Productive Computing Creating Efficiency Through Automation

FM Books Connector

Developer's Guide

Revised January 29, 2021

Table of Contents

I. Introduction

Description:
The FM Books Connector plug-in is a powerful tool used to move data between FileMaker® Pro and Intuit® QuickBooks applications. The Developer's Guide will explain the necessary integration steps, how the plug-in "talks" to QuickBooks and the concept of how to create your FileMaker scripts which can be applied to any area in QuickBooks. Understanding the concept of the script construction and how the data exchange functions will save a tremendous amount of time and confusion during the plug-in integration process.

Product Version History:
The version history can be found at the bottom of the product page on our website:
https://www.productivecomputing.com/products/filemaker-quickbooks-integration/

Intended Audience:
Intermediate to advanced developers or persons with knowledge of FileMaker Pro or FileMaker Pro Advanced, especially in the areas of scripting, calculations and relationships as proper use of the plug-in requires that FileMaker integration scripts be created in your FileMaker solution.

Successful Integration Practices:
Familiarize yourself with basic accounting practices and QuickBooks
Read the Developer's Guide:
https://www.productivecomputing.com/documents/FM_Books_Connector/Developers_Guide_FM_Books_Connector.pdf
Read the Functions Guide:
https://www.productivecomputing.com/documents/FM_Books_Connector/Functions_Guide_FM_Books_Connector.pdf
Review articles in our help center:
https://help.productivecomputing.com/help/fm-books-connector
Download the demo:
https://www.productivecomputing.com/products/filemaker-quickbooks-integration/
Watch our video tutorials on YouTube for installing the plug-in and using the demo file:
https://www.youtube.com/watch?v=zSEh8MerdSw&list=PLF9936A5DB331BCF2
Purchase our training course for more detailed instructions:
https://www.productivecomputinguniversity.com/courses/fm-books-connector-for-quickbooks-desktop
Use the OSR (Intuit's Onscreen Reference Manual):
OSR: https://static.developer.intuit.com/qbSDK-current/common/newosr/index.html

II. Integration Steps

Accessing and using the plug-in functions involve the following steps.

1) Installation Components - Prerequisites for Plug-in Installation

32-Bit and 64-bit
FM Books Connector is compatible with the 32-bit and 64-bit versions of FileMaker and QuickBooks, where applicable. 64-bit versions of FileMaker will require an additional component, the "QB Bridge", to be installed alongside the plug-in; this is handled during the installation process

QuickBooks Prerequisites
You will need to install various components in order to establish a connection with a QuickBooks file.

If you are using QuickBooks Online, please see FM Books Connector Online edition at https://www.fmbooksconnectoronline.com/.

Installing the QB Bridge File:
This installer is found in the FM Books Connector Extras folder. If you are installing the plug-in using the "Install FM Books Connector" installer package, this step can be ignored; otherwise, simply unzip the "Install FM Books QB Bridge" zip package and run the resulting "setup.exe" file to initiate the installation process.

Important note: As of version 9.0.0.2, the FM QB Bridge will only be utilized by the 64-bit version of the FM Books Connector plug-in. For more information on this, please review the version history for FM Books Connector v9.0.0.2.

Installing the Microsoft XML Processor:

Included in the package is a download link for all Windows users.

The name of the link is: "Install MSXML processor update from Microsoft (Required Install)"

This link will direct you to install version 6 of the MSXML Processor from Microsoft. We use version 6 of the MSXML processor in the plug-in and require that you install his processor. Please save the file to your desktop and then run the file locally from your machine in order to ensure proper installation.

Installing the Microsoft Visual C++ 2013 Redistributable Package:
Included in the package is a download link. The name of the link is: "Download Microsoft Visual C++ 2013 Redistributable Package (x86)"

This link will direct you to download the Microsoft Visual C++ Redistributable Package (x86). Some systems do not have a Visual C++ 2013 Redistributable Package installed by default. However, certain programs may have added it to your machine during their installation process.

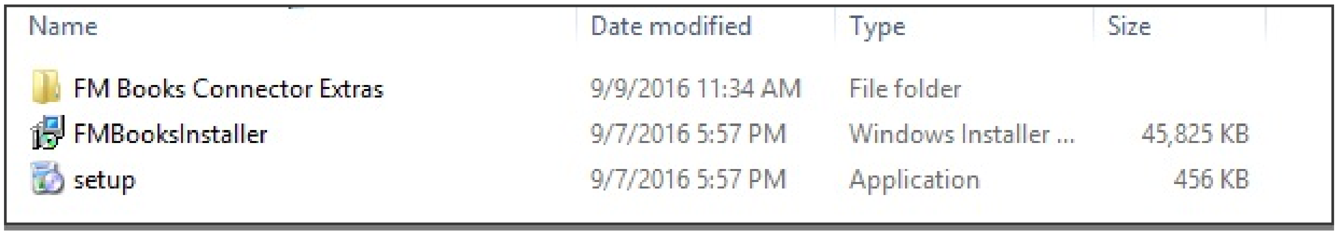
If the plug-in fails to be recognized by FileMaker after installation (i.e. does not show up in the Edit > Preferences > Plug-ins section), then please install the included redistributable package. Machines running 64-bit versions of Windows need to install the 64-bit ("x64") version of the redistributable package, which is also available from Microsoft.

Please note: For older versions, use the 2008 redistributable package.

2) Installing the Plug-in with the Installer

We have introduced installers to make installation of our plug-ins even easier. These installers will not only install the FileMaker plug-in file but will also install any third party software needed for the plug-in to function, the demo file, and additional resources you may need. We recommend using the installers to ensure that all components necessary for the plug-in to function are properly installed.

Contents of the FM Books Connector installation zip package:



Name	Date modified	Type	Size
FM Books Connector Extras	9/9/2016 11:34 AM	File folder	
FMBooksInstaller	9/7/2016 5:57 PM	Windows Installer ...	45,825 KB
setup	9/7/2016 5:57 PM	Application	456 KB

Once you download the FM Books Connector installation zip package, simply extract the package and open the resulting folder. Install the FM Books Connector with the following steps:

1) Run the "setup.exe" file
2) If prompted, install the Visual C++ 2008 or 2013 Runtime Libraries
3) If you are currently running FileMaker, please close FileMaker so that the plug-in will be installed correctly.
4) Accept the End User License Agreement ("EULA")
5) Select the location to install the plug-in*
6) Confirm the installation
7) If prompted by Windows user account control, allow the Installer to run
8) Your installation is complete!

*In order for FileMaker to properly recognize the plug-in, we suggest you do not change this default location. FileMaker plug-ins need to be installed in Extension folders recognized by the plug-in. By default, the plug-in will be installed to the base FileMaker/Extensions folder and will be available across multiple versions of FileMaker. However, if you wish to install the plug-in at a version-specific location like "FileMaker Pro Advanced/14.0/Extensions", you may browse to the folder location to do so.

If you install the FM Books Connector using the FM Books Connector installer package, the FM Books QB Bridge supplemental file will also be installed automatically, so you will not need to run that file, as well.

3) Installing the Plug-in Manually

You may seek to install the plug-in manually instead of using the FM Books Connector installer package. To do so, follow the steps below:

1) Install the FM Books QB Bridge
2) Unzip the "Install FM Books QB Bridge" zip package
3) In the resulting folder, run the "setup.exe" file and follow all prompts
4) Close the FileMaker application
5) Run the MSXML 6 Processor Update installer (downloadable from the web link file to Microsoft's download site, included in the main FM Books Connector download package)
6) Install the plug-in using either the FileMaker Pro demo file (if using FileMaker 12 or newer), or manually copying the appropriate FM Books Connector plug-in file to the FileMaker Extensions folder. This is normally found at:
    a. C:\Program Files\FileMaker\FileMaker Pro [Advanced] ##\Extensions
    b. C:\Program Files (x86)\FileMaker\FileMaker Pro [Advanced] ##\Extensions
7) The first path is for 64-bit FileMaker, the second path is for 32-bit FileMaker.
8) Start FileMaker. Confirm that the plug-in is successfully installed by navigating to Preferences in FileMaker, and select the "Plug-ins" tab; the plug-in should be visible in the list and checked with a corresponding checkbox.

4) Troubleshooting Plug-in Installation

When installing the plug-in using the "Install Plug-in" script step, there are certain situations that may cause a 1550 or 1551 error to arise. If such a situation occurs, please refer to the troubleshooting steps involving the most common problems that may cause those errors.

1) Invalid Bitness of FileMaker
    a. In some cases, FileMaker Pro may be attempting to install a plug-in with a different bitness than the FileMaker Pro application. This is most common with Windows plug-ins. The general rule is that the plug-in and FileMaker Pro must be the same bitness.

    b. To resolve this, ensure that the container field holding the plug-in contains the correct bitness of the plug-in. You can verify the plug-in's bitness by checking the file extension: if the extension is .fmx, the plug-in is a 32-bit plug-in; if the extension is .fmx64, the plug-in is a 64-bit plug-in. You can verify the bitness of FileMaker Pro itself by viewing the "About FileMaker Pro" menu option in the Help menu, and clicking the "Info" button to see more information; bitness is found under "Architecture".

2) Missing Dependencies
    a. Every plug-in has dependencies, which are system files present in the machine's operating system that the plug-in requires in order to function. If a plug-in is "installed" into an Extensions folder, but the plug-in does not load or is not visible in the Preferences > Plug-ins panel in FileMaker Pro's preferences, it's likely that there are files missing.

    b. To ensure that the appropriate dependencies are installed, please verify that the Visual Studio 2013 C++ Redistributable Package is installed. This can be located by opening Control Panel and checking the Installed Programs list (usually found under "Add/Remove Programs"). Older plug-ins may require the Visual C++ 2008 redistributable package, instead of the 2013 version.

    c. Some plug-ins also have a .NET Framework component that is also required. All such plug-ins of ours will require the .NET Framework 3.5, which can be downloaded from the following link: https://www.microsoft.com/en-us/download/details.aspx?id=21

3) Duplicate Plug-in Files
    a. When installing plug-ins, it is possible to have the plug-in located in different folders that are considered "valid" when FileMaker Pro attempts to load plug-ins for use. There is a possibility that having multiple versions of the same plug-in in place in these folders could cause FileMaker Pro to fail to load a newly-installed plug-in during the installation process.

    b. To resolve this, navigate to the different folders listed in the earlier installation steps and ensure that the plug-in is not present there by deleting the plug-in file(s). Once complete, restart FileMaker and attempt the installation again. If you installed the plug-in using a plug-in installer file, if on Windows, run the installer again and choose the "Uninstall" option, or if on Mac, run the "uninstall.tool" file to uninstall the plug-in.

If the three troubleshooting steps above do not resolve the issue, please feel free to reach out to our support team for further assistance.

5) Clearing out Existing Certificates to Upgrade Plug-in

In order up upgrade to a newer version of the FM Books Connector, you need to ensure you have cleared the certificate from the previous version of the plug-in.

Note: Be sure to quit/exit the FileMaker application before removing the plug-in file

Depending on the version of FileMaker, to uninstall the plug-in, you will need to close FileMaker and remove the file "PCFMBooksConnector.fmx" from the following locations below:

"C:\Program Files (x86)\FileMaker\FileMaker Pro Advanced\Extensions"
"C:\Program Files\FileMaker\FileMaker Pro Advanced\Extensions"
"C:\Users\<Current User>\AppData\Local\FileMaker\Extensions"
"C:\Users\<Current User>\AppData\Local\FileMaker\FileMaker Pro Advanced\"version"\Extensions"
"C:\Users\<Current User>\AppData\Local\FileMaker\FileMaker Pro\"version"\Extensions"

6) Registering the Plug-in

The next step is to register the plug-in which enables all plug-in functions.

1) Confirm that you have access to the internet and open our FileMaker demo file, which can be found the in "FileMaker Demo File" folder in your original download. If you are using the Australian version of QuickBooks, we recommend you utilize the US version of the demo files for testing and evaluation.

2) If you are registering the plug-in in Demo mode, then simply click the "Register" button and do not change any of the fields. Your plug-in should now be running in "DEMO" mode. The mode is always noted on the Setup tab of the FileMaker demo.

3) If you are registering a licensed copy, then simply enter your license number in the "LicenseID" field and select the "Register" button. Ensure you have removed the Demo License ID and enter your registration information exactly as it appears in your confirmation email. Your plug-in should now be running in "LIVE" mode. The mode is always noted on the Setup tab of the FileMaker demo, or by calling the PCQB_GetOperatingMode function.

Congratulations! You have now successfully installed and registered the plug-in!

Why do I need to register?

In an effort to reduce software piracy, Productive Computing, Inc. has implemented a registration process for all plug-ins. The registration process sends information over the internet to a server managed by Productive Computing, Inc. The server uses this information to confirm that there is a valid license available and identifies the machine. If there is a license available, then the plug-in receives an acknowledgment from the server and installs a certificate on the machine. This certificate never expires. If the certificate is ever moved, modified or deleted, then the client will be required to register again. On Windows, this certificate is in the form of a ".pci" file.

The registration process also offers developers the ability to automatically register each client machine behind the scenes by hard coding the license ID in the PCQB_Register function. This proves beneficial by eliminating the need to manually enter the registration number on each client machine. There are other various functions available such as PCQB_GetOperatingMode and PCQB_Version which can assist you when developing an installation and registration process in your FileMaker solution.

How do I hard code the registration process?

You can hard code the registration process inside a simple “Plug-in Checker” script. The “Plug-in Checker” script should be called at the beginning of any script using a plug-in function and uses the PCQB_Register, PCQB_GetOperatingMode and PCQB_Version functions. This eliminates the need to manually register each machine and ensures that the plug-in is installed and properly registered.

Below are the basic steps to create a “Plug-in Checker” script.

```
If [ PCQB_Version( "short" ) = "" or PCQB_Version( "short" ) = "?" ]
       Show Custom Dialog [ Title: "Warning"; Message: "Plug-in not installed."; Buttons:
“OK” ]
Else If [ PCQB_GetOperatingMode  ≠ “LIVE” ]
       Set Field [Main::gRegResult; PCQB_Register( “licensing.productivecomputing.com” ;
“80” ; “/PCIReg/pcireg.php” ; “your license ID” )
Else If [ Main::gRegResult   ≠  0 ]
       Show Custom Dialog [ Title: "Registration Error"; Message: “Plug-in Registration
Failed”; Buttons: “OK” ]
End If
```

7) FileMaker 16 Plug-in Script Steps

As of FileMaker Pro 16, all plug-ins have been updated to allow a developer to specify plug-in functions as script steps instead of as calculation results. The plug-in script steps function identically to calling a plug-in within a calculation dialog.

For an example of using plug-in script steps, compare two versions of the same script from the FM Books Connector demo file: Pull Customer__Existing Session.

Script 1 - Pull Customer__Existing Session with calculation ("traditional") plug-in scripting:

```
Set Error Capture [On]
Allow User Abort [Off]
# It is assumed the session is already opened from the previous script calling this
script.
# Query customers in QB (Request)

Set Variable [$$Result; Value: PCQB_RqNew( "CustomerQuery" ; "" )]
Set Variable [$$Result; Value: PCQB_RqAddFieldWithValue( "ListID" ;
Main::gCust_ListID )]
If [0 <> PCQB_RqExecute]
       Exit Script [Text Result:PCQB_SGetStatus]
End If

# Pull customer info into FileMaker (Response)
Set Variable [$$Result; Value: PCQB_RsOpenFirstRecord]
Set Field [main_CUST__Customers::ListID; PCQB_RsGetFirstFieldValue( "ListID" )]
Set Field [main_CUST__Customers::FullName; PCQB_RsGetFirstFieldValue( "FullName" )]
Set Field [main_CUST__Customers::First Name; PCQB_RsGetFirstFieldValue( "FirstName"
)]
Set Field [main_CUST__Customers::Last Name; PCQB_RsGetFirstFieldValue( "LastName" )]
Set Field [main_CUST__Customers::Company; PCQB_RsGetFirstFieldValue( "CompanyName" )]
Set Field [main_CUST__Customers::Bill_Address 1;
PCQB_RsGetFirstFieldValue( "BillAddress::Addr1" )]
Set Field [main_CUST__Customers::Bill_Address 2;
PCQB_RsGetFirstFieldValue( "BillAddress::Addr2" )]
Set Field [main_CUST__Customers::Bill_Address 3;
PCQB_RsGetFirstFieldValue( "BillAddress::Addr3" )]
Set Field [main_CUST__Customers::Bill_Address 4;
PCQB_RsGetFirstFieldValue( "BillAddress::Addr4" )]
Set Field [main_CUST__Customers::Bill_City;
PCQB_RsGetFirstFieldValue( "BillAddress::City" )]
Set Field [main_CUST__Customers::Bill_State;
PCQB_RsGetFirstFieldValue( "BillAddress::State" )]
Set Field [main_CUST__Customers::Bill_Postal Code;
PCQB_RsGetFirstFieldValue( "BillAddress::PostalCode" )]
Set Field [main_CUST__Customers::Phone; PCQB_RsGetFirstFieldValue( "Phone" )]
Set Field [main_CUST__Customers::Email; PCQB_RsGetFirstFieldValue( "Email" )]

Exit Script [Text Result:0]
```

Script 2 – Pull Customer   Existing Session with plug-in script steps:

```
Set Error Capture [On]
Allow User Abort [Off]
# It is assumed the session is already opened from the previous script calling this
script.
# Query customers in QB (Request)

PCQB_RqNew [Select; Results:$$Result; Request Type:"CustomerQuery"]
PCQB_RqAddFieldWithValue [Select; Results:$$Result; QB Field Name:"ListID"; Field
Value:Main::gCust_ListID]
PCQB_RqExecute [Select; Results:$$Result]
If [$$Result <> 0]
       Exit Script [Text Result:PCQB_SGetStatus]
End If

# Pull customer info into FileMaker (Response)
PCQB_RsOpenFirstRecord [Select; Results:$$Result]
PCQB_RsGetFirstFieldValue [Select; Results:main_CUST__Customers::ListID; Field
Name:"ListID"]
PCQB_RsGetFirstFieldValue [Select; Results:main_CUST__Customers::FullName;
FieldName:"FullName"]
PCQB_RsGetFirstFieldValue [Select; Results:main_CUST__Customers::First Name;
Field Name:" FirstName"]
PCQB_RsGetFirstFieldValue [Select; Results:main_CUST__Customers::Last Name;
Field Name:" LastName"]
PCQB_RsGetFirstFieldValue [Select; Results:main_CUST__Customers::Company;
Field Name:" CompanyName"]
PCQB_RsGetFirstFieldValue [Select; Results:main_CUST__Customers::Bill_Address 1;
Field Name:" BillAddress::Addr1"]
PCQB_RsGetFirstFieldValue [Select; Results:main_CUST__Customers::Bill_Address 2;
Field Name:" BillAddress::Addr2"]
PCQB_RsGetFirstFieldValue [Select; Results:main_CUST__Customers::Bill_Address 3;
Field Name:" BillAddress::Addr3"]
PCQB_RsGetFirstFieldValue [Select; Results:main_CUST__Customers::Bill_Address 4;
Field Name:" BillAddress::Addr4"]
PCQB_RsGetFirstFieldValue [Select; Results:main_CUST__Customers::Bill_City;
Field Name:" BillAddress::City"]
PCQB_RsGetFirstFieldValue [Select; Results:main_CUST__Customers::Bill_State;
Field Name:" BillAddress::State"]
PCQB_RsGetFirstFieldValue [Select; Results:main_CUST__Customers::Bill_Postal Code;
Field Name:" BillAddress::PostalCode"]
PCQB_RsGetFirstFieldValue [Select; Results:main_CUST__Customers::Phone; Field
Name:"Phone"]
PCQB_RsGetFirstFieldValue [Select; Results:main_CUST__Customers::Email; Field
Name:"Email"]

Exit Script [Text Result:0]
```

Using script steps instead of the more traditional methods can make scripting within a solution more direct, as well as help with data entry validation. Some functions accept calculation-style input, while others accept a Boolean "true" or "false" option, and others employ a drop-down list for the developer to choose an option from. As stated earlier, the functionality of the plug-in script step is identical to its functionality as a calculation function; PCQB_RsOpenFirstRecord as a script step will still open the first record in the response, and store the value in the $$Result global variable (as seen in Script 2), just the same as the Set Variable script step calls PCQB_RsOpenFirstRecord (which opens the first response record) and stores the result in the $$Result variable.

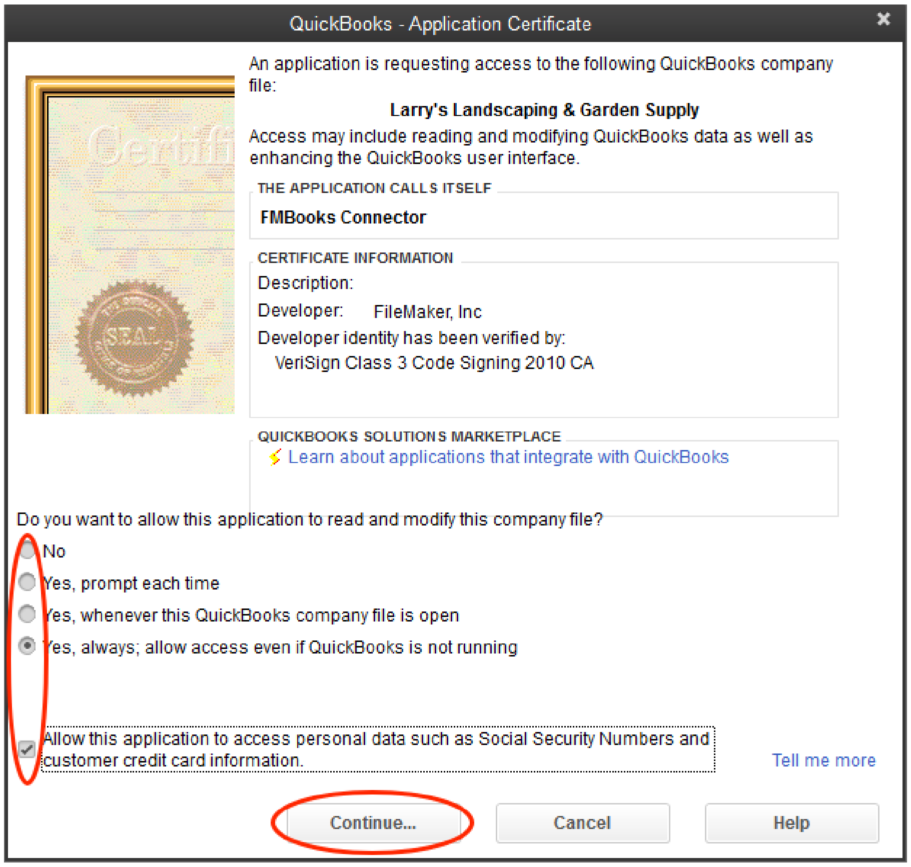
For all Productive Computing, Inc., plug-ins that provide plug-in script step functionality, calculation functions will still be provided for use in development. This is to ensure that scripts already integrated with any of our plug-ins will still be viable and functional, and the developer now has the option to utilize the plug-in script steps at their discretion.

8) Establish Initial QuickBooks Connection

Establish Initial Connection with QuickBooks Company File

First log into that QuickBooks file in Single User mode with "Admin" access. When making a call to QuickBooks via the FM Books Connector plug-in, you will see a screen similar to the one below. Once this screen appears, select the appropriate radio button to continue and allow communication with QuickBooks. We recommend selecting the 4th radio button to always allow access as shown below. These settings can be changed later in QuickBooks under the Edit > Preferences > Integrated Applications.

Figure 1.0 - Sample Screen Shot of QuickBooks Certificate


QuickBooks - Application Certificate

An application is requesting access to the following QuickBooks company file:

Larry's Landscaping & Garden Supply

Access may include reading and modifying QuickBooks data as well as enhancing the QuickBooks user interface.

THE APPLICATION CALLS ITSELF

FMBooks Connector

CERTIFICATE INFORMATION

Description:

Developer: FileMaker, Inc

Developer identity has been verified by:

VeriSign Class 3 Code Signing 2010 CA

QUICKBOOKS SOLUTIONS MARKETPLACE

Learn about applications that integrate with QuickBooks

Do you want to allow this application to read and modify this company file?

- No
- Yes, prompt each time
- Yes, whenever this QuickBooks company file is open
- [✓ Yes, always; allow access even if QuickBooks is not running]

[✓] Allow this application to access personal data such as Social Security Numbers and customer credit card information.

Connect to QuickBooks in Unattended Access Mode

QuickBooks allows a third-party application to connect to a company file in "Unattended Access Mode", which is where the QuickBooks application remains closed, but the third-party application can still push and pull data with the company file. When establishing the initial connection, the option "Yes, always; allow access even if QuickBooks is not running" in the Application Certificate is the option that controls whether unattended access mode is possible.

To use unattended access mode, the developer must provide the file path to the company file to the PCQB_BeginSession function. The function call would look like:

```
PCQB_BeginSession( "C:\Path\To\File.qbw" ; "" )
```

The first parameter will be the Windows-formatted path to the company file. The second parameter can be left blank to use the default "Do Not Care" connection type (between "Single"-user mode, "Multi"-user mode or "Do Not Care").

It is important to note that the company file path **must** be the same as the path that was used when opening the company file in QuickBooks. If the company file was opened in QuickBooks using a standard mapped drive (i.e. C:\folder\file.qbw), then the file path given to PCQB_BeginSession must also be a standard mapped drive. Consequently, if the company file was opened in QuickBooks using a network drive path (i.e. \\machine\folder\file.qbw), then the file path given to PCQB_BeginSession must also be a network drive path. If the path given to the plug-in is not the same as the original path, there will be errors and complications when attempting a connection.

9) Talking to QuickBooks

It is easiest to picture the information exchanged between the plug-in and QuickBooks as a very short conversation. Actually, this conversation is made simply of a request and a response to that request. That's it - a very short and simple conversation. See Figure 2.0 below.

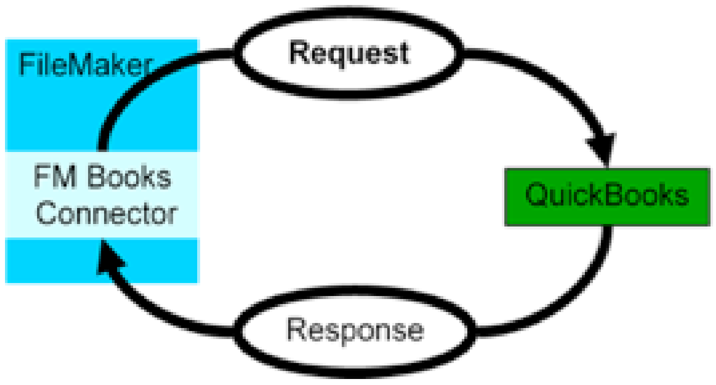


Figure 2.0 – FileMaker and QuickBooks Information Exchange
No matter what the request is, whether it is to add contacts to a QuickBooks company file or to find all unpaid invoices, the conversation always takes the form of a request to do something and a response suitable for the type of request made.

The following rules also apply:

1) The plug-in always initiates any conversation with a request.

2) QuickBooks always responds to a properly posed request.

3) The conversation is always finished after QuickBooks responds to the request. The request and response conversation is the foundation for exchanging information between FileMaker and QuickBooks. It is imperative to remember that this is the form of the conversation no matter what the plug-in is requesting of QuickBooks. It will be especially useful to remember this when it comes time to create scripts in your FileMaker solution for exchanging data with QuickBooks.

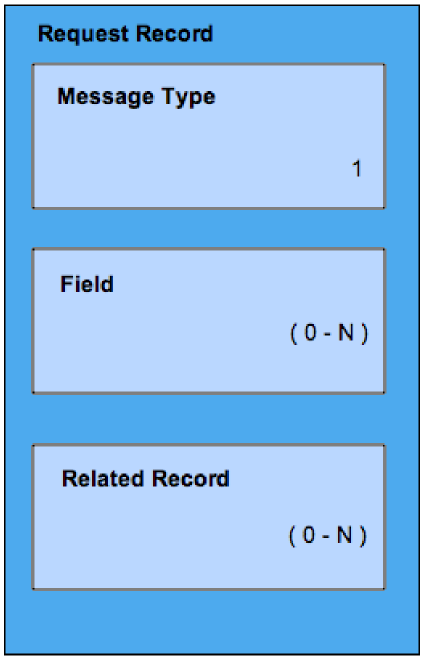
4) We now know how the plug-in talks to QuickBooks. We know that the conversation is short and to the point. What we have yet to learn is what is "said" during this conversation and how it is "spoken." We could go into all the details of the language that is used in this conversation, but we created this plug-in specifically so that the user does not need to know these details. With the plug-in all you need know is how to create a request, how to post the request to QuickBooks, and how to read QuickBooks' response.

Request and Response Explained

Requests are created and responses are read using external functions. Different requests and responses have different fields defined for them. Furthermore, requests and responses may have related items.
Please refer to Figure 2.1 for an explanation of the three different elements for both a request and response.

Figure 2.1 - Request and Response Models

Image 2: *Request* Model



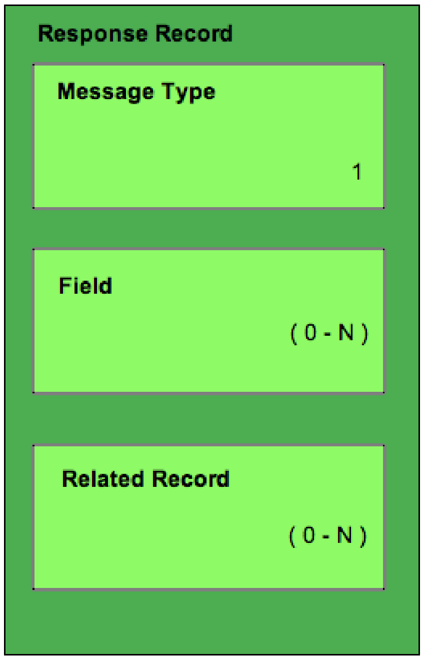
Each *Request* contains three elements: a single **Message Type**, zero or more **Fields**, and zero or more **Related Records**.

Message Type - Identifies what action is requested of QB and which class of QB records are to be affected. The element is set with the **PCQB_RqNew(...)** function.

Field - The fields of the record contain information about the request. The field types and the number of fields are determined by the request **Type**. These elements are set with the **PCQB_RqAddFieldWithValue(...)** function.

Related Record - Some request Types may allow for Related Records to be included in the request. Like the request record, each Related Record contains a Type and a set of Fields. Some related records may also contain their own Related Record. Each Related Record element is created with the **PCQB_RqAddRelatedRecord (...)** function.

Image 3: *Response* Model



Each *Response Record* contains three elements: a single **Message Type**, zero or more **Fields**, and zero or more **Related Records**.

Message Type - Identifies the Type for the Response Record. Each Record in the Response has the same Type. The Type for the records in the response are dictated by the Message Type specified by the preceding Request.

Field - The fields of the record contain information pertinent to the request. The number of fields and the type of fields are dictated by the correlating Request. Also, the number and type can be set by certain Request fields.

Related Record - Some response Types may allow for Related Records to be included for each Response Record. Each Related Record in turn contains a Type and a set of Fields. Some related records may also contain their own Related Records.

* Please come back and reference Figure 2.1 after you have reviewed this document. *

Create a Request

Each request has a certain predefined Message Type. A Message Type for a request has the distinct honor of telling QuickBooks what the request is aiming to do and which records are to be affected. In terms that are more technical, the Message Type defines which class of records we want to work with and which action we want applied to those records. For instance, the 'CustomerAdd' Message Type tells QuickBooks that the request wants to create a new Customer record in the QuickBooks file. The available Message Types for a request are listed in the OSR (Onscreen Reference Manual) and are left out of this document for the sake of brevity.

Some sample request Message Types and their definitions follow:

CustomerMod the request wants to modify an existing Customer record in QuickBooks.
InvoiceQuery the request wants to find Invoice records in QuickBooks.
SalesOrderAdd the request wants to add a Sales Order record to QuickBooks.

A request also contains fields that further define what the request is to do and which records are to be affected. The field names available for the different request Message Types are also listed in the OSR.

Two functions are used to create a basic request:

```
PCQB_RqNew( MessageType )
PCQB_RqAddFieldWithValue( Fieldname ; Value )
```

The first function is used to create a request of the desired Message Type, and the second is used to populate the fields of the request.

A simple request to add a Customer follows:

```
PCQB_RqNew( "CustomerAdd" )
PCQB_RqAddFieldWithValue( "Name" ; "Bob Jones" )
PCQB_RqAddFieldWithValue( "BillAddress::Addr1" ; "123 Any Street" )
PCQB_RqAddFieldWithValue( "BillAddress::City" ; "Any Town" )
PCQB_RqAddFieldWithValue( "BillAddress::State" ; "Any State" )
PCQB_RqAddFieldWithValue( "BillAddress::PostalCode " ; "11111" )
PCQB_RqAddFieldWithValue( "Email" ; "bill@someisp.com" )
```

For some request Message Types, the developer will want to include related records in the request. For example, when adding an invoice, the developer would like to add the line items also. To accomplish this, two other functions are also used. They are:

```
PCQB_RqAddRelatedRecord( ElementName )
PCQB_RqCloseRelatedRecord
```

A sample InvoiceAdd request demonstrates the usage of these two functions.

```
PCQB_RqNew( "InvoiceAdd" )
PCQB_RqAddFieldWithValue( "CustomerRef::FullName" ; "Bob Jones" )
PCQB_RqAddFieldWithValue( "TxnDate" ; "2006/01/01" )
PCQB_RqAddFieldWithValue( "RefNumber" ; "123456789" )
PCQB_RqAddFieldWithValue( "BillAddress::Addr1" ; "123 Any Street" )
PCQB_RqAddFieldWithValue( "BillAddress::City" ; "Any Town" )
PCQB_RqAddFieldWithValue( "BillAddress::State" ; "Any State" )
PCQB_RqAddFieldWithValue( "BillAddress::PostalCode " ; "11111" )
PCQB_RqAddRelatedRecord( "InvoiceLineAdd" )
PCQB_RqAddFieldWithValue( "ItemRef::FullName" ; "Widgit" )
PCQB_RqAddFieldWithValue( "Quantity" ; "20" )
PCQB_RqAddFieldWithValue( "Amount" ; "79.95" )
PCQB_RqCloseRelatedRecord
PCQB_RqAddRelatedRecord( "InvoiceLineAdd" )
PCQB_RqAddFieldWithValue( "ItemRef::FullName" ; "Gadget" )
PCQB_RqAddFieldWithValue( "Quantity" ; "3" )
PCQB_RqAddFieldWithValue( "Amount" ; "2.95" )
PCQB_RqCloseRelatedRecord
```

You will notice that the PCQB_RqAddFieldWithValue function operates in the context of the current record. When the PCQB_RqAddRelatedRecord function is called the context shifts from the parent record (the main request) to the related record, in this case the InvoiceLineAdd record. Subsequent calls to PCQB_RqAddFieldWithValue will add field values to the related record until the PCQB_RqCloseRelatedRecord function is called and the context is shifted back to the parent record. Another important note to make is that the fields of a request must be set in a specific order. The order to set the fields is listed in the OSR. It is also important to note that the PCQB_RqAddRelatedRecord must also be called in a specific order. Again, this order is listed in the OSR. Later in this document we will discuss how to use the OSR.

Post the Request

Now that we know how to create a request, we need to learn how to post the request to QuickBooks. This is a rather simple operation that requires only a single execute function. However, this is the perfect time to explain two other functions that the execute function relies upon. The three functions are:

```
PCQB_BeginSession( CompanyFile ; ShareMode ; optHideFMWindow )
PCQB_RqExecute
PCQB_EndSession
```

PCQB_BeginSession is used to establish a session with QuickBooks. This function must be returned successfully before we can post a request to QuickBooks. It may be called anytime during the script, but since it may block other applications from accessing the QuickBooks file it is proper etiquette to call it immediately before executing a request and then calling PCQB_EndSession immediately after executing. This minimizes the possibility of blocking other applications from accessing the file while you build script requests or processes responses.

Requests posted during PCQB_RqExecute will first be validated by the plug-in before being sent to QuickBooks.

The following table lists the effects of the mode in different operating environments:

Who started QuickBooks	**Selected Mode**	**Who else may obtain access**
FM Books Connector	“Single”	No one else
FM Books Connector	“Multi”	QuickBooks user on same machine = no access All other integrated applications = access QuickBooks users on other machines = access
QuickBooks User	“Single”	QuickBooks on same machine = access Only one integrated application = access
QuickBooks User	“Multi”	QuickBooks users = access Integrated applications = access

Parse the Response

Once a request is built and successfully executed, the plug-in will retain the response in memory. Depending on the type of request that is made the response may contain one or several records. For instance, after executing a request to add a customer (using a request of Message Type 'CustomerAdd') QuickBooks will respond with a 'CustomerRet' record that the plug-in will hold in memory. Query requests such as an ‘InvoiceQuery’ request may return several records in the response. Each of the records is accessed and parsed for the information they contain.

Accessing and parsing the records contained in the response is a rather simple process.

Six functions are used to read the contents of a response:

```
PCQB_RsOpenFirstRecord
PCQB_RsOpenNextRecord
PCQB_RsOpenFirstRelatedRecord( ElementName )
PCQB_RsOpenNextRelatedRecord
PCQB_RsCloseRelatedRecord
PCQB_RsGetFirstFieldValue( QBFieldName )
```

The first two functions are used to iterate through all the records in the response. PCQB_RsOpenFirstRecord opens the first record in the response and PCQB_RsOpenNextRecord is used to open successive records in the response. PCQB_RsOpenNextRecord will return "End" when there are no more records to be read in the response.

Once a record is opened with either of the open record functions the user is able to read the record contents using the PCQB_RsGetFirstFieldValue(QBFieldName) function. The name of the desired field is passed with the function and the contents of the field are returned. See the “OSR” for acceptable field names.

As with a request there may be related records to each record in the response. These related records are accessed with the PCQB_RsOpenFirstRelatedRecord(ElementName) and PCQB_RsOpenNextRelatedRecord functions. Once the related record is opened, its contents can be read with the PCQB_RsGetFirstFieldValue function.

An example of reading a response to an 'InvoiceQuery' request follows:

```
# Open the First record in the response
Set Field[ someField ; PCQB_RsOpenFirstRecord ]
Loop
  # Exit the loop on error or after last record is read
  Exit Loop If [ (someField < 0) or (someField = "End") ]
  # Get the name of the Customer and the Accounts Receivable
  Set Field [someField ; PCQB_RsGetFirstFieldValue( "CustomerRef::FullName" ) ]
  Set Field [someField ; PCQB_RsGetFirstFieldValue( "ARAccountRef::FullName" ) ]
  # Get any and all related transactions to the current invoice
  Set Field [someField ; PCQB_RsOpenFirstRelatedRecord( "LinkedTxn" ) ]
  Loop
    Exit Loop If [ (someField < 0) or (someField = "End") ]
    # Gets the information from the related transaction
    Set Field [someField ; PCQB_RsGetFirstFieldValue( "RefNumber" ) ]
    Set Field [someField ; PCQB_RsGetFirstFieldValue( "Amount" ) ]
    # Opens the next related transaction
    Set Field [someField ; PCQB_RsOpenNextRelatedRecord ]
  End Loop
  # Close the related record to return to the main record
  Set Field [someField ; PCQB_RsCloseRelatedRecord ]
  # Opens the invoice line item related records
  Set Field [someField ; PCQB_RsOpenFirstRelatedRecord( "InvoiceLineRet" )
  Loop
    Exit Loop If [ (someField < 0) or (someField = "End") ]
    # Gets the information from the line item
    Set Field [someField ; PCQB_RsGetFirstFieldValue( "ItemRef::FullName" ) ]
    Set Field [someField ; PCQB_RsGetFirstFieldValue( "Amount" ) ]
    # Opens the next line item
    Set Field [someField ; PCQB_RsOpenNextRelatedRecord ]
  End Loop
  # Close the related record to return to the main record
  Set Field [someField ; PCQB_RsCloseRelatedRecord ]
  # Open the next record in the response
  Set Field [someField ; PCQB_RsOpenNextRecord ]
End Loop
```

Using XML Attributes

Some objects in QuickBooks have attributes, whether for requests or for responses, that change the behavior of how the QBXML document is processed by QuickBooks. You may have noticed that there are a number of functions in the FM Books Connector plug-in that make reference to parameters called "optAttribute" or "optAttributes".

Here's a quick list of those functions:

```
PCQB_RqNew( Type ; optAttributes )
PCQB_RqAddFieldWithValue( QBFieldName ; Value ; optAttributes )
PCQB_RqAddRelatedRecord( Type ; optValue ; optAttributes )
PCQB_RsGetAttribute( AttributeName )
PCQB_RsGetResponseAttribute( AttributeName )
```

Let's take, for example, the use of attributes in PCQB_RqNew, specifically with the "Invoice" object. When performing a query, we can simply create a new query with PCQB_RqNew("InvoiceQuery"). However, if we wish to perform a query with a limited return set and need to make several successive calls to "page" over the return data, we can use the optAttributes parameter to provide an iterator and, later, an iterator ID; this is called "pagination". A simple pseudoscript can be written like this:

```
PCQB_RqNew( "InvoiceQuery" ; "iterator=start" )
PCQB_RqAddFieldWithValue( "MaxResults" ; 10 )
PCQB_RqExecute

# Store the iterator ID
Set Variable ( $iteratorID ; PCQB_RsGetResponseAttribute( "iteratorID" ) )

# Process first 10 results…


Loop
      # The iteratorRemainingCount will decrease by 1 per request until all records
      are returned.
      Set Variable ( $remainingPages ; PCQB_RsGetResponseAttribute(
      "iteratorRemainingCount" ) )
      PCQB_RqNew( "InvoiceQuery" ; "iterator=continue;iteratorID=" & $iteratorID )
      PCQB_RqAddFieldWithValue( "MaxResults" ; 10 )
      PCQB_RqExecute

      # Process next 10 results; repeat until $remainingPages = 0
End Loop
```

When paginating over a number of records, the first query request should always provide an iterator value of "start". This will inform QuickBooks that FileMaker is seeking to make several calls for information, and will request that information in piecemeal style. In the above example, we're only processing 10 records at a time in our pagination. When retrieving the next page, the script must execute requests with the exact same query criteria (in this case, MaxResults of 10), with an iterator attribute value of "continue" and an additional attribute of "iteratorID" with the iteratorID returned from the first request. The response attribute of "iteratorRemainingCount" provides a running count of how many iterations are left until the query is considered fully executed, and can be used as a sentinel value for FileMaker loop scripting.

Paginating is useful for when the developer knows that there is a possibility of a large dataset in QuickBooks that will be processed when performing a request. Pagination can be used in conjunction with other more conventional dataset-restricting search criteria, and has been shown to improve performance and stability when processing large volumes of data.

Another example of attribute use is found in the OSR reference for the “defMacro” and “useMacro” attribute values for certain objects. For our examples, again we will be using the Invoice object.

When creating a new request to add an invoice, the developer may seek to add the attribute “defMacro”, and providing a unique custom identifier that will reference that transaction for any later use of “useMacro”. This transaction macro can be used in substitution of a TxnID returned by QuickBooks; you can, for example, use the FileMaker internal ID of a record as the “defMacro” value instead of the TxnID.

See the sample pseudoscript below:

```
PCQB_RqNew( “InvoiceAdd” ; “defMacro=Invoice1234” )
PCQB_RqAddFieldWithValue( “CustomerRef::ListID” ; $$CustomerListID )
...
# Populate additional Invoice information...
...

# Later, we receive a payment for the invoice created above, so we reference it
# with “useMacro”
PCQB_RqNew( “ReceivePaymentAdd” )
PCQB_RqAddFieldWithValue( “CustomerRef::ListID” ; $$CustomerListID )
...
PCQB_RqAddRelatedRecord( “AppliedToTxnAdd” )
PCQB_RqAddFieldWithValue( “TxnID” ; “” ; “useMacro=Invoice1234” )
...
# Additional AppliedToTxnAdd information here..
...
PCQB_RqCloseRelatedRecord
```

A final example of attribute use can be seen in the sample script “Processing a Report Query” below. With the use of the PCQB_RsGetAttribute and PCQB_RsGetResponseAttribute functions, a developer can implement a script to pull in a report from QuickBooks, parse the rows and columns of data from the report, and display them in FileMaker in an organized, customized format. Please refer to the Functions Guide for further information about the PCQB_RsGetAttribute and PCQB_RsGetResponseAttribute functions.

Sample Script: Processing a Report Query

In some cases, a customer may require pulling in a report from QuickBooks into FileMaker. In general, the recommendation is to allow QuickBooks to handle the process of generating the report, and the plug-in will, in fact, provide QuickBooks with a requested report type and a set of criteria to use for the report, allowing QuickBooks’ reporting engine to generate the information and return an XML document containing complete, calculated details and values as QuickBooks business rules dictate. The act of parsing this report, however, is quite challenging, especially when reviewing the XML document itself. Below is a sample XML document containing an A/R Aging Summary report with a single customer entry. As a note, this information was generated using a sample QuickBooks file:

```
<?xml version="1.0"?>
<QBXML>
   <QBXMLMsgsRs>
      <AgingReportQueryRs statusCode="0" statusSeverity="Info" statusMessage="Status OK">
         <ReportRet>
            <ReportTitle>A/R Aging Summary</ReportTitle>
            <ReportSubtitle>As of December 15, 2019</ReportSubtitle>
            <ReportBasis>Accrual</ReportBasis>
            <NumRows>52</NumRows>
            <NumColumns>7</NumColumns>
            <NumColTitleRows>1</NumColTitleRows>
            <ColDesc colID="1" dataType="STRTYPE">
               <ColTitle titleRow="1"/>
               <ColType>Label</ColType>
            </ColDesc>
            <ColDesc colID="2" dataType="AMTTYPE">
               <ColTitle titleRow="1" value="Current"/>
               <ColType>Amount</ColType>
            </ColDesc>
            <ColDesc colID="3" dataType="AMTTYPE">
               <ColTitle titleRow="1" value="1 - 30"/>
               <ColType>Amount</ColType>
            </ColDesc>
            <ColDesc colID="4" dataType="AMTTYPE">
               <ColTitle titleRow="1" value="31 - 60"/>
               <ColType>Amount</ColType>
            </ColDesc>
            <ColDesc colID="5" dataType="AMTTYPE">
               <ColTitle titleRow="1" value="61 - 90"/>
               <ColType>Amount</ColType>
            </ColDesc>
            <ColDesc colID="6" dataType="AMTTYPE">
               <ColTitle titleRow="1" value="&gt; 90"/>
               <ColType>Amount</ColType>
            </ColDesc>
            <ColDesc colID="7" dataType="AMTTYPE">
               <ColTitle titleRow="1" value="TOTAL"/>
               <ColType>Total</ColType>
            </ColDesc>
            <ReportData>
               <TextRow rowNumber="1" value="Allard, Robert"/>
               <DataRow rowNumber="2">
                  <RowData rowType="name" value="Allard, Robert:Remodel"/>
                  <ColData colID="1" value="Remodel"/>
                  <ColData colID="2" value="14510.00"/>
                  <ColData colID="3" value="0.00"/>
                  <ColData colID="4" value="0.00"/>
                  <ColData colID="5" value="0.00"/>
                  <ColData colID="6" value="0.00"/>
                  <ColData colID="7" value="14510.00"/>
               </DataRow>
                <TotalRow rowNumber="3">
                   <ColData colID="1" value="TOTAL"/>
                   <ColData colID="2" value="14510.00"/>
                   <ColData colID="3" value="0.00"/>
                   <ColData colID="4" value="0.00"/>
                   <ColData colID="5" value="0.00"/>
                   <ColData colID="6" value="0.00"/>
                   <ColData colID="7" value="5100.00"/>
                </TotalRow>
            </ReportData>
         </ReportRet>
      </AgingReportQueryRs>
   </QBXMLMsgsRs>
</QBXML>
```

The initial fields of the report, more specifically the Report Title, Subtitle, Basis, and summary counts of columns and rows, are easy enough to parse; those can be accessed simply by opening the first record and using PCQB_RsGetFirstFieldValue function with the field names to retrieve their values.

The tricky part of the report comes in once the parsing script needs to access the column header data, and the report row data afterwards. For these, the FM Books Connector can make use of the PCQB_RsGetAttribute function combined with a creative application of the PCQB_RsOpenFirstRelatedRecord / PCQB_RsOpenNextRelatedRecord functions.

Below is a sample script that demonstrates the processing of the summary fields and the column header fields. For demonstration purposes, we assume there are two tables, "ReportHeader" and "ReportData", that are related by standard FileMaker internal keys, "ID" and "Header ID":

```
# Performing an Aging Report Query
Set Variable [$result; Value: PCQB_RqNew( "AgingReportQuery" )]

# Specifying an AR Aging Summary report, with some start and end timestamp values
Set Variable [$result; Value: PCQB_RqAddFieldWithValue( "AgingReportType" ; "ARAgingSummary")]
Set Variable [$result; Value: PCQB_RqAddFieldWithValue( "ReportPeriod::FromReportDate" ;
$$StartTimestamp )]
Set Variable [$result; Value: PCQB_RqAddFieldWithValue( "ReportPeriod::ToReportDate" ;
$$EndTimestamp )]

# Submit the report query
Set Variable [$result; Value: PCQB_RqExecute]
If [$result <> 0]
        # Error Capture...
End If

Set Variable [$result; Value: PCQB_RsOpenFirstRecord]
Go to Layout[ "ReportHeader" (ReportHeader); Animation: None]
New Record/Request

# Populate the primary report header fields
Set Field [ReportHeader::ReportTitle; PCQB_RsGetFirstFieldValue( "ReportTitle" )]
Set Field [ReportHeader::ReportSubtitle; PCQB_RsGetFirstFieldValue( "ReportSubtitle" )]
Set Field [ReportHeader::ReportBasis; PCQB_RsGetFirstFieldValue( "ReportBasis" )]
Set Field [ReportHeader::NumRows; PCQB_RsGetFirstFieldValue( "NumRows" )]
Set Field [ReportHeader::NumColumns; PCQB_RsGetFirstFieldValue( "NumColumns" )]
Set Field [ReportHeader::NumColTitles; PCQB_RsGetFirstFieldValue( "NumColTitles" )]

# Now we start parsing the title row, which stores the column header names and values
Set Variable [$result; Value: PCQB_RsOpenFirstRelatedRecord( "ColDesc" )
Set Variable [$iter; Value: 0]
Loop
        # We loop over every instance of "ColDesc", storing the column ID and value, if
        # applicable, in two repeating fields: ColType[] and ColTitle[].

        Exit Loop If [$iter > ARAgingHeader::NumColumns]

        Set Variable [$colType; Value:PCQB_RsGetAttribute( "dataType" )];
        If [$colType = "AMTTYPE"]
                Set Variable [$result; Value: PCQB_RsOpenFirstRelatedRecord( "ColTitle" )]
                Set Variable [$colTitle; Value: PCQB_RsGetAttribute( "value" )];
                Set Variable [$reslt; Value: PCQB_RsCloseRelatedRecord]
        End If

        # Store the results of $colType and (if applicable) $colTitle.
        Set Field [ReportHeader::ColType[$iter]; $colType]
        Set Field [ReportHeader::ColTitle[$iter]; If( not IsEmpty( $colTitle ) ; $colTitle ; ""
)]
End Loop

Set Variable [$result; Value: PCQB_RsCloseRelatedRecord]
```

```
# Now we have the header information stored. Moving on to process the contents of the report.
# Since we're working with related ReportData records, store the Header ID for the relationship.

Set Variable [$reportID; Value: ReportHeader::ID]
Set Variable [$numColumns; Value: ReportHeader::NumColumns]

# Test to see if any report data came down from the request
Set Variable [$result; PCQB_RsOpenFirstRelatedRecord( "ReportData" )]
If [$result <> 0]
        # Error: No report data came down
        Show Custom Dialog ["Error"; PCQB_SGetStatus]
        Exit Script []
Else
        # When processing the report data rows, there are many different kinds of rows:
        # TextRow - A label or other text-only information row, usually for parent Customers
        # DataRow - Actual report data, with values split across multiple columns
        # SubtotalRow - A subtotal row, with values split across multiple columns
        # TotalRow - The final row in a report, with values totaled from the data rows

        Go to layout ["Report Data" (ReportData); Animation:None]

        # We can parse the rows in any order, but let's start with TextRows.
        Set Variable [$result; Value: PCQB_RsOpenFirstRelatedRecord( "TextRow" )]
        If [$result = 0]
                Loop
                        Exit Loop If [$result = "END" or $result = "!!ERROR!!"]

                        New Record/Request
                        Set Field [ReportData::Header ID; $reportID]
                        Set Field [ReportData::RowType; "TextRow"]
                        Set Field [ReportData::RowNumber; PCQB_RsGetAttribute( "rowNumber" )]
                        Set Field [ReportData::RowData; PCQB_RsGetAttribute( "value" )]

                        Set Variable [$result; Value: PCQB_RsOpenNextRelatedRecord]
                End Loop
                Set Variable [$result; Value: PCQB_RsCloseRelatedRecord]
        End If

        # Next we'll parse all DataRows.
        Set Variable [$result; Value: PCQB_RsOpenFirstRelatedRecord( "DataRow" )]
        If [$result = 0]
                Loop
                        Exit Loop If [$result = "END" or $result = "!!ERROR!!"]

                        New Record/Request
                        Set Field [ReportData::Header ID; $reportID]
                        Set Field [ReportData::RowType; "DataRow"]
                        Set Field [ReportData::RowNumber; PCQB_RsGetAttribute( "rowNumber" )]

                        # DataRows store their values with one RowData and many ColData elements,
                        # so we must parse them individually. Start with the RowData element.
                        Set Variable [$result2; Value: PCQB_RsOpenFirstRelatedRecord( "RowData" )]
                        Set Field [ReportData::RowData; PCQB_RsGetAttribute( "value" )]
                        Set Variable [$result2; Value: PCQB_RsCloseRelatedRecord]

                        # Now pull in the ColData elements.
                        Set Variable [$result2; Value: PCQB_RsOpenFirstRelatedRecord( "ColData" )]
                        Set Variable [$iter; Value: 1]
                        Loop
                                Exit Loop If [$iter > $numColumns]

                                Set Field [ReportData::ColData[$iter]; PCQB_RsGetAttribute( "value"
                )]

                                Set Variable [$result2; Value: PCQB_RsOpenNextRelatedRecord]
                                Set Variable [$iter; Value: $iter + 1]
                        End Loop
                        Set Variable [$result2; Value: PCQB_RsCloseRelatedRecord]

                        # Proceed to the next DataRow
                        Set Variable [$result; Value: PCQB_RsOpenNextRelatedRecord]
                End Loop
                Set Variable [$result; Value: PCQB_RsCloseRelatedRecord]
        End If
        # Now we parse any SubtotalRows, if applicable.
        Set Variable [$result; Value: PCQB_RsOpenFirstRelatedRecord( "SubtotalRow" )]
        If [$result = 0]
                Loop
                        Exit Loop If [$result = "END" or $result = "!!ERROR!!"]

                        New Record/Request
                        Set Field [ReportData::Header ID; $reportID]
                        Set Field [ReportData::RowType; "SubtotalRow"]
                        Set Field [ReportData::RowNumber; PCQB_RsGetAttribute( "rowNumber" )]

                        # SubtotalRows store their values with one RowData and many ColData
                        # elements, so we must parse them individually. Start with the RowData
                        # element.
                        Set Variable [$result2; Value: PCQB_RsOpenFirstRelatedRecord( "RowData" )]
                        Set Field [ReportData::RowData; PCQB_RsGetAttribute( "value" )]
                        Set Variable [$result2; Value: PCQB_RsCloseRelatedRecord]

                        # Now pull in the ColData elements.
                        Set Variable [$result2; Value: PCQB_RsOpenFirstRelatedRecord( "ColData" )]
                        Set Variable [$iter; Value: 1]
                        Loop
                                Exit Loop If [$iter > $numColumns]

                                Set Field [ReportData::ColData[$iter]; PCQB_RsGetAttribute( "value"
                )]

                                Set Variable [$result2; Value: PCQB_RsOpenNextRelatedRecord]
                                Set Variable [$iter; Value: $iter + 1]
                        End Loop
                        Set Variable [$result2; Value: PCQB_RsCloseRelatedRecord]

                        # Proceed to the next DataRow
                        Set Variable [$result; Value: PCQB_RsOpenNextRelatedRecord]
                End Loop
                Set Variable [$result; Value: PCQB_RsCloseRelatedRecord]
        End If


        # Every report has a "TotalRow" which stores the totals of all report rows.
        Set Variable [$result; Value: PCQB_RsOpenFirstRelatedRecord( "TotalRow" )]
        If [$result = 0]
                New Record/Request
                Set Field [ReportData::Header ID; $reportID]
                Set Field [ReportData::RowType; "TotalRow"]
                Set Field [ReportData::RowNumber; PCQB_RsGetAttribute( "rowNumber" )]

                # Now pull in the ColData elements.
                Set Variable [$result2; Value: PCQB_RsOpenFirstRelatedRecord( "ColData" )]
                Set Variable [$iter; Value: 1]
                Loop
                        Exit Loop If [$iter > $numColumns]

                        Set Field [ReportData::ColData[$iter]; PCQB_RsGetAttribute( "value" )]

                        Set Variable [$result2; Value: PCQB_RsOpenNextRelatedRecord]
                        Set Variable [$iter; Value: $iter + 1]
                End Loop
                Set Variable [$result2; Value: PCQB_RsCloseRelatedRecord]
        End If

        # Now we close the ReportData record, as we're done processing everything.
        Set Variable [$result; Value: PCQB_RsCloseRelatedRecord]
End If

# From here, we would return to the main Report display layout, displaying the report with
# the ReportData records sorted by RowNumber, and custom filtering/formatting based on
# RowType, as needed by the business rule.
```

Custom Fields

Custom fields are accessed using DataExt objects. Each custom field is a DataExt object as the QBSDK uses a DataExt object for custom fields. If a QB object (invoice, customer, etc.) supports custom fields then the list of available fields in the response for the object will contain a DataExtRet object. Accessing the DataExtRet object in a response is the same as accessing any other related record of the response. Setting values for DataExt objects is accomplished with DataExtMod requests. One must be familiar with creating requests using the OSR to successfully modify DataExt objects.

This DataExt object is available in the following QB lists:

```
Accounts
Customers
Vendors
Items
OtherNames
Employees
```

The DataExt object is also available in the following transaction types:

```
ARRefundCreditCard
Bill
BillPaymentCheck
BillPaymentCreditCard
BuildAssembly
Charge
Check
CreditCardCharge
CreditCardCredit
CreditMemo
Deposit
Estimate
InventoryAdjustment
Invoice, ItemReceipt
JournalEntry
PurchaseOrder
ReceivePayment
SalesOrder
SalesReceipt
SalesTaxPaymentCheck
VendorCredit
```

Adding values to a custom field is not very well defined in the QBSDK, therefore it is difficult to explain how to use them with our plug-in. In some cases, the custom field is populated with a separate request and in other cases it is populated with the parent item request. For instance, when adding/modifying Customer in QuickBooks with the plug-in the DataExt aggregate is not available in the CustomerAdd nor CustomerMod request. But in the InvoiceAdd request a DataExt aggregate is available for each line item in the invoice. This inconsistency makes accessing custom fields difficult at best.

The most consistent way to add/mod/del the contents of a custom field is to use the DataExtAdd, DataExtMod, and DataExtDel requests. These requests can be found in the OSR.

Retrieving the contents of a custom field requires querying for the parent object and including the OwnerID field in the query, which is normally one of the last fields to be added to the request. Obtaining the DataExt values in the Response to a query objects (contacts, invoices,etc.) requires that the request contain the following PCQB_RqAddFieldWithValue("OwnerID" ; "0").

The above function adds the OwnerID field to the request, and populates it with a "0" . This causes QuickBooks to return the public data extensions (custom fields) with the response. (Advanced users can cause QuickBooks to return private data extensions by passing the GUID instead of "0", but this is only for advanced users). When QuickBooks returns the DataExt (custom fields) in the response, the plug-in user can access the information in the data extension. The following script demonstrates accessing the custom fields in a response:

```
If [ 0 = PCQB_RsOpenFirstRelatedRecord( "DataExtRet" ) ]
  Loop
    #the name of the custom field
    Set Field[ N_Field ; PCQB_RsGetFirstFieldValue( "DataExtName" ) ]
    #the value of the custom field
    Set Field[ D_Field ; PCQB_RsGetFirstFieldValue( "DataExtValue" ) ]
    #get next custom field/exit if there are no more
    Exit Loop If[ 0 <> PCQB_RsOpenNextRelatedRecord ]
  End Loop
  Set Field[ SomeField ; PCQB_RsCloseRelatedRecord ]
End If ...
```

Since custom fields are more advanced and can be quite complex, we are available for hire to assist with this development.

Working with Batches

When working with a multitude of requests to be transferred between FileMaker and QuickBooks, the most common way of handling them would be to loop and iterate over the found set of records, or to loop and iterate over the records in the response of a query. With the introduction of version 11, however, new functionality has been added to help make processing batches of requests a bit easier, as well as improving performance.

Batch queues are, put simply, a queue of requests to be executed in sequence with QuickBooks. When setting up requests (add, modify, or query), the request can be added to the queue using the PCQB_RqAddRequestToBatch function. This will take the request in memory and place it in the queue, readying it for being executed with other requests. Requests placed in the queue are not limited to being the same kinds of requests, either; a system can mix different kinds of Add requests with Mod requests, even Query requests, and they all will be handled as appropriate to their type.

After adding at least one request to the batch queue, calling PCQB_RqExecute will then process the queue by taking one requerst, submitting it to QuickBooks, then placing the response into the response queue, for each request in the request queue. The final result is a response queue filled with responses to each request, in the same order as the request queue: the first request maps to the first response, the third request maps to the third response, and so forth.

Accessing the batch response records is similar to accessing response records, by first calling PCQB_RsOpenFirstBatchRecord and then looping over the results with PCQB_RsOpenNextBatchRecord until the function returns "END". The act of opening a response record means that the entirety of the response, whether it is a single record, a group of records, or an error result, will be loaded as the response object in memory, exposing that response to future calls of any of the Response functions. When a script using batches proceeds to the processing portion, the developer should take care to properly nest the loops of batch responses and responses, so that all appropriate data is pulled into FileMaker.

Below is an example of using batch processing to submit a series of requests containing a mix of CustomerAdd and CustomerMod request types.

```
Allow User Abort [Off]
Set Error Capture [On]

# Verify the plug-in is ready to function
Perform Script ["Plug-in Checker"]

# Start batch loop
Go to Record/Request [First]
Loop
       If [not IsEmpty( Customers::QBID )]
              Set Variable [$result; Value: PCQB_RqNew( "CustomerMod" )]
              Set Variable [$result; Value: PCQB_RqAddFieldWithValue( "ListID" ;
Customers::QBID )]
              Set Variable [$result; Value: PCQB_RqAddFieldWithValue( "EditSequence" ;
Customers::EditSequence )]
       Else
              Set Variable [$result; Value: PCQB_RqNew( "CustomerAdd" )]
              Set Variable [$result; Value: PCQB_RqAddFieldWithValue( "Name" ;
Customers::FullName )]
       End If

              Set Variable [$result; Value: PCQB_RqAddFieldWithValue( "CompanyName" ;
Customers::CompanyName )]
              Set Variable [$result; Value: PCQB_RqAddFieldWithValue( "FirstName" ;
Customers::FirstName )]
              Set Variable [$result; Value: PCQB_RqAddFieldWithValue( "LastName" ;
Customers::LastName )]
```

```
        # Add the request to the request batch
        Set Variable [$result; Value: PCQB_RqAddRequestToBatch]

        Go to Record/Request [Next; Exit After Last:On]
End Loop

# Batch is complete, so execute the batch requests (We assume a session is active)
Set Variable [$result; Value: PCQB_RqExecute]

# Loop over the batch responses
Set Variable [$result; Value: PCQB_RsOpenFirstBatchRecord]

# Developer's Note: In this example script, we are only performing Add or Mod
# operations, so each batch record in the response will be one complete
# Customer record.  If any queries were included, then there would need to be
# additional scripting for a second loop that iterates over that batch record
# query's response set.

Loop
        Exit Loop If [$result = "END" or $result = "!!ERROR!!"]

        # When loading a response record from the batch, the status results
        # of that response are also loaded, so PCQB_SGetStatus will reflect
        # whether the request was successful or not for each response.
        If [PatternCount( PCQB_SGetStatus ) ; "Success" ) > 0]
                # Process successful response
                Set Variable [$res; Value: PCQB_RsOpenFirstRecord]

                Set Variable [$qbID; Value: PCQB_RsGetFirstFieldValue( "ListID" )]
                # Find Criteria: Find all where Customers::QBID == $qbID
                Perform Find [Restore]

                # Since these are all single-record responses to Add or Mod requests,
                # we can simply store the QB list information and move on to the next
                # record.
                Set Field [Customers::QBID; PCQB_RsGetFirstFieldValue( "ListID" )]
                Set Field [Customers::EditSequence; PCQB_RsGetFirstFieldValue(
        "EditSequence" )]
        Else
                # Process error response
                Set Field [Customers::QBError; PCQB_SGetStatus]
        End If

        Set Variable [$result; Value: PCQB_RsOpenNextBatchRecord]
End Loop

# After all the batching has been done, we need to "clean up" the batch queues,
# so that the requests and responses are cleared out and won't be included in the
# next batch.
Set Variable [$result; Value: PCQB_SClearBatchQueues( "Both" )]
```

On the surface, the structure of handling batches is similar to handling sets of records, with the key exception being the single call to PCQB_RqExecute and the looped handling of the response queue. This method may appeal to developers who prefer a process to queue up the requests before sending them off to QuickBooks (such as an "end of day" run of records), rather than performing each request at the time the request is generated.

10) Use the OSR (Onscreen Reference Manual)

Now that you thoroughly understand how FileMaker and QuickBooks "talk" to each other by making requests and responses, we are ready to introduce Intuit's OSR. The OSR contains a complete list of all available fields/filters, detailed descriptions of the request types, errors and specifies field order. This will be crucial during your development.

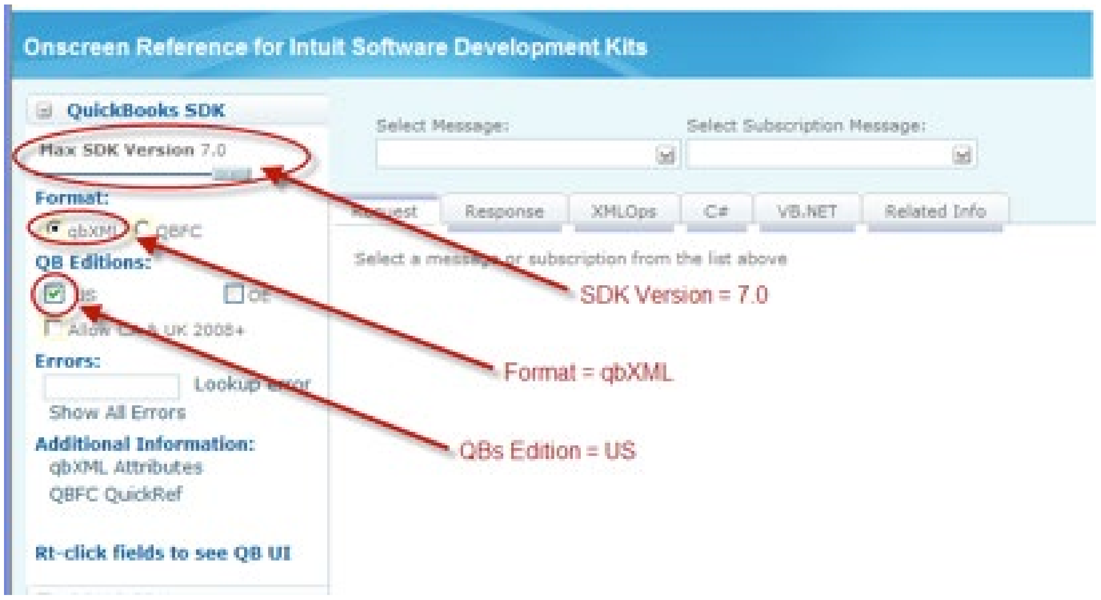
The OSR can be found at the following link:
https://static.developer.intuit.com/qbSDK-current/common/newosr/index.html

Step 1 - Set Up

First use the control panel on the left to select the proper settings.
- The **SDK Version** should be set to the latest SDK version. For example, if using QB 2014 then the SDK should be set to 13.0. If using QB 2013 then the SDK should be set to 12.0. The formula is typically the QB version minus 1.
- The **Format** will need to be changed to qbXML. The format should <u>ALWAYS</u> be set to qbXML.
- The **QB Editions** specify what international version of QB you are using. If you are using the US edition, then select "US". If you are using the Canadian or UK edition, then please select "Allow CA and UK" and select the appropriate edition. For the Australian version select UK.

For example, in Figure 5.0 below the SDK Version is 7.0, the format is qbXML and the US edition of QuickBooks has been selected.

Figure 5.0



Max SDK Version 7.0

Format: [✓ qbXML] QBFC

QB Editions: [✓ US] OE

Allow CA & UK 2008+

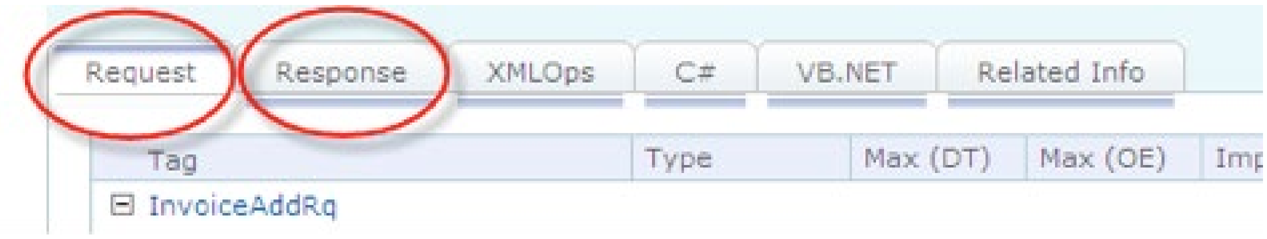
SDK Version = 7.0

Format = qbXML

QBs Edition = US

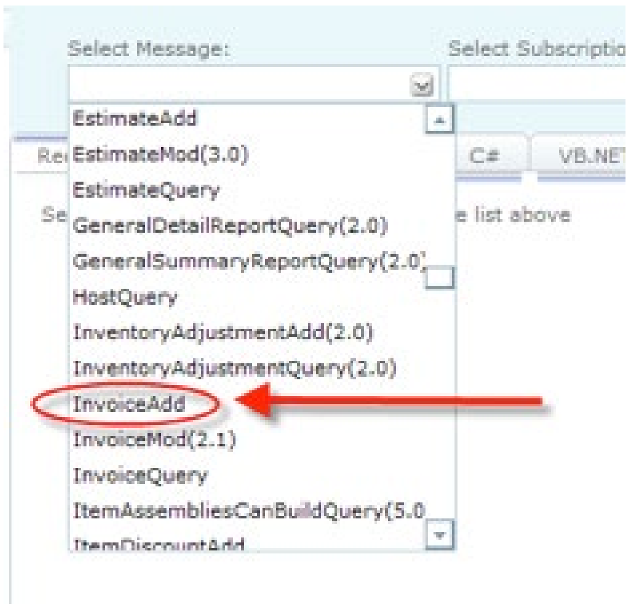
Step 2 - Using the OSR
You will only be working with the Request and Response Tabs as shown below in Figure 5.1. All other tabs can be ignored.

Figure 5.1

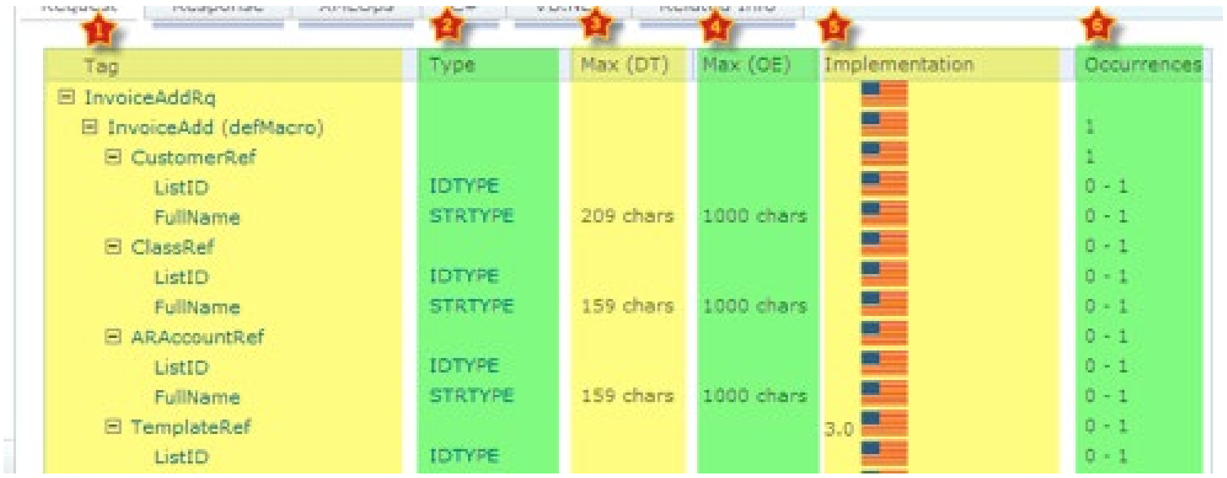


Since you already understand what Requests and Responses are, select the appropriate tab to begin. After specifying the Response or Request tab, then you will select a Message. The Message defines which class of records we want to work with and which action we want applied to those records. In our example in Figure 5.2 we will select the Message called "InvoiceAdd" in order to add an Invoice to QuickBooks. The "InvoiceAdd" Message tells QuickBooks that the request wants to create a new Invoice record in the QuickBooks file.

Figure 5.2


Select Message:
Select Subscriptio
EstimateAdd
EstimateMod(3.0)
EstimateQuery
GeneralDetailReportQuery(2.0)
GeneralSummaryReportQuery(2.0)
HostQuery
InventoryAdjustmentAdd(2.0)
InventoryAdjustmentQuery(2.0)
InvoiceAdd
InvoiceMod(2.1)
InvoiceQuery
ItemAssembliesCanBuildQuery(5.0
C#
VB.NE
e list above

You will now be looking at a screen with columns titled Tag, Type, Max (DT), Max (OE), Implementation and Occurrences. Let's have a closer look at what these columns mean in Figure 5.3.

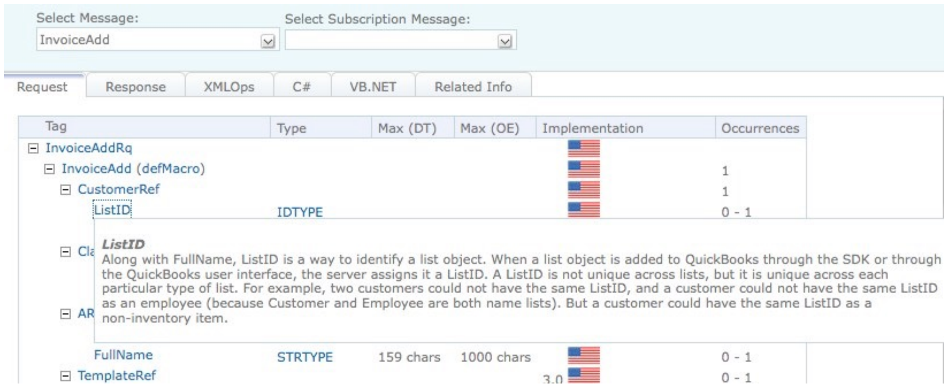


Tag	Type	Max (DT)	Max (OE)	Implementation	Occurrences
⊟ InvoiceAddRq				🇺🇸	
⊟ InvoiceAdd (defMacro)				🇺🇸	1
⊟ CustomerRef				🇺🇸	1
ListID	IDTYPE			🇺🇸	0 - 1
FullName	STRTYPE	209 chars	1000 chars	🇺🇸	0 - 1
⊟ ClassRef				🇺🇸	0 - 1
ListID	IDTYPE			🇺🇸	0 - 1
FullName	STRTYPE	159 chars	1000 chars	🇺🇸	0 - 1
⊟ ARAccountRef				🇺🇸	0 - 1
ListID	IDTYPE			🇺🇸	0 - 1
FullName	STRTYPE	159 chars	1000 chars	🇺🇸	0 - 1
⊟ TemplateRef				3.0 🇺🇸	0 - 1
ListID	IDTYPE			🇺🇸	0 - 1

Figure 5.3

```
1. Tag = name of the element or field and the value that you put into the function
For example, PCQB_RqAddFieldWithValue( "CustomerRef::FullName" ; "Bob Jones" )
Please left click on the element name to obtain a detailed description as shown in Figure
5.4 below.
Some element names will also allow you to right click and obtain a corresponding QB field
mapping image.
```

Figure 5.4



Tag	Type	Max (DT)	Max (OE)	Implementation	Occurrences
⊟ InvoiceAddRq				🇺🇸	
⊟ InvoiceAdd (defMacro)				🇺🇸	1
⊟ CustomerRef				🇺🇸	1
ListID	IDTYPE			🇺🇸	0 - 1
⊟ Cla					
⊟ AR					
FullName	STRTYPE	159 chars	1000 chars	🇺🇸	0 - 1
⊟ TemplateRef				3.0 🇺🇸	0 - 1

ListID
Along with FullName, ListID is a way to identify a list object. When a list object is added to QuickBooks through the SDK or through the QuickBooks user interface, the server assigns it a ListID. A ListID is not unique across lists, but it is unique across each particular type of list. For example, two customers could not have the same ListID, and a customer could not have the same ListID as an employee (because Customer and Employee are both name lists). But a customer could have the same ListID as a non-inventory item.

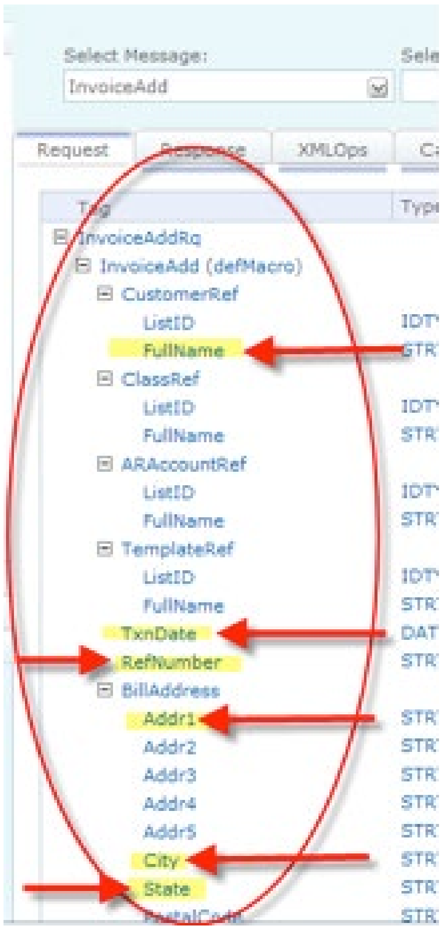
```
2. Type = type of data that goes into this field.

3. Max (DT) = number of allowable characters in the field for desktop versions.

4. Max (OE) = No longer applicable.

5. Implementation = Required SDK version indicated by each country's flag. If a number
accompanies the flag such as 3.0 as shown above, then that is the minimum version of the
SDK that is required for that field.

6. Occurrences = 1 implies a required field, 0-1 implies an optional field, and 0-n implies
an optional field that multiple records can use also known as 0-many.
```

Step 3 - Proper Field Order

The proper field order is crucial. If one required field is missing or one field is out of order, then you will encounter errors. The field order is specified from top to bottom in the OSR and must be referenced in the script in that exact order. For example, in Figure 5.5 when adding an invoice to QuickBooks we must specify the proper field order which is determined by the OSR as shown below.

Figure 5.5 - Sample script showing field order taken directly from the OSR

```
PCQB_RqAddFieldWithValue( "CustomerRef::FullName" ; "Bob Jones" )
PCQB_RqAddFieldWithValue( "TxnDate" ; "2006/01/01" )
PCQB_RqAddFieldWithValue( "RefNumber" ; "123456789" )
PCQB_RqAddFieldWithValue( "BillAddress::Addr1" ; "123 Any Street" )
PCQB_RqAddFieldWithValue( "BillAddress::City" ; "Any Town" )
PCQB_RqAddFieldWithValue( "BillAddress::State" ; "Any State" )
```



11) Error Handling

We find that most developers run into issues due to a lack of error trapping. Please ensure that you properly trap for errors in your solutions.

Typically, the plug-in functions will return a 0 for a success. However, any of the plug-in functions may encounter an error during processing and will return the !!ERROR!! string. When an !!ERROR!! occurs during processing, immediately call the PCQB_SGetStatus function in order to obtain a full description of the error. This function returns the message associated with the last error. The status is used to identify errors in the request or the processing of requests. The text returned by this function will help troubleshoot script or logic failures.

This makes it simple to check for errors. If a plug-in function does not return a 0 or returns !!ERROR!!, then immediately after call PCQB_SGetStatus function for a detailed description of what the exact error was. Here are a few samples of how you can check for errors.

Example 1:
```
Set Variable [ $result = PCQB_SomePluginFunction( “a” ; “b” ) ]
If [ $result = !!ERROR!! ]
    Show Custom Dialog [ “An error occurred: “ & PCQB_SGetStatus ]
End If
```

Example 2:
```
Set Variable [ $result = PCQB_SomePluginFunction( “a” ; “b” ) ]
If [ $result ≠ 0 ]
    Show Custom Dialog [ “An error occurred: “ & PCQB_SGetStatus ]
End If
```

It is good practice to ALWAYS trap for errors immediately after the PCQB_BeginSession and PCQB_RqExecute functions.

Some features of the QBXML SDK have not been fully implemented. When such a feature is used, QuickBooks will return a "warning" status code of 530. The request that generates this status code response will still be processed by QuickBooks, and the plug-in will return an error result of "!!WARNING!!". Calling PCQB_SGetStatus will return the fields used in the request that have generated the warning message.

Function Notes for Error Handling:
Please see the accompanying FM Books Connector Functions Guide for further details.

- PCQB_RqExecute - Upon receiving a status code 530 response, the function result will be "!!WARNING!!". Response will still be processed as though a "0" result is received.

- PCQB_SGetStatus - Will return the proper error code result.
  - Code: 530
  - Severity: Warning
  - Message: <message from QB detailing what caused the 530 response>

QuickBooks Error Codes:
During the course of development and active use, certain errors may arise that are generated from QuickBooks directly. To better understand these errors and resolve them, we recommend reviewing the error code on the list at the following website: http://wiki.consolibyte.com/wiki/doku.php/quickbooks_error_codes

12) Tips

Tax Tips

Tax varies depends on the version of QuickBooks you are using and requires knowledge of how QuickBooks handles tax. Since we do not provide QuickBooks training, we assume that you already have knowledge of how your QuickBooks file handles tax. For example, when using the Canadian version of QuickBooks, you will probably want to turn off Sales Tax. When using the US version sales tax will be applicable and should be turned on.

The following are some helpful tips when using the various QuickBooks versions available.

U.S. Version

- The tax rate is pre-entered in QuickBooks. However, you can modify the tax rate from FM to QB.
- The tax rate is set up at invoice level and not the line item level.
- An item must be specified as taxable or non-taxable. This is set up at the line item level for an invoice or at the item level. It is synonymous.
- If you do not specify the tax rate in QB, then the QB file default will be applied.

Canadian Version

When using the Canadian version please make the following two important changes.

- Change "State" to "Province".
- Remove or adjust any reference to Sales Tax.

UK and Australian Version

- If using UK QuickBooks versions 2007 or earlier, then we recommend referencing the old OSR for your development as Intuit's new OSR only appears to support UK QuickBooks versions 2008 or later. Please download the full SDK 6.0 to access the old OSR. Here is the direct link to download the full SDK 6.0. http://www.synqware.com/download/QBSDK60.exe. You will need to navigate through the SDK folders to locate the OSR. The old OSR will list all available fields including field types for multi-currency.
- The Australian versions of QuickBooks typically adhere to the UK version requirements. Please note, If you are using Reckon, the FM Books Connector is not compatible with this application.

Formatting Tips

By default, the plug-in will attempt to format any values passed to it via the PCQB_RqAddFieldWithValue function by prepping the values and ensuring they are acceptable according to XML format guidelines. This means that some characters may become "escaped" in order to protect the integrity of the underlying XML document that QuickBooks works with. An example of this would be the changing of the ampersand character (&) to "&amp;", or the quotation character (") to "&quot;".

In some newer versions of QuickBooks, the underlying QBXML processor may also be applying this kind of formatting on any submitted QBXML document from a third-party source such as the FM Books Connector plug-in. This may result in the appearance of "double-escaped" characters in QuickBooks locations such as a customer name, item description, and so forth. As of version 12.0.0.5 of the FM Books Connector plug-in, a new function has been added called "PCQB_SToggleApplyFilter" that will instruct the plug-in to either apply the special character format filter or to not apply the special character format filter. Developers can troubleshoot instances of double-escaped characters by calling PCQB_SToggleApplyFilter and providing a Boolean value of "False" at the beginning of the user's session to instruct the plug-in to refrain from applying the special character filter.

13) Known Issues

FileMaker 13
We have observed the following behavior in testing for FileMaker 13 compatibility:

1) When attempting to begin a session with QuickBooks, a series of Windows configuration windows will appear, attempting to "set up" QuickBooks.
2) When attempting to begin a session with a QuickBooks company file with QuickBooks closed (in "unattended mode"), the result of the PCQB_BeginSession function will sometimes fail, with an error code result of "0x80040402" or "0x80040408".

This behavior appears to be independent of the FM Books Connector. To resolve this behavior, we recommend the following steps:

- Resolving Issue #1:
  Download and install the Microsoft® .NET 4.5 Framework from Microsoft at the following link:
  http://www.microsoft.com/en-us/download/details.aspx?id=30653

- Resolving Issue #2:
  - Before each call to PCQB_BeginSession or PCQB_EndSession, insert a Pause Script step for one (1) to three (3) seconds. This will allow the qbXML Request Processor (QBXMLRP2) to complete any background processes, such as closing down the access point to QuickBooks or resolving any pending requests, before the plug-in makes another call to the Request Processor, and minimize the error results passed back from the PCQB_BeginSession function call.
  - If possible, begin the session with the company file opened in QuickBooks on the machine running FileMaker. The behavior experienced when connecting in unattended mode does not happen when connecting to an already-opened session of QuickBooks.
  - If having the QuickBooks application open is not feasible for your solution, you can minimize the amount of times that PCQB_BeginSession is called. Before performing a series of QuickBooks transactions (pushing/pulling customers, pushing/pulling invoices, pushing/pulling items, etc.), you can call PCQB_BeginSession once, with the QuickBooks company file path designated as the first parameter, then perform the needed requests and responses within the existing session. There can be any number of request/response transactions that take place between PCQB_BeginSession and PCQB_EndSession. Once all transactions have been completed, and no further transactions are necessary, call PCQB_EndSession to gracefully end the session. We also recommend placing a Pause/Resume Script step with a duration of one (1) to three (3) seconds after calling PCQB_EndSession, to give the QBXMLRP2 background processes to gracefully close all related background processes before calling PCQB_BeginSession again.

We have made Intuit® aware of issue #2, and we are working with them to find a more concrete resolution. Once this resolution is reached, an update will be released, and the resolution will be documented.

Session Timeout Issue
In version 9.0.0.0, we encountered an error in which the plug-in would return an error of "The server threw an exception" when calling PCQB_BeginSession, PCQB_EndSession, and PCQB_RqExecute in certain situations. This was due to an inherent 2 minute "timeout" for the QuickBooks session that was previously unknown. In version 9.0.0.1, the plug-in will automatically attempt to refresh its session in order to attempt to mitigate these particular issues, so that it can "extend" the timeout. As long as sessions are properly begun and ended in close succession to their intended requests, the plug-in will not encounter this particular error. However, if this error is encountered, the user should force-close QuickBooks using the Task Manager (after saving all of their changes) and re-start it, then begin a fresh session with the plug-in.

In 9.0.0.2, the 32-bit version of the plug-in has been adjusted to utilize a more direct conduit to communicate with QuickBooks from FileMaker. This conduit has resolved the "server threw an exception" error. The 64-bit version of the plug-in, however, still will utilize the FM QB Bridge to handle the translation from 64-bit FileMaker to 32-bit QuickBooks.

This issue has been fixed as of version 10.0.0.0.

PCQB_Browse
The error returned when attempting to "open" a file that is already opened on a Windows network is intended Windows functionality; it is tied to the Windows file security model, and cannot be overridden.

If a developer's solution requires the use of PCQB_Browse, it is best to attempt to use that when no one else has opened the QuickBooks company file, so that the file path can be acquired successfully, or otherwise manually enter in the file path for PCQB_BeginSession to utilize.

PCQB_BeginSession with Unattended Access Mode
When beginning a session with a company for the first time in order to set up access for the FM Books Connector, if the developer plans to use Unattended Access mode (transferring data with the QuickBooks company file while QuickBooks is closed), the developer must make note of whether they opened the company file by a mapped network path (e.g. "H:\") or a UNC (e.g. "\\server\share\") path. When calling PCQB_BeginSession with the company file path, that path must be the exact same as the path that was used when setting up the connection certificate in QuickBooks. This is intended QuickBooks functionality.

FileMaker Plug-in Function Script Step with PCQB_BeginSession's "Connection Mode" Parameter
We have discovered a possible bug with the FileMaker Plugin API in regard to handling plug-in function script steps where the default value of a drop-down list type parameter is not the first entry in the list. Specifically, when specifying the "PCQB_BeginSession" function as a script step, if the developer does not explicitly choose the "Do Not Care" option, which is the default value, the plug-in will behave as though the "Single User Mode" option was selected. This issue has been reported and we are awaiting an update to the API to resolve the problem.

The workaround for this issue is to explicitly specify the "Do Not Care" option simply by clicking the dropdown list and choosing the option from the list. This will properly set the value for the plug-in.

III. Contact Us

Successful integration of a FileMaker plug-in requires the creation of integration scripts within your FileMaker solution. A working knowledge of FileMaker Pro, especially in the areas of scripting and calculations is necessary. If you need additional support for scripting, customization or setup (excluding registration) after reviewing the videos, documentation, FileMaker demo and sample scripts, then please contact us via the avenues listed below.

Phone: 760-510-1200
Email: support@productivecomputing.com
Help Center: https://help.productivecomputing.com/help/fm-books-connector
FM Forums: https://fmforums.com/forum/290-fm-books-connector/

Please note assisting you with implementing this plug-in (excluding registration) is billable at our standard hourly rate. We bill on a time and materials basis billing only for the time in minutes it takes to assist you.

QuickBooks integration training is available through Productive Computing University. We offer a training course to provide you with the fastest way to master the skills needed to integrate FileMaker with QuickBooks Desktop.
https://www.productivecomputinguniversity.com/courses/fm-books-connector-for-quickbooks-desktop

We will be happy to create your integration scripts for you and can provide you with a free estimate if you fill out a Request For Quote (RFQ) at www.productivecomputing.com/rfq. We are ready to assist and look forward to hearing from you!