also offers general information on how to create an application that integrates effectively with QBPOS.

## Who Should Read This Guide?

---

This guide is the starting point for all developers who are creating applications that integrate with QBPOS. It provides practical information on how to create request messages and interpret response messages using the QBPOS SDK.

## Before You Begin Programming

---

Before you start programming, make sure you are familiar with using QBPOS at the user interface (UI) level. Knowing how the UI works can save you quite a bit of time when you start using the SDK's application interface (API) to automate or augment QBPOS features. The QBPOS UI provides the best hints as to how these features work.

In addition, make sure you are familiar both with the concepts presented in this programming guide and also with the QBPOS requests and responses contained in the *Onscreen Reference* (OSR) for QBPOS. The OSR contains the syntax for each request and response message type for all of the SDK APIs. Finally, if you choose to write a web services application, you should know about web services, you should know about SOAP, and so forth.

## Common Abbreviations and Acronyms

---

Throughout this document, abbreviations and acronyms are used as shortcuts. The following is a list of the ones we commonly use:

- QuickBooks Point of Sale (QBPOS)
- QuickBooks Point of Sale Software Development Kit (QBPOS SDK, or simply, SDK)
- QuickBooks financial software (QBFS)
- User interface (UI)

## What's New in This Guide

---

This version of the guide provides information on new or changed features in the latest QBPOS release, QBPOS 6.0, including support for the following:

- Adding, modifying and querying for Time Entries

- Adding and modifying Tax Categories
- Adding and modifying Tax Codes
- Unit of measure support in the following: inventory items, sales receipts, sales orders, purchase orders, vouchers, inventory quantity adjustments, inventory cost adjustments, and transfer slips.

# CHAPTER 1

## OVERVIEW

This chapter provides an overview of the QBPOS Point of Sale (QBPOS) SDK.

### What is the QBPOS SDK?

---

The QBPOS SDK is a set of redistributable runtime and support files, programming documentation, software tools, and programming samples that enable you to write your own applications that add, modify, query, and delete data from the QBPOS application. It is programming language-independent, so you should be able to use it in the environment of your choice.

### Architecture: How Applications Integrate with QBPOS

---

Applications integrate with QBPOS via the qbposXML request processor. As shown in Figure 1-1 on page 2, an application communicates either directly with the request processor using the request processor API, or it uses the QBPOS Foundation Class Library (qbposFC).

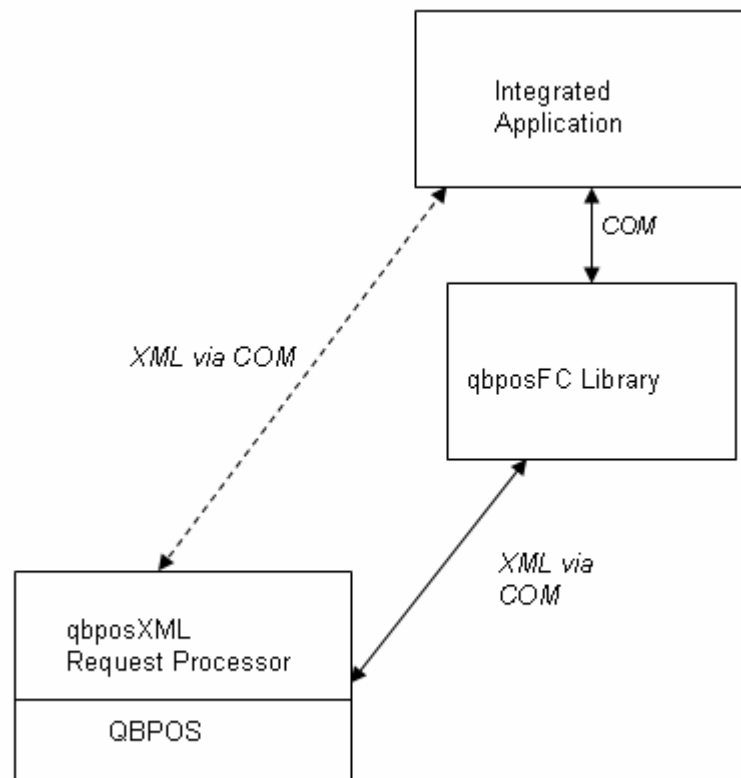


Figure 1-1 How Applications Integrate with QBPOS

As shown in the figure, if an application uses the request processor API, it uses XML to communicate with QBPOS. If an application uses the qbposFC library, which is a thin wrapper layer over the request processor, the application just uses objects and method calls to communicate with qbposF: qbposFC does the translation into XML for the application before communicating with the request processor. Similarly, when handling responses, qbposFC does the translation from XML.

## Supported Configurations

---

An integrated application can be on the same machine as QBPOS or it can be on a remote machine in a local network. However, as shown in Figure 1-2 on page 3, the QBPOS SDK runtime and support DLLs must be on the same machine as the integrated application.

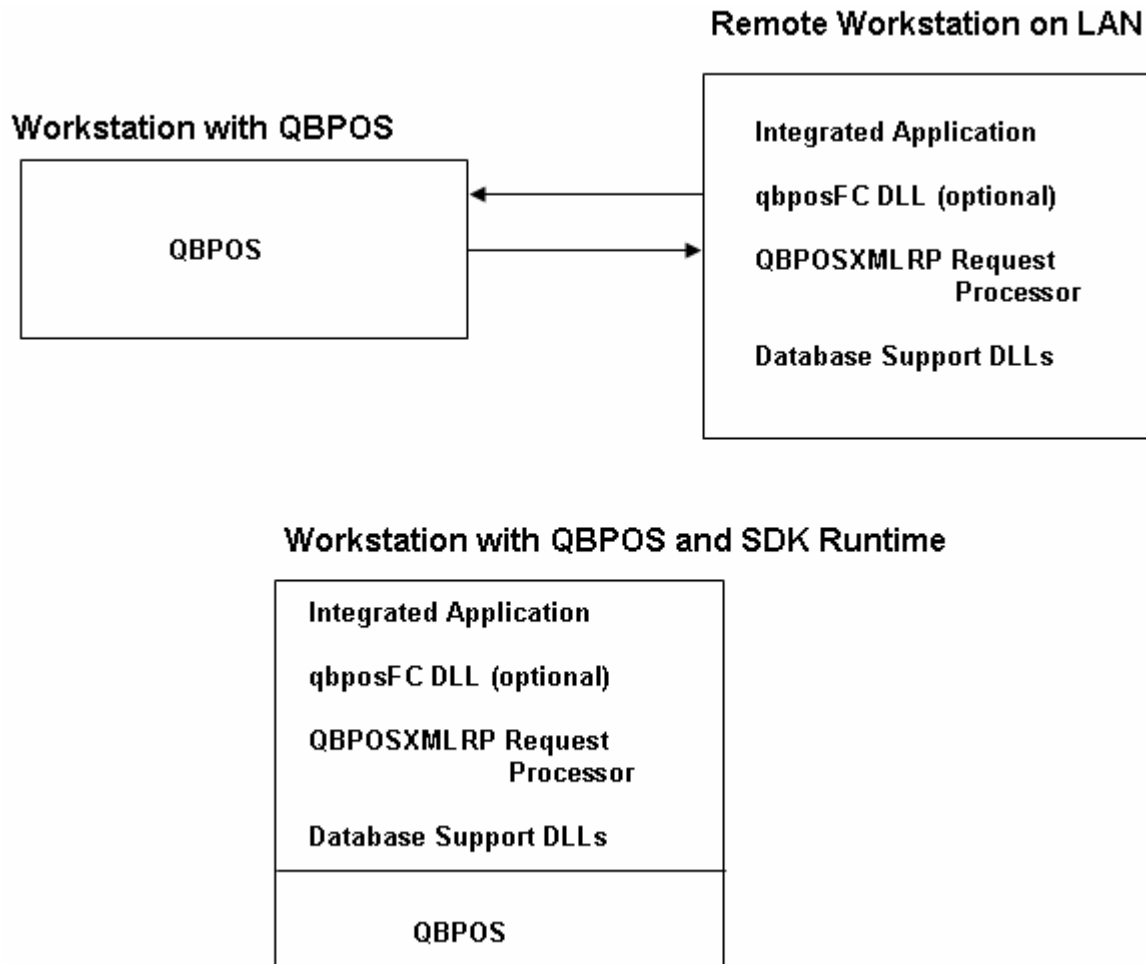


Figure 1-2 How Applications Access a QBPOS Company

As shown in the figure, an integrated application can reside on the same system as QBPOS, or it can reside on some remote system in a local area network. Notice that for this remote configuration, the QBPOS SDK runtime and support DLLs are required on the application machine, but not on the machine hosting QBPOS. (The runtime and support DLLs can be present on the QBPOS host, but aren't required.)

#### **NOTE**

QBPOS need not be running in order for your requests to be successful.

## Compatibility and QBPOS Version Issues

Applications designed for the current level of QBPOS can be expected to be forward compatible with future versions of QBPOS provided that an updated version of the QBPOS SDK runtime is installed locally. Although the QBPOS SDK runtime can *detect* newer

versions of QBPOS, applications cannot automatically *use* a newer version of QBPOS without an updated SDK runtime. The SDK runtime must be updated before an application can successfully connect to a newer version of the QBPOS database. It is expected that new versions of QBPOS will include the required SDK runtime.

There is another potential issue that your product documentation and online help should make clear to your end user. When a new version of QBPOS is installed, the QBPOS user has the opportunity to upgrade existing company data to the new version. This process creates a new company with the upgraded format required to be used in the new QBPOS version.

However, the old company with its old version is kept and this is separate from the new company. The end result is that your application can keep logging into the old company and writing data there, while the end user is using the new version of the company.

To avoid this, you can use the `POSVersions` method of the request processor to determine whether there are multiple versions of a given company and then require the user to select the version that the application should be logged into.

## Requirements

---

The QBPOS SDK requires QBPOS Point of Sale version 4.0 R5 or later.

## Integrating Applications with Both QBPOS and QuickBooks

---

An application can integrate with QuickBooks and with QBPOS at the same time, using the QB SDK and the QBPOS SDK. However, if you are concerned mainly with sharing customer data, you should consider whether your needs can be met using the built-in data sharing supported by QBPOS and the QBPOS SDK.

From the QBPOS user interface, the end user can specify this type of data sharing via the `Use With QuickBooks` checkbox in the company preferences and via the related account mappings in preferences. Using QBPOS with QuickBooks requires the end user to make certain mappings to QuickBooks accounts in QBPOS company preferences.

In the QBPOS SDK, if the end user has set up QBPOS to work with QuickBooks, an application can make use of this feature programmatically in requests supporting this feature by setting the `UseWithQuickBooks` element in the request.

## Multi-User vs Single User

---

If you are familiar with programming with the QBSDK, you may wonder about the multi-user/single user issue, which is important for QuickBooks. No need to worry about this in QBPOS. QBPOS is always multi-user.

## The QBPOS UI and SDK Applications: Behavioral Notes

---

The QBPOS SDK, unlike the QB SDK, does not work through the code of the main program. The advantage to this is that you don't need a QBPOS installation on the machine you're deploying the POS SDK application to. One consequence of this, however, is that when you have both QB POS and a QB POS SDK application on the same machine, to QBPOS that POS SDK application looks like another QBPOS workstation. This has the following impacts when the application communicates with QBPOS on that machine:

- The local QBPOS never refreshes the information updated by another workstation or your QBPOS SDK application with regard to a record that is open in the QBPOS UI, unless you're trying to edit the record.
- When switching the current record, the QBPOS UI tries to detect changes to the list and refresh the information.
- If a record is being edited in the QBPOS UI, and a QBPOS SDK application changes it in the background, an error is given in QBPOS when you try to save it in the QBPOS UI.

Another behavior to note about the QBPOS SDK is that you can do things you are prevented from doing within the QBPOS UI itself. For example, in the SDK you could invoke a `SaleReceiptAdd` multiple times against the same `SalesOrder`, in effect closing the `SalesOrder` many times, which you wouldn't be allowed to do from the UI. So some caution needs to be exercised.

## QBPOS Objects and the QBPOS Transaction Flow

---

When you begin to use the QBPOS SDK, you need to become familiar with the way QBPOS works and also to what extent you can use SDK functionality to accomplish a desired type of implementation. You need to become familiar with

- The typical QBPOS setup and workflow.
- The relationships between objects. For example, when you construct a new inventory item, you should assign it a department and one or more preferred vendors, which means that the required department and vendor objects should exist in the QBPOS company before the `ItemInventoryAdd` request is invoked. (Alternatively, the department or vendor could be added using macros in the same message set, but in a prior request within that message set.)
- The QBPOS settings that are used, but not set by the SDK. For example, Tax Codes and Price Levels, which are set through QBPOS preferences by the end user.

The remainder of this section provides more details.

### QBPOS Setup

---

A central feature of QBPOS is its item inventory control and reporting. In order for these features to be useful, the inventory items must be carefully grouped into categories called departments. (See the online Help for QBPOS for more details.) That is, the departments

are established first, and are assigned later to new inventory items at item creation time. Because much of the QBPOS functionality centers around inventory items, setting up the department structure is normally done first.

#### **NOTE**

Programmatically, the grouping of items under departments is carried out in the `ItemInventoryAdd` request where the new item is assigned a department.

Similarly, vendor information can be useful when creating inventory items. So vendors are usually set up after departments.

Once the departments and vendors are created, the inventory items are set up. At this point, when the items are established, the user can order merchandise (`PurchaseOrder`) and receive it into QBPOS using a voucher.

Finally, customers are set up, if these can be known in advance of a sale. (Customers can also be added at the time of sale or sale order.) Customer information is normally needed for `SalesOrders` and `SalesReceipts`. Notice that the QBPOS end user can specify in Preferences whether customer information is required or not for all receipts.

## QBPOS Workflow

---

Figure 1-3 on page 7 shows the typical workflow of transactions in QBPOS, along with the SDK requests used to carry out the workflow. The center of the workflow is the inventory system provided by the QBPOS application. The inventory system maintains the “documents” used in these activities and also maintains a historical record of movement of items into and out of inventory.

As shown in the figure, items move into the inventory via goods purchased from vendors. These items are received using Vouchers. Items move out of the inventory when customers purchase them, with the purchase (and item movement) recorded via `SalesReceipts`.

Another way for items to move in inventory, as shown in Figure 1-3 on page 7, is in multi-store configurations where items may be physically moved from one store to another. This movement is recorded in `TransferSlips`.

Finally, as shown in the figure, there may be a need to manually adjust the inventory system in the case of spoilage, errors, unforeseen vendor discounts, etc. These adjustments are made by the `InventoryQtyAdjustments` and `InventoryCostAdjustments`, respectively.

The SDK objects that are maintained as part of the historical record of the inventory system as mentioned above are Vouchers, `SalesReceipts`, `TransferSlips`, `InventoryQtyAdjustments`, and `InventoryCostAdjustments`. Notice that these objects do not have SDK Mod requests, as that would put at risk the historical transaction record. Notice that beginning with QBPOS 5.0 (and supported in QBPOS SDK 1.1), some of these documents can be marked as Held, meaning the transaction is not yet complete but is pending.



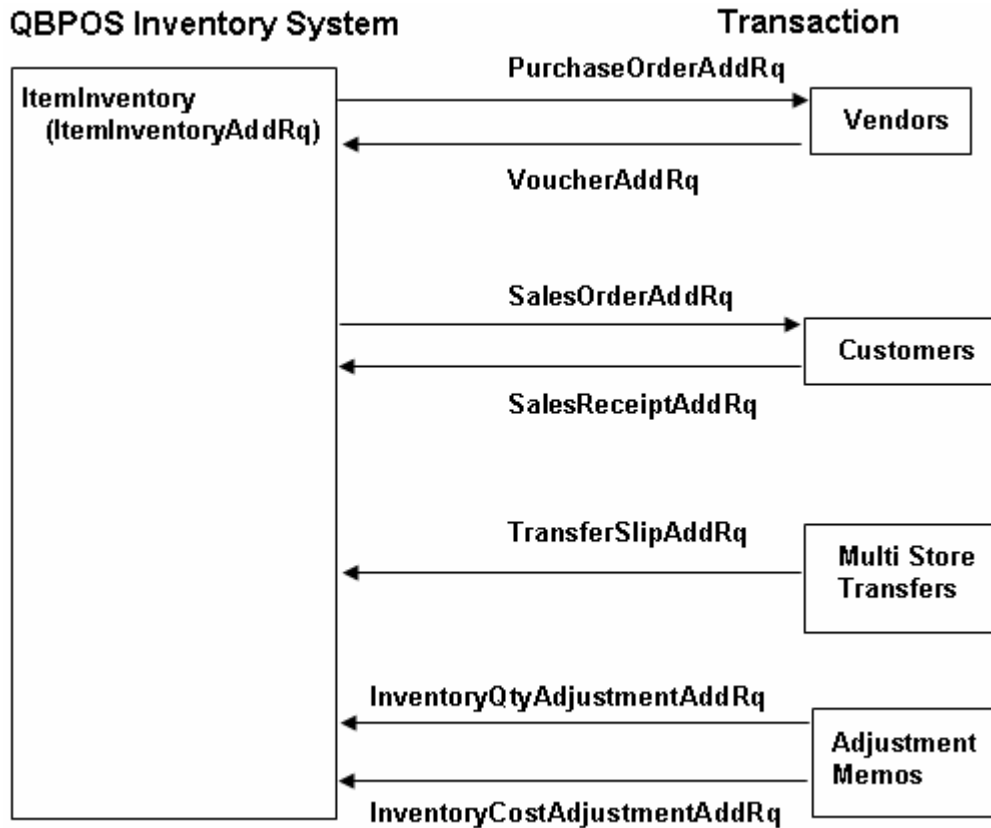


Figure 1-3 QBPOS Transaction Workflow

## Two Ways to Access QBPOS Via the SDK

There are two ways to use the SDK to access QBPOS. You can use the QBPOS request processor directly by building and sending qbposXML requests and handling the qbposXML responses. This requires you to build the XML and walk the XML response to get the values you are interested in. We strongly recommend using an XML-aware API (such as MSXML) to build and parse XML exchanged with QBPOS to avoid problems with XML reserved characters.

If you are not comfortable working with XML, an alternative way to access QBPOS via SDK is the QBPOS Foundation Class library. This library provides a wrapper layer over the same QBPOS request processor used in the XML method described above. There is no significant performance difference between either of these methods, only the programmer's choice of working with XML or using the thin object layer to get and set information from object properties.

Both methods are described in this document. Usually, the qbposXML method is described first and qbposFC described immediately after it, where appropriate.

Notice that all applications integrated with QBPOS use the qbposxmlrp.dll, even those written using qbposFC. Applications implemented with qbposFC must, in addition, also use the qbposFC DLL. This DLL is versioned.

## A Note About Programming with qbposFC

---

If you program with qbposFC, in addition to the OSR you should read Chapter 4, “Using the qbposFC Convenience Library.” Notice that the qbposFC objects and methods are also available from your programming IDE if it supports Intellisense or object browsing.

## Which QBPOS Edition Supports Which Features?

---

There is an appendix (currently Appendix B) in the User Guide provided with QBPOS that lists each QBPOS feature and the QBPOS editions (QBPOS Basic, Pro, and Multi-Store) that support the feature. You can access the QBPOS User Guide from the QBPOS main menubar by selecting Help->User Guides.

If you look at the functionality table in that user guide appendix, you’ll notice that the Basic edition doesn’t support some functionality that is supported by Pro, and Pro doesn’t support some functionality that is supported by Multi-Store.

However, you should be aware that these differences in feature support are applicable only to the QBPOS UI. If you use the QBPOS SDK, the same features are available across the board to integrated applications, with some notable exceptions, which we’ll cover shortly. For example, you can use the SDK to create and use SalesOrders, even if your application is running against the Basic edition, which doesn’t support SalesOrder in its UI. The only thing to remember is that in this case, although the SalesOrder data will be stored in the QBPOS company, only your application or Pro editions and above will be able to access it. (The QBPOS Basic edition won’t be able to use access these SalesOrders because it lacks the UI to do so.)

Some Multi-Store features, however, can only be used (even with the SDK) with the Multi-Store edition, because only that edition has the UI support required for the creation and setup of stores 2 through 10. For example, if you have Basic or Pro you could not perform a transaction involving store 2.

## Can My Application Access QB POS Merchant Services?

---

The QBPOS SDK does not expose any QBPOS Merchant Service functionality. Therefore, you cannot currently process credit card transactions in your application using QBPOS merchant services.

## CHAPTER 2

# COMMUNICATING WITH QBPOS

This chapter describes communicating with QBPOS using the request processor API or alternatively using the qbposFC library. This chapter does not describe the construction or content of the requests sent to QBPOS, just the initial work your application needs to do before sending the requests to QBPOS. This chapter covers the following topics:

- “Call Sequence for QBPOS Communication”
- “How End Users Authorize Application-QBPOS Communication”
- “What Happens When a QBPOS Company has Login Protection?”
- “Can I Access a QBPOS Company from a Remote Computer?”
- “Request Processor API and qbposFC Object Methods”
- “Communicating with QBPOS from Web Services”

## Call Sequence for QBPOS Communication

---

Figure 2-1 on page 10 shows the call sequence and session life cycle pseudocode.

## Communicating with QBPOS

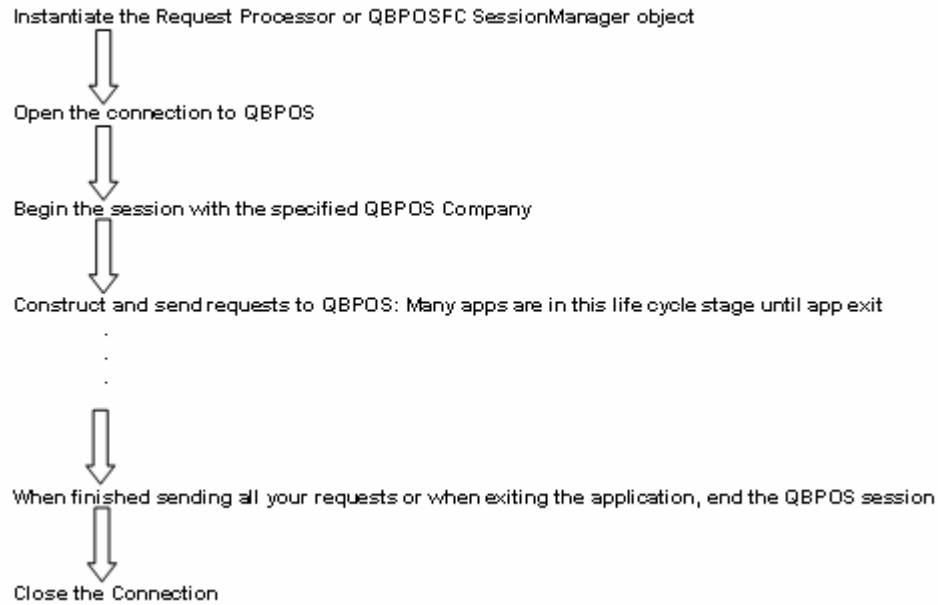


Figure 2-1 Call Sequence Pseudo Code

Table 2-1 shows Visual Basic examples of the code supporting the full life cycle of QBPOS communication for both methodologies: using the request processor API and using the qbposFC library.

Table 2-1 The Two Methodologies of QBPOS Communication

Communicating via Request Processor API	Communicating via the qbposFC Library
<pre> Dim qbposXMLRP As New QBPOSXMLRPLib.                                 RequestProcessor  Dim qbFileName As String Dim ticket As String Dim post As String Dim cAppID As String Dim cAppName As String cAppID = " " cAppName = "My Sample" 'This is how to build the connection string: qbFileName = "Computer Name=John's; +             Company Data=My Company Data; +             Version=5" 'Supplying an empty connection string would let 'the user pick 'from a list of servers and companies qbposXMLRP.OpenConnection cAppID, cAppName ticket = qbposXMLRP.BeginSession(qbFileName) 'xmlStream is a fully constructed qbposXML 'message request post = qbposXMLRP.ProcessRequest(ticket,                                 xmlStream)  qbposXMLRP.EndSession (ticket) qbposXMLRP.CloseConnection </pre>	<pre> Dim sessionManager As New QBPOSSessionManager Dim qbFileName As String Dim ticket As String Dim post As String Dim cAppID As String Dim cAppName As String cAppID = " " cAppName = "My Sample" 'This is how to build the connection string: qbFileName = "Computer Name=John's; +             Company Data=My Company Data; +             Version=5" 'Supplying an empty connection string would let 'the user pick 'from a list of servers and companies sessionManager.OpenConnection cAppID, cAppName ticket = sessionManager.BeginSession(qbFileName) 'requestMsgSet is a fully constructed request 'message set Dim responseMsgSet As IMessageSetResponse Set responseMsgSet = sessionManager.DoRequests                                 (requestMsgSet)  sessionManager.EndSession sessionManager.CloseConnection </pre>

Notice that the basic communication is very similar whether you use the request processor or qbposFC. The real differences (and ease of use of the qbposFC library) come when you build the request set, which is glossed over in our table.

As shown in Table 2-1, to communicate with QBPOS, an application must do the following:

1. Instantiate the request processor object, if using the request processor API. If using qbposFC, instantiate the QBPOS session manager object.
2. Open the connection. The AppID and the AppName values can be any values you want.
3. Begin the session. The BeginSession call in both methodologies takes a string value of this format: "Computer Name = Your Computer Name;Company Data = Your Company Name;Version=YourVersion", where "Your Computer Name" is the end user's computer name, "Your Company Name" is the name of the QBPOS company you are logging into, and "YourVersion" is the version number of QBPOS that you want to use. Optionally, you can supply the parameter Practice=Yes if you want to work with the practice version of that company file.

(For more information about versions, see "Compatibility and QBPOS Version Issues.") Alternatively, you could simply supply an empty string here, which would cause the local network to be searched for a list of QBPOS servers and companies for user selection. This method can be time consuming, however, due to the time required

to search for available QBPOS servers.

The begin session call makes a database connection, which involves overhead and time. So if your application is going to support ongoing access to a QBPOS company, you should keep the connection and close it only when the application no longer needs it. Avoid implementations where you open the connection, make a request, close the connection, then reopen it, then make a request, and so on repeatedly. You should close a connection when you are finished with it.

Notice that BeginSession returns the session ticket, which you must supply in the subsequent calls to the request processor.

4. Build the requests and send them to QBPOS. Constructing the requests is where most of your work as a programmer will occur.
5. End the session upon application close or when finished sending requests.
6. Close the connection.

## My Language isn't VB: Where Do I Find Communication Syntax Information?

---

You can find examples of QBPOS communication in languages other than Visual Basic in the QBPOS SDK subdirectory \Samples\qbpos. For languages that are not sampled in that subdirectory, refer to the COM interface definitions for the Request Processor API or, optionally for the qbposFC library. These are in the qbposXML and qbposFC type libraries, respectively, and can be viewed with various object browsers, such as the Visual Studio Object Browser.

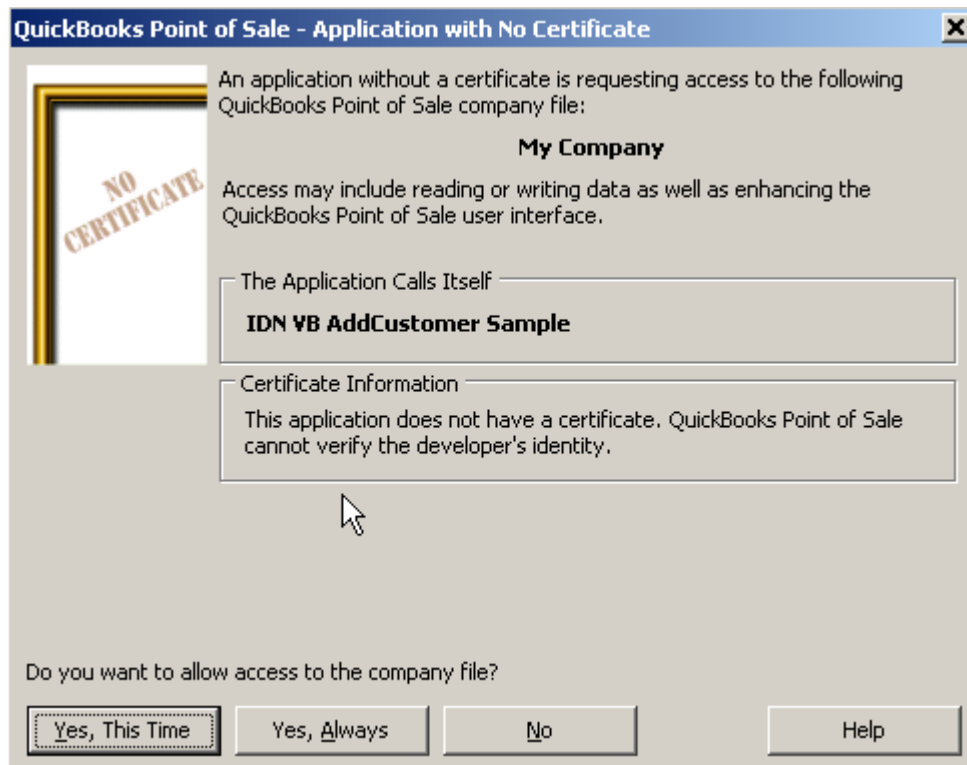
## How End Users Authorize Application-QBPOS Communication

---

The first time a QBPOS-integrated application accesses a QBPOS company, the call to BeginSession causes an authorization dialog to be displayed. This prompts the QBPOS administrative user (SysAdmin) to grant or deny authorization to the application. The user can choose either a one-time only authorization, or a permanent authorization, or the user can deny access altogether.

### **IMPORTANT**

Only the QBPOS administrative user (SysAdmin) can authorize applications. Once an application is authorized, however, other non-admin users can use that application.



If the user selects a one time (this time) authorization, the dialog is displayed the next time the application makes a call to BeginSession. If the user selects Yes, Always, the authorization dialog will not appear again.

#### **IMPORTANT**

During the development of your application, each time you rebuild your application you will be prompted to re-authorize that application, even though you authorized it by selecting "Yes, Always". The reason for this is that the application is seen as a new application for security purposes. For this reason, your customers will need to reauthorize your application when they install a new version of it.

## What Happens When a QBPOS Company has Login Protection?

The QBPOS user may at any point change the QBPOS company preferences to require login protection. Logins for a company are set and removed in the QBPOS Company Preferences page checkbox labelled "Require users to sign in." (This page is displayed by selecting Edit->Preferences->Company from the main QBPOS menubar.)

What this means for SDK applications is that if logins are required, a login prompt is automatically displayed for the end user when the application attempts to begin a session. Only after the login data is successfully entered can the application access the company. There is no way to handle this programmatically.

## Security and SDK Applications

---

When access is given to an SDK application, that application has full access. The QBPOS administrator, however, is able to limit user access to SDK applications. But the security right for "access SDK apps" is true by default.

## Can I Access a QBPOS Company from a Remote Computer?

---

An SDK application on one machine in a local network can access a QBPOS company on another machine in that network, whether QBPOS is installed on the local computer or not.

## Request Processor API and qbposFC Object Methods

---

The following tables list the object methods available for the request processor (QBPOSXMLRPLib.RequestProcessor) and the qbposFC session manager (QBPOSSessionManager), respectively.

These methods are part of the required call sequence described in this chapter, or are convenience methods for determining the current qbposXML version and request processor version. There are also qbposFC methods for converting valid qbposXML request/response strings into the corresponding qbposFC objects, and methods for converting qbposFC request/response objects into qbposXML strings.

Table 2-2 on page 15 lists the methods for the request processor object. The syntax and more description are provided in Appendix B, "Request Processor API Reference."



Table 2-2 QBPOSXMLRPLib.RequestProcessor Methods

Method	Description
BeginSession	Begins a session operating on a specific company file.
CloseConnection	Closes the connection between your application and QBPOS.
EndSession	Ends the QBPOS session.
GetCurrentCompanyFileName	Returns the name of the company file that is currently open (requires a ticket)
MajorVersion	Returns the major version number of the qbposXML Request Processor (for version 1.0, the major version number is 1)
MinorVersion	Returns the minor version number of the qbposXML Request Processor (for version 1.0, the minor version number is 0).
OpenConnection	Opens a connection between your application and QBPOS.
POSServers	Returns the QBPOS servers that are available in the local network.
POSVersions	For the specified company, returns the versions of that company database that are available, permitting user selection of the desired version.
ProcessRequest	Sends a request message set to QBPOS and returns with a response message set from QBPOS.
QBPOSXMLVersionsForSession	Returns the versions of the qbposXML specification supported by the request processor that your application is currently using to connected to QBPOS (requires a ticket)
ReleaseLevel	Returns the release description of the qbposXML Request Processor (for example, <i>alpha</i> , <i>beta</i> , or <i>release</i> )
ReleaseNumber	Returns the release number of the qbposXML Request Processor (for example, the release number for version 1.0 is 1)

Table 2-3 on page 16 lists the methods for the qbposFC Session Manager object.

Table 2-3 QBPOSSessionManager Methods

Method	Description
BeginSession	Begins a session with a QBPOS company.
CloseConnection	Closes the connection with QBPOS.
CreateMsgSetRequest	Creates the IMsgSetRequest object that contains the actual individual requests. The individual requests are appended to the IMsgSetRequest object via the various IMsgSetRequest object methods.
DoRequests	Sends the fully constructed IMsgSetRequest object to QBPOS.
DoRequestsFromXMLString	Like DoRequests, but takes an XML string. This is useful if you already have the fully constructed XML request string and don't need to use qbposFC to build it.
EndSession	Ends a QBPOS session.
GetCurrentCompanyFileName	Returns the name of the QBPOS company data file that the application is currently logged into.
GetVersion	Returns the current qbposFC version.
OpenConnection	Establishes a connection with QBPOS.
POSServers	Returns the QBPOS servers that are available in the local network.
POSVersions	For the specified company, returns the versions of that company database that are available, permitting user selection of the desired version.
QBPOSXMLVersionsForSession	Returns the supported qbposXML versions.
ToMsgSetRequest	Converts the supplied valid qbposXML message request string into an IMsgSetRequest object.
ToMsgSetResponse	Converts the supplied valid qbposXML message response string into an IMsgSetResponse object.

## Communicating with QBPOS from Web Services

Communicating with QBPOS from a web service requires the use of the QuickBooks Web

Connector (QBWC). For more information see the QBWC Programmer's Guide, which is available in the QuickBooks SDK.



## CHAPTER 3

# RUNNING, DEPLOYING, AND DISTRIBUTING YOUR APPLICATION

This chapter details what you need to install on your system and your customer's systems in order to enable your application to access a QBPOS company. It also describes how to distribute your application with any QBPOS SDK merge modules and/or installers.

## Required Files

---

The following files are included in the QBPOS SDK and are redistributable, following the terms of the QBPOS SDK license:

- dbcon8.dll
- dbghelp.dll
- dblgen8.dll
- dblib8.dll
- qbposfc1\_1.dll
- Interop.QBPOSXMLRPLIB.dll
- QBPOSXMLRPLib.dll
- stlport\_vc745.dll
- stlport\_vc746.dll
- xerces-c\_2\_5\_0\_qb.dll

The files must be installed on the system where your application is going to run, and the system path must include their location.

QBPOSXMLRPLib.dll and qbposfc1.dll both expose COM objects and must be registered as COM servers in the windows registry.

## Distributing Your Application

---

If you are using the qbposFC API or even if you are just using QBPOSXMLRPLib you need to redistribute the SDK runtime. There are only two supported ways in which you can redistribute the SDK runtime:

1. You can use the stand-alone compressed-image installers that we provide.
2. You can use the merge modules that we provide.

## NOTE

*It is a violation of your SDK license agreement to redistribute the SDK runtime or qbposFC without using either our stand-alone installers or our merge modules.*

Automatic installation programs and packaging wizards, such as the wizard in Microsoft® Visual Studio®, will not perform the install properly (even if you are using .NET):

- Automatic solutions will redistribute the qbposxmlrp.dll file. Redistributing this file is against your license agreement, and could also cause significant problems for your end users.
- Automatic solutions will redistribute the qbposFC DLL file or files, but not the Xerces files that must go with them.

## Using the Stand-Alone Installers

---

If your install process does not support merge modules, you will need to use the stand-alone installers provided with the SDK. Using these installers will result in a good install.

To install the qbposFC Library on your end-users' machines:

- Distribute the QBFC installer, QBPOSFC1\_0Installer.exe, located in the QBPOS SDK install subdirectory /tools/installers folder. (Merge modules are in the QBPOS SDK install subdirectory /tools/MergeModules.)
- Call the installer. Exactly how you call it depends on the underlying technology you are using to drive your installation.

## Using the Merge Modules

---

If your install process supports Microsoft merge modules, you can use the merge modules that are provided.

### What Is a Merge Module?

The Microsoft Installer (MSI) service is built into Windows 2000 and XP. MSI solves a number of installation problems, such as getting clean uninstalls and protecting system components, and includes redistributable install engines that support Win98, WinNT, and Win ME. To get a “Designed for Windows” logo, your application must be installed using MSI.

“Merge modules” are a key part of MSI. They encode the logic and files needed to correctly redistribute shared components, which aren’t removed from a system until all of the applications that installed them are removed.

Any installation that is built for an MSI-engine installer can use merge modules. Many proprietary install tools that are not strictly based on MSI (for example, newer versions of InstallShield Professional) can also take advantage of merge modules.

## How Do I Use a Merge Module from the SDK?

The SDK merge modules are located in the /tools/MergeModules folder. Here's how to use them:

1. Make sure you have the Microsoft VC (VC\_CRT.msm) and VC++ (VC\_STL.msm) runtime library merge modules, which are required because the SDK merge modules install components that depend on the Visual C and C++ version 7 runtime libraries.  
  
These Microsoft merge modules are included with most MSI-based install builders, or you can get them directly from Microsoft. When the VC\_CRT.msm and VC\_STL.msm modules are added to the installer, the install author is responsible for configuring them to set their target directory to the windows system directory.
2. Set your installation development tool to include the SDK MergeModules directory in the MergeModule search path.
3. Each MSI "feature" refers to components and/or merge modules. For any feature that installs components of your application that depend on the SDK capabilities provided by a merge module, specify that particular merge module as part of that feature.  
If a merge module is dependent on some other module, the other module will be added to your installer automatically. (For example, the various versions of the QBFC merge modules depend on various versions of Xerces, which are packaged in separate merge modules: the correct one is automatically added to the installation.)
4. Build your installation as usual. All the logic from the included merge modules will be merged into your install.

## What Installation Logic Is Built Into the Provided Merge Modules?

The qbposFC merge modules provide qbposFC DLL files and COM registration information for qbposFC. The QBFC merge modules depend on the Xerces XML parser module and on the QBPOSXMLRP merge module—in other words, installing qbposFC installs the Xerces files and QBPOSXMLRP.





## CHAPTER 4

# USING THE QBPOSFC CONVENIENCE LIBRARY

This chapter describes the main QBPOS Foundation Classes (qbposFC) objects and methods you need to use to create and send requests to QBPOS and to process the data returned. It points you to the OSR for the details you need and also mentions how to use the OSR to look up this information. Although the central qbposFC objects and methods are described in this chapter, you'll notice that not every possible object, property, and method are listed. There are a couple of reasons for this approach:

- First, there is really no need to do this: the objects and methods are available in the type library and can be seen via an object browser (such as Microsoft's OLEView included with Visual Studio).
- Second, much of the information is redundant: most of what you need to know is already documented in the OSR or in the relevant chapters in this programming guide.
- Third, such an approach is confusing, due to the plethora of objects in qbposFC.

## Understanding the Context and Flow of qbposFC Objects

---

If you try to understand the qbposFC object model from the lengthy list of available objects, you're in for a long day's work. The qbposFC library is a thin wrapper, which means there are a great many objects to wade through if you are just looking at objects.

### Objects, Objects Everywhere: Where Do I Start?

---

The best way to understand the qbposFC objects is to consider the way in which the objects are used, starting with the central qbposFC object: *QBPOSSessionManager*. This object is used for all communication with QBPOS, including:

- Making the initial QBPOS connection
- Beginning the QBPOS session,
- Creating requests
- Sending the requests to QBPOS
- Returning responses from QBPOS.

In this chapter, we are concerned mainly with the *QBPOSSessionManager*'s role in creating and sending requests and returning responses because most of the qbposFC objects and data processing via those objects are exercised during these activities.

#### **NOTE**

For more information on *QBPOSSessionManager*, see Chapter 2, "Communicating with QBPOS."

## Which Objects Do I Need to Create a Request?

As shown in Figure 4-1, you first use the QBPOSSessionManager's CreateMsgSetRequest method to create an IMsgSetRequest object that will contain one or more requests to be sent to QBPOS. You then add an empty request object to that IMsgSetRequest object via an Append method. As indicated in the figure, IMsgSetRequest has one Append method for each possible SDK request. (The OSR for qbposFC lists all of the SDK requests, if you don't have an object browser handy.)

### IMPORTANT

Only invoke an Append method once for each desired request!  
Each time you invoke an Append method, you are adding a new and separate request object to the request set.

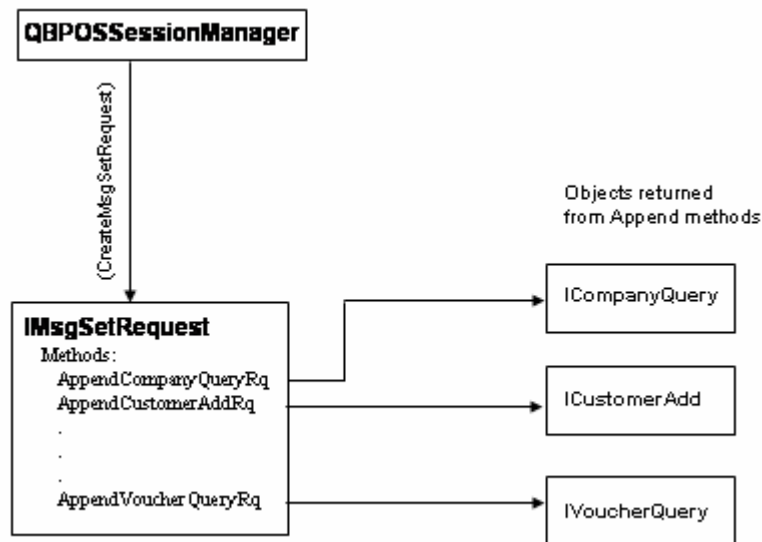


Figure 4-1 Objects used to build requests

In Figure 4-1, notice that when you invoke an Append method, it returns you an empty object that you subsequently must construct prior to sending the message set to QBPOS. For example, **AppendCustomerAddRq** returns the **ICustomerAdd** object. Once you have this object, you can go to work constructing your request by filling in the properties for this object.

Notice that the various Append methods are not listed in the OSR. If your programming environment supports Intellisense, you can see all of the Append methods in your code editor when you instantiate an **IMsgSetRequest** object. If you don't have this, and don't have an object browser that can look into the qbposFC library, you can easily construct the Append method name simply by looking at the request names listed in the OSR: just put "Append" in front of each request name and "Rq" at the end of the request name.

**NOTE**

For each request in the OSR, the OSR provides code samples that show the construction in VB or VB.Net code. Simply click on the VB6 Code or VB.NET Code tabs in the OSR.

## How Do I Use the OSR to Construct the Request?

You need to use the OSR to program in qbposFC, both to get an idea of what properties are required/available for a given request or response object, and for documentation of those properties. Figure 4-2 shows the OSR listing for the CustomerAdd request.

QBPOSFC1				Home   VB6 Code   VB.NET Code
CustomerAdd				Show
Tag	Type	Max	Implementation	
<b>ICustomerAdd</b>				
Notes	IQBStringType	245 Chars		
UseWithQB	IQBBoolType			
CustomerDiscPercent	IQBAmountType			
AcceptChecks	IQBBoolType			
CustomerDiscType	IQBENCustomerDiscTypeType			
EMail	IQBStringType	99 Chars		
TaxCategory	IQBStringType	20 Chars		
Phone	IQBStringType	21 Chars		
AltPhone	IQBStringType	21 Chars		
Salutation	IQBStringType	15 Chars		
LastName	IQBStringType	30 Chars		
FirstName	IQBStringType	30 Chars		
CompanyName	IQBStringType	40 Chars		
PriceLevelNumber	IQBENPriceLevelNumberType			
ShipAddress	IShipAddress			
Addr1	IQBStringType	41 Chars		
City	IQBStringType	31 Chars		
State	IQBStringType	21 Chars		

Figure 4-2 OSR listing for CustomerAdd request

Notice the name directly under the Tag heading, circled in this figure, ICustomerAdd. This is the name of the qbposFC object for the request or response, depending on whether the OSR is showing requests or responses.

In your code, you need to create an object of this type by invoking the corresponding Append method that returns it, and then fill its fields as you want, using the field names listed under that object in the OSR. For example, for a new customer, you would set the first name field like this if you were coding in Visual Basic:

```
Dim MyICustomerAdd as ICustomerAdd
Set MyICustomerAdd = MyRequestMsgSet.AppendCustomerAddRq
MyICustomerAdd.FirstName.SetValue("Fred")
```

## The OSR Has Everything You Need to Set Request Values

The OSR contains all the information you need to set request object values, including any available enumerated values. Figure 4-3 shows the price level number field for a customer, which is an enumerated value. To see the available values, click on the field type to the left of the field name. In our example, this would be

IQBENPriceLevelNumberType

Clicking on the type displays the enumerated values, as in the circled area in the figure where the values displayed are pln1, pln2, pln3, and pln4.

CustomerAdd

AccountLimit	IQBPriceType
AccountBalance	IQBPriceType
TaxCategory	IQBStringType
Phone	IQBStringType
AltPhone	IQBStringType
Salutation	IQBStringType
LastName	IQBStringType
FirstName	IQBStringType
CompanyName	IQBStringType
StoreExchangeStatus	IQBENStoreExchangeStatusType
FullName	IQBStringType
TimeModified	IQBDateTimeType
PriceLevelNumber	IQBENPriceLevelNumberType
ShipAddress	IShipAddress
Addr1	IQBStringType
City	IQBStringType
State	IQBStringType
PostalCode	IQBStringType
ShipSameAs	IQBBoolType
BillAddress	IBillAddress
PostalCode	IQBStringType
State	IOBStringType

**IQBENPriceLevelNumberType**

typedef enum {pln1, pln2, pln3, pln4} ENPriceLevelNumber

'VB6 Methods and Properties

SetValue ( val As ENPriceLevelNumber )

GetValue ( ) As ENPriceLevelNumber

Figure 4-3 Looking up enumerated values in the OSR

## Other Useful IMsgSetRequest Methods

---

The IMsgSetRequest Append methods are already covered in this document and in the OSR. However, the request message set object has other methods you need to know about, and these methods are shown in the following table.

<b>IMsgSetRequest Method/Property</b>	<b>Parameters</b>	<b>Description</b>
HRESULT Attributes ([out, retval] IAttributesRqSet**pVal);	-pVal Pointer to the returned IAttributeRqSet object	This property returns the IAttributeRqSet object, which you would need if you wanted to determine the current attribute settings in the request set.  IAttributeRqSet contains the attributes that are currently in effect for all requests in the message set.  QBPOS currently supports one such attribute: OnError, which can be set to roeContinue or roeStop.  roeContinue means that a failure in one request will not prevent the other requests from being processed.  roeStop means that a failure in any request will prevent any subsequent requests in the message set from being processed.
HRESULT ClearRequests();	NA	Removes all requests currently appended to the request message set.
HRESULT RequestList ([out, retval] IRequestList* *pVal);	pVal Pointer to the returned IRequestList object	You probably will seldom use this property. You would use this property if you wanted to get one or more requests from the request message set. The IRequestList object returned has a count and a GetAt method for returning individual IRequest objects from the list.  Once you have the IRequest object, you can use its RequestID, Type, or Detail methods as desired. The Detail is processed exactly like its IResponse counterpart, which is thoroughly covered in Chapter 5, "Building Requests and Processing Responses."

<b>IMsgSetRequest Method/Property</b>	<b>Parameters</b>	<b>Description</b>
HRESULT ToXMLString([out, retval] BSTR* qbposXMLRequest);	-qbposXMLRequest Pointer to the returned string containing the request message set in qbposXML format.	This method is very handy during diagnostics where you need to examine the complete XML representation of the requests that were built in qbposFC. Useful for making sure you are getting the requests you expect.
HRESULT Verify([out] BSTR* errorMsg, [out, retval] VARIANT_BOOL* isOK)	errorMsg Contains an error message for every request that failed validation. If there is no failure, this string is empty.  isOk Returns True in VB and Variant_True in C++ if all the requests are valid. Returns False in VB and Variant_False in C++ otherwise.	The DoRequests method causes validation to be run automatically. However, if you need to validate the requests for proper construction before you invoke DoRequests, you can use this method.

## Which Objects Do I Need to Process a Response?

As shown in Figure 4-4, you use the QBPOSSessionManager's DoRequests method to send a request message set to QBPOS. The input to this method is the IMsgSetRequest described earlier in this chapter. The return from this method is an IMsgSetResponse object.

The IMsgSetResponse has a property called ResponseList that returns the IResponseList object containing all of the responses to the requests that were made in the IMsgSetRequest. If all of the requests were successful, the number of responses will match the number of requests in the corresponding IMsgSetRequest, but this won't always be the case, due to the possible request errors.

Notice that the DoRequest method returns successfully without errors even if any or all of the requests fail. You can't use the method return to get information on whether requests were successful. To do this, you need to process the individual request objects (IResponse) contained in the ResponseList. Once you have the individual IResponse, you can query it for the StatusCode and StatusMessage you need to determine success or failure and get some idea of the nature of the failure.

We'll describe the IResponse methods later. But in the figure, notice the methods that are listed, such as Detail, StatusCode, and Type. You will use those every time you process a Response: you need to check the StatusCode for success, then you need to check the Type. You need the type because this determines what kind of object you need to use to receive the response Detail, and, in some languages, make an upcast to that type.

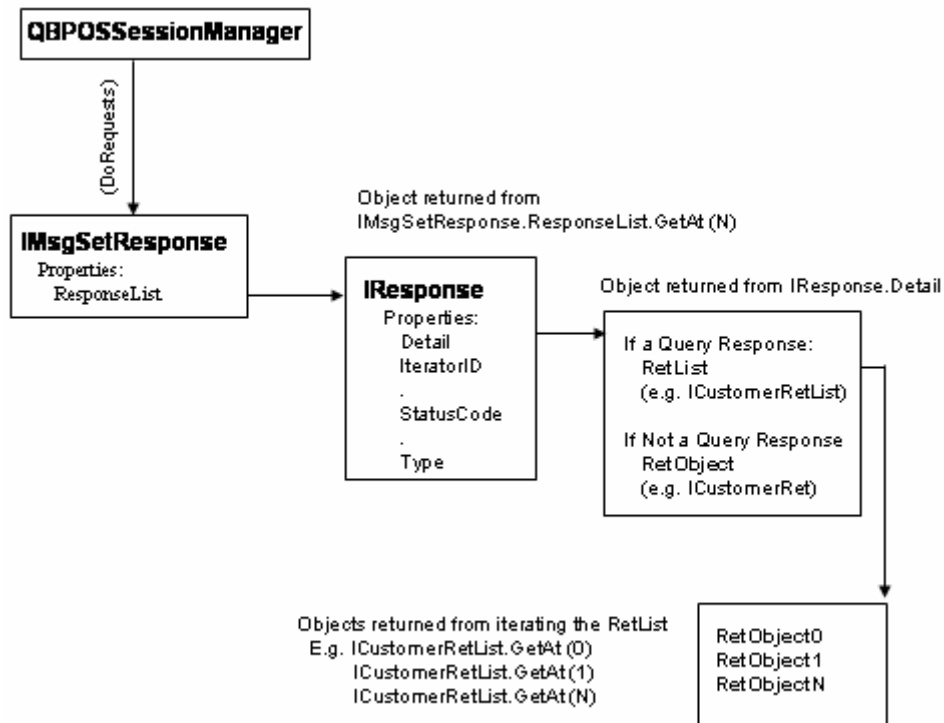


Figure 4-4 Objects used to process data from a response

As shown in the figure, the response Detail differs for queries. For non-query requests, the Detail contains a Ret object with the actual field values you are interested in. For query requests, the Detail contains a Ret list that contains the individual Ret objects.

An explanation of how to process Ret objects from this list is described in Chapter 5, “Building Requests and Processing Responses.” It is fairly straightforward, using the RetList object’s Count property to get the number of Ret objects and then using the GetAt method to return the individual Ret object from the RetList.

## Getting Data from the Ret Object

You can see the Ret object for each request in the OSR by clicking on the Response link in the upper right of the OSR page.

Figure 4-5 shows the Ret object for the ICustomer object in the OSR, with the object name circled. Remember that the OSR contains primarily information about each request and Ret object proper (field).

If you click on the VB6 or VB.Net links at the upper right of the OSR (circled in the figure), you’ll get context-sensitive source code samples that show how to get data from a response message set (IMsgSetResponse) that contains a customer Ret object.

QBPOSFC1		Home   VB6 Code   VB.NET Code	
CustomerAdd		Sho	
Tag	Type	Max	Implementation
<b>ICustomerRet</b>			
Notes	IQBStringType	245 Chars	
UseWithQB	IQBBoolType		
CustomerDiscPercent	IQBAmountType		
LastSale	IQBDateType		
AcceptChecks	IQBBoolType		
CustomerDiscType	IQBENCustomerDiscTypeType		
EMail	IQBStringType	99 Chars	
ListID	IQBIDType		
AccountLimit	IQBPriceType		
AccountBalance	IQBPriceType		
TaxCategory	IQBStringType	20 Chars	
Phone	IQBStringType	21 Chars	
AltPhone	IQBStringType	21 Chars	
Salutation	IQBStringType	15 Chars	
LastName	IQBStringType	30 Chars	
FirstName	IQBStringType	30 Chars	
CompanyName	IQBStringType	40 Chars	

Figure 4-5 The OSR Response object (Ret object) for Customer

Just like the request object described earlier, the properties listed for each Ret object in the OSR are the actual property names you need to supply in order to get the data for that property. For example, to get the Notes data from the customer Ret, you'd use this code in VB:

```
MycustomerRet.Notes.GetValue
```

## Objects and Methods Used in Processing Response Data

The following tables list the objects described in the previous paragraphs, along with their methods and properties. The request and Ret objects are not listed here because they are documented fully in the OSR.



## IMsgSetResponse

The following table shows the properties and methods for the message set response object:

<b>IMsgSetResponse Method/Property</b>	<b>Parameters</b>	<b>Description</b>
HRESULT ResponseList ([out, retval] IResponseList* *pVal)	-pVal Pointer to the returned IResponseList object.	You need to invoke this on every IMsgSetResponse object to get the response list.
HRESULT ToXMLString ([out, retval] BSTR* qbposXMLResponse);	-qbposXMLResponse Pointer to the returned string containing the response message set in qbposXML format.	This method is very handy during diagnostics where you need to examine the complete XML representation of the responses that were returned from QBPOS. Useful for making sure you are getting the results you expect.

## IResponseList

The following table shows the properties and methods for the IResponseList object.

<b>IResponseList Method/Property</b>	<b>Parameters</b>	<b>Description</b>
HRESULT Count ([out, retval] long *pVal)	-pVal Pointer to the returned number of IResponse objects contained in the list.	Useful for setting up a loop, with Count used as the loop limit.
RESULT GetAt (long index, [out,retval] IResponse** retVal);	-index The supplied index specifying which IResponse object in the list is to be returned.  -retVal Pointer pointer to the returned IResponse object.	The index is zero based, so the first IResponse on the list has the index of 0.

## IResponse

The following table shows the properties and methods for the IResponse object.

IResponse Method/Property	Parameters	Description
HRESULT Detail ([out, retval] IQBBase** pVal);	-pVal Pointer pointer to the response contents	This returned value must be upcast in most languages. For example, in VB.NET, for example, you would get the Response detail data into an ItemInventoryRet object like this:  itemInventoryRet = response.Detail as itemInventoryRet
HRESULT iteratorID ([out, retval] BSTR *pVal);	-pVal	The iteratorID is returned only for queries that use iterators to manage the amount of data returned. You would need to get this ID from the first response, then use it in the succeeding iterations of the query.  For more information on iterators, see Chapter 7, "Creating Queries."
HRESULT iteratorRemainingCount ([out, retval] long *pVal);	-pVal	This property indicates the number of objects remaining to be iterated through. This is helpful in optimizing the MaxReturned value, among other things.
HRESULT retCount ([out, retval] long *pVal);	-pVal	This value is available if the response is a query response. It indicates the count of Ret objects in the list.
HRESULT StatusCode ([out, retval] long *pVal);	-pVal	Indicates success or the nature of any failure. The value 0 indicates success.
HRESULT StatusMessage ([out, retval] BSTR *pVal);	-pVal	A text message that provides more information than just the status code regarding the nature of the failure
HRESULT StatusSeverity ([out, retval] BSTR *pVal);	-pVal	The severity level of the error.
HRESULT Type ([out, retval] IResponseType** pVal);	-pVal	The type of RetList or Ret object contained in the IResponse. You need this to specify the object to receive the IResponse.Detail data.

## CHAPTER 5

# BUILDING REQUESTS AND PROCESSING RESPONSES

This chapter describes how to build requests and handle responses using both the request processor API and using the qbposFC library. Because the request processor API method requires you to build valid qbposXML request strings and parse the returned qbposXML responses, this chapter shows the use of Microsoft's XML Services 4.0 DOM to build and parse the XML using DOM documents.

If you are already familiar with the QuickBooks SDK and qbXML and QBFC, the material in the chapter is very similar to what you already know. But you may want to skim through this chapter anyway, looking for areas where QBPOS differs. These are usually noted in the text.

### Building a qbposXML Request

---

Building a qbposXML request and sending it to QBPOS is simply a matter of writing out a valid qbposXML string that contains the requests you want and then sending them to QBPOS. However, because building syntactically correct XML "by hand" is so tedious and prone to error, we strongly recommend the use of technology that does all the tedious stuff for you (angle brackets, start and end tags, and so on), so you can focus on including those qbposXML elements that you want.

Our samples show the use of Microsoft XML (MSXML) API DOMDocument, a technology which is currently free from Microsoft. When you build the qbposXML, you should refer to the Onscreen Reference (OSR) for details on required request elements.

## Building a Request Message Set: Pseudocode

---

Figure 5-1 shows the general steps in building a request using a DOM document.

```
Instantiate a DOM Document
Set the Node to QBPOSXML
  Add the message set (QBPOSXMLMsgRq) to the Node as the first DOM element
  Set any attributes on the message set
  Add the Request tag DOM element (i.e. AddCustomerRq) to the message set
  Set any attributes on the request
    Add a DOM element containing a desired request element tag (i.e., FirstName)
    Set the value for that element (i.e., John)
    Repeat for each desired request element
  If desired, add other Request tags and populate them as per the above pseudocode
```

Figure 5-1 Building a qbposXML Request

## Building a Request Message Set: Sample Code

---

The following code snippet shows the qbposXML for adding a new customer, with just a few of the available fields filled out, to keep things simple. Once the request is filled out, our code snippet prepends the required header information and sends the completed XML string to QBPOS via the request processor.

```
' Build the request XML
Dim builder As New XmlDocument40
Dim QBPOSXML As IXMLDOMNode

'After the header tags, <QBPOSXML> is always the first element tag,
'followed by the message set <QBPOSXMLMsgsRq>

Set QBPOSXML = builder.createElement("QBPOSXML")
builder.appendChild QBPOSXML
Dim msgsRq As IXMLDOMElement
Set msgsRq = QBPOSXML.appendChild(builder.createElement("QBPOSXMLMsgsRq"))

'Set attributes on the message set
msgsRq.setAttribute "onError", "continueOnError"
Dim CustomerAddRq As IXMLDOMElement
Dim CustomerAdd As IXMLDOMElement

Set CustomerAddRq =
msgsRq.appendChild(builder.createElement("CustomerAddRq"))
CustomerAddRq.setAttribute "requestID", "1"
Set CustomerAdd = CustomerAddRq.appendChild(builder.createElement
("CustomerAdd"))
```

```

Dim dataElement As IXMLDOMElement
If firstName <> "" Then
    Set dataElement = CustomerAdd.appendChild(builder.createElement
        ("FirstName"))
    dataElement.appendChild builder.createTextNode(firstName)
End If

If lastName <> "" Then
    Set dataElement = CustomerAdd.appendChild(builder.createElement
        ("LastName"))
    dataElement.appendChild builder.createTextNode(lastName)
End If

'The request is built except for the headers: so build these and
'append the request to them:
requestXML = "<?xml version="" & "1.0" & ""?">"
requestXML = requestXML + "<?qbposxml version="" & "1.0" & ""?">" +
    builder.xml

'Start a QBPOS session and send the request
Dim qbposXMLRP As New QBPOSXMLRPLib.RequestProcessor
Dim ticket As String
qbposXMLRP.OpenConnection cAppID, cAppName

ticket = qbposXMLRP.BeginSession(qbfilename)
responseXML = qbposXMLRP.ProcessRequest(ticket, requestXML)
qbposXMLRP.EndSession ticket
qbposXMLRP.CloseConnection

```

## Building a Request using qbposFC

---

The qbposFC library provides a convenience layer that allows you to construct requests using the familiar object and object property paradigm. You will notice that the convenience layer is a thin layer, so consequently there are a lot of objects to contend with in qbposFC. In fact, each request is a separate object, and each request object has its own unique sub objects, for example the line item add object for SalesOrders is different than the line item add object for SalesReceipts.

Fortunately, the abundance of objects is not as daunting as it might seem. All of the requests are built in the same way and in the same order, and the line items are appended to their parent object in the same way as well.

### IMsgSetRequest: the Central Object in Sending QBPOS Requests

If you look at the QBPOSSessionManager method (DoRequests) used to send requests to QBPOS, you'll notice that the method takes the input parameter of an IMsgSetRequest type. This is a message set object that contains a request list that has one or more IRequest objects, as shown in Figure 5-2.

## The Request Message Set Object

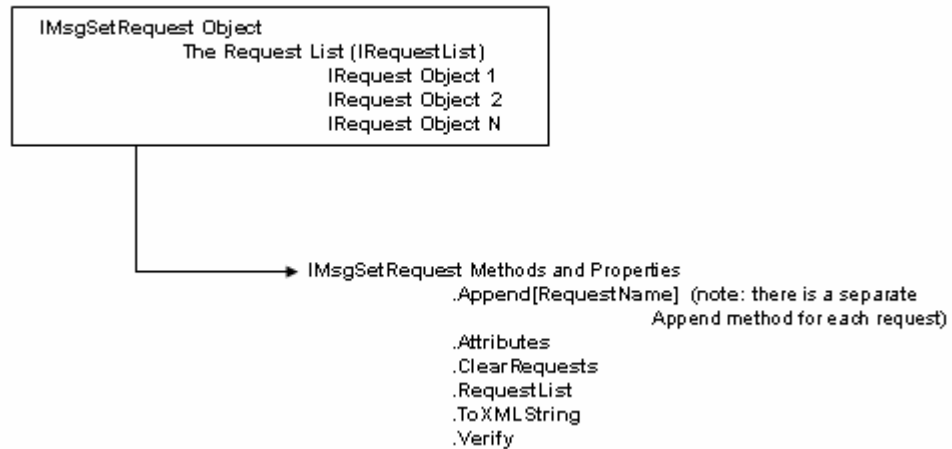


Figure 5-2 The Request Message Set Object

You need to be aware of the request message set methods and properties that are available for your use:

- The various Append methods are used to add request objects of a specific type to the message set. Each Append method call returns a corresponding request object that must be filled with property values as desired.
- The Attributes property is used to set message set-level attributes: currently continue on error and stop on error are the supported attributes for all requests; iterator and iteratorID are attributes supported for query requests. We'll describe the first two attributes later in this chapter; iterators are described in Chapter 7, "Creating Queries."
- The ClearRequests method empties the message set once you've invoked DoRequests and are otherwise finished with the original request. After you clear out the message set object, you can fill it again with new requests that you want to send. It saves some overhead to do this rather than instantiate a new message set object.
- The RequestList method returns the list (IRequestList) of request objects in the message set. IRequestList has a count property and a GetAt (index) method to support retrieval of requests from the list.
- The ToXMLString returns a complete and valid qbposXML string that represents the message set object and all its requests. This is useful for diagnostic purposes or if you simply want to make sure you are building the objects as you expect.
- The Verify method can be used before the call to DoRequests to make sure the requests are fully formed (all required fields have been set) and valid. However, the DoRequests method call also performs this checking automatically.

## Understanding the Request Message Set Structure in Detail

---

Figure 5-3 shows a more detailed view of the logical arrangement of the individual requests in a request message set, along with sample property values. The message set object contains one or more request objects.

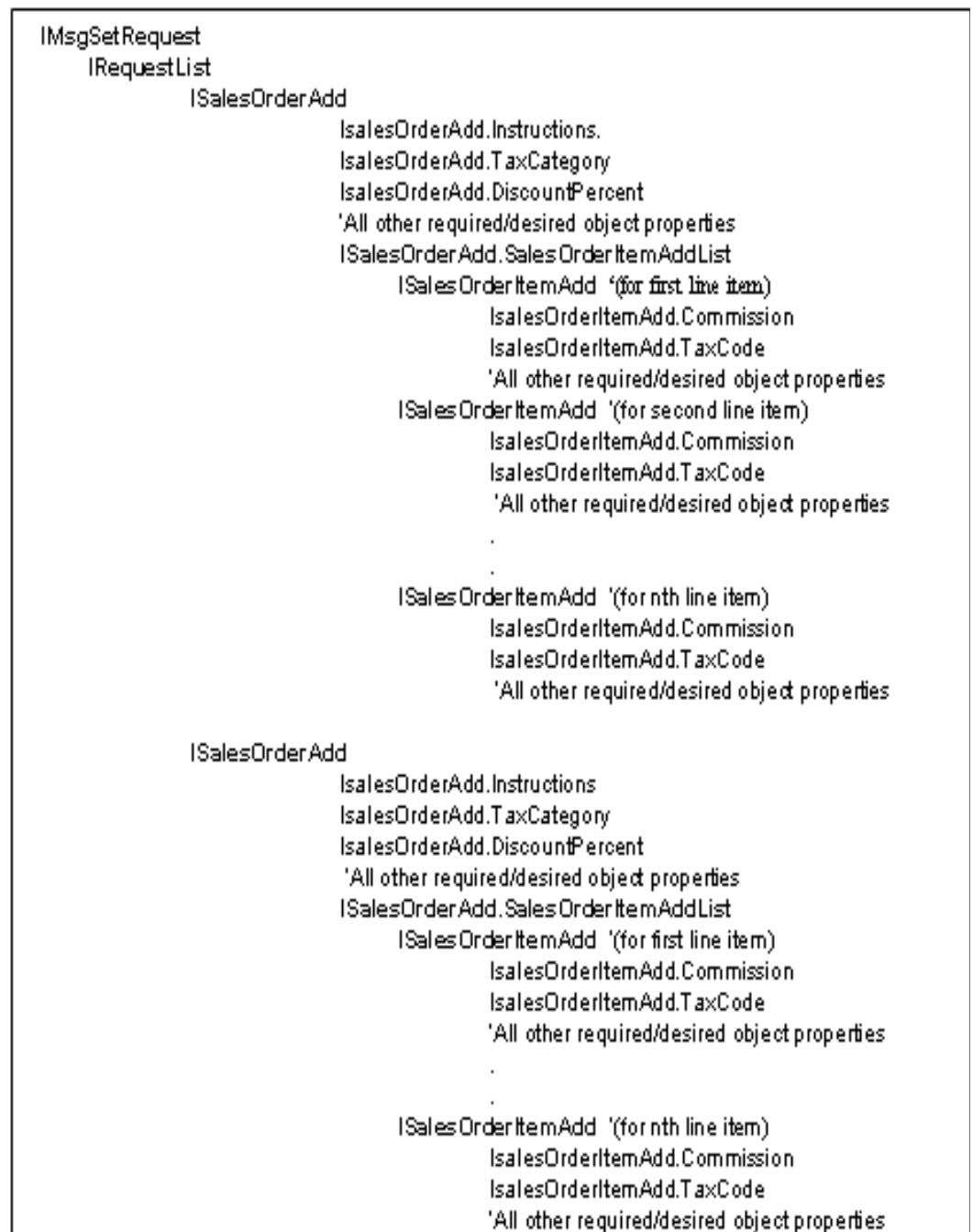


Figure 5-3 Request Message Set Structure

The figure shows a message set *transaction* object for a transaction containing two SalesOrderAdd objects. Each SalesOrder object has properties that need to be set in order for the request to be constructed. Moreover, each SalesOrder usually has a separate line item for each item sold. The line items contained in the SalesOrderItemAdd object are



added to the line item add list in the SalesOrder object. Notice that the line item objects also have properties that need to be set in order to finish constructing the request. There can be as many line item objects as you need.

The basic message set structure shown in Figure 5-3 is the same for every request object that has line items: these are usually transactions. If the object is a *list* object, which does not contain line items, such as Customer, you don't add any line item sub objects as shown in the sample figure. Instead you simply set the object properties as shown in the top of the figure.

For query requests, you can optionally supply one or more groups of filter conditions. For more information on building queries, refer to Chapter 7, "Creating Queries."

## How Do I build a Fully Constructed Message Set?

---

The following pseudocode shows how to build a complete request to send to QBPOS. Notice that the open connection and begin session calls are shown after the request is constructed. This is not required. You can perform the open connection and begin session calls at any time before the call to DoRequests. We made these calls at the end mainly to point out that you don't need to be connected to QBPOS in order to construct requests.

### **Building a QBPOS Request Message: Pseudocode**

You build a request message by doing the following:

1. Instantiate the QBPOSFC SessionManager object.
2. Create the request message set object (SessionManager.CreateMsgSetRequest).
3. Set any desired message set-level attributes in that message set object.
4. Append the desired request objects to the message set object.
5. Set all required or desired field values (see the OSR) in that request object.
6. If the request contains line items, append the appropriate line item add object to the request and set its field values: you must append one line item add object for each line item.
7. If you want to add another request, append the desired request object to the message set object and set its values as described above: you can add as many request objects to the message set as you want.
8. Open the connection to QBPOS.
9. Begin the session with the specified QBPOS Company.
10. Invoke DoRequests to send the requests to QBPOS.

### **Building a QBPOS Request Message: Sample Code**

The following Visual Basic code sample snippets and commentary show how to build a SalesOrderAdd request, just to be consistent with our examples in this chapter. Notice that we just filled a few properties for each object, in order to keep the sample easy to follow.

## 1: Creating the Message Set Object and Setting Its Attributes

The first order of business is to create the message set object:

### **'Create the message set object first**

```
Dim sessionManager As New QBPOSSessionManager
Dim requestMsgSet As IMsgSetRequest
Set requestMsgSet = sessionManager.CreateMsgSetRequest("1", "0")
requestMsgSet.Attributes.OnError = roeContinue
```

When you create the message set object, you need to specify the qbposXML spec version that this message set supports. The sample above creates a message set that supports version 1.0 of the qbposXML specification. The purpose of this is to make sure that the currently installed qbposFC library and request processor can support the type of requests your application is going to make.

Notice the OnError attribute setting of the message set. This specifies how errors in the individual requests are handled. If you specify roeContinue, as the sample does, then the failure of one request in the message set won't prevent other requests from being processed. If you specify roeStop, then an error in one request stops the processing of any succeeding requests.

## 2: Appending the Request Object to the Message Set and Filling its Values

After you create the message set object, you need to fill it with the requests you want it to contain. To add a request to the message set, simply invoke the Append method that creates the request you want. In the sample below, we used AppendSalesOrderAddRq. Keep in mind that the message set object has a separate Append method for each individual request, with each Append method returning the corresponding type of Request object for you to populate with values.

### **NOTE**

If your development environment and language supports Intellisense, a list of the message set's Append methods are available from within your environment. You could also use the Visual Studio Object Browser to look up the Append methods.

Alternatively, you can look up the request name in the OSR and prefix that name with Append and suffix it with Rq. Also, the top level object Tag name in the OSR for each request entry is the name of the object that is returned by the message set's corresponding AppendRequest method.

### **'Append the Request object and fill property values as you want**

```
Dim salesOrderAdd As ISalesOrderAdd
Set salesOrderAdd = requestMsgSet.AppendSalesOrderAddRq

salesOrderAdd.TaxCategory.SetValue "val"
salesOrderAdd.DiscountPercent.SetValue 2.00
salesOrderAdd.CustomerListID.SetValue "val"
salesOrderAdd.SalesOrderType.SetValue sotSalesOrder
salesOrderAdd.TxnDate.SetValue #2005/12/31#
salesOrderAdd.PriceLevelNumber.SetValue pln1
```

The SalesOrderAdd request's property values in the sample above are representative only. You need to become familiar with each request object property and how it relates to the underlying QBPOS features. This is key to programming a QBPOS-integrated application.

### 3: Adding Line Items to a Request

If a request supports line items and you want to use line items, you add them using the \*ItemAddList method available in the request object. Notice that each request object that supports line items has an ItemAddList method that is prefixed with the Request type name. For example, the SalesOrderAdd request object has a line item add list named SalesOrderItemAddList.

The \*ItemAddList method returns the line item object so you can proceed to fill its property values, as shown below:

#### **`Append each line item object and fill its values**

```
Dim salesOrderItemAdd1 As ISalesOrderItemAdd
Set salesOrderItemAdd1 = salesOrderAdd.SalesOrderItemAddList.Append

salesOrderItemAdd1.TaxCode.SetValue "val"
salesOrderItemAdd1.Desc1.SetValue "val"
salesOrderItemAdd1.ExtendedPrice.SetValue 2.00
salesOrderItemAdd1.Qty.SetValue 2.00
salesOrderItemAdd1.Associate.SetValue "val"
salesOrderItemAdd1.ListID.SetValue "val"
```

### 4: Sending Requests to QBPOS

Once the request message set is fully constructed you can send it to QBPOS using the QBPOSSessionManager method DoRequests. You need to be connected to QBPOS and have a valid QBPOS session before invoking this method.

Notice that DoRequests returns a response object. This returned object needs to be processed as described under “Processing a Response Using qbposFC.”

#### **` Start the QBPOS session and send the request set to QBPOS**

```
sessionManager.OpenConnection "IDN", "Sample qbposFC Code"
sessionManager.BeginSession "qbposfilename"

Dim responseMsgSet As IMessageSetResponse
Set responseMsgSet = sessionManager.DoRequests(requestMsgSet)
```

## Processing a qbposXML Response

---

The response XML string is returned from the ProcessRequests API call that sent the original request set to QBPOS. The suggested way to process that qbposXML response message set is to load it into a DOM document and walk its node list as shown in the following pseudocode and Visual Basic code sample.

### Processing a Response Message Set: Pseudocode

---

Figure 5-4 shows the general process of getting response data using DOM documents.

### Processing qbposXML Responses Pseudocode

```
Instantiate a DOM Document
Load the response XML string into the DOM document
Return the responses into a DOM node list
Foreach response in the list
  Get the response attributes (status code, etc)
  Walk the node list to get the response child nodes (elements) and data
```

Figure 5-4 Processing qbposXML Responses

## Processing a Response Message Set: Sample Code

---

The following code snippet shows how to load the response message set into the DOM document and walk the node list for response data. Notice that there can be more than one response in the message set to process.

```
Dim retStatusCode As String
Dim retStatusMessage As String
Dim retStatusSeverity As String
' Create xmlDoc Obj

Dim xmlDoc As New XmlDocument40
Dim objNodeList As IXMLDOMNodeList
' Node objects
Dim objChild As IXMLDOMNode
Dim custChildNode As IXMLDOMNode

Dim attrNamedNodeMap As IXMLDOMNamedNodeMap
Dim i As Integer
Dim ret As Boolean
Dim errorMsg As String

' Get CustomerAddRs nodes list
Set objNodeList = xmlDoc.getElementsByTagName("CustomerAddRs")
For i = 0 To (objNodeList.length - 1)
  ' Get the CustomerRetRs
  Set attrNamedNodeMap = objNodeList.Item(i).Attributes
  ' Get the status Code, info and Severity
  retStatusCode = attrNamedNodeMap.getNamedItem("statusCode").nodeValue
  retStatusSeverity = attrNamedNodeMap.getNamedItem
    ("statusSeverity").nodeValue
  retStatusMessage = attrNamedNodeMap.getNamedItem
    ("statusMessage").nodeValue
```

```

' Walk through the child nodes of CustomerAddRs node
For Each objChild In objNodeList.Item(i).childNodes
    ' Get the CustomerRet block
    If objChild.nodeName = "CustomerRet" Then
        ' Get the elements in this block
        For Each custChildNode In objChild.childNodes
            If custChildNode.nodeName = "ListID" Then
                resListID = custChildNode.Text
            ElseIf custChildNode.nodeName = "Name" Then
                resCustName = custChildNode.Text
            ElseIf custChildNode.nodeName = "FullName" Then
                resCustFullName = custChildNode.Text
            End If
        Next
    End If ' End of customerRet
Next ' End of customerAddret
Next

```

## Processing a Response Using qbposFC

---

Processing responses using qbposFC is somewhat like building requests, only in reverse. The DoRequests method used to send the requests to QBPOS returns an IMsgSetResponse object. This object contains all of the response data. We'll describe how to get at the data in this object in the following sections.

### IMsgSetResponse: the Central Object in Processing Responses

---

If you look at the QBPOSSessionManager method (DoRequests) used to send requests to QBPOS, you'll notice that the method returns an IMsgSetResponse object. This object is always returned unless DoRequest fails. IMsgSetResponse contains a response list that has one or more response objects, as shown in Figure 5-5.

### The Response Message Set Object

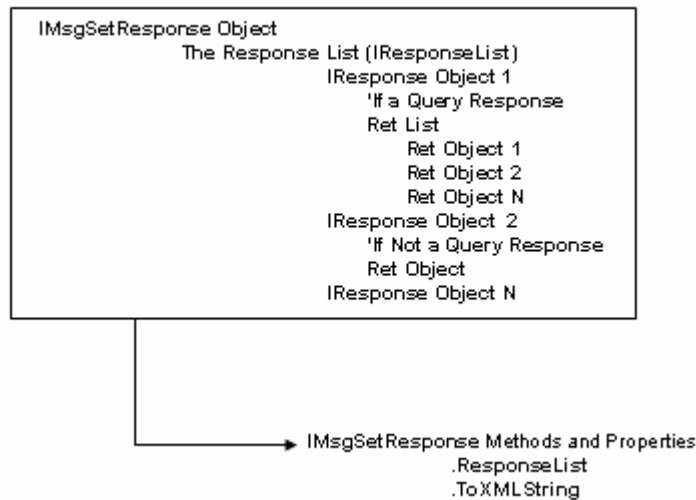


Figure 5-5 Response Message Set Structure

As shown in the figure, a response object that is a query response contains a Ret list object that contains potentially multiple Ret objects. A response object that is not a query response contains only one Ret object and no Ret list. This difference is crucial when it comes to processing the response data. We'll describe these in more detail shortly.

Also as shown in the figure, you'll need to be aware of these response message set methods:

- The `ResponseList` method returns the `IResponseList` object containing the individual response objects in the message set. `IResponseList` has a count property and a `GetAt` (index) method to support retrieval of responses from the list.
- The `ToXMLString` returns a complete and valid qbposXML string that represents the response message set object and all its individual responses. This is useful if you want to dump out the response message to see the contents either for diagnostic purposes or if you want to make sure you are getting the data you expect.

## How Do I Get Data from the Response Message Set?

To get the individual responses contained in the response message set, you use the `ResponseList` method to get the `IResponseList`, then you use that list's `Count` and `GetAt` methods to get the individual `IResponse` objects.

Getting the data from `IResponse` is a bit trickier, and requires a bit more detailed information about `IResponse`. Figure 5-6 shows the `IResponse` object.

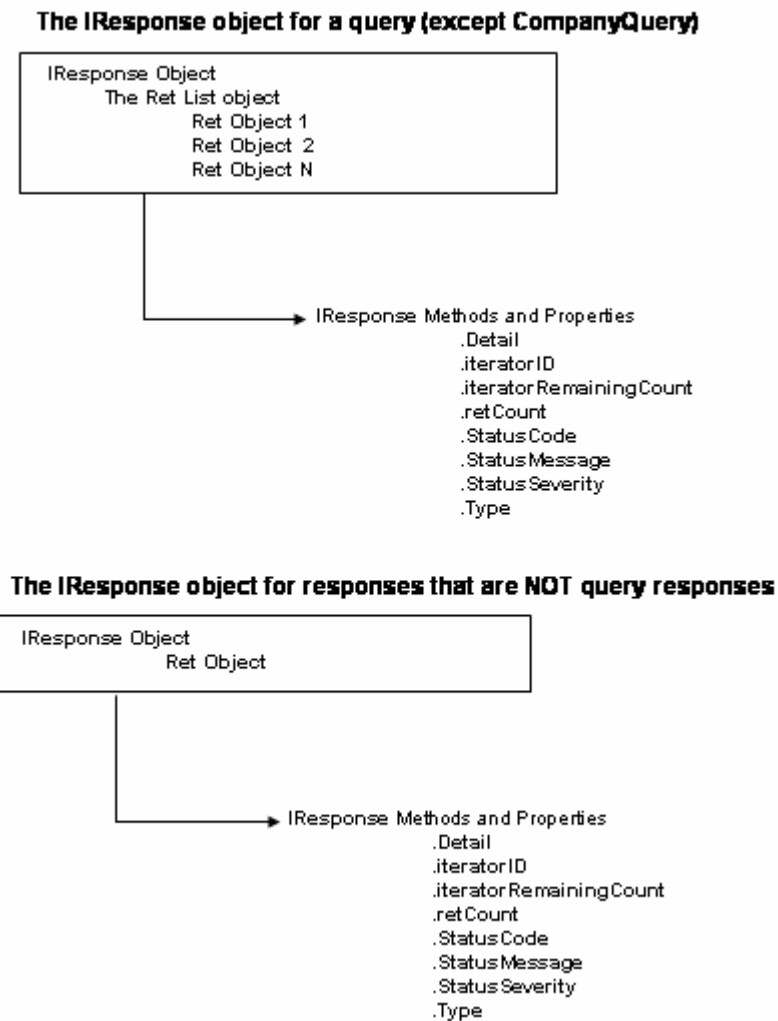


Figure 5-6 The IResponse Object

As shown in the figure, a query response object has a Ret list object containing one or more Ret objects. The retCount, iteratorID, and iteratorRemainingCount properties are provided to support getting the individual Ret objects from the Ret list.

The non query response has no ret list. It has only the single Ret object, from which data can be extracted.

The IResponse object has these methods and properties:

- The Detail method does different things depending on whether the response is a query response or not. For a query response, the Detail method returns the Ret list, which can be looped through to get the individual Ret objects. For non query responses, Detail returns the Ret object itself.
  - Notice that each request has a corresponding Ret object, for example, ItemInventoryAddRq has the response Ret object of IInventoryRet.

- Notice also that CompanyQuery is a special request that is treated as a non-query request and response
- The iteratorID property returns the iterator ID, which is used only for query responses whose originating request was set up to use iterators. For information on iterators, see Chapter 7, “Creating Queries.”
- The iteratorRemainingCount property returns the number of objects remaining in the iteration. This is used only for query responses whose originating request was set up to use iterators.
- The retCount property returns the number of Ret objects contained in the Ret list. It is used only for query responses.
- The StatusCode property returns the status code of the response. Every response has a status code indicating success (the value zero) or a non zero code that indicating the nature of the failure.
- The StatusMessage property returns the a text description of the status code in the response. Every response has a status message.
- The StatusSeverity property returns the severity level of the error.
- The Type property identifies the response type. This is useful for making sure that the response you are processing is of the expected type. You can invoke GetValue to get the type as an enumerated value, which is the easiest way to do a comparison with an expected type. Alternatively, you can invoke GetAsString to get the string representation of the type.

## Getting Response Data: Pseudocode

Figure 5-7 shows the generalized method for getting response data.

### Getting Response Data Pseudocode

```

IMsgSetResponse Object
Get the Response List Object (IMsgSetResponse.ResponseList)
Get each IResponse object from the IResponseList
For index = 0 To IResponseList.Count - 1
    IResponse = IResponseList.GetAt(index)

    Check the status code. If there are any errors, there
    normally is no Detail data.

    If IResponse is not a query response get the Ret object
        RetObject = IResponse.Detail
        Then get desired data from Ret object fields

    If IResponse IS a query response get the Ret list
        RetList = IResponse.Detail
        Get the Ret object from the list
        For j = 0 To RetList.RetCount - 1
            RetObject = RetList.GetAt(j)
            Then get desired data from Ret object fields
  
```

Figure 5-7 Getting Response Data



## Getting Response Data: Sample Code

The following code snippet shows the response to an ItemInventoryAdd request. The response is not a query response so there is only one Ret object and no Ret list. In this sample, notice the response type. You can look up all the response types in the Visual Studio object browser under the ENResponseType or you can construct them from the response name by prefixing the response name with rt, for example, the type for an ItemInventoryAddRs response is rtItemInventoryAddRs.

Also, the OSR is useful in looking up the Ret object and Ret list names for each Response. The Ret or Ret list for the response is listed at the top of the OSR entry for each response.

```
'Send the request message and get the response message: then get the first
'response message. This sample only expects one response
Dim responseMsgSet As IMessageSetResponse
Dim response As IResponse
Set responseMsgSet = sessionManager.DoRequests(requestMsgSet)
Set response = responseMsgSet.ResponseList.GetAt(0)

'Make sure the response type is the expected ItemInventoryAddRs type
'then get the expected IItemInventoryRet object data: we only want ListID
Dim itemInventoryListID As String
If (Not response.Detail Is Nothing) Then
    Dim responseType As Integer
    responseType = response.Type.GetValue

    Dim j As Integer
    'Notice that we make an implicit upcast here (supported in VB),
    'upcasting the detail to the itemInventoryRet type. In other languages,
    'you must do an explicit upcast, for example, in VB.Net:
    'itemInventoryRet = response.Detail as itemInventoryRet
    If (responseType = rtItemInventoryAddRs) Then
        Dim itemInventoryRet As IItemInventoryRet
        Set itemInventoryRet = response.Detail
        itemInventoryListID = itemInventoryRet.ListID.GetValue
    End If
End If
```

The code snippet below shows how to get data from a query response, which does have a Ret list of Ret objects.

```
'Send the request message and get the response message: then get the first
'response message. This sample only expects one response
Dim responseMsgSet As IMessageSetResponse
Dim response As IResponse
Set responseMsgSet = sessionManager.DoRequests(requestMsgSet)
Set response = responseMsgSet.ResponseList.GetAt(0)

'We're expecting an ItemInventoryQuery response.
If (Not response.Detail Is Nothing) Then
    Dim responseType As Integer
    responseType = response.Type.GetValue
    Dim j As Integer
```

```

'Get the ret list
If (responseType = rtItemInventoryQueryRs) Then
    Dim itemInventoryRetList As IItemInventoryRetList
    Set itemInventoryRetList = response.Detail
    Dim itemInventoryRet As IItemInventoryRet
    ItemFlexGrid.Rows = (1 + itemInventoryRetList.Count)

    'load each ret object from the list into a separate row in the
    'grid control to display certain ret object fields
    For j = 0 To itemInventoryRetList.Count - 1
        Set itemInventoryRet = itemInventoryRetList.GetAt(j)
        Set itemInventoryRet = itemInventoryRetList.GetAt(j)
        If (Not itemInventoryRet.Desc1 Is Nothing) Then
            ItemFlexGrid.Col = 0
            ItemFlexGrid.Row = j + 1
            ItemFlexGrid.Text = itemInventoryRet.ListID.GetValue
            ItemFlexGrid.Col = 1
            ItemFlexGrid.Text = itemInventoryRet.DepartmentListID.GetValue
            ItemFlexGrid.Col = 2
            ItemFlexGrid.Text = itemInventoryRet.Desc1.GetValue
            ItemFlexGrid.Col = 3
            ItemFlexGrid.Text = itemInventoryRet.Price1.GetValue
        End If
    Next j
End If
End If

```

In the preceding code sample, an item inventory query is sent to get a list of inventory items and display the list ID, department, description, and price from each item in a VB flex grid control. We just get the ret list, and in the same loop that we use to go through the ret list, we also populate each row in the flex grid control.

## CHAPTER 6

# MACROS, DATA EXTENSIONS, OBJECT DELETION

This chapter provides details on certain useful SDK features, such as using macros in your requests, accessing custom data fields that are accessible also from the UI, adding private data, and deleting objects. Accessing custom data fields and adding private data both require the use of data extensions, which are described in this chapter as well.

## What are Macros and How Do You Use Them?

---

A *macro* is a request attribute used to define an element that is created in an Add request, so that the element can subsequently be used (via an object reference) in a later request in the same message set. For example, you could create an inventory item using the macro attribute and then later in that same message set refer to that inventory item in a SalesOrderAdd request.

Macros eliminate the need to wait and parse the response to obtain the ListID for an element before you reference it in a subsequent request. The macro name defined in the macro attribute is used to represent the item ListID that QBPOS will assign to the item when it actually creates it.

Listing 6-1 shows how to use macros. Notice how the macro attribute is constructed in the request:

```
<ItemInventoryAdd defMacro="ListID:Inv1">
```

You use the defMacro attribute to set the macro: notice the macro name defined in the attribute setting (ListID:Inv1). When you want to refer to that macro, you use the macro attribute useMacro, as shown in the listing:

```
<ListID useMacro="ListID:Inv1" />
```

The macro name defined is used by the SDK runtime to substitute the actual ListID created in QBPOS as a result of the Add request when it encounters the corresponding useMacro attribute later in the message set.

Listing 6-1 Using Macros in Add Requests

```
<?xml version="1.0" ?>
<?qbposxml version="1.0"?>
<QBPOSXML>
  <QBPOSXMLMsgsRq onError="stopOnError">
    <ItemInventoryAddRq requestID="1">
      <ItemInventoryAdd defMacro="ListID:Inv1">
        <DepartmentListID>1000000001</DepartmentListID>
```

```

    <Desc1>SDK Item Sales Order</Desc1>
</ItemInventoryAdd>
</ItemInventoryAddRq>
<SalesOrderAddRq requestID = "63">
<SalesOrderAdd>
    <CustomerListID>3033159058971328769</CustomerListID>
    <Instructions>Instructions</Instructions>          <!-- opt, field max = 2000 -->
    <PromoCode>PromoCode</PromoCode>                  <!-- opt, field max = 10 -->
    <!-- SalesOrderType is one of these: SalesOrder,Layaway,WorkOrder -->
    <SalesOrderType>SalesOrder</SalesOrderType>        <!-- opt -->
    <Associate>Associate</Associate>                    <!-- opt, field max = 40 -->
    <Cashier>Cashier</Cashier>                          <!-- opt, field max = 40 -->
    <TxnDate>2005-4-13</TxnDate>                        <!-- opt -->
    <!-- PriceLevelNumber is one of these: 1,2,3,4 -->
    <PriceLevelNumber>2</PriceLevelNumber> <!-- opt -->
    <SalesOrderItemAdd>                                <!-- opt, may rep -->
        <ALU>ZnUBHoiKG!NssbB!GBW3</ALU>                <!-- opt, field max = 20 -->
        <UPC>990742993074670439</UPC>                  <!-- opt, field max = 18 -->
        <Size>Size</Size>                                <!-- opt, field max = 8 -->
        <Attribute>Attribut</Attribute>                  <!-- opt, field max = 8 -->
        <Desc2>Desc2</Desc2>                            <!-- opt, field max = 30 -->
        <Desc1>Desc1</Desc1>                            <!-- opt, field max = 30 -->
        <ExtendedPrice>5</ExtendedPrice>                <!-- opt -->
        <Qty>2</Qty>                                      <!-- opt -->
        <Associate>Associate</Associate>                  <!-- opt, field max = 40 -->
        <ListID useMacro="ListID:Inv1" />
    </SalesOrderItemAdd>
</SalesOrderAdd>
</SalesOrderAddRq>
</QBPOSXMLMsgsRq>
</QBPOSXML>

```

## What are Data Extensions and How Do I Add Them?

Occasionally, a QBPOS user may need to add some custom data to some customer, vendor, employee, or inventory item that is not currently available in the UI. To satisfy this need, QBPOS provides custom fields for these objects. The user simply goes into company preferences and sets the My Field Labels for the object type (customer, and/or vendor and so forth). Alternatively, the user can go into the object's New or Edit forms and do the same thing in the Notes and Custom tab. Using either method, the custom field is immediately visible and available for use by the QBPOS interactive user.

Data extensions are how the SDK programmatically accesses those *public* custom fields to read and write data to them. (We'll describe this in detail in a moment.) We call the custom fields *public* because any authorized application can read and write to them.

However data extensions can also be used to store private data in an object. This private data is not visible either in the QBPOS UI or to other applications unless you provide the GUID needed to read and write. (More on this later.) Unlike the public custom fields, private data extensions can be stored in almost every object, not just customer, vendor, employee, and inventory item.

## What are DataExt and DataExtDef?

---

The term *data extension* refers to the whole data extension feature. DataExt and DataExtDef are how you actually use the feature. To see how these work together, let's take a look at a private data extension. To add the private extension to, for example, a customer, you must first define the data extension and attach it to Customer using DataExtDefAddReq. That definition is now available for all customers. When you write the private data for a specific customer, say John Williams, you use that existing definition in a DataExtAdd request, which does the write.

Notice that you *do not* use DataExtDefAdd or Mod for custom fields. The data extension definition is done automatically when the QBPOS user selects one of the custom fields. For custom fields, the DataExtType is always STR255TYPE (a maximum 255 characters string). The DataExtName is whatever the QBPOS user enters for the custom field name, and the OwnerID is always 0 for custom fields.

## How Do I Read and Write Data to a Custom Field?

---

An application cannot read or write to any custom field unless the QBPOS user first adds it to one or more object type (customer and/or vendor, and so forth). Notice that the SDK does not currently allow an application to perform this task. Since your application cannot know any custom fields in advance, how does an application discover the name of the custom fields that can be written to?

To find out the list of custom fields for a given object, you perform a DataExtDefQuery, supplying the type of object you are searching, via the AssignToObject field. The returned list contains all of the public data extensions for that object type. (A key field within each Ret object is the DataExtName, which contains the name of the custom field.) This will give you the public fields for a particular object type. If you want to use a broader net, using OwnerID with a value of 0 will return all public data extensions for every object type.

Once you have located (or your end user has selected) the desired custom field from the returned list, and once you have the ListID of the particular object you want to write data to, you can do a DataExtAdd or Mod.

Listing 6-2 shows how to query for all of the public data extensions for Customers. Notice that in addition to custom fields, other public data extensions may be available. Any data extension def created with an OwnerID of 0 will be public.

## Listing 6-2 Querying for Public Data Extensions

```
<?xml version="1.0" ?>
<?qbposxml version="1.1"?>
<QBPOSXML>
  <QBPOSXMLMsgsRq onError="stopOnError">
    <DataExtDefQueryRq requestID = "13">
      <AssignToObject>Customer</AssignToObject>
      <DataExtType>STR255TYPE</DataExtType>
    </DataExtDefQueryRq>
  </QBPOSXMLMsgsRq>
</QBPOSXML>
```

If you wanted to query for private data extensions, you would have to use the OwnerID field in the query, instead of the AssignToObject field, and you would have to specify the OwnerID of that private extension. It won't be 0, as we'll show later. You can also cause data extension data to be returned in other queries, for example CustomerQuery, by including the OwnerID field set to the desired ID (0 for public, some other GUID for private extensions).

Listing 6-3 shows how to write to a data extension that doesn't contain any data yet. You'll use the DataExtAdd request in this case. If the data extension already has data, you'll have to use the DataExtMod request, which is very similar.

## Listing 6-3 Writing Data to the Public Data Extension

```
<?xml version="1.0" ?>
<?qbposxml version="1.1"?>
<QBPOSXML>
  <QBPOSXMLMsgsRq onError="stopOnError">
    <DataExtAddRq requestID = "13">
      <DataExtAdd>
        <OwnerID>0</OwnerID>
        <DataExtName>Route Number</DataExtName>
        <ListDataExtType>Customer</ListDataExtType>
        <ListObjRef>
          <ListID>3648633554738249985</ListID>
        </ListObjRef>
        <DataExtValue>Old Highway 99</DataExtValue>
      </DataExtAdd>
    </DataExtAddRq>
  </QBPOSXMLMsgsRq>
</QBPOSXML>
```

Looking at the sample above, notice that we can assume the OwnerID of 0. The DataExtName value we got in our query. The only other piece of information we need is the ListID of the customer to which we are adding the data extension data.

## How Do I Read and Write Data to Private Data Extensions?

---

Data extensions that are private have a GUID for the OwnerID, not the value 0. They can be accessed only by applications that know that OwnerID for the data extension. Through the SDK, your application can add, modify, query, and delete any data extensions whose OwnerID is known by your application. To query for application-defined data extensions, set the OwnerID to a valid GUID (as described in the following paragraphs).

Adding a private data extension is a two-part process:

1. First you have to define the data extension—that is, assign it a name, a data type, and an identifying GUID. (This is unlike custom data, which has only one data type, STR255TYPE.) Your application must specify the object types to which your data extension will be attached. This type can be specified when the data extension is first created, or the data extension definition can be modified later to add the object types to which this extension will be attached. Data extension definitions can be queried, modified, and deleted as well as added.
2. After a data extension is defined, you can add data to an existing object in QBPOS that is a member of the object types you defined the extension for. To add data, use a DataExtAdd request.

To delete data from an object that has a data extension, use a DataExtDel message. To query data from an object that has a data extension, use the Query request that corresponds to the type of object the extension is assigned to. For example, to get information about Customer data extensions within the QBPOS company, include your OwnerID (a GUID) in a CustomerQuery request.

### Limits of Private Data Extensions

The maximum length of a value added to QBPOS as a private data extension depends on the data type of the extension. For example, if DataExtType is STR255TYPE, the maximum length of DataExtValue is 255 characters. If DataExtType is STR1024TYPE, the maximum size of DataExtValue is 1k (that is, 1024 bytes). For more details, see the *Onscreen Reference*.

There is no limit to the number of private data extensions that a QBPOS object can have.

## Why Aren't Data Extensions Returned in My Queries?

---

By default, most queries do not return data extension data. You have to ask a query to return that kind of data, and the way you ask is to include the OwnerID field with the value of 0 for public extensions, or a valid OwnerID GUID for the private extensions identified by that GUID. You can only know your own private extensions or those that have been made available to you.

## How Do You Delete an Object?

---

Unlike the many forms of Add, Mod, and Query requests, which specify the object explicitly (for example, CustomerAdd, InventoryItemAdd, TransferSlipAdd, and so on), delete requests take only one form (ListDelRq). You cannot delete a whole list at once; instead, you delete individual objects in the list, such as a single employee or customer.

## Why Can't I Delete (or Mod) Some QBPOS Objects?

---

Certain transactions cannot be modified or deleted after they are created in QBPOS. For example, sales receipts, transfers, or vouchers cannot be modified or deleted either from the QBPOS user interface or via the SDK. The reason for this is that any such modification or deletion would jeopardize the transaction history.

Other objects such as purchase orders or sales orders do not represent finalized transactions and thus are not part of any transaction history. So these can be modified and deleted at will.



## CHAPTER 7

# CREATING QUERIES

This chapter provides details on how to use query filters and iterators to get the data you want in the most efficient way in a query request. As mentioned earlier in this guide, query requests retrieve data from QBPOS that meets criteria specified in one or more *query filters*. The use of filters enables you to

- Obtain a set of objects that meet specified criteria
- Limit the number of objects returned in a response to a manageable quantity

Along with filters, for queries that are expected to return large amounts of data, you should also use query iterators, which provide the most efficient way to walk through a large list of returned data.

Each of these QBPOS SDK features, query filters and iterators, is covered separately in this chapter.

## About Query Requests

---

If you scan through the OSR, you'll notice that each QBPOS object that supports queries has its own query request. For example, Customer has CustomerQuery, ItemInventory has ItemInventoryQuery, and so on.

If you look at the OSR a little more closely, you'll notice that the QBPOS SDK allows you to query on virtually any field in an object using the available filters. (If you are familiar with the QuickBooks SDK, you'll notice that these filters are a lot more flexible than their counterparts in the QB SDK.)

## Query Filter Groups and How They Work

---

If you haven't programmed in the QB SDK, certain aspects of the OSR query descriptions may be puzzling at first, especially if you are working in qbposFC. Take a look at the qbposFC OSR, for example, the CustomerQuery (Figure 7-1 on page 56 ).

Tag	Type
<b>ICustomerQuery</b>	
<b>iterator</b>	IQBENiteratorType
<b>iteratorID</b>	IQBUUIDType
<b>MaxReturned</b>	IQBIntType
<b>OwnerIDList</b>	IGUIDList
<b>ORNotesFilters</b>	IORNotesFilters
<b>NotesFilter</b>	INotesFilter
<b>MatchStringCriterion</b>	IQBENMatchStringCriterionType
<b>Notes</b>	IQBStringType
<b>NotesRangeFilter</b>	INotesRangeFilter
<b>FromNotes</b>	IQBStringType
<b>ToNotes</b>	IQBStringType
<b>UseWithQBFilter</b>	IQBBoolType
<b>ORCustomerDiscPercentFilters</b>	IORCustomerDiscPercentFilters
<b>CustomerDiscPercentFilter</b>	ICustomerDiscPercentFilter
<b>MatchNumericCriterion</b>	IQBENMatchNumericCriterionType
<b>CustomerDiscPercent</b>	IQBAmountType
<b>CustomerDiscPercentRangeFilter</b>	ICustomerDiscPercentRangeFilter

Figure 7-1 OSR OR filter lists in qbposFC

What does the ORNotesFilters item mean, the ORCustomerDiscPercentFilters mean, and so forth? (See the circled items in the figure.) Each of these identifies a *filter group* that can be set using one and only one of the individual filters listed under the group name. For example, to set a notes filter that looks for a certain ISBN string value, you could invoke the following in qbposFC:

```
customerQuery.ORMNotesFilters.NotesFilter.Notes.SetValue("ISBN 5432")
```

Or, you could set the filter to use one of the other filters within the filter group, in our example, you could set a range filter to get values within a range instead of a single value:

```
customerQuery.ORMNotesFilters.NotesRangeFilter.FromNotes.SetValue("ISBN 2344")
customerQuery.ORMNotesFilters.NotesRangeFilter.ToNotes.SetValue("ISBN 3500")
```

Notice that you can set only one of the filters listed within the filter group. If you tried to set both the NotesFilter and the NotesRangeFilter in the ORNotesFilters group, for example, you'd get a runtime error. The individual filters within the filter group are mutually exclusive.

In the qbposXML OSR, this becomes a little more obvious (see Figure 7-2 on page 57).

Tag	Type
<b>CustomerQueryRq</b>	
<b>MaxReturned</b>	INTTYPE
<b>OwnerID</b>	GUIDTYPE
BEGIN OR	
<b>NotesFilter</b>	
<b>MatchStringCriterion</b>	ENUMTYPE
<b>Notes</b>	STRTYPE
OR	
<b>NotesRangeFilter</b>	
<b>FromNotes</b>	STRTYPE
<b>ToNotes</b>	STRTYPE
END OR	
<b>UseWithQBFilter</b>	BOOCTYPE
BEGIN OR	
<b>CustomerDiscPercentFilter</b>	
<b>MatchNumericCriterion</b>	ENUMTYPE
<b>CustomerDiscPercent</b>	AMTTYPE
OR	
<b>CustomerDiscPercentRangeFilter</b>	
<b>FromCustomerDiscPercent</b>	AMTTYPE
<b>ToCustomerDiscPercent</b>	AMTTYPE

Figure 7-2 OSR OR filter lists in qbposXML

In qbposXML, the usage is a bit more transparent: the filter group is the set of items between the BEGIN OR and the END OR, with the individual filters separated by an OR. In qbposXML as well, the individual filters within the group are mutually exclusive.

But what happens if you want to use more than one filter in a request? Can you specify more than one filter if they are in different filter groups? The answer is yes, but this needs to be explained in a bit more detail.

## Specifying Multiple Filters

You can specify more than one filter in a query *if* they are from different filter groups. If you specify more than one filter, these filters are ANDed, with the result that your query yields increasingly more precise results as you add more filters.

## Building a Query

When you build a query, you need to consider which fields in the object best suit your needs. For example, in a CustomerQuery, you might want to include a FirstNameFilter and a LastNameFilter to restrict searches to a specified name.

## Sample qbposXML Query with Multiple Filters

---

The following sample customer query filters for all customers whose last name starts with K and whose first name is Henry.

```
<?xml version="1.0"?>
<?qbposxml version="1.0"?>
<QBPOSXML>
  <QBPOSXMLMsgsRq onError="stopOnError">
    <CustomerQueryRq requestID="1">
      <LastNameFilter>
        <MatchStringCriterion>StartsWith
      </MatchStringCriterion>
      <LastName>K
    </LastName>
    </LastNameFilter>
    <FirstNameFilter>
      <MatchStringCriterion>Equal
    </MatchStringCriterion>
    <FirstName>Henry
    </FirstName>
    </FirstNameFilter>
    </CustomerQueryRq>
  </QBPOSXMLMsgsRq>
</QBPOSXML>
```

## Sample qbposFC Query with Multiple Filters

---

The following Visual Basic sample customer query filters for all customers whose first and last name contains the strings obtained from text boxes (txtFN and txtLN, respectively).

```
Dim customerQuery As ICustomerQuery
Set customerQuery = requestMsgSet.AppendCustomerQueryRq
If txtFN.Text <> "" Then
  customerQuery.ORFirstNameFilters.FirstNameFilter.MatchStringCriterion.SetValue mscContains
  customerQuery.ORFirstNameFilters.FirstNameFilter.FirstName.SetValue txtFN.Text
End If

If txtLN.Text <> "" Then
  customerQuery.ORLastNameFilters.LastNameFilter.MatchStringCriterion.SetValue mscContains
  customerQuery.ORLastNameFilters.LastNameFilter.LastName.SetValue txtLN.Text
End If

Set responseMsgSet = sessionManager.DoRequests(requestMsgSet)
```

# The Importance of Limiting the Number of Objects Returned

When you issue queries that could return large amounts of data, you need to take care to limit the data returned. Otherwise, you can run into performance issues due to the large memory allocations and re-allocations as the string is built and marshalled across COM process boundaries. In addition, there is the issue of XML parse time based on size of the XML data being parsed.

The best way to limit the data returned from a query is to use the Iterator attribute that is available to most queries.

## **IMPORTANT**

If you use the Iterator attribute, you must also use MaxReturned to specify the size of each iteration.

## Using Iterators to Walk Through Large Query Returns

The iterator attribute provided with most query types also allows you to break down query results in smaller and more manageable chunks of data. An iterator results in responses that contain only the specified number of objects. Iterators are only valid for the application that starts them, and they are only valid for the current session with QBPOS.

### Starting an Iteration

How do iterators work? The iterator is created when a query contains the iterator attribute set to Start, along with a MaxReturn value specifying how many records are to be returned in each iteration. The response to that first query iteration contains, along with the response data, an IteratorID value that uniquely identifies that iterator. This is important because you can have many iterators active at the same time. The following example shows how to start an iteration

```
<?xml version="1.0" ?>
<?qbposxml version="1.0" ?>
  <QBPOSXML>
    <QBPOSXMLMsgsRq onError="stopOnError">
      <CustomerQueryRq requestID="5001" iterator="Start">
        <MaxReturned>10</MaxReturned>
        <AccountBalanceRangeFilter>
          <FromAccountBalance>100.00</FromAccountBalance>
        </AccountBalanceRangeFilter>
      </CustomerQueryRq>
    </QBPOSXMLMsgsRq>
  </QBPOSXML>
```

The response contains the returned objects, with the iteration-related information returned in the response attributes, as shown in the following snippet:

```
<CustomerQueryRs requestID = "5001" statusCode = "0" statusSeverity =
"INFO" statusMessage = "..." iteratorRemainingCount = "50" iteratorID =
"{D7355385-A17B-4f5d-B34D-F34C79C3E6FC}">
```

In the query responses, notice the attribute called `iteratorRemainingCount` that contains the number of objects left in the iteration. You must check for a value of 0 because that means not only that there are no more objects, but more importantly, it means that the Iterator is now removed from memory and therefore further attempts to access it will fail: its `iteratorID` is no longer valid. Other than checking for this remaining count value of 0, you can use this `iteratorRemainingCount` value any way you want, for example, to change the current `MaxReturned` settings.

### **IMPORTANT**

When the `remainingCount` value returned in the query response contains a value of 0, this indicates that the corresponding Iterator can no longer be used.

## **Continuing the Iteration**

To continue a particular iteration, you simply issue the same query request again, but this time with the iterator value now set to `Continue`, and the `IteratorID` field set to the `IteratorID` value returned from the first query iteration. A `qbposXML` example is shown below:

```
<?xml version="1.0" ?>
<?qbposxml version="1.0" ?>
  <QBPOSXML>
    <QBPOSXMLMsgsRq onError="stopOnError">
      <CustomerQueryRq requestID="5001" iterator="Continue"
        iteratorID="{D7355385-A17B-4f5d-B34D-F34C79C3E6FC}">
        <MaxReturned>10</MaxReturned>
        <AccountBalanceRangeFilter>
          <FromAccountBalance>100.00</FromAccountBalance>
        </AccountBalanceRangeFilter>
      </CustomerQueryRq>
    </QBPOSXMLMsgsRq>
  </QBPOSXML>
```

Once a query has been issued with an iterator, subsequent iterations (i.e., queries using that same `iteratorID` and the iterator attribute set to `Continue`) cannot change any filtering. The only thing that can be changed during an iteration is the `MaxReturned` value.

## **Stopping the Iteration**

At any point during an iteration, you can stop the iteration and destroy the iterator (freeing up memory) by issuing the query request with the iterator attribute set to `Stop` and the `iteratorID` set to the proper iterator ID. You should always do this if you want to stop an iteration before it is complete, otherwise the iterator will continue to be held in memory.

### IMPORTANT

If you exhaust the iteration, so that the last query shows an iteratorRemainingCount value of 0, you need not and in fact cannot issue the query with the iterator attribute set to "Stop." The iteratorID you would have to supply is no longer valid.

## Miscellaneous Details on Query Filters

### Default Values for Date/Time

The following tables list default values assigned by the SDK when all or part of the time is not specified. When a time zone is not specified, local time is assumed.

Table 7-1 Default values for "From" date/times

"From" is specified	Start period	Example
Date only	Beginning of the day	10/1/02 means 10/1/02 00:00:00
Date and hour only	Beginning of the hour	10/1/02 8 means 10/1/02 8:00:00
Date, hour, and minute only	Beginning of the minute	10/1/02 8:20 means 10/1/02 8:20:00
Date, hour, minute, and second	The second	10/1/02 8:20:40 means 10/1/02 8:20:40

NOTE: If "From" is not specified at all, the start period is the earliest date possible.

Table 7-2 Default values for "To" date/times

"To" is specified	Start period	Example
Date only	End of the day	10/1/02 means 10/1/02 23:59:59
Date and hour only	Beginning of the hour	10/1/02 8 means 10/1/02 8:00:00
Date, hour, and minute only	Beginning of the minute	10/1/02 8:20 means 10/1/02 8:20:00
Date, hour, minute, and second	The second	10/1/02 8:20:40 means 10/1/02 8:20:40

NOTE: If "To" is not specified at all, the end period is up to the last date used.

### Date Ranges

The acceptable range of dates are from 1970-01-01 to 2038-01-19T03:14:07 (2038-01-18T19:14:07-08:00 PST).

## Match Criterion for Names

---

Another useful filter available in most list queries are the various name filters, which allow you to obtain objects that match a specified string. You specify both the string itself and the `MatchCriterion`, which can be `StartsWith`, `Contains`, `Equal`, or `EndsWith`. For example, you could use this filter with a customer query and request all customers whose names start with “Mac.”

## Ranges for Names

---

Another alternative is to create a query that specifies a *range* of names for the objects to be returned. Using a name range filter, you can specify both the starting and ending points of the range, only the start of the range, or only the end of the range. String values for names are case-insensitive. In a sorted list, punctuation characters are first, followed by numeric characters, and then alphabetical characters. Values progress from 0 to 9 and from A to Z. If both `FromName` and `ToName` are specified, `ToName` must be lexicographically higher than `FromName`. See the section “The Importance of Limiting the Number of Objects Returned,” beginning on page 59, for information on how to use the name range filter in conjunction with `MaxReturned`.

## Understanding the Time Modified Field

---

Queries can return the time modified field if the object being queried has been modified. If an object is created but not modified, there is no `TimeModified` value. If you are used to programming in the QBSDK, you’ll notice that this behavior is different, as the creation of an object does set the time modified field for QBFS.



## CHAPTER 8

# MOVING ITEMS INTO AND OUT OF INVENTORY: SALESORDERS PURCHASEORDERS, VOUCHERS AND SALESRECEIPTS

This chapter describes the process of receiving inventory items from vendors using sales orders, purchase orders and receiving vouchers. The chapter also covers the process of removing items from inventory using return vouchers and sales receipts.

Notice that if the QBPOS administrator sets the QBPOS company preferences to share data with a QuickBooks company, voucher and sales receipt data can be shared. Purchase order and sales order data is not shared with QuickBooks.

## Process Overview

---

Typically, a business receives items purchased from vendors into its own inventory using purchase orders and receiving vouchers (see Figure 8-1.) The purchase order is used to obtain the items from the vendor and the receiving voucher is used to receive items into inventory against that purchase order. In some cases, as shown in the figure, this process can be initiated by a customer sales order that orders some item not currently in stock, with the purchasing order being generated from the sales order later. (This is a QBPOS UI feature, but you could also do the same thing indirectly via the SDK.)

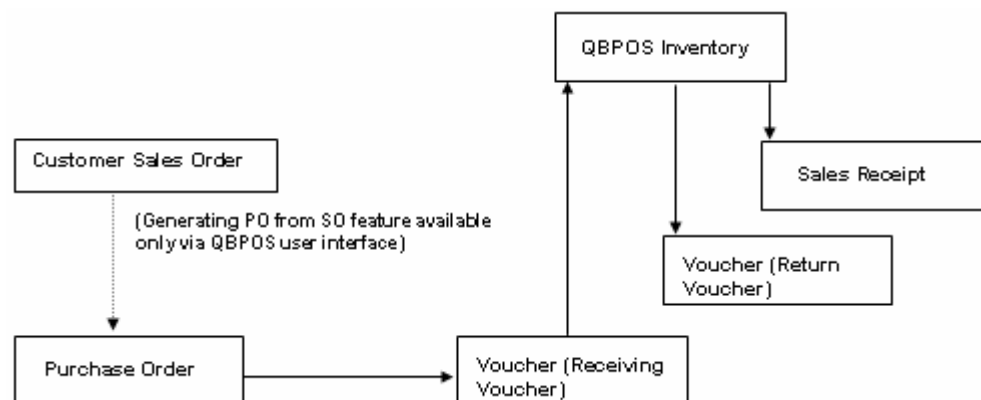


Figure 8-1 Receiving Items into inventory

Items are moved out of inventory when customers purchase items or when items are returned to the vendor, as shown in Figure 8-1. You use a return voucher to return items to the vendor and a sales receipt to record a sale to a customer.

Notice that receiving items against purchase orders is a common way to receive items from vendors. However, you can use vouchers to receive items without using purchase orders.

## Vouchers Update PurchaseOrder, SalesReceipts Update SalesOrder

---

If purchase orders are used, inventory items received against them with the receiving voucher in the SDK (VoucherAdd) contain a reference (purchase order TxnID) to the purchase order being received against. The VoucherAdd request, *if purchase orders are used*, thus automatically updates several fields in the corresponding purchase order, such as PurchaseOrderStatus, QtyReceived, QtyDue, UnfilledPercent, and of course, TimeModified.

If sales orders are used, sales receipts against a sales order also contain a reference to that sales order (sales order TxnID). The SalesReceiptAdd request, if sales orders are used, thus automatically updates several fields in the SalesOrder, such as SalesOrderStatus, UnfilledPercent, and BalanceDue.

## Creating Sales Orders

---

The term “SalesOrder” in the SDK may lead you to believe that the SDK supports only sales orders, and not layaways or work orders. However, you should be aware that what the SDK calls “SalesOrder” is actually a “customer order” in QBPOS. Customer order is a feature available from the UI only in QBPOS Pro and greater. (However, customer order is available via the SDK in all QBPOS editions.) There are several types of customer orders supported in the SDK:

- Sales orders, which are typically customer orders of out-of-stock merchandise.
- Work orders, which can be quotes for services.
- Layaways, which are sales of in-stock items, but paid for via a series of deposits prior to customer receiving the items.

These customer orders are added, modified, and deleted using the SDK requests SalesOrderAdd, SalesOrderMod, and ListDel, respectively. You specify the type of SalesOrder (customer order) you want via the SalesOrderType field.

## Creating a SalesOrder

---

Take a look at the New Sales Order form (or New Work Order or New Layaway) in the QBPOS UI, which is posted if you select Point of Sale->New Sales Order from the main QBPOS menubar. (See Figure 8-2). If you compare the UI to the OSR listing of the request fields, you'll notice that most of the SDK's SalesOrderAdd request fields map directly to the form. Most of this mapping is self explanatory: Desc1 and Desc2 map to Description 1 and Description 2 on the form, PromoCode maps to Promo Code, and so forth.

However, other UI form features are replaced in the SDK request. For example, the UI's customer lookup (bottom left in the figure) is replaced by the CustomerListID field, which specifies a particular customer. This is also true for the Item lookup in the upper left of the form, where the UI lookup is replaced by a reference (ListID) to a specific QBPOS item.

Notice the UI components that are greyed out in the New Sales Order form, such as Sales Order #, Deposit Balance, and Balance Due. These represent data that can appear only after adding the new sales order. Sales Order # is generated only after the sales order is successfully created: in the SDK, this number is returned in the SalesOrderAdd response. Deposit Balance and Balance Due in the UI form are filled in automatically if a deposit receipt is created and linked to that sales order.

**New Sales Order**

Edit Item Delete Item Find Item E-mail Help

Sales Order #  Deposit Balance  Associate

Status  Balance Due  Cashier

Enter Item(s)

Item #	Description 1	Qty	Sold	Qty Due	Price	Disc %	Disc \$	Disc T...

Customer

Address

Promo Code

Instructions

Price Level  SubTotal

Disc %  Disc \$

Tax %  Tax \$

Shipping

**Total**

Sales Order Tasks Print Save Cancel

Figure 8-2 The New Sales Order form

Finally, notice the discount fields in the UI form. The line item-level discount is located in the line item area: you can apply discounts to any of the line items. The order-level discount is located in the bottom right of the New Sales Order form: you can apply this discount only to the subtotal of the line items. (It is an additional discount to any line item discounts.)

## Rules for Using SalesOrder Exclusive Fields

SalesOrder contains certain fields that are mutually exclusive, which means that you cannot set both fields in an SDK request. The fields are exclusive because filling one field automatically causes the other field to be calculated by QBPOS.

The following fields are exclusive and cannot both be set in the same SalesOrder request:

- The DiscountAmount and DiscountPercentage in the same line item are exclusive.
- Any discount field and any price field in the same line item are exclusive.
- The price field and the extended price field in the same line item are exclusive.
- The order-wide Discount and DiscountPercent fields applied to the SalesOrder subtotal are exclusive.

If you try to set fields that are exclusive, you get a runtime error.

## Creating a SalesOrder Using qbposXML

Listing 8-1 shows a construction of a sales order in qbposXML. This sample is a special order for a new item not yet carried in inventory. Accordingly, the inventory item is created first in the message set using the macro attribute:

```
defMacro="ListID:Inv1"
```

That item is subsequently referenced by its macro name in the SalesOrderAdd line item that is in boldface font in the listing:

```
<ListID useMacro="ListID:Inv1" />
```

However, the listing does not show all of the possible request fields. For that information, please refer to the OSR.

### Listing 8-1 Constructing a Sales Order in qbposXML

```
<?xml version="1.0" ?>
<?qbposxml version="1.0"?>
<QBPOSXML>
  <QBPOSXMLMsgsRq onError="stopOnError">
    <ItemInventoryAddRq requestID="1">
      <ItemInventoryAdd defMacro="ListID:Inv1">
        <DepartmentListID>1000000001</DepartmentListID>
        <ALU>ZnUBHoIKG!NssbB!GBW3</ALU>      <!-- opt, field max = 20 -->
        <UPC>990742993074670439</UPC>        <!-- opt, field max = 18 -->
        <Size>Small</Size>                    <!-- opt, field max = 12 -->
      </ItemInventoryAdd>
    </ItemInventoryAddRq>
  </QBPOSXMLMsgsRq>
</QBPOSXML>
```

```

        <Attribute>Black</Attribute>      <!-- opt, field max = 12 -->
    <Cost>1.00</Cost>
    <Price1>3.00</Price1>
    <Desc2>Batched</Desc2>               <!-- opt, field max = 30 -->
    <Desc1>SDK Item Sales Order</Desc1>   <!-- opt, field max = 30 -->
</ItemInventoryAdd>
</ItemInventoryAddRq>
<SalesOrderAddRq requestID = "63">
<SalesOrderAdd>
    <CustomerListID>3033159058971328769</CustomerListID>
    <Instructions>Instructions</Instructions>      <!-- opt, field max = 2000 -->
    <PromoCode>PromoCode</PromoCode>              <!-- opt, field max = 10 -->
    <!-- SalesOrderType is one of these: SalesOrder,Layaway,WorkOrder -->
    <SalesOrderType>SalesOrder</SalesOrderType>    <!-- opt -->
    <Associate>Associate</Associate>               <!-- opt, field max = 40 -->
    <Cashier>Cashier</Cashier>                     <!-- opt, field max = 40 -->
    <TxnDate>2005-4-13</TxnDate>                  <!-- opt -->
    <!-- PriceLevelNumber is one of these: 1,2,3,4 -->
    <PriceLevelNumber>2</PriceLevelNumber>          <!-- opt -->
    <SalesOrderItemAdd>                            <!-- opt, may rep -->
        <Qty>2</Qty>                                <!-- opt -->
        <ListID useMacro="ListID:Inv1" />
    </SalesOrderItemAdd>
</SalesOrderAdd>
</SalesOrderAddRq>
</QBPOSXMLMsgsRq>
</QBPOSXML>

```

In Listing 8-1, notice that the only SalesOrder line item information that is required is the quantity ordered and the item ordered (referenced by its ListID). No other information is needed. The inventory item should contain all the other data required for the sales order line item. QBPOS will automatically fill in and calculate all of the other line item data from the referenced inventory item, such as price, UPC, ALU, and so forth.

## Creating a SalesOrder Using qbposFC

Creating a SalesOrder in qbposFC is the same as creating one in qbposXML so far as the fields are concerned. The same fields are available to qbposFC as are available in qbposXML. The only difference is that you don't need to construct any XML, but simply use the qbposFC objects and object properties to build the request.

Listing 8-2 shows a Visual Basic example that builds a Sales Order.

### Listing 8-2 Building a SalesOrder in qbposFC: VB Sample

```

'Define some qbposFC objects we need to begin construction:
Dim sessionManager As New QBPOSSessionManager
Dim xmlMajorVersion As String

```

```

Dim xmlMinorVersion As String
Dim responseMsgSet As IMsgSetResponse
xmlMajorVersion = "1"
xmlMinorVersion = "0"

'Create the message set and append the SalesOrder request. We just added this one request
'to the message set: you could add as many requests as you want, using the appropriate
'Append method for each request.
Set requestMsgSet = sessionManager.CreateMsgSetRequest(xmlMajorVersion, xmlMinorVersion)
requestMsgSet.Attributes.OnError = roeContinue
Dim SalesOrderAdd As ISalesOrderAdd
Set SalesOrderAdd = requestMsgSet.AppendSalesOrderAddRq

'When you append a request, the corresponding request object is returned for you to populate.
'You must set the properties in this object as desired before executing the request.
SalesOrderAdd.SalesOrderType.SetValue ENSalesOrderType.sotSalesOrder
SalesOrderAdd.DiscountPercent.SetValue (5)
SalesOrderAdd.CustomerListID.SetValue ("3293374402841706753")
SalesOrderAdd.PriceLevelNumber.SetValue (pln1)
SalesOrderAdd.ShippingInformation.FullName.SetValue ("Jack Higgins")
SalesOrderAdd.ShippingInformation.Street.SetValue ("123 Hopkins")
SalesOrderAdd.ShippingInformation.City.SetValue ("Mountain View")
SalesOrderAdd.ShippingInformation.State.SetValue ("CA")
SalesOrderAdd.ShippingInformation.PostalCode.SetValue ("94043")
SalesOrderAdd.ShippingInformation.ShipBy.SetValue ("UPS")
SalesOrderAdd.ShippingInformation.Shipping.SetValue (9.53)
SalesOrderAdd.Instructions.SetValue ("Please Rush")
SalesOrderAdd.TxnDate.SetValue (txtTDate)

'We've finished setting the main object properties. However, SalesOrder normally has
'line items, so we now add one line item. You can add more just by repeating what we
'do here. Price and Quantity are obtained from UI components not defined here.
Dim salesOrderItemAdd As ISalesOrderItemAdd
Set salesOrderItemAdd = SalesOrderAdd.SalesOrderItemAddList.Append
salesOrderItemAdd.ListID.SetValue ListID

'Only Quantity is required. We also set Price, because we are allowing the default
'item price to be replaced, for example by a sale price.
salesOrderItemAdd.Qty.SetValue ItemFlexGrid.Text
salesOrderItemAdd.Price.SetValue ItemFlexGrid.Text

'SalesOrder is fully constructed, now connect to QBPOS and Add it to QBPOS.
sessionManager.OpenConnection "", ""
sessionManager.BeginSession "Computer Name=;Company Data=My Company;Version=5"
Set responseMsgSet = sessionManager.DoRequests(requestMsgSet)

```

## Generating a PurchaseOrder from a SalesOrder

Generating a purchase order from a sales order is a feature available within the QBPOS UI. This feature requires the user to select sales order line items and quantities and creates the PO using those line items. You can “mimic” the same thing through the SDK but you’ll need to do the work of creating the PurchaseOrder and getting the line item data (quantity and items ordered) from the Sales Order. However, notice that there is no intrinsic link between PurchaseOrder and SalesOrder .

One way to make a quasi-link is to do what the QBPOS UI does, which is to specify the PO number manually using the SalesOrder number as part of that PO number. For example, if SalesOrder 4 is used to generate a PO number, you could supply the PO number as SO4. However, notice that this “quasi-link” does NOT result in automatic updates between sales order and purchase order, whether from the SDK or from the QBPOS UI.

## Creating Purchase Orders

---

PurchaseOrders are sent to vendors by merchants who want to restock inventory items or add new inventory items. Accordingly, when you build the PurchaseOrder using the SDK PurchaseOrderAdd request, you must always include the following required fields:

- VendorListID, which specifies the vendor for the PurchaseOrder
- One or more inventory item ListIDs, (one for each line item in the PurchaseOrder)
- The quantity field in each line item. All other data can be generated automatically for the line item from the item ListID reference, except quantity. You need to supply the quantity.

Other fields that are typically supplied are:

- The transaction date, which indicates the date the PurchaseOrder was generated.
- The StartShip date, which indicates the date the order is expected to ship from the vendor.
- The cancel date, which indicates the ending date when the PurchaseOrder is no longer valid.
- Other optional fields such as Associate, store number and so on.

PurchaseOrders are added, modified, and deleted using the SDK requests PurchaseOrderAdd, PurchaseOrderMod, and ListDel, respectively.

### **NOTE**

Notice that PurchaseOrder data is not shared between QBPOS and your QuickBooks Financial Software.

## Creating a PurchaseOrder

Take a look at the New Purchase Order form in the QBPOS UI, which is posted if you select Purchasing->New Purchase Order from the main QBPOS menubar. (See Figure 8-3 on page 70). If you compare the UI to the OSR listing of the request fields, you'll notice that most of the SDK's PurchaseOrderAdd request fields map directly to the form. Most of this mapping is self explanatory: TxnDate maps to Order Date, StartShipDate maps to Ship Date on the form, Fee maps to Fee, and so forth.

As with SalesOrder, other New Purchase Order UI form features are replaced in the SDK request. The UI's vendor lookup (upper left in the figure) is replaced by the VendorListID field, which specifies a particular vendor. The Item lookup, also in the upper left of the form is replaced by a reference (ListID) to a specific QBPOS item in each line item.

The screenshot shows the 'New Purchase Order' window. At the top, there's a title bar and a menu bar with 'Edit Item', 'Delete Item', 'Find Item', 'E-mail', and 'Help'. Below the menu bar, there are input fields for 'PO #', 'Status', 'Order Date', 'Vendor', 'Ship Date', 'Company', 'Cancel Date', and an 'Enter Item(s)' field. The 'Enter Item(s)' field has a placeholder text '< Type/Scan item info here >'. Below these fields is a table with columns: 'Item #', 'De...', 'Description 1', 'Attribute', 'Size', 'Qty', 'Qty Due', and 'Qty Rc...'. The bottom section of the form contains summary fields: 'Qty Ordered' (0), 'Qty Received' (0), 'Unfilled %' (0.00), 'Qty Due' (0), 'Instructions', 'SubTotal' (\$0.00), 'Disc %' (0.00), 'Disc \$' (\$0.00), 'Fee' (\$0.00), and a large 'Total' field showing '\$0.00' in green. At the bottom, there are buttons for 'Purchase Order Tasks', 'Print', 'Save', and 'Cancel', along with a 'Store' dropdown set to 1 and a status bar showing 'WS:1'.

Figure 8-3 New Purchase Order form

Notice the UI components that are greyed out in the New Purchase Order form, such as the quantity ordered, quantity received, unfilled percentage, and quantity due. The quantity ordered is filled automatically by QBPOS when the PurchaseOrderAdd request is



successful. The other fields are changed automatically when items are received into inventory via receiving vouchers either in the QBPOS UI or in the VoucherAdd request, provided that the VoucherAdd request contains a reference to that purchase order (purchase order list ID).

Notice the Fee and Discount fields in the UI form. You can specify either a Discount amount or percentage to apply to the subtotal field in the Purchase Order. The Fee is any special handling fee that applies to the order.

## Rules for Using PurchaseOrder Exclusive Fields

PurchaseOrder contains certain fields that are mutually exclusive, which means that you cannot set both fields in an SDK request. The fields are exclusive because filling one field automatically causes the other field to be calculated by QBPOS.

The following fields are exclusive and cannot both be set in the same PurchaseOrder request:

- The cost field and the extended cost field in the same line item are exclusive.
- The order-wide Discount and DiscountPercent fields applied to the PurchaseOrder subtotal are exclusive.

If you try to set fields that are exclusive, you get a runtime error.

## Creating a PurchaseOrder Using qbposXML

Listing 8-3 shows a construction of a purchase order in qbposXML. This sample first adds a new item not yet carried in inventory. Accordingly, the inventory item is created first in the message set using the macro attribute:

```
defMacro="ListID:Inv1"
```

That item is subsequently referenced by its macro name in the PurchaseOrderAdd line item that is in boldface font in the listing:

```
<ListID useMacro="ListID:Inv1" />
```

Notice that the listing does not show all of the possible request fields. For that information, please refer to the OSR.

Listing 8-3 Constructing a PurchaseOrderAdd Request in qbposXML

```
<?xml version="1.0" ?>
<?qbposxml version="1.0"?>
<QBPOSXML>
<QBPOSXMLMsgsRq onError="stopOnError">
  <ItemInventoryAddRq requestID="1">
    <ItemInventoryAdd defMacro="ListID:Inv1">
      <DepartmentListID>1000000001</DepartmentListID>
      <Desc1>SDK Item Purchase Order</Desc1>
```

```

    <Cost>5.00</Cost>
    <Price1>10.00</Price1>
    <ALU>dHbrVMEvQAnJxK@DATr2</ALU>          <!-- opt, field max = 20 -->
    <UPC>113741896773315807</UPC>             <!-- opt, field max = 18 -->
    <Size>Medium</Size>                        <!-- opt, field max = 12 -->
    <Attribute>Blue</Attribute>                <!-- opt, field max = 12 -->
    <Desc2>BigShoes22</Desc2>                  <!-- opt, field max = 30 -->
    <Desc1>Large Size Shoes</Desc1>            <!-- opt, field max = 30 -->
</ItemInventoryAdd>
</ItemInventoryAddRq>
<PurchaseOrderAddRq requestID = "21">
<PurchaseOrderAdd>
    <Fee>5</Fee>                                <!-- opt -->
    <StartShipDate>2005-9-13</StartShipDate>    <!-- opt -->
    <CancelDate>2005-10-13</CancelDate>          <!-- opt -->
    <Instructions>Instructions</Instructions>      <!-- opt, field max = 2000 -->
    <DiscountPercent>5</DiscountPercent>         <!-- opt -->
    <Associate>Associate</Associate>              <!-- opt, field max = 40 -->
    <VendorListID>1000000002</VendorListID>
    <TxnDate>2005-7-13</TxnDate>
<PurchaseOrderItemAdd>                          <!-- opt, may rep -->
    <Qty>2</Qty>
    <ListID useMacro="ListID:Inv1" />
</PurchaseOrderItemAdd>
</PurchaseOrderAdd>
</PurchaseOrderAddRq>
</QBPOSXMLMsgsRq>
</QBPOSXML>

```

In Listing 8-3, notice that the only PurchaseOrder line item information that is required is the quantity ordered and the item ordered (referenced by its ListID). No other information is needed. The inventory item should contain all the other data required for the purchase order line item. QBPOS will automatically fill in and calculate all of the other line item data from the referenced inventory item, such as price, UPC, ALU, and so forth.

## Creating a PurchaseOrder Using qbposFC

Creating a PurchaseOrder in qbposFC is the same as creating one in qbposXML so far as the fields are concerned. The same fields are available to qbposFC as are available in qbposXML. The only difference is that you don't need to construct any XML, but simply use the qbposFC objects and object properties to build the request.

Listing 8-4 shows a Visual Basic example that builds a Purchase Order.

#### Listing 8-4 Constructing a PurchaseOrder in qbposFC

```
'Define some qbposFC objects we need to begin construction:
Dim sessionManager As New QBPOSSessionManager
Dim xmlMajorVersion As String
Dim xmlMinorVersion As String
Dim responseMsgSet As IMessageSetResponse
xmlMajorVersion = "1"
xmlMinorVersion = "0"

'Create the message set and append the PurchaseOrder request. We just added this one request
'to the message set: you could add as many requests as you want, using the appropriate
'Append method for each request.
Set requestMsgSet = sessionManager.CreateMsgSetRequest(xmlMajorVersion, xmlMinorVersion)
requestMsgSet.Attributes.OnError = roeContinue
Dim PurchaseOrderAdd As IPurchaseOrderAdd
Set PurchaseOrderAdd = requestMsgSet.AppendPurchaseOrderAddRq

'When you append a request, the corresponding request object is returned for you to populate.
'You must set the properties in this object as desired before executing the request.
Dim PurchaseOrderAdd As IPurchaseOrderAdd
Set PurchaseOrderAdd = requestMsgSet.AppendPurchaseOrderAddRq
PurchaseOrderAdd.VendorListID.SetValue ("3304098223232024833")
PurchaseOrderAdd.Fee.SetValue (5#)
PurchaseOrderAdd.CancelDate.SetValue #12/31/2005#
PurchaseOrderAdd.DiscountPercent.SetValue (5)

PurchaseOrderAdd.StartShipDate.SetValue #10/31/2005#
PurchaseOrderAdd.Instructions.SetValue txtComments.Text
PurchaseOrderAdd.TxnDate.SetValue #9/31/2005#

'We've finished setting the main object properties. However, PurchaseOrder normally has
'line items, so we now add one line item. You can add more just by repeating what we
'do here. Quantity is obtained from UI components not defined here.
Dim purchaseOrderItemAdd As IPurchaseOrderItemAdd
Set purchaseOrderItemAdd = PurchaseOrderAdd.PurchaseOrderItemAddList.Append
purchaseOrderItemAdd.ListID.SetValue ListID

'PurchaseOrder is fully constructed, now connect to QBPOS and Add it to QBPOS.
sessionManager.OpenConnection "", ""
sessionManager.BeginSession "Computer Name=;Company Data=My Company;Version=5"
Set responseMsgSet = sessionManager.DoRequests(requestMsgSet)
```

## Creating Receiving Vouchers

You create receiving vouchers by using the VoucherAdd request, with the voucher type field set to “Receiving”. QBPOS inventory is augmented by the quantities of each item specified in the voucher. Notice that although the QBPOS SDK allows you to receive items

directly into inventory or receive items into inventory against a specified purchase order. (For this latter feature, you must include the purchase order list ID in the VoucherAdd request.)

When you build the receiving voucher, you must include the following fields:

- VendorListID, which specifies the vendor you purchased the items from.
- One or more inventory item ListIDs, (one for each line item in the PurchaseOrder)
- The quantity field in each line item. All other data can be generated automatically for the line item from the item ListID reference, except quantity. You normally need to supply the quantity.

Other fields that are typically supplied are:

- The invoice date, which indicates the date the vendor's invoice was generated.
- The invoice number, which indicates the vendor's invoice number.
- Other optional fields such as Associate, store number and so on.

Vouchers are added and queried using the SDK requests VoucherAdd and VoucherQuery. They cannot be modified or deleted either in the QBPOS UI or via the SDK: this preserves the integrity of the transaction history.

#### **NOTE**

Notice that Voucher data can be shared between QBPOS and your QuickBooks Financial Software, if the QBPOS company is set up to work with QuickBooks.

## The Receiving Voucher Request and the New Voucher Form

Take a look at the New Voucher form in the QBPOS UI, which is posted if you select Purchasing->New Receiving Voucher from the main QBPOS menubar. (See Figure 8-4). Most of the SDK's VoucherAdd request fields map directly to the form, but some UI form features are replaced in the SDK request. The UI's vendor lookup (upper left in the figure) is replaced by the VendorListID field, which specifies a particular vendor. The Item lookup is replaced by a reference (ListID) to a specific QBPOS item in each line item.

Notice that the Purchase Order lookup is replaced by the purchase order listID field, which links a specific PO to the voucher. This causes the PO to be automatically updated by the voucher.

Notice the greyed out UI components for the company and payee names: these are automatically filled in by QBPOS from the vendor and payee. Notice also that the payee name is automatically filled in by default with the vendor whose VendorListID is specified in the request.

**New Receiving Voucher**

Edit Item Delete Item Find Item Help

PO #  Date 9/5/2005

Vendor  Associate

Company

Enter Item(s)

Item #	Description 1	Qty Rc...	Vouc...	Ext Cost	Attribute	Size

Payee  SubTotal \$0.00

Payee Name  Disc % 0.00 Disc \$ \$0.00

Invoice #  Freight \$0.00

Invoice Date  Fee \$0.00

Store 1

**Total \$0.00**

Receiving Voucher Tasks Held Vouchers Print/Update - F12 Close

WS:1

Figure 8-4 New Voucher form

### Creating a Receiving Voucher Using qbposXML

Listing 8-5 shows a construction of a purchase order in qbposXML. This sample receives a quantity of two in the specified inventory item. QBPOS calculates the costs from the inventory item. You can supply the cost in the voucher line item if the cost differs from the one entered for the item in QBPOS.

In the listing notice the PurchaseOrderTxnID tag in bold. This links this voucher to that specific purchase order, causing it to be updated automatically. Notice also that this linkage is the **ONLY** function performed by this TxnID reference in this request: no line items or line item fields in VoucherAdd get filled in as a consequence of the PurchaseOrderTxnID line.

## Listing 8-5 Constructing the VoucherAdd Request in qbposXML

```
<?xml version="1.0" ?>
<?qbposxml version="1.0"?>
<QBPOSXML>
<QBPOSXMLMsgsRq onError="stopOnError">
  <VoucherAddRq requestID = "37">
    <VoucherAdd>
      <Fee>5.00</Fee>                                <!-- opt -->
      <DiscountPercent>5</DiscountPercent>           <!-- opt -->
      <VoucherType>Receiving</VoucherType>           <!-- opt -->
    > <VoucherType>Receiving</VoucherType>           <!-- opt -->
      <PurchaseOrderTxnID>4304097136480133124</PurchaseOrderTxnID> <!-- opt -->
      <InvoiceDate>2005-7-13</InvoiceDate>
      <TxnDate>2005-8-30</TxnDate>                   <!-- opt -->
      <Freight>5.00</Freight>                         <!-- opt -->
      <InvoiceNumber>6</InvoiceNumber>               <!-- opt -->
      <VendorListID>1000000002</VendorListID>
      <VoucherItemAdd>                                <!-- opt, may rep -->
        <QtyReceived>2</QtyReceived>                 <!-- opt -->
        <ListID>3304097136580133121</ListID>
      </VoucherItemAdd>
    </VoucherAdd>
  </VoucherAddRq>
</QBPOSXMLMsgsRq>
</QBPOSXML>
```

## Creating a Receiving Voucher Using qbposFC

Listing 8-6 shows a construction of a purchase order in qbposXML. This sample receives a quantity of two in the specified inventory item, and it receives that item quantity against the purchase order specified in the PurchaseOrderTxnID line. QBPOS calculates the costs from the inventory item. You can supply the cost in the voucher line item if the cost differs from the one entered for the item in QBPOS.

## Listing 8-6 Constructing a Receiving Voucher in qbposFC

```
'Define some qbposFC objects we need to begin construction:
Dim sessionManager As New QBPOSSessionManager
Dim xmlMajorVersion As String
Dim xmlMinorVersion As String
Dim responseMsgSet As IMessageSetResponse
xmlMajorVersion = "1"
xmlMinorVersion = "0"

'Create the message set and append the VoucherAdd request. We just added this one request
'to the message set: you could add as many requests as you want, using the appropriate
'Append method for each request.
```

```

Set requestMsgSet = sessionManager.CreateMsgSetRequest(xmlMajorVersion, xmlMinorVersion)
requestMsgSet.Attributes.OnError = roeContinue

'When you append a request, the corresponding request object is returned for you to populate.
'You must set the properties in this object as desired before executing the request.
Dim VoucherAdd As IVoucherAdd
Set VoucherAdd = requestMsgSet.AppendVoucherAddRq
VoucherAdd.VoucherType.SetValue (vtReceiving)

VoucherAdd.VendorListID.SetValue ("3304098223232024833")
VoucherAdd.InvoiceNumber.SetValue ("6")
VoucherAdd.InvoiceDate.SetValue #8/31/2005#
VoucherAdd.Fee.SetValue (5#)
VoucherAdd.Freight.SetValue (5#)
VoucherAdd.DiscountPercent.SetValue (5)

'Link this voucher to a purchase order to receive against that purchase order and update it.
VoucherAdd.PurchaseOrderTxnID.SetValue ("3315104180040794369")

If txtTDate.Text <> "" Then
    VoucherAdd.TxnDate.SetValue txtTDate.Text
End If

'We've finished setting the main object properties. However, Voucher normally has
'line items, so we now add one line item. You can add more just by repeating what we
'do here. Item listID and Quantity are obtained from UI components not defined here.
Dim voucherItemAdd As IVoucherItemAdd
Set voucherItemAdd = VoucherAdd.VoucherItemAddList.Append
voucherItemAdd.ListID.SetValue ListID
If ItemFlexGrid.Text <> "" Then
    voucherItemAdd.QtyReceived.SetValue ItemFlexGrid.Text
End If

'Voucher is fully constructed, now connect to QBPOS and Add it to QBPOS.
sessionManager.OpenConnection "", ""
sessionManager.BeginSession "Computer Name=;Company Data=My Company;Version=5"
Set responseMsgSet = sessionManager.DoRequests(requestMsgSet)

```

## Creating Return Vouchers

---

The return voucher is constructed in the same way as the Receiving voucher. The VoucherType field is simply set to “Return” instead of “Receiving”. The return voucher decreases the inventory by the quantities specified for each inventory item.

## Creating SalesReceipts

The term “SalesReceipt” in the SDK, like SalesOrder, is potentially confusing. It actually refers to several different types of customer receipt, including the following::

- Sales receipts, which represents a customer sale.
- Return receipts, which represent customer returns.
- Payout receipts, which represent payments from the cash drawer.
- Deposit receipts, which represent deposits taken on customer orders such as layaways.

You specify the type you want via the SalesReceiptType field. SalesReceipts are added to QBPOS using the SalesReceiptAdd request. You cannot modify or delete SalesReceipts.

### Creating a SalesReceipt

The New Sales Receipt form in the QBPOS UI is posted if you select Point of Sale->New Sales Receipt from the main QBPOS menubar. (See Figure 8-5).

The screenshot shows the 'New Sales Receipt' window. At the top, there are buttons for 'Edit Item', 'Delete Item', 'Find Item', and 'Help'. The 'Date' field is set to 9/5/2005. Below this are fields for 'Cashier' and 'Associate'. A section labeled 'Enter Item(s)' contains a text box with the placeholder '< Type/Scan item info here >' and a magnifying glass icon. Below this is a table with columns: Item #, Description 1, Qty, Price, Disc %, Disc \$, and Disc T... The table is currently empty. Below the table are fields for 'Customer' (with a placeholder 'Enter Customer here'), 'Address', 'Price Level' (set to 'Reg'), 'SubTotal' (\$0.00), 'Disc %' (0.00), 'Disc \$' (\$0.00), 'Tax %' (0.000), 'Tax \$' (\$0.00), 'Ship Date', 'Shipping' (\$0.00), 'Promo Code', and 'Store' (set to 1). A large 'Total' field shows \$0.00 in green. At the bottom, there are buttons for 'Sales Receipt Tasks', 'Held Receipts', 'Payment - F12', and 'Close'.

Figure 8-5 New Sales Receipt form



If you compare the UI to the OSR listing of the request fields, you'll notice that most SalesReceiptAdd request fields map to the form in obvious ways: PriceLevelNumber maps to Price Level, Cashier maps to Cashier, and so forth

Notice that the UI customer lookup (bottom left in the figure) is replaced by the CustomerListID field specifying a particular customer. The Item lookup in the upper left of the form is replaced by a reference (ListID) to a specific QBPOS item. Also, the greyed-out customer information is filled out automatically when the customer is specified.

If the sales receipt is based on an existing sales order, you can specify the sales order (via the SalesOrderListID field) in the SalesReceiptAdd request. This will automatically update the sales order in QBPOS. For example, it can automatically update quantities and set the SO status to Close if all items are accounted for in the sales receipt.

## Taxes

If the QBPOS company collects sales taxes, you'll notice that the taxable individual items in the sales receipt already have a tax code assigned to them. The tax assessed against this tax code depends on the TaxCategory the code belongs to. You can see this hierarchy in the company tax preferences.

Notice that the actual total tax amount for the sale is calculated after the SalesReceiptAdd is invoked, and returned in the response. You cannot calculate this. If you want to change the rates, do so in the company tax preferences. If you want to exempt the sale from any taxes, use the value "Exempt" in the TaxCategory field of the SalesReceiptAdd request.

## Creating a SalesReceipt Using qbposXML

Listing 8-7 shows a construction of a sales receipt in qbposXML. In the listing notice the sales order transaction ID line in bold. This links this receipt to that specific salesorder, causing it to be updated automatically. Notice also that this linkage is the ONLY function performed by this TxnID reference in this request: no line items or line item fields in SalesReceiptAdd get filled in as a consequence of the SalesOrderTxnID line.

Listing 8-7 Constructing the SalesReceiptAdd Request in qbposXML

```
<?xml version="1.0"?>
<?qbposxml version="1.0"?>
<QBPOSXML>
<QBPOSXMLMsgsRq onError="stopOnError">
  <SalesReceiptAddRq requestID="50">
    <SalesReceiptAdd>
      <SalesReceiptType>Sales</SalesReceiptType>
      <ShipDate>2005-9-13</ShipDate>
      <Comments>Completes Sales Order</Comments>          <!-- opt, field max = 300 -->
      <TxnDate>2005-9-05</TxnDate><!-- opt -->
      <CustomerListID>3293374402841706753</CustomerListID>
      <SalesOrderTxnID>3304105812489568513</SalesOrderTxnID>
      <SalesReceiptItemAdd>
```

```

        <Qty>2</Qty>
        <ListID>3304097136580133121</ListID>
        </SalesReceiptItemAdd>
    </TenderCashAdd>
        <TenderAmount>4.00</TenderAmount>
    </TenderCashAdd>
</SalesReceiptAdd>
</SalesReceiptAddRq>
</QBPOSXMLMsgsRq>
</QBPOSXML>

```

## Creating a SalesReceipt Using qbposFC

Listing 8-8 shows a construction of a sales receipt in qbposFC.

### Listing 8-8 Constructing a SalesReceipt in qbposFC

```

'Define some qbposFC objects we need to begin construction:
Dim sessionManager As New QBPOSSessionManager
Dim xmlMajorVersion As String
Dim xmlMinorVersion As String
Dim responseMsgSet As IMsgSetResponse
xmlMajorVersion = "1"
xmlMinorVersion = "0"

'Create the message set and append SalesReceiptAdd request. We just added this one request
'to the message set: you could add as many requests as you want, using the appropriate
'Append method for each request.
Set requestMsgSet = sessionManager.CreateMsgSetRequest(xmlMajorVersion, xmlMinorVersion)
requestMsgSet.Attributes.OnError = roeContinue

'When you append a request, the corresponding request object is returned for you to populate.
'You must set the properties in this object as desired before executing the request.
Dim SalesReceiptAdd As ISalesReceiptAdd
Set SalesReceiptAdd = requestMsgSet.AppendSalesReceiptAddRq

SalesReceiptAdd.SalesReceiptType.SetValue ENSalesReceiptType.srtSales
SalesReceiptAdd.ShipDate.SetValue txtSDate.Text
SalesReceiptAdd.PriceLevelNumber.SetValue pln1
SalesReceiptAdd.Comments.SetValue txtComments.Text
SalesReceiptAdd.TxnDate.SetValue txtTDate.Text

'We've finished setting the main object properties. However, SalesReceipts normally have
'line items, so we now add one line item. You can add more just by repeating what we
'do here. Item listID and Quantity are obtained from UI components not defined here.
Dim salesReceiptItemAdd As ISalesReceiptItemAdd

```

```

Set salesReceiptItemAdd = SalesReceiptAdd.SalesReceiptItemAddList.Append
salesReceiptItemAdd.ListID.SetValue ListID
salesReceiptItemAdd.Qty.SetValue ItemFlexGrid.Text

'Need to indicate payment type and amount. Get total from UI components.
'This sample is a cash transaction.
Dim total As Double
total = CDbl(txtQtySold.Text) * CDbl(txtPrice.Text)

'Add the cash tender indicating the amount involved in the transaction
Dim salesReceiptTenderCashAdd As ITenderCashAdd
Set salesReceiptTenderCashAdd = SalesReceiptAdd.TenderCashAddList.Append
salesReceiptTenderCashAdd.TenderAmount.SetValue Str(total)

'Voucher is fully constructed, now connect to QBPOS and Add it to QBPOS.
sessionManager.OpenConnection "", ""
sessionManager.BeginSession "Computer Name=;Company Data=My Company;Version=5"
Set responseMsgSet = sessionManager.DoRequests(requestMsgSet)

```



## CHAPTER 9

# ADDING, MODIFYING, AND QUERYING TIME ENTRIES

This chapter describes how to add, modify, and query employee time entries.

The POS SDK provides access to parts of the QBPOS employee time clock functionality, allowing applications to add new time entries, modify existing time entries, and query for specified time entries within QBPOS.

There are some key limitations, however. The QB POS time clock reports and the weekly time sheet are not currently accessible through the SDK, although those reports do reflect time entry Adds and Mods performed via the SDK. Also, no time entries, whether created via the QBPOS SDK or via the QBPOS UI, can be sent to QuickBooks Financial Software (QBFS): there is no way, currently, for QBPOS time entries to be automatically reflected in QuickBooks.

You should be aware that the QBPOS administrator can turn off the time clock feature in the QBPOS company preferences (this feature is turned on by default), in which case the various time clock and time entry UI forms may not be visible to your user. However, the SDK can Add, Modify, or Query time entries, whether time clock features are turned on in company preferences or not.

## About Time Entries in QBPOS and SDK Support of Them

---

In the QBPOS UI, there are two ways to get time data into QBPOS for an employee, assuming the time clock functions are turned on in the QBPOS company preferences.

- Using the clock in and clock out functionality, which automatically calculates times and makes time entries.
- Making a time entry directly for an employee, supplying clock in/clock out times and store number, for multi store configurations.

The QBPOS SDK supports only the making and modifying of time entries. It doesn't currently support the clock in and clock out functionality.

## Adding a Time Entry in QBPOS SDK

---

To add a time entry, you use the TimeEntryAdd request, in which you supply the employee ListID, a clock in date/time, a clock out date/time, and a store number. You can optionally use a macro for the employee ListID if you are creating an employee at the same time (within the same message set). The following examples show how to do this using qbposXML and the qbposFC convenience library.

## Adding a Time Entry in qbposXML

---

The following XML shows how to add a time entry

```
<?xml version="1.0" encoding="UTF-8" ?>
<?qbposxml version="2.0"?>
<QBPOSXML>
  <QBPOSXMLMsgsRq onError="continueOnError">
    <TimeEntryAddRq requestID="0">
      <TimeEntryAdd>
        <ClockInTime>2006-09-14T08:00:00</ClockInTime>
        <ClockOutTime>2006-09-14T16:00:00</ClockOutTime>
        <CreatedBy>JGoodall</CreatedBy>
        <EmployeeListID>5171502750302634241</EmployeeListID>
        <StoreNumber>1</StoreNumber>
      </TimeEntryAdd>
    </TimeEntryAddRq>
  </QBPOSXMLMsgsRq>
</QBPOSXML>
```

## Adding a Time Entry in qbposFC

---

The following VB code snippet shows how to add that same time entry using qbposFC:

```
Dim sessionManager As QBPOSSessionManager
Set sessionManager = New QBPOSSessionManager
sessionManager.OpenConnection "123", "Time Entry Add Sample"
sessionManager.BeginSession "Computer Name=yourComputerName;
                             Company Data=Al's Sports Hut;
                             Version=6;practice=Yes"

Dim requestMsgSet As IMessageSetRequest
Set requestMsgSet = sessionManager.CreateMsgSetRequest("2", "0")
requestMsgSet.Attributes.OnError = roeContinue

Dim MyTimeEntry As ITimeEntryAdd
Set MyTimeEntry = requestMsgSet.AppendTimeEntryAddRq
MyTimeEntry.ClockInTime.SetValue #9/14/2006 8:00:00 AM#, False
MyTimeEntry.ClockOutTime.SetValue #9/14/2006 4:00:00 PM#, False
MyTimeEntry.CreatedBy.SetValue "JGoodall"
MyTimeEntry.EmployeeListID.SetValue "5171502750302634241"
MyTimeEntry.StoreNumber.SetValue 1

Dim responseMsgSet As IMessageSetResponse
Set responseMsgSet = sessionManager.DoRequests(requestMsgSet)
sessionManager.EndSession
sessionManager.CloseConnection
```

## Modifying a Time Entry in QBPOS SDK

---

To modify an existing time entry for an employee, you use the `TimeEntryMod` request, in which you supply the `ListID` of the time entry you want to modify, and the clock in and/or clockout date/time you want to modify. The following examples show how to do this using `qbposXML` and the `qbposFC` convenience library.

### Modifying a Time Entry in qbposXML

---

The following XML shows how to modify an existing time entry:

```
<?xml version="1.0" encoding="UTF-8" ?>
<?qbposxml version="2.0"?>
<QBPOSXML>
  <QBPOSXMLMsgsRq onError="continueOnError">
    <TimeEntryModRq requestID="0">
      <TimeEntryMod>
        <ListID>5171654100671955201</ListID>
        <ClockInTime>2006-09-14T08:00:00</ClockInTime>
        <ClockOutTime>2006-09-14T16:00:00</ClockOutTime>
        <CreatedBy>JGoodall</CreatedBy>
      </TimeEntryMod>
    </TimeEntryModRq>
  </QBPOSXMLMsgsRq>
</QBPOSXML>
```

### Modifying a Time Entry in qbposFC

---

The following XML shows how to modify an existing time entry in `qbposFC`:

```
Dim sessionManager As QBPOSSessionManager
Set sessionManager = New QBPOSSessionManager
sessionManager.OpenConnection "123", "Time Entry Add Sample"
sessionManager.BeginSession "Computer Name=yourComputerName;
                             Company Data=Al's Sports Hut;
                             Version=6;practice=Yes"

Dim requestMsgSet As IMessageSetRequest
Set requestMsgSet = sessionManager.CreateMsgSetRequest("2", "0")
requestMsgSet.Attributes.OnError = roeContinue

Dim MyTimeEntry As ITimeEntryMod
Set MyTimeEntry = requestMsgSet.AppendTimeEntryModRq
MyTimeEntry.ListID.SetValue "5171654100671955201"
MyTimeEntry.ClockInTime.SetValue #9/14/2006 8:00:00 AM#, False
MyTimeEntry.ClockOutTime.SetValue #9/14/2006 4:00:00 PM#, False
MyTimeEntry.CreatedBy.SetValue "JGoodall"
```

```

Dim responseMsgSet As IMsgSetResponse
Set responseMsgSet = sessionManager.DoRequests(requestMsgSet)
sessionManager.EndSession
sessionManager.CloseConnection

```

## Querying For Time Entry Records in QBPOS SDK

---

The TimeEntryQuery request is used to retrieve the time entry records specified in the query. You can get all time records or filter them by clock-in or clock-out dates, employee first, last, or login names, and so forth, as listed in the OSR.

### Using qbposXML

---

In this example, we filter time records by a range of employee login names:

```

<?xml version="1.0" encoding="UTF-8" ?>
<?qbposxml version="2.0"?>
<QBPOSXML>
  <QBPOSXMLMsgsRq onError="continueOnError">
    <TimeEntryQueryRq requestID="0">
      <EmployeeLoginNameRangeFilter>
        <FromEmployeeLoginName>Alberta</FromEmployeeLoginName>
        <ToEmployeeLoginName>Fred</ToEmployeeLoginName>
      </EmployeeLoginNameRangeFilter>
    </TimeEntryQueryRq>
  </QBPOSXMLMsgsRq>
</QBPOSXML>

```

### Using qbposFC

---

In this example we do the same thing as we did in XML, filtering by employee login names:

```

Dim sessionManager As QBPOSSessionManager
Set sessionManager = New QBPOSSessionManager
sessionManager.OpenConnection "123", "Time Entry Add Sample"
sessionManager.BeginSession "Computer Name=yourComputerName;
                             Company Data=Al's Sports Hut;
                             Version=6;practice=Yes"

Dim requestMsgSet As IMsgSetRequest
Set requestMsgSet = sessionManager.CreateMsgSetRequest("2", "0")
requestMsgSet.Attributes.OnError = roeContinue

Dim MyTimeEntry As ITimeEntryQuery
Set MyTimeEntry = requestMsgSet.AppendTimeEntryQueryRq
MyTimeEntry.OREmployeeLoginNameFilters.EmployeeLoginNameRange
    Filter.FromEmployeeLoginName.setValue ("Alberta")
MyTimeEntry.OREmployeeLoginNameFilters.EmployeeLoginNameRange
    Filter.ToEmployeeLoginName.setValue ("Fred")

```



```
Dim responseMsgSet As IMsgSetResponse
Set responseMsgSet = sessionManager.DoRequests(requestMsgSet)
sessionManager.EndSession
sessionManager.CloseConnection
```



## CHAPTER 10

# SETTING AND MODIFYING SALES TAX PREFERENCES

This chapter describes the QBPOS sales tax preferences and how these are supported in the SDK. We'll also briefly describe how the tax preferences are applied to inventory items, which determines actual tax at transaction time.

The SDK sales tax preference requests must be understood in the context of the QBPOS sales tax preferences capability in the UI, which we'll describe next. However, before we start, there are a few things we want to alert you to because of their importance and because they are not immediately obvious:

- Company preferences don't reflect new or modified tax categories, tax codes, or tax rates that are performed through the SDK, nor are they available for use elsewhere in QBPOS, *until* you close and re-open the company.
- What the SDK calls a TaxCategory is called a "Tax Location" in the QBPOS UI.
- In the SDK, you access/manipulate sales tax preferences via the following requests:
  - > You create a TaxCategory (tax location) with TaxCategoryAdd and modify it with TaxCategoryMod
  - > You create a TaxCode with TaxCodeAdd and modify it with TaxCodeMod.
  - > You set tax rate(s) for a TaxCode with TaxRecordMod.
  - > You delete TaxCodes and TaxCategories using the ListDel request.
  - > There is currently no TaxCategory or TaxCode query. To get a list of all the tax codes (and their rates) under each tax category, you use a CompanyQuery.

## About Sales Tax Preferences in the QBPOS UI

---

In the QBPOS UI, you specify that you want to support sales tax in your company by checking the Collect Sales Tax checkbox in the Setup Tax form within company preferences (Figure 10-1). Then, you create sales tax locations (which are called tax categories in the SDK) and tax codes in preferences, and apply rates to those tax codes.

Figure 10-1 shows some tax locations and tax codes with some tax rates set. Notice that in the figure each of the tax locations have the same set of tax codes defined. This happens automatically. When you create a tax code, it is automatically added to all existing tax locations (tax categories) and to all new ones.

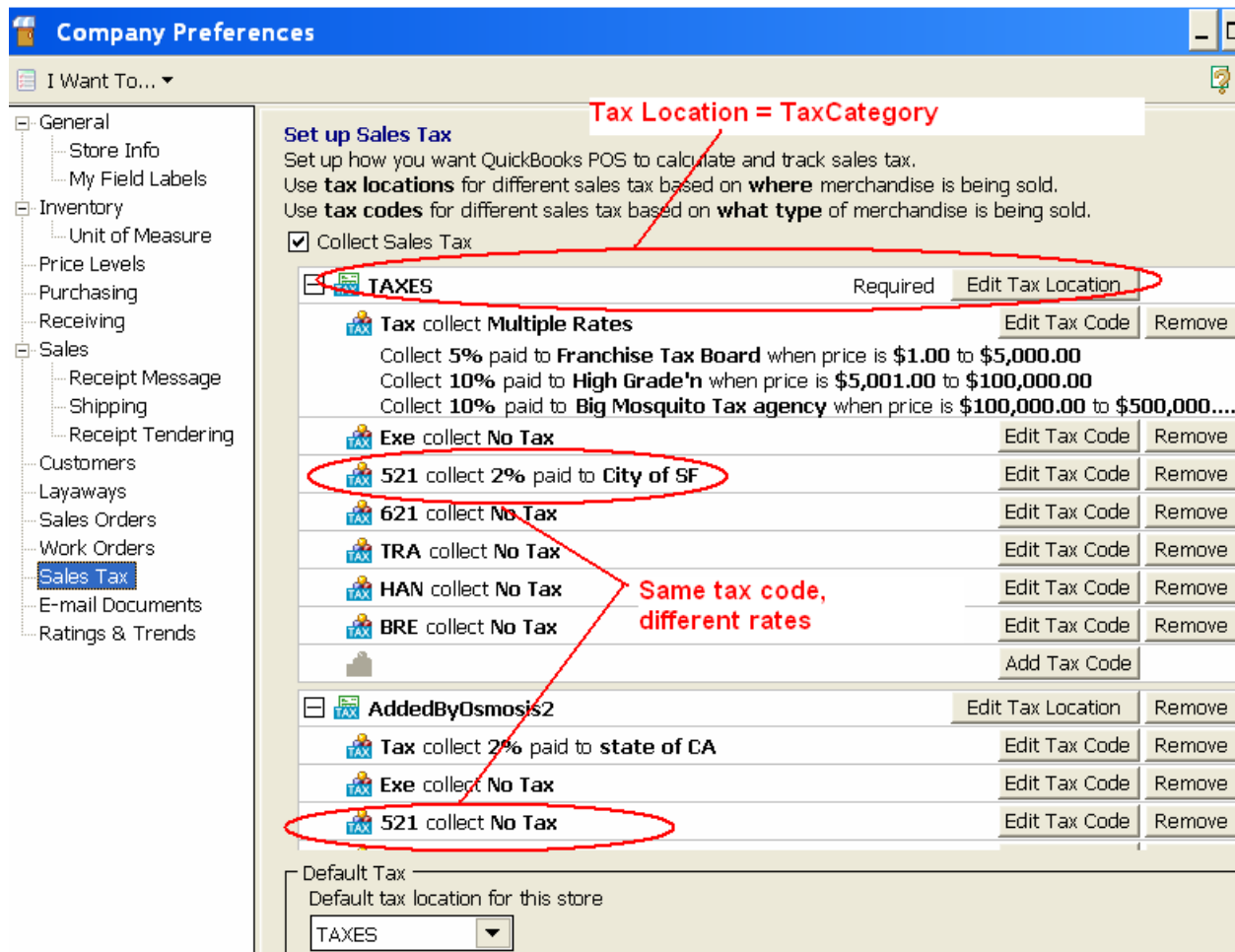


Figure 10-1 Setting Up Sales Tax in Preferences

Notice that a tax code under one tax category (location) can have a different rate from that same tax code under a different tax category.

Finally, notice the default Tax location, "TAXES", in Figure 10-1. You specify which set of tax code rates will be in effect when you create or modify an inventory item. We'll go into the inventory item/tax code and how this affects sales transactions later.

You can change the default tax category at any time in QBPOS, which will have the effect of changing the effective tax rates used in transactions to whatever rates are defined in the tax codes under the new default tax category. This has NO impact on any transactions made previously with different tax code rates under a different tax category! Whatever tax category was used in those transactions at transaction time will not change. Changing the tax category only affects the next transaction made using that new default tax category. However, note that you can only change the default tax category from the UI, not from the SDK!

## Tax Records: an SDK Concept

As mentioned in the previous section, the tax rates for a given tax code can vary depending on the tax category. Another way to think about this is to use the concept of Tax Record, which is introduced in the SDK.

A tax category and a tax code under that tax category constitute a tax record (see Figure 10-2).

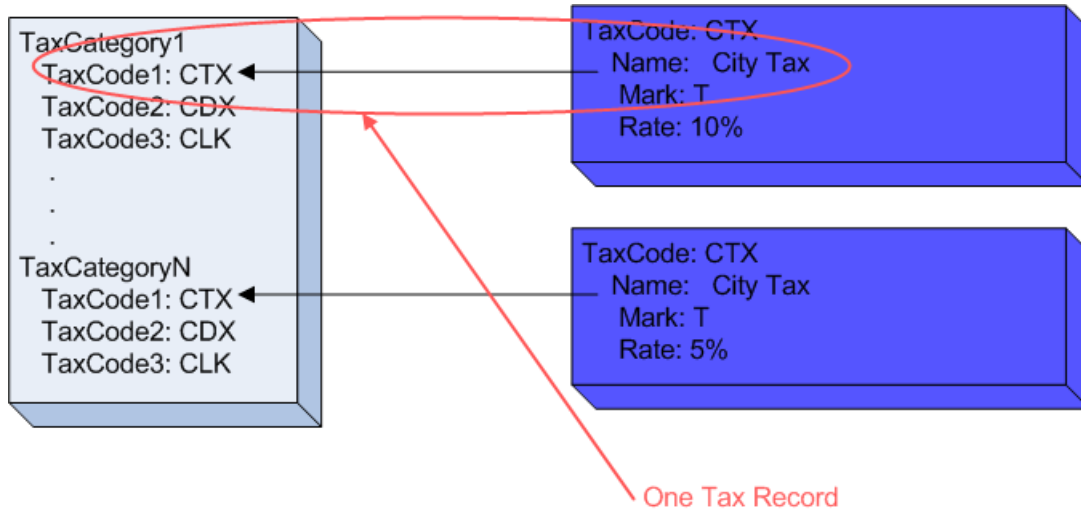


Figure 10-2 Each tax category-taxcode pair is one tax record

You can get all of the tax records in a QBPOS company by invoking a CompanyQuery. The following sample is taken from a Ret from a CompanyQuery. It shows two tax records:

```
<TaxRecord>
  <TaxCategoryListID>EastPA</TaxCategoryListID>
  <TaxCategory>EastPA</TaxCategory>
  <POSTaxCodeListID>PAT</POSTaxCodeListID>
  <POSTaxCode>PAT</POSTaxCode>
  <TaxPercent>7</TaxPercent>
  <QBTaxGroup />
  <QBTaxCode />
</TaxRecord>
```

```

<TaxRecord>
  <TaxCategoryListID>TAXES</TaxCategoryListID>
  <TaxCategory>TAXES</TaxCategory>
  <POSTaxCodeListID>WestPA</POSTaxCodeListID>
  <POSTaxCode>WestPA</POSTaxCode>
  <TaxRate>
    <TaxPercent>2</TaxPercent>
    <TaxLowRange>1000.00</TaxLowRange>
    <IsTaxAppliedOnlyWithinRange>1</IsTaxAppliedOnlyWithinRange>
    <QBTaxGroup />
  </TaxRate>
  <QBTaxGroup />
  <QBTaxCode />
</TaxRecord>

```

## How TaxCategories/TaxCodes are Applied in Transactions

Ok. We've set up the tax categories (locations) and the tax codes with the rates we want to use. How do we get the right category and tax codes applied in a sales transaction? The key concept is that you always apply the tax code to the inventory item first (see Figure 10-3).

The screenshot shows the 'New Item' form with the following details:

- Item Info:** Item Type: Inventory, Dept Name: Animal Feed, Vendor Name: LET'S RUN, Description 1: Reely good Dawg Food, Description 2: , Attribute: , Size: Med.
- Price & Cost (Base Unit):** Regular Sales Price: \$10.00, Average Unit Cost: \$3.00, Tax Code: 921 (highlighted with a red circle).
- Quantity (Base Unit):** On-Hand Quantity: 100, Reorder Point: 10.
- Units of Measure:** (empty field).

Figure 10-3 Assigning a TaxCode (and it's rate) to an inventory item

As shown Figure 10-3, you can create or modify inventory items that has any of the tax codes that are currently defined. However, both in the UI and in the SDK, you cannot specify any tax category (tax location) for the inventory item. Instead, whatever is currently the default tax category for the QBPOS company is used to determine the tax rates of the assigned tax code.

## Creating/Editing a Tax Category (Location) in the UI

To create a tax category, in the Set Up Sales Tax form click on Add Tax Location to display the Tax Category form which is shown in Figure 10-4. (To edit an existing category, click on the Edit Tax Location button to the right of the category.)

**Enter Tax Location information**  
 Use Tax Locations to calculate different sales taxes for different tax jurisdictions.

Tax Location Name  
 Enter the name of the tax location. This name will be used on all documents and receipts.

Tax Location Name:  Example printed receipt with tax location

Example printed receipt with tax location:

1 Item(s)	Subtotal:	\$ 21.99
Sales Tax	8.25 % Tax	\$ 1.81

Shipping  
 Choose whether or not the shipping amount for merchandise shipped to this location should be taxed.

☐ Shipping to this location is taxable using the tax:

Help OK Cancel

Figure 10-4 Specifying a Tax Category

You need to supply the tax category name (TAXES, in our example) and indicate whether shipping fees are to be taxed when this tax category is used. In our example, shipping is not taxed, so the checkbox is left unchecked. If shipping is to be taxed, then the tax code to be used for that tax must be specified from the drop down list to the right of the checkbox.

When you create a tax category in the SDK using `TaxCategoryAdd`, or modify one using `TaxCategoryMod`, you can set the same pieces of information as you can in the UI. You can set or change the tax category name and you can indicate whether shipping tax is collected and if so the name of the tax code.

## Adding a TaxCategory via the SDK

Adding a tax category requires you to specify the tax category name, and optionally, to specify whether items shipped under this category will incur shipping tax. (By default, if you don't specify this, no shipping tax is charged.)

### Adding a TaxCategory using qbposXML

The following XML adds a TaxCategory (tax location) named GreaterTuna and specifies that tax be collected on the shipping using the tax code TUN. The shipping tax code must be an already existing tax code in the company, and as such, is a maximum of three characters.

```

<?xml version="1.0" encoding="UTF-8"?>
<?qbposxml version="2.0"?>
<QBPOSXML>
<QBPOSXMLMsgsRq onError = "continueOnError">
<TaxCategoryAddRq requestID = "0">
<TaxCategoryAdd>
<TaxCategory>GreaterTuna</TaxCategory>
<ShippingTaxCode>TUN</ShippingTaxCode>
</TaxCategoryAdd>
</TaxCategoryAddRq>
</QBPOSXMLMsgsRq>
</QBPOSXML>

```

## Adding a TaxCategory using qbposFC

---

The following VB code does the same thing as the preceding XML:

```

Dim sessionManager As QBPOSSessionManager
Set sessionManager = New QBPOSSessionManager
sessionManager.OpenConnection "123", "Tax Category Add Sample"
sessionManager.BeginSession "Computer Name=yourComputerName;
                             Company Data=Al's Sports Hut;
                             Version=6;practice=Yes"

Dim requestMsgSet As IMessageSetRequest
Set requestMsgSet = sessionManager.CreateMsgSetRequest("2", "0")
requestMsgSet.Attributes.OnError = roeContinue

Dim MyTaxCategory As ITaxCategoryAdd
Set MyTaxCategory = requestMsgSet.AppendTaxCategoryAddRq
MyTaxCategory.ShippingTaxCode.setValue "TUN"
MyTaxCategory.TaxCategory.setValue "GreaterTuna"

Dim responseMsgSet As IMessageSetResponse
Set responseMsgSet = sessionManager.DoRequests(requestMsgSet)
sessionManager.EndSession
sessionManager.CloseConnection

```

## Modifying a TaxCategory using the SDK

---

Modifying a tax category is similar to adding one, except that you need to supply the TaxCategoryID. Conveniently enough, the tax category name and its list ID are always the same!

## Querying for TaxCategories using the SDK

---

You get the tax categories, tax codes (and their rates) for a given company by using a CompanyQuery, which, along with other company data, returns a separate tax record for all tax category/tax code pair, as in the following example:



```

<TaxRecord>
  <TaxCategoryListID>TAXES</TaxCategoryListID>
  <TaxCategory>TAXES</TaxCategory>
  <POSTaxCodeListID>521</POSTaxCodeListID>
  <POSTaxCode>521</POSTaxCode>
  <TaxPercent>2</TaxPercent>
  <TaxRate>
    <TaxPercent>2</TaxPercent>
    <TaxLowRange>1000</TaxLowRange>
    <IsTaxAppliedOnlyWithinRange>1</IsTaxAppliedOnlyWithinRange>
    <QBTaxGroup />
  </TaxRate>
  <QBTaxGroup />
  <QBTaxCode />
</TaxRecord>

<TaxRecord>
  <TaxCategoryListID>TAXES</TaxCategoryListID>
  <TaxCategory>TAXES</TaxCategory>
  <POSTaxCodeListID>621</POSTaxCodeListID>
  <POSTaxCode>621</POSTaxCode>
  <TaxPercent>4</TaxPercent>
  <QBTaxGroup />
  <QBTaxCode />
</TaxRecord>

```

## Creating/Editing a Tax Code in the UI

---

To create a tax code, in the Set Up Sales Tax form click on Add Tax Code to display the Tax Code form which is shown in Figure 10-5. (To edit an existing code, click on the Edit Tax Code button to the right of the tax code.)

**Enter Tax Code information**

Use Tax Codes to calculate the sales tax for different types of merchandise or services

Tax Code Name

Enter a code and name for this tax code. This name will be used on all documents and receipts.

Tax Code:  (max. 3 characters)

Tax Code Name:

Tax Code Mark

You can optionally assign a 1-character "mark" to identify items sold with this tax code.

Printed "Mark":  (max. 1 character)

Example printed receipt with mark:

DESCRIPTION	QTY	PRICE	EXT PRICE
Sweatshirt	1	\$ 21.99	\$ 21.99 <b>T</b>

Help < Back Next > Cancel

Figure 10-5 Specifying a Tax Code

You need to supply the tax code, which is normally a three character value, but can be any value so long as it doesn't exceed three characters (621, in our example) and supply a longer name for that tax code. This name is displayed in the QBPOS UI to aid the user in applying the proper tax code. You can also supply a one-character mark to be printed on sales receipts to help identify the tax code that was used.

When you create a tax code in the SDK using `TaxCodeAdd`, or modify one using `TaxCodeMod`, you can set the same pieces of information as you can in the UI. You can set or change the tax code, the tax code name, and the tax code mark.

## Adding a TaxCode via the SDK

Adding a tax code requires you to specify the tax code characters, which can be anything so long as it doesn't exceed three characters. Optionally, you can also supply a longer name as your tax code name. The tax code name is displayed in the QBPOS UI, and can be useful in helping the QBPOS user select the right tax code to use. You can also optionally supply a one-character mark to be printed on sales receipts, to identify which tax applied to the item.

## Adding a TaxCode using qbposXML

---

The following XML adds a TaxCode of BRE with a tax code name of Bremen City Tax and specifies a character “B” to be printed on sales receipts to identify the tax.

```
<?xml version="1.0" encoding="UTF-8" ?>
<?qbposxml version="2.0"?>
<QBPOSXML>
  <QBPOSXMLMsgsRq onError="continueOnError">
    <TaxCodeAddRq requestID="0">
      <TaxCodeAdd>
        <POSTaxCode>BRE</POSTaxCode>
        <POSTaxCodeName>Bremen City Tax</POSTaxCodeName>
        <POSTaxCodeMark>B</POSTaxCodeMark>
      </TaxCodeAdd>
    </TaxCodeAddRq>
  </QBPOSXMLMsgsRq>
</QBPOSXML>
```

## Adding a TaxCode using qbposFC

---

The following VB code does the same thing as the preceding XML:

```
Dim sessionManager As QBPOSSessionManager
Set sessionManager = New QBPOSSessionManager
sessionManager.OpenConnection "123", "Tax Category Add Sample"
sessionManager.BeginSession "Computer Name=yourComputerName;
                             Company Data=Al's Sports Hut;
                             Version=6;practice=Yes"

Dim requestMsgSet As IMsgSetRequest
Set requestMsgSet = sessionManager.CreateMsgSetRequest("2", "0")
requestMsgSet.Attributes.OnError = roeContinue

Dim MyTaxCode As ITaxCodeAdd
Set MyTaxCode = requestMsgSet.AppendTaxCodeAddRq
MyTaxCode.POSTaxCode.setValue "BRE"
MyTaxCode.POSTaxCodeName.setValue "Bremen City Tax"
MyTaxCode.POSTaxCodeMark.setValue "B"

Dim responseMsgSet As IMsgSetResponse
Set responseMsgSet = sessionManager.DoRequests(requestMsgSet)
sessionManager.EndSession
sessionManager.CloseConnection
```

## Modifying a TaxCode using the SDK

---

Modifying a tax code is similar to adding one, except that you need to supply the tax code list ID (POSTaxCodeListID), which is conveniently always the same as the tax code itself.

## Querying for TaxCodes using the SDK

---

You get the tax categories, tax codes (and their rates) for a given company by using a CompanyQuery, as described previously under “Querying for TaxCategories using the SDK”.

## Setting TaxCode Rates in QBPOS

---

Once you’ve set the tax code, the tax code name, and the tax code mark to your satisfaction, you need to assign one or more tax rates to this code, unless you want this code to have no tax. In the QBPOS UI, tax rates are assigned within each tax code under the tax category (location). You assign the rates by clicking the **Next>** button in the bottom of the tax code edit or add form (Figure 10-5).

Clicking **Next>** displays the Tax Options form (see Figure 10-6):

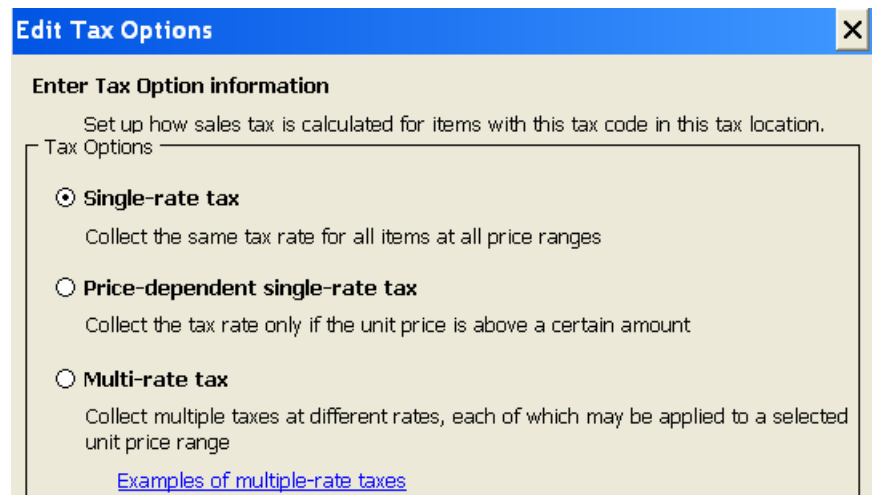


Figure 10-6 Choosing a tax rate structure

How you assign the rates in the Tax Options form depends on the tax requirements of the tax location (category). In some locations you can use the simplest option, which is to collect a single rate for all taxable items regardless of price, payable to one tax authority. In other locations, you might not have any applicable sales tax until a price threshold is crossed; for example, the first \$5000 is tax free, then everything after is taxed. In still other locations, you may have multiple rates and/or need to pay taxes to multiple tax authorities.

## Setting a Single Rate Tax (UI)

---

From the Tax Option form (Figure 10-6) select the single-rate tax and click **Next** to display the Tax Calculation form:

Figure 10-7 Setting a Single Rate Tax

In the form, supply the tax percentage, and optionally the tax authority that is to be paid the tax. If the QBPOS company is set up to work with QBFS, you can specify the QuickBooks tax item/item group and tax code that you want to map to this tax code and rate. If the company is set up to use QBFS and you don't specify the QuickBooks tax information here, QBFS will create a new tax item and tax code using the current QBPOS tax code data.

## Setting a Single Rate Tax (SDK)

To set the tax rate in the QBPOS SDK, you use the `TaxRecordMod` request. You need to supply the tax category list ID and the tax code list ID of an existing tax category and an existing tax code, which identify the tax record you are setting the rate for. You need to supply the tax percentage value you want.

### Setting a Single-Rate Tax Using qbposXML

```
<?xml version="1.0" encoding="UTF-8" ?>
<?qbposxml version="2.0"?>
<QBPOSXML>
  <QBPOSXMLMsgsRq onError="continueOnError">
    <TaxRecordModRq requestID="0">
      <TaxRecordMod>
        <TaxRecord>
          <TaxCategoryListID>TAXES</TaxCategoryListID>
          <POSTaxCodeListID>TRA</POSTaxCodeListID>
          <TaxPercent>4.1</TaxPercent>
        </TaxRecord>
      </TaxRecordMod>
    </TaxRecordModRq>
  </QBPOSXMLMsgsRq>
</QBPOSXML>
```

## Setting a Single-Rate Tax Using qbposFC

The following VB code does the same thing as the preceding XML:

```
Dim sessionManager As QBPOSSessionManager
Set sessionManager = New QBPOSSessionManager
sessionManager.OpenConnection "123", "Tax Record Mod Sample"
sessionManager.BeginSession "Computer Name=yourComputerName;
                             Company Data=Al's Sports Hut;
                             Version=6;practice=Yes"

Dim requestMsgSet As IMsgSetRequest
Set requestMsgSet = sessionManager.CreateMsgSetRequest("2", "0")
requestMsgSet.Attributes.OnError = roeContinue

Dim MyTaxRecordMod As ITaxRecordMod
Set MyTaxRecordMod = requestMsgSet.AppendTaxRecordModRq
MyTaxRecordMod.TaxRecord.TaxCategoryListID.SetValue "TAXES"
MyTaxRecordMod.TaxRecord.POSTaxCodeListID.SetValue "TRA"
MyTaxRecordMod.TaxRecord.TaxPercent.SetValue (4.1)

Dim responseMsgSet As IMsgSetResponse
Set responseMsgSet = sessionManager.DoRequests(requestMsgSet)
sessionManager.EndSession
sessionManager.CloseConnection
```

## Setting a Price-Dependant Single Rate Tax (UI)

From the Tax Option form (Figure 10-6) select the single-rate tax and click **Next** to display the Tax Calculation form:

**Edit Tax Calculation**

**Enter Tax Rate information** **<TaxLowRange>**

Set up how sales tax is calculated for items with this tax code in this tax location.

Enter your sales tax rate as a percentage (e.g. "5.2%")  %

the name of the government agency to which you pay sales tax

this tax on an item only if the unit price or shipping is more than

☐ Apply sales tax only to the amount over the unit price or shipping threshold

**QuickBooks Tax Info** **<IsTaxAppliedOnlyWithinRange>**

Assigned to QuickBooks Tax Item/Group:

Assigned to QuickBooks Tax Code:

Figure 10-8 Setting a Price-Dependant Single Rate Tax

In the form, supply the tax percentage, and optionally the tax authority that is to be paid the tax. Then specify the threshold dollar amount that will trigger the tax. If you want the whole item to be taxed after this threshold is reached, leave the checkbox unchecked. If you want sales tax applied to the item only on the amount exceeding the threshold, DO check the checkbox.

If the QBPOS company is set up to work with QBFS, you can specify the QuickBooks tax item/item group and tax code that you want to map to this tax code and rate. If the company is set up to use QBFS and you don't specify the QuickBooks tax information here, QBFS will create a new tax item and tax code using the current QBPOS tax code data.

## Setting a Price-Dependant Single Rate Tax (SDK)

---

You can set a price-dependant single rate tax using the SDK TaxRecordMod request. There is a trick to this, however,. What you need to do is use the TaxRate aggregate such that you specify the low range (<TaxLowRange>) to the amount you want for your tax threshold, but don't set the high range (<TaxHighRange>).

If you want the threshold to trigger a tax for the whole amount of the item, set <IsTaxAppliedOnlyWithinRange> to False. If you want the tax to apply only to amount over the threshold amount, set <IsTaxAppliedOnlyWithinRange> to True.

### Setting a Price-Dependant (Single-Rate) Tax Using qbposXML

The following XML shows the setting of a price-dependant tax rate that triggers once the item price reaches \$1000, and results in a tax only on the price amount that is greater than \$1000.

```
<?xml version="1.0" encoding="UTF-8" ?>
<?qbposxml version="2.0"?>
<QBPOSXML>
  <QBPOSXMLMsgsRq onError="continueOnError">
    <TaxRecordModRq requestID="0">
      <TaxRecordMod>
        <TaxRecord>
          <TaxCategoryListID>TAXES</TaxCategoryListID>
          <POSTaxCodeListID>HAN</POSTaxCodeListID>
          <TaxRate>
            <TaxPercent>3.3</TaxPercent>
            <TaxAgency>Bigger Hands Helping Institute</TaxAgency>
            <TaxLowRange>1000.00</TaxLowRange>
            <IsTaxAppliedOnlyWithinRange>True</IsTaxAppliedOnlyWithinRange>
          </TaxRate>
        </TaxRecord>
      </TaxRecordMod>
    </TaxRecordModRq>
  </QBPOSXMLMsgsRq>
</QBPOSXML>
```

## Setting a Price-Dependant (Single-Rate) Tax Using qbposFC

The following VB code does the same thing as the preceding XML:

```
Dim sessionManager As QBPOSSessionManager
Set sessionManager = New QBPOSSessionManager
sessionManager.OpenConnection "123", "Tax Record Mod Sample"
sessionManager.BeginSession "Computer Name=yourComputerName;
                             Company Data=Al's Sports Hut;
                             Version=6;practice=Yes"

Dim requestMsgSet As IMsgSetRequest
Set requestMsgSet = sessionManager.CreateMsgSetRequest("2", "0")
requestMsgSet.Attributes.OnError = roeContinue

Dim MyTaxRecordMod As ITaxRecordMod
Set MyTaxRecordMod = requestMsgSet.AppendTaxRecordModRq
MyTaxRecordMod.TaxRecord.TaxCategoryListID.setValue "TAXES"
MyTaxRecordMod.TaxRecord.POSTaxCodeListID.setValue "HAN"

Dim MyRate As ITaxRate
Set MyRate = MyTaxRecordMod.TaxRecord.TaxRateList.Append
MyRate.TaxLowRange.setValue 1000.00
MyRate.IsTaxAppliedOnlyWithinRange.setValue True
MyRate.TaxAgency.setValue "Bigger Hands Helping Institute"
MyRate.TaxPercent.setValue 3.3

Dim responseMsgSet As IMsgSetResponse
Set responseMsgSet = sessionManager.DoRequests(requestMsgSet)
sessionManager.EndSession
sessionManager.CloseConnection
```

## Setting a Multi-Rate Tax (UI)

---

From the Tax Option form (Figure 10-6) select the single-rate tax and click **Next** to display the Tax Calculation form:



**Edit Tax Calculation** [X]

**Multi-rate tax information** this is <TaxRateName> in the SDK

Set up the different tax rates that apply to this tax code.

**Enter information for one of the rates that belongs to this tax code:**

Tax Component:  Tax rate:  %

Tax agency:

☐ Apply this tax rate to a specific item price range:  to

☐ Apply tax to the amount of an item's price within this

Assigned to QuickBooks Tax Item/Group:

<TaxLowRange>  
<TaxHighRange>  
<IsTaxAppliedOnlyWithinRange>

3 rates per tax code maximum

QuickBooks Tax Info

Assigned to QuickBooks Tax Code:

Figure 10-9 Setting a Multi-Rate Tax

In the tax rate form,

1. Supply the name of the tax rate (Tax Component).
2. Supply the tax percentage.
3. Optionally, supply the name of the tax authority that is to be paid the tax.
4. If you are creating a simple tax rate, leave the first checkbox unchecked.
5. If instead you want to set an amount threshold that will trigger a tax, check that first checkbox.
  - a. Specify the threshold dollar amount that will trigger the tax (the lower tax range).
  - b. Optionally, specify the higher tax range if you want to limit the tax on the upper side. (If you want the tax to apply to an item no matter how high the dollar amount, leave the higher tax range empty.)
  - c. If you check the first checkbox, to establish an amount threshold to trigger a tax, this enables the second checkbox, where you have the option of applying the tax rate to the entire amount (leaving the second checkbox unchecked), or applying the tax rate only within the tax range specified (checking the second checkbox).
6. When you are finished with the first rate, click **Add another tax rate** to fill out another tax rate following the previous steps. You can add a maximum of three tax rates per tax code.

7. If the QBPOS company is set up to work with QBFS, you can specify the QuickBooks tax code that you want to map to this tax code and rates. If the company is set up to use QBFS and you don't specify the QuickBooks tax information here, QBFS will create a new tax code using the current QBPOS tax code data.

## Setting a Multi-Rate Tax (SDK)

---

Setting a multi-rate task in the SDK follows the way you set this in the UI. For each tax rate you want to set in the tax code, you fill out a `TaxRate` aggregate which contains the tax rate name, tax percentage, tax agency, and information on how to apply the tax (e.g., `TaxLowRange/TaxHighRange`).

### Setting a Multi-Rate Tax Using `qbposXML.xml`

In the following XML, a multi-rate tax is assigned to the tax code HAN under the tax category TAXES. In this sample, the `TaxLowRange` is used to set the beginning threshold amount at which each tax starts to be applied. Because there is no `TaxHighRange` specified, each of these tax rates will continue to be applied once the beginning threshold is reached.

That is, the State Rate starts at amount 0 and is applied to all purchases regardless of amount, the Municipal Rate is an additional tax applied at all amounts greater than 100, and the Luxury tax is an additional tax applied to all purchases over the amount of 2000.

```
<?xml version="1.0" encoding="UTF-8" ?>
<?qbposxml version="2.0"?>
<QBPOSXML>
  <QBPOSXMLMsgsRq onError="continueOnError">
    <TaxRecordModRq requestID="0">
      <TaxRecordMod>
        <TaxRecord>
          <TaxCategoryListID>TAXES</TaxCategoryListID>
          <POSTaxCodeListID>HAN</POSTaxCodeListID>
          <TaxRate>
            <TaxPercent>3.3</TaxPercent>
            <TaxRateName>State Rate</TaxRateName>
            <TaxAgency>State Excess Money Dept</TaxAgency>
            <TaxLowRange>0.00</TaxLowRange>
            <IsTaxAppliedOnlyWithinRange>1</IsTaxAppliedOnlyWithinRange>
          </TaxRate>
          <TaxRate>
            <TaxPercent>2.3</TaxPercent>
            <TaxRateName>Municipal Rate</TaxRateName>
            <TaxAgency>City of East Berkeley</TaxAgency>
            <TaxLowRange>100.00</TaxLowRange>
            <IsTaxAppliedOnlyWithinRange>1</IsTaxAppliedOnlyWithinRange>
          </TaxRate>
          <TaxRate>
            <TaxPercent>5</TaxPercent>
```

```

        <TaxRateName>Luxury Tax</TaxRateName>
        <TaxAgency>State Tax Board</TaxAgency>
        <TaxLowRange>2000.00</TaxLowRange>
        <IsTaxAppliedOnlyWithinRange>1</IsTaxAppliedOnlyWithinRange>
    </TaxRate>
</TaxRecord>
</TaxRecordMod>
</TaxRecordModRq>
</QBPOSXMLMsgsRq>
</QBPOSXML>

```

## Setting a Multi-Rate Tax Using qbposFC

The following VB code does the same thing as the preceding XML:

```

Dim sessionManager As QBPOSSessionManager
Set sessionManager = New QBPOSSessionManager
sessionManager.OpenConnection "123", "Tax Record Mod Sample"
sessionManager.BeginSession "Computer Name=yourComputerName;
                             Company Data=Al's Sports Hut;
                             Version=6;practice=Yes"

Dim requestMsgSet As IMsgSetRequest
Set requestMsgSet = sessionManager.CreateMsgSetRequest("2", "0")
requestMsgSet.Attributes.OnError = roeContinue
Dim MyTaxRecordMod As ITaxRecordMod
Set MyTaxRecordMod = requestMsgSet.AppendTaxRecordModRq
MyTaxRecordMod.TaxRecord.TaxCategoryListID.SetValue "TAXES"
MyTaxRecordMod.TaxRecord.POSTaxCodeListID.SetValue "HAN"

Dim MyRate As ITaxRate
Set MyRate = MyTaxRecordMod.TaxRecord.TaxRateList.Append
MyRate.TaxLowRange.SetValue 0.00
MyRate.IsTaxAppliedOnlyWithinRange.SetValue True
MyRate.TaxAgency.SetValue "State Excess Money Dept"
MyRate.TaxPercent.SetValue 3.3
MyRate.TaxRateName.SetValue "State Rate"

Set MyRate = MyTaxRecordMod.TaxRecord.TaxRateList.Append
MyRate.TaxLowRange.SetValue 100.00
MyRate.IsTaxAppliedOnlyWithinRange.SetValue True
MyRate.TaxAgency.SetValue "City of East Berkeley"
MyRate.TaxPercent.SetValue 5.3
MyRate.TaxRateName.SetValue "Municipal Rate"

Set MyRate = MyTaxRecordMod.TaxRecord.TaxRateList.Append
MyRate.TaxLowRange.SetValue 2000.00
MyRate.IsTaxAppliedOnlyWithinRange.SetValue True
MyRate.TaxAgency.SetValue "State Tax Board"
MyRate.TaxPercent.SetValue 5.3
MyRate.TaxRateName.SetValue "Luxury Tax"

```

```
Dim responseMsgSet As IMsgSetResponse
Set responseMsgSet = sessionManager.DoRequests(requestMsgSet)
sessionManager.EndSession
sessionManager.CloseConnection
```

## CHAPTER 11

# USING UNITS OF MEASURE (UOM)

This chapter describes the QBPOS unit of measure feature and how it is supported in the SDK.

## About the QBPOS UOM Feature

QBPOS supports the use either of a single unit of measure for inventory items, or multiple units of measure. If only single UOM is turned on in company preferences (Figure 11-1), then the single UOM specified for the item is used when ordering or selling, and to track inventory quantities.

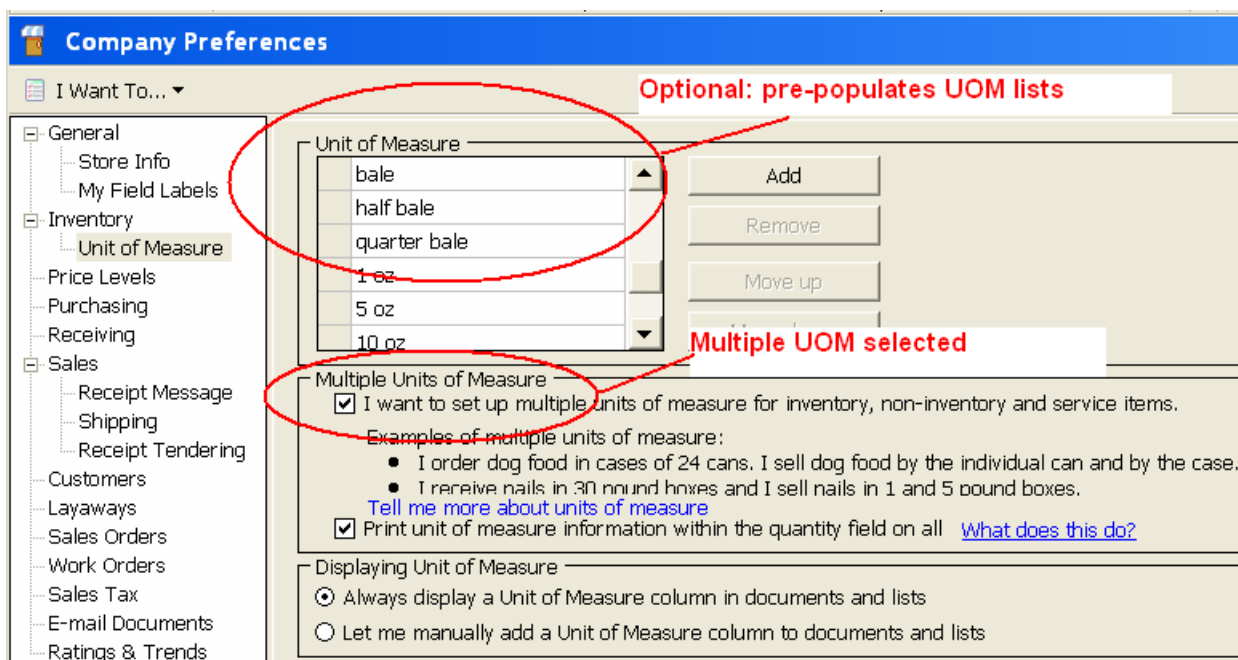


Figure 11-1 Multiple UOMs turned on in Preferences

In the preferences form for UOM, notice the Unit of Measure fields circled in red. UOMs that you add here will show up in the various UOM selection lists in the UI within inventory item forms, purchase order forms, and so forth. This is optional. You can alternatively just create the units of measure “on the fly” as you need them when you create inventory items, either through the UI or the SDK.

However, in the SDK, if you add new UOMs “on the fly” when creating new inventory items those UOMs will *not* be added to Preferences nor will they show up in UI selection lists when you create a new inventory item in the UI.

## Multiple UOM Features in the QBPOS UI

If multiple UOM is turned on in company preferences (Figure 11-1), other supporting UI features are turned on in related forms, such as the New Item or Edit Item forms as shown in Figure 11-2. However, in the SDK, the multiple UOM feature is available whether multiple UOM is turned on or off in company preferences!

The screenshot shows the 'New Item' form with several sections. The 'Item Info' tab is active, showing fields for Item Type (Inventory), Dept Name, Vendor Name, Description 1, Description 2, and Attribute. The 'Additional Info' tab is also visible, showing fields for Price & Cost (Base Unit), Quantity (Base Unit), and Units of Measure. The 'Units of Measure' section is highlighted with a red circle and contains three selection lists labeled '<UnitOfMeasure1>', '<UnitOfMeasure2>', and '<UnitOfMeasure3>'. The 'Base Unit of Measure' selection list is also circled in red. The 'Manage Units of Measure...' button is located at the bottom right of the Units of Measure section.

Figure 11-2 Multiple UOM in the New Item UI

Notice the Base Unit of Measure selection list shown circled in red in Figure 11-2. If you use multiple UOMs, you'll need to specify a base unit when you create an inventory item. This base unit is used to track quantities: inventory quantities are expressed in these base units. The base unit should be the smallest unit of that item that you sell. In the SDK, the base unit has the tag `<UnitOfMeasure>`.

Also in Figure 11-2, notice the three selection lists under the label "Unit of Measure". You use these to specify up to three other units of measure. These should be units that you purchase and/or sell the item in. In the SDK these have the tags `<UnitOfMeasure1>`, `<UnitOfMeasure2>`, and `<UnitOfMeasure3>` as shown in the figure.

The units can be ones already defined in company preferences for UOM, or you can specify new ones, both in the UI and in the SDK.

### Managing Units of Measure

In the low right of Figure 11-2, notice the Manage Units of Measure button. If you click this, the UI displays the aptly named Units of Measure form (Figure 11-3).

Figure 11-3 Manage UOM form

In the upper right of Figure 11-3, notice that you can specify different units to be used for purchasing and for selling, and the SDK tags corresponding to them. These form values don't establish a fixed unit for ordering or selling. They just establish a default that you can override later when purchasing or selling. (Either in the UI or using the `PurchaseOrderAdd` or `SalesReceiptAdd` requests.)

## Price Levels and UOM

In the SDK, for example in an `ItemInventoryAdd` request, you'll notice a set of price levels (Price1 through Price5) that apply to the item, and sets of price levels that apply to `UnitOfMeasure1` through `UnitOfMeasure3`. (See Figure 11-4 and Figure 11-5, respectively.)

Notice that if you have price levels set up in company preferences, you don't have to specify Price 2 through Price 5. You only need to specify Price 1. The price level rules in company preferences will automatically be used to calculate Price 2 through Price 5.

OnHandStore10	QUANTITY
OrderByUnit	STRTYPE
OrderCost	AMTTYPE
Price1	
Price2	
Price3	AMTTYPE
Price4	AMTTYPE
Price5	AMTTYPE
ReorderPoint	QUANTITY
SellByUnit	STRTYPE
SerialFlag	ENUMTYPE
Size	STRTYPE
TaxCode	STRTYPE
UnitOfMeasure	STRTYPE
UPC	STRTYPE

Item price for base unit

Figure 11-4 Inventory item price (base unit)

UPC	STRTYPE
VendorListID (useMacro)	IDTYPE
UnitOfMeasure1	
ALU	
MSRP	
NumberOfBaseUnits	QUANTITY
Price1	AMTTYPE
Price2	AMTTYPE
Price3	AMTTYPE
Price4	AMTTYPE
Price5	AMTTYPE
UnitOfMeasure	STRTYPE
UPC	STRTYPE
UnitOfMeasure2	
ALU	STRTYPE
MSRP	AMTTYPE
NumberOfBaseUnits	QUANTITY
Price1	AMTTYPE
Price2	AMTTYPE
Price3	AMTTYPE

Item price for UOM 1

Figure 11-5 Inventory item price (for non-base unit UOMs)



## Specifying UOMs for Inventory Items (SDK)

---

The units of measure for an inventory item (base unit and alternate units available for buying and selling) are specified in the inventory item itself. In the SDK, you specify this when you create or modify the inventory item.

### Specifying UOMs for an Item in qbposFC

---

The following VB snippet shows the adding of an inventory item with multiple units of measure: the statuette comes in packages of 1, 2, and 3. These UOMs don't exist in the company yet, so they'll be created for this item.

```
Dim sessionManager As QBPOSSessionManager
Set sessionManager = New QBPOSSessionManager
sessionManager.OpenConnection "123", "Tax Category Add Sample"
sessionManager.BeginSession "Computer Name=yourComputerName;
                             Company Data=Al's Sports Hut;
                             Version=6;practice=Yes"

Dim requestMsgSet As IMsgSetRequest
Set requestMsgSet = sessionManager.CreateMsgSetRequest("2", "0")
requestMsgSet.Attributes.OnError = roeContinue

'Adding a small statuette of Anthony Trollope to our inventory
Dim MyInventoryItem As IItemInventoryAdd
Set MyInventoryItem = requestMsgSet.AppendItemInventoryAddRq
MyInventoryItem.ALU.setValue "1234567899876"
MyInventoryItem.Attribute.setValue "sm"
MyInventoryItem.Cost.setValue 2.5
MyInventoryItem.DepartmentListID.setValue "1000000001"
MyInventoryItem.Desc1.setValue "Bust of Anthoney Trollope"
MyInventoryItem.MSRP.setValue 8.5
MyInventoryItem.OnHandStore01.setValue 11

'We'll order in packages of three and sell in packages of 1, 2, and 3.
MyInventoryItem.OrderByUnit.setValue "3-bust"
MyInventoryItem.OrderCost.setValue 7.5
MyInventoryItem.SellByUnit.setValue "1-bust"

'The price for the item in its base unit. We use price levels in
'preferences, so we just supply Pricel here.
MyInventoryItem.Pricel.setValue 5.5
MyInventoryItem.UnitOfMeasure.setValue "1-bust"
MyInventoryItem.TaxCode.setValue "TRA"
MyInventoryItem.UPC.setValue "1020101010134"
MyInventoryItem.VendorListID.setValue "1000000002"
```

```

    "We sell this item in 1, 2, and 3 packs.
    MyInventoryItem.UnitOfMeasure1.ALU.setValue "UoM1611121111211111"
    MyInventoryItem.UnitOfMeasure1.MSRP.setValue 10.5
    MyInventoryItem.UnitOfMeasure1.NumberOfBaseUnits.setValue 2
    MyInventoryItem.UnitOfMeasure1.Pricel.setValue 10.5
    MyInventoryItem.UnitOfMeasure1.UnitOfMeasure.setValue "2-bust"
    MyInventoryItem.UnitOfMeasure1.UPC.setValue "3030303060504"

    MyInventoryItem.UnitOfMeasure2.ALU.setValue "UoM1621121111211111"
    MyInventoryItem.UnitOfMeasure2.MSRP.setValue 14.5
    MyInventoryItem.UnitOfMeasure2.NumberOfBaseUnits.setValue 3
    MyInventoryItem.UnitOfMeasure2.Pricel.setValue 14.5
    MyInventoryItem.UnitOfMeasure2.UnitOfMeasure.setValue "3-bust"
    MyInventoryItem.UnitOfMeasure2.UPC.setValue "3030303070504"

    Dim responseMsgSet As IMsgSetResponse
    Set responseMsgSet = sessionManager.DoRequests(requestMsgSet)
    sessionManager.EndSession
    sessionManager.CloseConnection

```

## Specifying UOMs for an Item in qbposXML

---

The following XML constructs the same request as the qbposFC code does in the preceding section.

```

<?xml version="1.0" encoding="UTF-8" ?>
<?qbposxml version="2.0"?>
<QBPOSXML>
  <QBPOSXMLMsgsRq onError="continueOnError">
    <ItemInventoryAddRq requestID="0">
      <ItemInventoryAdd>
        <ALU>1234567899876</ALU>
        <Attribute>sm</Attribute>
        <Cost>2.50</Cost>
        <DepartmentListID>1000000001</DepartmentListID>
        <Desc1>Bust of Anthoney Trollope</Desc1>
        <MSRP>8.50</MSRP>
        <OnHandStore01>50</OnHandStore01>
        <OrderByUnit>3-bust</OrderByUnit>
        <OrderCost>7.50</OrderCost>
        <Pricel>5.50</Pricel>
        <SellByUnit>1-bust</SellByUnit>
        <TaxCode>TRA</TaxCode>
        <UnitOfMeasure>1-bust</UnitOfMeasure>
        <UPC>1020101010134</UPC>
        <VendorListID>1000000002</VendorListID>
      </ItemInventoryAdd>
    </ItemInventoryAddRq>
  </QBPOSXMLMsgsRq>
</QBPOSXML>

```

```

<UnitOfMeasure1>
  <ALU>UoM16111211111211111</ALU>
  <MSRP>10.50</MSRP>
  <NumberOfBaseUnits>2</NumberOfBaseUnits>
  <Price1>10.50</Price1>
  <UnitOfMeasure>2-bust</UnitOfMeasure>
  <UPC>3030303060504</UPC>
</UnitOfMeasure1>
<UnitOfMeasure2>
  <ALU>UoM16211211111211111</ALU>
  <MSRP>14.50</MSRP>
  <NumberOfBaseUnits>3</NumberOfBaseUnits>
  <Price1>14.50</Price1>
  <UnitOfMeasure>3-bust</UnitOfMeasure>
  <UPC>3030303070504</UPC>
</UnitOfMeasure2>
</ItemInventoryAdd>
</ItemInventoryAddRq>
</QBPOSXMLMsgsRq>
</QBPOSXML>

```



# APPENDIX A

## STATUS CODES FOR qbposxml RESPONSES

This appendix lists the status codes returned in the qbposxml statusCode attribute.

Table A-1 lists the status code, gives its meaning, and explains the condition that the code represents.

Table A-1 Status Codes and Their Meaning

Status Code	Meaning	Explanation
20000	Unknown field(%s)	In Add or Mod request, an element was specified that was not recognized.
20001	Unknown field type (%s)	The field mentioned is in schema, but has a type which is not supported by SDK. This is an SDK bug.
20002	Unsupported object %s	Object name part of the request is not one of the recognized by the POS SDK.
20003	Item %d not found	ListID reference points to a non-existing inventory item when adding an item to a document.
20004	The receipt is not balanced	The sum of tender items on the receipt does not match the receipt total.
20005	Unknown tender type %s	One of the TenderXXXAdd elements in SalesReceiptAdd refers to an unknown tender type.
20006	Customer %d not found	A customer referred by ListID (on Receipt or Sales Order) is not found.
20007	SalesOrder %d not found	Sales order, referred by TxnID on the Receipt, is not found.
20008	Enumerated value %s not found for field %s	For the mentioned enumerated field, a mentioned value is not found in the lookup list. Affects Add, Mod, or Query.
20009	Items are only allowed on Sales and Return Receipts	Deposit, Refund, Payout and Payin types of Sales Receipt cannot have inventory items.
20010	Department Code %s not found	In ItemInventoryAdd or ItemInventoryMod, a non-existent department code is given.
20011	Vendor Code %s not found	When referring to an unexisting Vendor code in Inventory, PurchaseOrder or Voucher requests.
20012	Unknown macro field %s	defMacro refers to a field that doesn't exist in the schema.
20013	Unsupported macro field %s	Currently, only bigint (ListID and TxnID) are supported by macros. It can change, though, over time.
20014	Macro %s not found	Attempt to use a macro which was not defined.
20015	Customer Ship Same As should be set to false before setting shipping info	Customer "Ship Same As" boolean value is set to true, but user tries to assign some shipping info on the customer record.

Status Code	Meaning	Explanation
20016	The value (%s) for the field (%s) is greater than the allowed size (%d)	Field size limit exceeded.
20017	Invalid Store Number	Store number is less than 1 or greater then the number of stores defined.
20018	Invalid UPC value - %s	UPC code is a string, but it should contain only digits. If a string value cannot be converted to an UPC code, this error is given.
20019	The field %s is read-only and cannot be assigned	The mentioned field is marked as read-only in the schema.
20020	BUG - Lookup list is not formed for %s	When accessing an enumerated field, SDK did not form internal list of choices. This is an SDK bug.
20021	Duplicate Department Code %s	In DepartmentAdd or DepartmentMod request, department code matches one of another department record.
20022	Duplicate Vendor Code %s	In VendorAdd or VendorMod, the VendorCode matches the one of another vendor
20023	Exclusive fields - %s	Some fields are mutually-exclusive, for example Discount amount and discount %. A comma-delimited list of fields is given in the error message.
20024	The field %s cannot be assigned - invalid store number	OnHandStoreX is for Store which number exceeds the number of stores defined.
20025	Duplicate Vendor Code within item %s	Inventory item allows 5 vendor codes. They should be unique within the item.
20026	Duplicate ALU within item %s	5 ALU's can be placed on inventory item. They should be unique within the item.
20027	Duplicate UPC within item %s	5 UPC's are allowed per inventory item. They should be unique within the item.
20028	Duplicate ALU %s	Any of the ALU's on the inventory item should not be the same as any of the ALU's on another item.
20029	Duplicate UPC %s	Any of the UPC's on the inventory item should not be the same as any of the UPC's on another item.
20030	Duplicate attribute/size combination for style %s,%s,%s,%s	The style is a set of items with the same department code and description1. Within a style, all items should have unique attribute/size combination. Exception - blank attribute and size. Parameters to this error message are Department Code, Description, Attr, Size.
20031	%s - a filter for this field is already defined	In one query, there is more than one filter for the same field.
20032	Unsupported filter (%s)	There was an error parsing the filter mentioned other than incorrect criterion value (code 20033)
20033	Unknown filter criteria (%s)	The criterion part of the match filter is no one of the known values.
20034	QBPOSXML: Unknown request version	request version is missing.

Status Code	Meaning	Explanation
20035	%s Error processing %s. Source %s. Line %d, Position %d.	Parse error.
20036	Extension error (%s)	One of the errors in data extension system
20037	The field %s is required	If a required field was not assigned on "Add" request or erased on "Mod" request.
20038	Internal error (%s)	A non-SDK exception was raised.
20039	Customer Price Level can be set only if Customer Discount Type is set to "Price Level"	Following the business logic of POS UI for customer.
20040	Customer Discount Percentage can be set only if Customer Discount Type is set to "Percentage"	Following the business logic of POS UI for customer.
20041	Account can not be assigned - "Use with QBPOS" setting is off	Inventory item can have Income Account and COGS account assigned, but only if QBPOS integration is on.
20042	Account %s is not found	When trying to assign Income or COGS account to an inventory item, but the account is not found by POS.
21000	Can't connect to the database	Possible reasons: -Invalid computer name -Invalid company data name -The computer mentioned has different company data open -User Name/Password is absent, but the data file requires user login. -Invalid User Name/Password -Client app and POS DB server are on the same machine, but POS database service is not started.
21001	Session is not created	When trying to execute a request without establishing a connection
21002	Ticket mismatch	A ticket is not from the currently established session.
21003	Connection is not open	When trying to create a session before opening a connection.
21004	Application ID is empty	Application ID is required when establishing a session.
21005	The database is empty	The database does not contain data needed for POS functioning. This error shouldn't occur under normal conditions, but can be a result of a bug or user experiments.
21006	The database is shutting down	Some POS processes require shutting down the database. It is done through setting a special flag in the DB that makes it reject all connections from POS clients. We can't allow SDK clients connect if the flag is set.
21007	Database schema version mismatch	If the DB version is newer than SDK version.
21008	Access Denied	User clicked "No" in the access control dialog.

<b>Status Code</b>	<b>Meaning</b>	<b>Explanation</b>
21009	An active session already exists	Begin session is called while another session is already created.
21010	Already connected	You need to disconnect before you connect again.
21011	This user is not permitted to run third-party applications. See 'Login from an integrated application' security right in POS	Some POS users can be prevented from running a 3 <sup>rd</sup> party apps.



## APPENDIX B

### REQUEST PROCESSOR API REFERENCE

This appendix provides alphabetical reference pages for the methods included in the qbposXML Request Processor COM API.

Table B-1 Methods and Properties for RequestProcessor2

Name	Description
BeginSession	Begins a session operating on a specific QBPOS company.
CloseConnection	Closes the connection between your application and QBPOS
EndSession	Ends the session.
GetCurrentCompanyFileName	Returns the name of the company file that is currently open (requires a ticket)
MajorVersion	Returns the major version number of the qbposXML Request Processor.
MinorVersion	Returns the minor version number of the qbposXML Request Processor.
OpenConnection	Opens a connection between your application and QBPOS.
POSServers	Returns the QBPOS servers that are available in the local network.
POSVersions	For the specified company, returns the versions of that company database that are available, permitting user selection of the desired version.
ProcessRequest	Sends the specified request message set to QBPOS and returns with a response message set from QBPOS.
QBPOSXMLVersionsForSession	Returns the versions of the qbXML specification supported by the QBPOS product your application is currently connected to (requires a ticket)
ReleaseLevel	Returns the release description of the qbposXML Request Processor (for example, <i>alpha</i> , <i>beta</i> , or <i>release</i> )
ReleaseNumber	Returns the release number of the qbposXML Request Processor.

# BeginSession

---

```
HRESULT BeginSession(  
    [in] BSTR qbFileName,  
    [out, retval] BSTR* pTicket);
```

After a QBPOS connection has been established, this method begins a session working on the QBPOS company specified in. The returned session ticket is passed in to subsequent calls to ProcessRequest.

## Parameters

*qbFileName* String containing the connection parameters. The string contains the parameters Computer Name=*MyComputer*;Company Data=*My Company*;Version=*version\_number*, where you replace *MyComputer* with the network machine name of the system running QBPOS, replace *My Company* with the name of the QBPOS company that your application is going to access, and *version\_number* with the number of the QBPOS version you want to access. Optionally, you can supply the parameter Practice=Yes if you want to work with the practice version of that company file.

Notice that the parameters are separated by a semicolon.

You can alternatively supply qbFileName as an empty string. If you do this, the local network will be searched for QBPOS instances and QBPOS companies and the resulting list of them is automatically displayed to the QBPOS user at runtime for the user's selection.

*ticket* Pointer to the returned session ticket. Your application must save this ticket (store it or keep it in memory) and pass it to any ProcessRequest, GetCurrentCompanyFileName, and QBPOSXMLVersionsForSession calls made in this session only. When you are finished with the session, you release the ticket by passing it to the EndSession call that terminates the session. QBPOS frees the memory allocated for the ticket when you return it.

## Usage

For more details on using this call, see Chapter 2, "Communicating with QBPOS." That chapter also contains sample code.

## Sample Connection String

```
MyComputer=JohnsSystem;Company Data=My Company;Version=5;Practice=No
```

## CloseConnection

---

```
HRESULT CloseConnection();
```

Closes the connection with QBPOS.

### **Usage**

For more details on using this call, see Chapter 2, “Communicating with QBPOS.” That chapter also contains sample code.

# EndSession

---

```
HRESULT EndSession([in] BSTR ticket);
```

Frees resources, closes the company data access, and ends the session.

## Parameters

*ticket*

Ticket for the current session. This method releases the memory allocated for the ticket, and the ticket can no longer be used.

## Usage

# GetCurrentCompanyFileName

---

```
HRESULT GetCurrentCompanyFileName([in] BSTR ticket,  
                                  [out, retval] BSTR* pFileName);
```

Returns the name of the currently open company file.

## Parameters

<i>ticket</i>	Ticket for the current session.
<i>pFileName</i>	Pointer to the returned name of the company currently open in QBPOS.

## Usage

This method can be used any time the application needs to display the name of the company—for example, when it asks the user to confirm modification of data contained in a particular company. If you call `BeginSession` and do not explicitly specify a company name (and the call succeeds), you should then call `GetCurrentCompanyFileName` to obtain the name of the company that is currently open.

## Example (Visual Basic)

```
Dim qbposXMLRP as QBPOSXMLRPLib.RequestProcessor  
Dim strQBFileName as String  
strQBFileName = qbposXMLRP.GetCurrentCompanyFileName(strTicket)
```

## MajorVersion

---

```
HRESULT MajorVersion([[out, retval] short* pMajorVersion);
```

Returns the major version number of the qbposXML Request Processor.

### Parameters

<i>pMajorVersion</i>	Pointer to the returned major version number of the qbposXML Request Processor.
----------------------	---

### Usage

For version 1.0, the major version number of the qbposXML Request Processor is 1.

### Example (Visual Basic)

```
Dim qbposXMLRP as QBPOSXMLRPLib.RequestProcessor
Dim iMajorVersion as integer
iMajorVersion = qbposXMLRP.MajorVersion
```

## MinorVersion

---

```
HRESULT MinorVersion([out, retval] short* pMinorVersion);
```

Returns the minor version number of the qbposXML Request Processor.

### Parameters

<i>pMinorVersion</i>	Pointer to the returned minor version number of the qbposXML Request Processor.
----------------------	---

### Usage

For version 1.0, the minor version number of the qbposXML Request Processor is 0.

### Example (Visual Basic)

```
Dim qbposXMLRP as QBPOSXMLRPLib.RequestProcessor  
Dim iMinorVersion as integer  
iMinorVersion = qbposXMLRP.MinorVersion
```

## OpenConnection

---

```
HRESULT OpenConnection(  
    [in] BSTR appID,  
    [in] BSTR appName);
```

Opens the connection with QBPOS. The AppID and the AppName values can be any values you want.

### **Usage**



# POSServers

---

```
HRESULT POSServers(  
    [in] VARIANT_BOOL IsPractice,  
    [out, retval] BSTR* ServersXML);
```

Returns data about the available QBPOS instances, companies, and versions in the local network.

## Parameters

*IsPractice* Specify True if the company is the QBPOS practice company. This can decrease the lookup time by restricting the search to the test company. Useful for development. For real deployments, however, this value would be set to False to make sure all companies were listed.

*ServersXML* Pointer to the returned string containing an XML document containing the list of available servers.

## Usage

This method returns the available servers and companies in a string that has the following format:

```
<POSServers>  
  <POSServer>  
    <ServerName>QBPOS HostMachineName</ServerName>  
    <CompanyName>QBPOS Company Name</CompanyName>  
    <Version>QBPOS version number</Version>  
  </POSServer>  
</POSServers>
```

The example above is the string returned when only one QBPOS instance, company, and version is found. If there were multiples found, then there would be a separate group of server name, company name, and version elements nested under a separate <POSServer> tag.

The example

## Example (Visual Basic)

```
Private Sub getServers()  
    Dim sessionManager As New QBPOSSessionManager  
    Dim posserversXML As String  
    posserversXML = sessionManager.POSServers(True)  
    Dim xmlDoc As New DOMDocument40  
    Dim objNodeList As IXMLDOMNodeList  
    Dim objChild As IXMLDOMNode  
    Dim childNode As IXMLDOMNode  
    Dim i As Integer  
    Dim ret As Boolean  
    Dim errorMsg As String
```

```

errorMsg = ""
ret = xmlDoc.loadXML(posserversXML)
If Not ret Then
    errorMsg = "loadXML failed, reason: " & xmlDoc.parseError.Reason
    GoTo ErrHandler
End If

Dim server As String
Set objNodeList = xmlDoc.getElementsByTagName("POSServer")
For i = 0 To (objNodeList.length - 1)
    For Each objChild In objNodeList.Item(i).childNodes
        If objChild.nodeName = "ServerName" Then
            server = objChild.Text
        End If
        If objChild.nodeName = "CompanyName" Then
            server = server + " - " + objChild.Text
        End If
        If objChild.nodeName = "Version" Then
            server = server + " - " + objChild.Text
        End If
        ComboServer.List(i) = server
    Next
Next
Exit Sub

ErrHandler:
MsgBox Err.Description, vbExclamation, "Error"
Exit Sub

End Sub

```

# POSVersions

---

```
HRESULT POSVersions(  
    [in] BSTR ServerName,  
    [in] BSTR CompanyName,  
    [in] VARIANT_BOOL IsPractice,  
    [out, retval] VARIANT* VersionArray);
```

Returns an array of the available versions for the specified company and QBPOS host machine. This allows you to select the latest version or present the user with a choice of available company versions.

## Parameters

<i>ServerName</i>	The machine name of the computer hosting the desired QBPOS instance.
<i>CompanyName</i>	The name of the QBPOS company to be checked for versions.
<i>IsPractice</i>	Specify True if the company is the QBPOS practice company. This can decrease the lookup time by restricting the search to the test company. Useful for development. For real deployments, however, this value would be set to False to make sure all companies were listed.
<i>VersionArray</i>	Pointer to the returned array containing a list of versions.

## Usage

When a new major release of QBPOS is installed over an existing installation that has companies, those companies are copied and the copies are upgraded to the new schema version. However, the old companies are still there, but have the old schema. For example, QBPOS 4.0 has the version 4. The 5.0 version of QBPOS will have the version 5.

By default your application will continue to log into the existing (old) version, which may be undesirable if the user has transitioned to the company that was upgraded to the new schema version. This means your application might write data to the wrong company.

To avoid this, your application should save its current version number during its session. Then, at application startup time, your application should check the versions using this method call to determine whether a new version is available. You can display this information to the user for the user to select the proper version.

# ProcessRequest

---

```
HRESULT ProcessRequest(  
    [in] BSTR ticket,  
    [in] BSTR inputRequest,  
    [out, retval] BSTR* outputResponse);
```

Sends the request message set to QBPOS for processing and receives the corresponding response message set from QBPOS.

## Parameters

<i>ticket</i>	Ticket for the current session, returned by the BeginSession method.
<i>inputRequest</i>	Text stream containing the qbposXML request message set to be processed by QBPOS.
<i>outputResponse</i>	Pointer to the returned qbposXML response message set from QBPOS. The memory for this string is allocated on behalf of your application; however, it is your application's responsibility to release the memory when it is finished using it.

## Usage

The ProcessRequest method sends the request message set to QBPOS. It waits while QBPOS validates your qbposXML document, processes the requests, and creates the response qbposXML document. Upon successful return of this method, the *outputResponse* parameter contains the response from QBPOS.

You may want to validate the qbposXML text stream contained in the *inputRequest* parameter before you issue this request. The SDK contains an example of an external qbposXML validation tool that you can use during the design and development phases of your application. Later, you may want to build a qbposXML validator into your application.

## Example (Visual Basic)

```
Dim qbposXMLRP as QBPOSXMLRPLib.RequestProcessor  
Dim strXMLResponse as String  
strXMLResponse = qbposXMLRP.ProcessRequest(strTicket, strXMLRequest)
```

## QBPOSXMLVersionsForSession

---

```
HRESULT QBPOSXMLVersionsForSession(  
    [in] BSTR ticket,  
    [out, retval] VARIANT* ppsa);
```

Returns an array containing the version numbers of the DTDs supported by the Request Processor. Note that this information may be different from the information returned by a Host Query request, as described in the *Concepts Manual*. HostQuery returns the complete list of all qbXML versions supported by the *currently open connection*, which is usually the information your application will require.

### Parameters

<i>ticket</i>	Session ticket (returned by BeginSession).
<i>ppsa</i>	Pointer pointer to an array of binary strings that specify the versions of the qbposXML specification that are supported by the current QBPOS Request Processor. For example, the array contains 1.0, 1 if your application is using QBPOSXMLRP from the QBPOS SDK 1.0

### Usage

Your application is responsible for freeing the memory used for the *ppsa* array. For example, to release the memory for the array when using the C++ language, call `SafeArrayDestroy(ppsa)`.

### Example (Visual Basic)

```
Dim qbposXMLRP as QBPOSXMLRPLib.RequestProcessor  
Dim strXMLVersionsArray() as String  
strXMLVersionsArray = qbposXMLRP.QBXMLVersionsForSession(strTicket)
```

## ReleaseLevel

---

```
HRESULT ReleaseLevel  
([out, retval] QBPOSXMLRPReleaseLevel* Value);
```

Returns the release level of the qbposXML Request Processor.

**Parameters**

<i>pReleaseLevel</i>	Pointer to the returned release level. This value can be <i>preAlpha</i> , <i>alpha</i> , <i>beta</i> , or <i>release</i> .
----------------------	---

## ReleaseNumber

---

```
HRESULT ReleaseNumber([out, retval] short* Value);
```

Returns the release number of the qbposXML Request Processor.

**Parameters**

*pReleaseLevel*            Pointer to the returned release number.





## APPENDIX C

### QBPOSFC LANGUAGE REFERENCE

This chapter is a brief language reference for the main objects and object methods in the qbposFC library. If you are not yet comfortable with qbposFC programming, you should also read Chapter 5, “Building Requests and Processing Responses.”

## QBPOSSessionManager Object and Methods

The following table lists the QBPOSSessionManager methods by functional area.

Table B-1 QBPOSSessionManager Methods/Properties Grouped by Functional Area

Functionality	Supporting Methods/Properties	Notes
Connection and session management.	BeginSession CloseConnection EndSession OpenConnection POSServers POSVersions	You need to invoke POS Servers and POSVersions  OpenConnection followed by BeginSession before you can send requests to QB. (When you're finished, you need to invoke EndSession to release resources.)
Create message set request object.	CreateMsgSetRequest	CreateMsgSetRequest creates an empty IMsgSetRequest object to which you then append your desired individual requests.
Send requests to the request processor.	DoRequests DoRequestsFromXMLString	DoRequests sends the fully constructed IMsgSetRequest object to QBPOS.  DoRequestsFromXMLString does the same thing, but uses a fully constructed qbposXML message set instead of qbposFC objects.
Convert qbposXML to qbposFC objects	ToMsgSetRequest ToMsgSetResponse	These convenience methods take fully constructed qbposXML message sets and constructs the appropriate qbposFC objects.
Get context information.	GetCurrentCompanyFileName, GetVersion QBPOSXMLVersionsForSession	These method calls respectively return the name of the company currently connected to, the POS version currently being used, and the qbposXML spec versions supported by the QBPOS installation currently connected to.

## QBPOSSessionManager.BeginSession

---

```
void BeginSession(  
    [in] BSTR qbFileName);
```

After a QBPOS connection has been established, this method begins a session working on the QBPOS company under the QBPOS server and version specified in *qbFileName*.

### Parameters

*qbFileName* String containing the connection parameters. The string contains the parameters Computer Name=*MyComputer*;Company Data=*My Company*;Version=*version\_number*, where you replace *MyComputer* with the network machine name of the system running QBPOS, replace *My Company* with the name of the QBPOS company that your application is going to access, and *version\_number* with the number of the QBPOS version you want to access. Optionally, you can supply the parameter Practice=Yes if you want to work with the practice version of that company file.

You can alternatively supply qbFileName as an empty string. If you do this, the local network will be searched for QBPOS instances and QBPOS companies and the resulting list of them is automatically displayed to the QBPOS user at runtime for the user's selection.

### Usage

For more details on using this call, see Chapter 2, "Communicating with QBPOS." That chapter also contains sample code.

### Sample Connection String

```
MyComputer=JohnsSystem;Company Data=My Company;Version=5;Practice=No
```

## QBPOSSessionManager.CloseConnection

---

```
void CloseConnection ()
```

Closes the connection with QBPOS.

### **Usage**

For more details on using this call, see Chapter 2, “Communicating with QBPOS.” That chapter also contains sample code.

## QBPOSSessionManager.CreateMsgSetRequest

---

```
IMsgSetRequest* CreateMsgSetRequest(  
    short xmlMajorVersion,  
    short xmlMinorVersion);
```

Tells qbposFC which version of qbXML your application is using, and returns the request message set object in return, which you then fill with individual requests via the IMsgSetRequest object's Append\* methods. .

### Parameters

<i>xmlMajorVersion</i>	The major version of qbposXML to be used by your application. (For example, if the version is 2.0, the major version is 2.)
<i>xmlMinorVersion</i>	The minor version of qbposXML to be used by your application. (For example, if the version is 2.0, the minor version is 0.)

### NOTE

qbposFC supports qbposXML versions 1.0, 1.1, 1.2, and 2.0.

### Usage

It is important to set the qbXML version appropriately:

- Make sure your end-users have a version of QBPOS that supports the qbposXML version you specify.
- Note that you cannot use any qbposXML functionality that was added since the version you specify.

## QBPOSSessionManager.DoRequests

---

```
IMsgSetResponse* DoRequests( IMsgSetRequest* request );
```

Sends the specified XML request to QuickBooks and returns a response.

### Parameters

<i>request</i>	The request message set object containing any number of requests to be processed by QuickBooks.
----------------	---

### Return Value

<i>IMsgSetResponse</i>	The response message set object, which will contain a list of responses, one for every request in the request message set.
------------------------	--

### Usage

Before sending the request set to QBPOS for processing, this method verifies the given request message set to make sure that all the mandatory fields in each of its requests have been set and that no two mutually exclusive fields have been set.

### DoRequests Code Sample

What is difficult about DoRequests is the constructing of the request message set required prior to invoking DoRequests and the processing of the response that is returned from it. These topics are covered in Chapter 5, “Building Requests and Processing Responses.”

## QBPOSSessionManager.DoRequestsFromXMLString

---

```
IMsgSetResponse* DoRequestsFromXMLString(BSTR xmlRequest);
```

Sends an XML request as a string rather than in an IMsgSetRequest object (as happens in DoRequests).

### Parameters

<i>xmlRequest</i>	The request is not validated before it is sent. Any <i>requestIDs</i> within the request must start with 0 and continue with 1, 2, and so on.
-------------------	---

### Return Values

IMsgSetResponse	A parsed list of responses, one for every request in the request message set.
-----------------	---

## QBPOSSessionManager.EndSession

---

```
void EndSession ()
```

Terminates the session with the current company. When the QBPOSSessionManager object goes out of scope, qbposFC will automatically call EndSession if it wasn't called explicitly before. (It also calls CloseConnection, if necessary.)

## QBPOSSessionManager.GetCurrentCompanyName

---

```
BSTR GetCurrentCompanyName();
```

Returns the name (including path) of the QBPOS company data being accessed in the current session.

This method can be used any time the application needs to display the name of the company—for example, when it asks the user to confirm modification of data contained in a particular company. If you call `BeginSession` and do not explicitly specify a company name (and the call succeeds), you should then call `GetCurrentCompanyName` to obtain the name of the company that is currently open.



## QBPOSSessionManager.GetVersion

---

```
void GetVersion(  
    [out] short* majorVersion,  
    [out] short* minorVersion,  
    [out] ENReleaseLevel* releaseLevel,  
    [out] short* releaseNumber);
```

Returns version and release information for the qbposFC Library currently being used.

### Parameters

<i>majorVersion</i>	The major version for the qbposFC Library. For example, for qbposFC2 the major version is 2.
<i>minorVersion</i>	The minor version for the qbposFC Library. For example, for qbposFC2 the minor version is 0.)
<i>releaseLevel</i>	The release level for the qbposFC Library. It can be one of four values: rlPreAlpha, rlAlpha, rlBeta, or rlRelease.
<i>releaseNumber</i>	The release number for the qbposFC Library. It corresponds to a specific build of the product software.

## QBPOSSessionManager.OpenConnection

---

```
void OpenConnection(  
    BSTR appID,  
    BSTR appName);
```

Opens the connection with QBPOS. The AppID and the AppName values can be any values you want.

## QBPOSSessionManager.POSServers

---

```
BSTR POServers([in] VARIANT_BOOL isPractice);
```

Returns data about the available QBPOS instances, companies, and versions in the local network.

### Parameters

*IsPractice* Specify True if the company is the QBPOS practice company. This can decrease the lookup time by restricting the search to the test company. Useful for development. For real deployments, however, this value would be set to False to make sure all companies were listed.

### Return Values

*BSTR* Returned list of available QBPOS servers.

### Usage

This method returns the available servers and companies in a string that has the following format:

```
<POSServers>
  <POSServer>
    <ServerName>QBPOS HostMachineName</ServerName>
    <CompanyName>QBPOS Company Name</CompanyName>
    <Version>QBPOS version number</Version>
  </POSServer>
</POSServers>
```

The example above is the string returned when only one QBPOS instance, company, and version is found. If there were multiples found, then there would be a separate group of server name, company name, and version elements nested under a separate <POSServer> tag.

## QBPOSSessionManager.POSVersions

---

```
SAFEARRAY(BSTR) POSVersions(  
    [in] BSTR serverName,  
    [in] BSTR CompanyName,  
    [in] VARIANT_BOOL isPractice);
```

Returns an array of the available versions for the specified company and QBPOS host machine. This allows you to select the latest version or present the user with a choice of available company versions.

### Parameters

<i>ServerName</i>	The machine name of the computer hosting the desired QBPOS instance.
<i>CompanyName</i>	The name of the QBPOS company to be checked for versions.
<i>IsPractice</i>	Specify True if the company is the QBPOS practice company. This can decrease the lookup time by restricting the search to the test company. Useful for development. For real deployments, however, this value would be set to False to make sure all companies were listed.

### Return Values

<i>SAFEARRAY</i>	Array containing a list of versions.
------------------	--------------------------------------

### Usage

When a new major release of QBPOS is installed over an existing installation that has companies, those companies are copied and the copies are upgraded to the new schema version. However, the old companies are still there, but have the old schema. For example, QBPOS 4.0 has the version 4. The 5.0 version of QBPOS will have the version 5.

By default your application will continue to log into the existing (old) version, which may be undesirable if the user has transitioned to the company that was upgraded to the new schema version. This means your application might write data to the wrong company.

To avoid this, your application should save its current version number during its session. Then, at application startup time, you application should check the versions using this method call to determine whether a new version is available. You can display this information to the user for the user to select the proper version.

## QBPOSSessionManager.QBXMLVersionsForSession

---

```
SAFEARRAY(BSTR) QBPOSXMLVersionsForSession();
```

Returns an array containing the version numbers of the DTDs supported by the Request Processor. Note that this information may be different from the information returned by a Host Query request, as described in the *Concepts Manual*. HostQuery returns the complete list of all qbXML versions supported by the *currently open connection*, which is usually the information your application will require.

### Return Values

BSTR	Array of strings that specify the versions of the qbposXML specification that are supported by the current QBPOS Request Processor. For example, the array contains 2.0, 1.2, 1.1, 1.0 if your application is using qbposFC from the QBPOS SDK 2.0
------	--

## QBPOSSessionManager.ToMsgSetRequest

---

```
IMsgSetRequest* ToMsgSetRequest(BSTR xmlRequest);
```

Takes a validly constructed qbposXML request string and parses it into an IMsgSetRequest object. Reads the qbposXML major and minor version numbers from the request.

## QBPOSSessionManager.ToMsgSetResponse

---

```
IMsgSetResponse* ToMsgSetResponse(  
    BSTR xmlResponse,  
    short xmlMajorVersion,  
    short xmlMinorVersion);
```

Parses the XML response and returns it in a qbposFC response message set object.

### **IMPORTANT**

This method performs NO version or qbXML validation of the supplied XML string. You are responsible for supplying a valid string to this method call.

#### **Parameters**

<i>xmlResponse</i>	The qbposXML response string that gets returned from QBPOS after processing a request message set sent via the qbposXML request processor.
<i>xmlMajorVersion</i>	The major version of qbposXML to be used for this operation. (For example, if the version is 2.0, the major version is 2.)
<i>xmlMinorVersion</i>	The minor version of qbposXML to be used for this operation. (For example, if the version is 2.0, the minor version is 0.)

#### **Parameters**

<i>IMsgSetResponse</i>	The response message set object, which will contain a list of responses, one for every request in the request message set.
------------------------	--

