Function Reference for FX.php

This document is intended to demonstrate the functions in FX.php. Almost all of the functions within FX can be called in any order. There are, however, some exceptions: 1) the declaration of an instance of FX must come before the others; and 2) ***one*** of the query functions must be called very last. A list of the functions follows with descriptions of each relevant parameter. Please note that in the original release, the FX class was called FMData() instead.

Important: Whenever FX is used, you *must* call FX() and *one* of the query functions. Some of the other functions are required as well, depending on the type of query you're trying to execute. Within this document, the description of each of the functions in the 'Next Functions' Group indicates where it is required and where it is optional.

Conventions used:
+ $ReturnedData is used to signify the array containing the data returned by a query. For examples of the syntax used to populate $ReturnedData, see the 'Last Functions' portion of this document.
+ Lines where parameter descriptions begin are indicated by a leading '>> '.

First Function to Call

FX ($dataServer, $dataPort=591, $dataType='', $dataURLType='')

>> The first parameter, **$dataServer**, should be the IP address (I think a domain name would work, too) of the machine where you have the FileMaker Pro Web Companion running.

>> $dataPort is an optional parameter specifying on which port FileMaker data is accessible.

In FileMaker 5 thru 6, this can be configured in FileMaker Unlimited (under the 'Edit' menu in most versions or the 'FileMaker Pro' menu in Mac OS X) under:

Preferences->Application->Plug-Ins

Then select 'Web Companion' and click the 'Configure...' button. There will be a box where the Port Number can be entered. This is set to port 80 by default, but it states in the FileMaker help (under 'Web Companion, Specifying a port number for web publishing'):

FileMaker, Inc. has registered port number 591 with the Internet Assigned Numbers Authority (IANA) for use with FileMaker Pro Web Companion.

If neither of these ports works for you, you might also consider using one of the ports between 49152 and 65535. According to IANA, these ports are considered Dynamic and/or Private. That is to say, they aren't registered to any specific application.

In the case of FileMaker 7 Server Advanced, FIleMaker data will be accessible on the same port on which the associated web server is running. This will usually be port 80, or port 443 for SSL connections.

>> The thrid parameter, **$dataType**, is new as of version 3.x of FX.php. This optional parameter is used to specify what sort of database will be accessed. Please note that if you will always be accessing the same type of data source, you can simply change this line in FX.php:

var $dataServerType = 'xxxx';

Where 'xxxx' is the parameter from the list below corresponding to your data source.

The possible values for this parameter (case is not important) are:

FMPro5/6 - for FileMaker version 5 thru 6 databases.

FMPro7 - for FileMaker 7 Server Advanced data sources.

CAFEphp4PC - when using FMWebschool's CAFEphp to connect to FileMaker client on the PC.

ODBC - to connect to an ODBC data source.

OpenBase - if connecting to an OpenBase data source.

MySQL - if connecting to a MySQL data source.

>> The final parameter is **$dataURLType**. This parameter is new as of version 4.x of FX.php, and is only relevant when connecting to a FileMaker 7 data source. $dataURLType is an optional parameter with two possible values (case insensitive): HTTP or HTTPS. When HTTPS isn't specified (or when not connecting to FileMaker 7), HTTP is assumed. This parameter is used when connecting to FileMaker 7 Server Advanced via SSL.

Typically, you would call this function thus:

$InstanceName = new FX('127.0.0.1'); // if you are using port 591

or,

$portNumber = '49494'; // where the value is the port you want to use
$InstanceName = new FX('127.0.0.1', $portNumber);

or,

$InstanceName = new FX('127.0.0.1', 591, 'FMPro5/6'); // to connect to FileMaker 5 or 6

or,

$InstanceName = new FX('127.0.0.1', 591, 'FMPro7', 'HTTPS'); // to connect to FM7SA using HTTPS

Next Functions to Call

The functions within this group can be called in any order relative to one another. Some of these functions are optional, and some might be used multiple times in a particular query.

SetDBData ($database, $layout="", $groupSize=50, $responseLayout="")

SetDBData() is a required function ***except*** when the action called is FMDBNames(). When performing an FMDBNames() action, the SetDBData() function should not be called. In any other instance, if you fail to specify the data relating to where FX can find your data, only an error will be returned.

>> The **$database** parameter is a required parameter for every action ***except*** FMDBNames(). It should contain the name of the database to be accessed. The parameter can either be a variable, as shown here, or the literal name of the database: e.g. 'Books.fp5'.

>> $layout is technically an optional parameter for FileMaker versions 5 thru 6, but is required for most queries in version 7 (failure to specify a layout when connecting to a FileMaker 7 database will result in an error. It indicates the name of the layout to access. It is optional in the older versions of FileMaker because by default FileMaker 5 and 6 access an internal layout which contains every field. However, using the default layout ***will almost always result in reduced performance***. On the other hand, this parameter should not be passed, regardless of FileMaker version, when that action to be called is FMLayoutNames(). The $layout parameter is also unnecessary for

FMScriptNames(), a new function in FX.php version 3.x specifically for FileMaker 7. For all other actions (e.g. FMFind(), FMAdd(), etc.), I recommend that the developer create layouts which contain ***only*** those fields necessary for a specific query (this is generally good practice, no matter how you access FileMaker via the web.) This will result in a measurable improvement in the Web Companion's performance.

>> The next parameter, **$groupSize**, is used to set how many records FX returns at once. As is shown above, this is 50 by default. The special value 'All' may be used as well. The example below sets the database, layout, and a group size of 10 records:

```
$returnCount = 10;
$InstanceName->SetDBData('MyDatabase', 'web_layout', $returnCount);
```

>> Finally, **$responseLayout** is a new parameter in version 3.x of FX.php which is specific to version 7 of FileMaker. When accessing a FileMaker version 7 data source, it is possible to perform the request on one layout, and then use a different layout to pull the returned data. Caution should be used with this parameter since a change in layout in FileMaker 7 may correspond to a change in context. In the following example, the query will be performed on a layout called "web_find", but the data returned will use the layout "web_details":

```
$InstanceName->SetDBData('MyDatabase', 'web_find', 20, 'web_details');
```

SetDBPassword ($DBPassword, $DBUser='FX')

SetDBPassword() is an optional function unless your FileMaker database is password protected. However, if your database is password protected, you need to provide a password that provides the level of access a user would need to perform the actions your script will be asked to do. Should you fail to specify a password for a database that is configured to require one, only an error will be returned.

In the event that you're using FileMaker Pro's Web Security database for Web Companion security, or you're accessing a secured FileMaker 7 database, you'll need to specify both a user name and password. As is always the case with FileMaker Pro passwords, be sure that the user name and password combination that you're using is allowed to perform the desired action. In the case of version 7 of FileMaker, this includes insuring that the password used is allowed to access data via the web.

>> $DBPassword is the only parameter needed by SetDBPassword() if you're using FileMaker Pro's built in security. It can either be the literal password for the database, or a variable containing the password. For an extra measure of security, it may be a good idea to keep the file containing your password outside of your web folder and include it via PHP. A literal password would be set like this:

```
$InstanceName->SetDBPassword('knock-knock');
```

>> $DBUser need only be passed if: 1) you're using FileMaker Pro's Web Security database for Web Companion security (check your plug-in settings); **or** 2) you're accessing a secured FileMaker 7 database. Please note that because **$DBUser** is only needed for a specific type of FileMaker authentication, it is always passed **second** when it is used. Usage and security concerns are similar to those for **$DBPassword**. A literal user and password could be set like this:

$InstanceName->SetDBPassword('knock-knock', 'webuser');

SetDBUserPass ($DBUser, $DBPassword ='')

This function -- SetDBUserPass() -- is identical to the one above, but it may seem friendlier to users of FileMaker 7 and SQL databases. The notes for SetDBPassword() should be consulted for details about the use of this script. Notes about the minor differences related to the parameters follow:

>> FileMaker 7 and SQL databases generally always operate with a username, even if there is no password for a given user. For this reason, **$DBUser** is the only parameter required by SetDBUserPass(). In contrast to the previous function, SetDBUserPass() expects **$DBUser** to be the first parameter passed.

>> $DBPassword is passed to SetDBUserPass() second. This password is optional, and should contain the password associated with the user specified via $DBUser (see above.)

SetCharacterEncoding ($encoding)

This function, and the one below -- SetDataParamsEncoding() -- are new in version 4.x of FX.php and are used when connecting to a multi-byte data-source without using UTF-8 encoding. Note that this function and the one below **both require that multi-byte support be compiled into PHP** (on the Mac, Complete PHP includes multi-byte support.)

Even though if your site is based on UTF-8, it's better to call SetCharaterEncoding method as like. You could see any multi-byte characters as you want, but they are numeric character reference style as like "&#nnnnn;. They could show as valid letter but it's not good for string operations. The setCharacterEncoding method realize you could get the string of valid encoded characters.

```
$fx = new FX('127.0.0.1', '80', 'FMPro7', 'HTTP');
$fx->setCharacterEncoding( 'UTF-8' );
```

>> The **$encoding** parameter passed to this function is used to determine both how data is passed to FileMaker, and how data is returned from FileMaker. If you need to set separate encoding for each direction, ***first*** use SetCharacterEncoding() to set the encoding of data coming from FileMaker, then use SetDataParamsEncoding(), below, to set the encoding of data being passed to FileMaker.

SetDataParamsEncoding ($encoding)

This function, and the one above -- SetCharacterEncoding() -- are new in version 4.x of FX.php and are used when connecting to a multi-byte data-source without using UTF-8 encoding. Note that this function and the one above **both require that multi-byte support be compiled into PHP** (on the Mac, Complete PHP includes multi-byte support.) Usually you don't specify this method on FileMaker Server 7 or above and your site is based on UTF-8. This is required for before FileMaker 6.

>> The **$encoding** parameter passed to this function is only used to determine how data is passed to FileMaker. See SetCharacterEncoding() for information on how to use the encoding functions together.

SetDefaultOperator ($op)

By default, FX.php performs begins with searches when performing a find. If desired, SetDefaultOperator() can be used to change this behavior -- if, for example, most of your queries compare using equals.

>> $op is the only parameter taken by SetDefaultOperator(). Possible values for **$op** are covered in detail in the **$op** parameter section of AddDBParam().

AddDBParam ($name, $value, $op="")

The AddDBParam function is not technically required (you won't see a query related error if you omit it), but calls to FMFind(), FMDelete(), and FMEdit() won't do much without it. AddDBParam() is the method by which query criteria are specified. It is common to use more than one instance of this function in a query, especially when performing complex searches or updating an existing record.

>> The **$name** parameter is used to specify which data you wish to make a query against. Usually, this will be a field name. The examples at the bottom of this function will be of this type. In such cases, there are a few things to keep in mind:

- If the field is from a related database, you'll need to make sure that the relationship is appended to the field name, e.g. relationship_name::field_name.

- When working with FileMaker versions 5 thru 6, the field specified must appear on the layout to be used for the current query.

- For FileMaker 7, if the field specified doesn't occur on the table associated with the current layout, the field name must be fully qualified. (e.g. tableName::fieldName(repetitionNumber).recordID)

Beyond the use of field names in AddDBParam(), there are a few special cases:

1) When performing an FMEdit() or FMDelete() query, you'll need some code similar to this:

```
$InstanceName->AddDBParam('-recid', $currentRecord);
```

Where $currentRecord is the FileMaker RecordID of the record you wish to update. The FileMaker RecordID is part of the information returned by FX.php for each record in a found set. If an array called $ReturnedData contains the data returned by FX.php, the keys in the $ReturnedData['data'] subarray contain both the RecordID and the ModifyID (the next exception that will be discussed) separated by a period. Both IDs could be extracted from the current key like this:

```
$recordPointers = explode ('.', $key);        // where $key is the current key from $ReturnedData['data']
$currentRecord = $recordPointers[0];      // here's the RecordID
$currentModified = $recordPointers[1];   // and here's the ModifyID
```

2) When performing an FMEdit() query, there might be cases that you want to ensure that the data being submitted is indeed newer than what's in the database. This can be done by adding an instance of AddDBParam() where **$name** is '-modid' (as mentioned in conjunction with -'recid' above.) Like this:

```
$InstanceName->AddDBParam('-modid', $currentModified);
```

If the record in the database is more recent than the data upon which the current update is based, $ReturnedData['errorCode'] will contain the value 306. This condition could be checked in this manner:

```
if ($ReturnedData['errorCode'] == 306) {
    echo "Sorry the data submitted is older than the existing data.";
}
```

Complete documentation on the error codes that may be returned to FX by the Web Companion can be downloaded from:

http://www.filemaker.com/downloads/pdf/xml_Appendix_C.pdf

3) Searches by default are logical 'and' searches. This means that if you pass in multiple AddDBParam() statements, the resulting find will return only those records that match ***all*** of the criteria. If you would instead like to perform a search that returns items that match ***any*** of the criteria specified, you can set a logical 'or' search using the special '-lop' **$name** parameter:

$InstanceName->AddDBParam('-lop', 'or');

This special parameter ***only*** needs to be used when requesting a logical 'or' search. Logical 'and' is always assumed otherwise (this makes sense if one considers the ways that logical 'or' and 'and' searches are performed in FileMaker.)

You can use FMFindQuery() to perform a “compound find request” (like adding new Find requests in FileMaker). You’ll want to make sure you include the ‘-query’ parameter like so:

$InstanceName->AddDBParam(‘-query’, ‘(q1,q2)’);

Where q1 and q2 are your find requests.

$InstanceName->AddDBParam(‘-q1’, ‘PetName’);
$InstanceName->AddDBParam(‘-q1.value’, ‘Spot’);
$InstanceName->AddDBParam(‘-q2’, ‘Breed’);
$InstanceName->AddDBParam(‘-q2.value’, ‘Shiba Inu’);

4) There may be cases when you would like to perform a FileMaker script on the returned data set. In such cases, use one of the '-script' **$name** parameters: '-script', '-script.prefind', or '-script.presort'. When these parameters are used, the **$value** parameter should be the name of the FileMaker script that you wish to be executed. Usage is fairly self-explanatory. When '-script' is used, the specified FileMaker script will be executed ***after*** the current find and sort (if any) are performed. Using '-script.prefind' causes the specified FileMaker script to execute ***before*** the current find and sort are executed. The '-script.presort' **$name** parameter runs the specified FileMaker script ***after*** the current find, but ***before*** the current sort is executed. Usage of these parameters would be similar to the following examples:

$InstanceName->AddDBParam('-script', 'Relookup Addresses');
$InstanceName->AddDBParam('-script.prefind', 'Delete Expired Records');
$InstanceName->AddDBParam('-script.presort', 'Omit Dulicates');

>> $value contains the value that a query should locate within the field specified by **$name**. In the case of an FMEdit() query, **$value** is used to hold the new values to be assigned to the fields in the record specified with the '-recid' AddDBParam() statement, as described in the special cases above. **$value** may also hold values corresponding to special cases as previously described.

Wild cards can be used in **$value** variables just like they can in FileMaker. If your wild cards seem to be behaving strangely, try using PHP's urlencode() function on **$value**.

>> The third parameter in AddDBParam statements is **$op**. This parameter is optional and without it, FileMaker's default 'begins with' search is performed. This parameter should not be used with the special cases described above. **$op** is the operator that you wish to use to compare data at the field level. A value for **$op** must be specified for each field that you wish to search in a manner other than the default search. The possible values for $op are as follows (syntax is the same for all of these; see the examples at the end of this section for an example):

'eq' - an 'equals' search.
'cn' - a 'contains' search.
'bw' - a 'begins with' search.
'ew' - an 'ends with' search.
'gt' - a 'greater than' search.
'gte' - a 'greater than or equal to' search.
'lt' - a 'less than' search.
'lte' - a 'less than or equal to' search.
'neq' - a 'not equal to' search.

For additional information related to FileMaker versions 5 thru 6, see page B-7 in the following documentation from FileMaker:

http://www.filemaker.com/downloads/pdf/xml_Appendix_B.pdf

AddDBParam() could be used for a FMFind() query something like this:

```
$InstanceName->AddDBParam('FirstName', 'John');
$InstanceName->AddDBParam('LastName', 'Smith', 'eq');
```

On the other hand, the AddDBParam() statements for an FMEdit() query might look something like the following:

```
$InstanceName->AddDBParam('-recid', $currentRecord);
$InstanceName->AddDBParam('FullName', $fullName);
$InstanceName->AddDBParam('Address', $address);
```

Please note that operators are not used for FMEdit() queries. Other than that (and the special '-recid' parameter) AddDBParam() is used in the same manner in these examples. In either case, variables or literal values can be used as the parameters.

AddDBParamArray ($paramsArray, $paramOperatorsArray=array())

This function is designed to streamline the creation of FX.php database queries. Instead of calling AddDBParam() multiple times, the field names and parameters can be combined into an array and a single call made to AddDBParamArray().

>> **$paramsArray** is the primary parameter for this function and should contain an array with each key the name of a field, and each value the data which should be passed.

>> The second parameter, **$paramOperatorsArray**, should be an array containing one operator for each field where the default operator should not be used. As with $paramsArray, each key should be a field name, but the values should be the operator to be used. Note that operators are only meaningful for find requests. For more information about operators, see the information in AddDBParam(), above.

An basic example use follows:

```
$formNames2FieldNames = array('fname' => 'FIrst_Name', 'lname' => 'Last_Name', 'id'
=> 'Rep_ID');
$fieldOperators = array('Rep_ID', 'eq');
$searchArray = array();
// the next lines take data from a form, and associate it with the appropriate fields
foreach ($formNames2FieldNames as $formName => $fieldName) {
      if (isset($_POST[$formName])) {
            $searchArray[$fieldName] = $_POST[$formName];
      }
}
// since the data is in arrays, the find requires very few lines
$InstanceName = new FX('127.0.0.1', 591);
$InstanceName->AddDBParamArray($searchArray, $fieldOperators);
$InstanceData = $InstanceName->FMFind();
```

SetPortalRow ($fieldsArray, $portalRowID=0, $relationshipName='')

The SetPortalRow() function is used to edit a portal row, or to create a record via a portal row. It's the equivalent of making several AddDBParam() calls using the full FileMaker portal syntax for the field name: "relationship_name::fieldName.<recID>".

>> The first parameter, **$fieldsArray**, should be an array of the data to be passed into the portal row. Each element of the array should have the FileMaker field name as the key, with the desired value as the value.

>> **$portalRowID** is the ID of the relevant portal row. For FileMaker 7, this means the RecordID of the row in question, whereas for FIleMaker 5/6, this is the number of the portal row (e.g. the third row of the portal would be '3'.) The default value, 0 (zero),

creates a new portal row. Note that only one row may be created in this manner per query (although multiple calls to SetPortalRow() could be used to *edit* multiple records in a portal via a single FX.php action.)

>> The **$relationshipName** parameter is optioned and is intended as a method of minimizing repetition in field names. For example, instead of prefixing all of your field names with "myRelationship::", you could just pas "myRelationship" (no double colon) as the last parameter.

SetRecordID ($recordID)

This function is equivalent to using AddDBParam() per special case (1), above.

>> $recordID is the only parameter and must contain the internal FileMaker RecordID of the record to be edited or deleted.

SetModID ($modID)

This function is equivalent to using AddDBParam() per special case (2), above.

>> $modID is the only parameter and must contain the internal FileMaker ModID of the record to be edited.

SetLogicalOR ()

This function is equivalent to using AddDBParam() per special case (3), above. Since all FileMaker searches are 'AND' searches by default, the function only need be used when performing a logical 'OR' search. SetLogicalOR() does just that, and toggles the current request to use logical 'OR'. This function has no parameters.

SetFMGlobal ($globalFieldName, $globalFieldValue)

This function is FileMaker 7 specific and is equivalent to appending ".global" to a field name in order to set a global field. In order to ensure that a global is set in FIleMaker 7 (before performing a script, for example), use this function instead of AddDBParam() to set the global field.

>> The $globalFieldName parameter should contain the name of the FileMaker global field to be set.

>> The value passed via **$globalFieldValue** will be stored in the specified global field before other FileMaker actions are performed.

PerformFMScript ($scriptName)
PerformFMScriptPrefind ($scriptName)
PerformFMScriptPresort ($scriptName)

These functions are equivalent to using AddDBParam() per special case (4), above. Please see that discussion for a detailed description of how FileMaker scripts work with FX.php. In brief, PerformFMScript() causes the specified script to exit just before data is returned; PerformFMScriptPrefind() executes the specified script before executing any actions in FileMaker; and a script specified for PerformFMScriptPresort() is executed after FileMaker has assembled a data set (via a find, for example), but before the resulting records are sorted.

>> $scriptName is the only parameter for each of these functions, and must contain the name of the FileMaker script to be executed.

AddSortParam ($field, $sortOrder="", $performOrder=0)

AddSortParam() is used to determine how FileMaker will sort the found records before returning the data to FX. If functions in much the same way that Sort mode does in FileMaker Pro. This function is optional with FMFind() and FMFindAll() actions and not relevant to any of the other final functions.

By default, the order in which multiple calls to AddSortParam() appear determines sort precedence -- i.e. the results will first be sorted as specified in the first call, next by the second call, etc. The behavior can be modified by using the **$performOrder** parameter (see below.)

>> $field specifies which field you wish to sort by. (Also, see above note about determining sort order for FileMaker 7 databases.)

>> $sortOrder can contain one one of three values in FileMaker versions 5 through 6: Ascend, Descend, or Custom. Custom sorting works the same way that it does in FileMaker (i.e. it uses the value list formatted for the field specified on the current layout, as specified by SetDBData() in the case of FX.) Syntax is as follows:

```
$InstanceName->AddSortParam('FirstName', 'Descend');
```

In FileMaker 7, the possible values are "ascend", "descend", or the name of a value list. In the event that the name of a value list is passed, the returned data will be sorted based on that value list. The following code would sort a field called "departmentName" based on a value list named "department":

```
$InstanceName->AddSortParam('departmentName', 'department');
```

Regardless of data source, if AddSortParam() is called with only a **$field** parameter, an ascending sort is assumed by default.

>> The **$performOrder** parameter can be used to place multiple sort requests in a specified order. The value passed should be an integer greater than zero. For example, the following lines would cause the returned data to be sorted first by a field called "lastName", and then by the field "firstName" in ascending order:

```
$InstanceName->AddSortParam('firstName', 'ascend', 2);
$InstanceName->AddSortParam('lastName', 'ascend', 1);
```

FMSkipRecords ($skipSize)

This function is used to specify which record in the found set should be the first returned to FX. FMSkipRecords() is optional.

>> $skipSize is the only parameter taken by FMSkipRecords(). By incrementing or decrementing this value, records in the found set could be paged through in groups whose size was specified with the **$groupSize** parameter in SetDBData(). For example:

```
$skipSize = $skipSize + $groupSize;
$InstanceName->FMSkipRecords($skipSize);
```

This would increment the number of records skipped by the value stored in **$groupSize**, and then pass that parameter to FX so that **$skipSize** records will be skipped and the next **$groupSize** number of records will be returned by the current query.

FMPostQuery ($isPostQuery = true)

This optional function is used to change FX's method of accessing FileMaker from an HTTP GET to an HTTP POST. Generally, the two methods are interchangeable and FX uses POST by default, because there is a limit to how long URLs can be. In the event that you wish to use GET requests for one reason or another, FMPostQuery(false) can be called. Data should always be passed with an HTTP POST when working with large amounts of data (a few hundred characters will cause an HTTP GET problems.)

IMPORTANT: FMPostQuery() determines the method in which data is sent from your server to FileMaker Pro's web companion, ***NOT*** the method by which data is sent from a user's browser to your server (this is almost always done in the HTML <form> tag.)

>> $isPostQuery is set to true by default, so a call to FMPostQuery() to use GET

instead of POST will look like this:

$InstanceName-> FMPostQuery(false); // sets the current request to the Web Companion as a GET

FMUseCURL ($useCURL = true)

This optional function is used to specify whether FX.php accesses FileMaker Pro using cURL or sockets. FX.php has logic built in to detect cURL support, but there may be times when it may be helpful to control this explicitly. By default, cURL is used by FX. Should you want or need to use sockets instead, FMUseCURL(false) can be called.

IMPORTANT: Although cURL and sockets are generally interchangeable, if there are several calls to FileMaker from a single page using sockets may result in the truncation of your data. For this reason, it's usually best to allow FX.php to communicate with FileMaker via cURL (the default.)

>> $useCURL is set to true by default, so a call to FMUseCURL() to use sockets instead of cURL will look like this:

$InstanceName-> FMUseCURL(false); // sends the current request to the FileMaker via sockets

FlattenInnerArray ()

By default, FX.php adds an inner layer to the returned array to allow for FileMaker repeating fields and/or portals. Since most fields on a layout usually do not fall into these criteria, often code to display field data will have a trailing "[0]". FlattenInnerArray() can be used to remove this inner layer if desired, and so obviate the need for the trailing, bracketed zero. This function has no associated parameters.

Last Functions to Call

Only ***one*** of the following functions should be used for each query with FX.php. Calling one of these functions is what actually causes the interaction with FileMaker Pro. The result of the current query (if relevant) can then be stored in an array, if desired.

>> Whether data is returned by the function or not is determined by the first parameter that is passed to each of these functions: **$returnDataSet**. Setting **$returnDataSet** to

"true" will cause FX to return the data that it received from FileMaker. In instances where data is only being sent to FileMaker, **$returnDataSet** can be set to "false". (Not returning any data will increase performance somewhat.)

>> In addition to determining whether data is returned, the extent of the returned data can be determined by setting **$returnData**. This parameter can have one of three values: 'basic', 'full', or 'object'. Setting **$returnData** to 'basic' will return all data from FileMaker ***except*** the contents of the records in the current found set. Passing 'full' as a parameter will return the full set of data from FileMaker Pro. If 'object' is passed as the value of **$returnData**, ***only*** the data from the retrieved records will be returned, and the topmost level of the return array (with 'data', 'foundCount', etc. as keys) is omitted; the other metadata can be retrieved via FX.php instance variables -- all of whose names begin with 'last'.

>> The final parameter, **$useInnerArray**, determines whether the returned data should make allowance for repeating fields and/or portals. This is discussed in greater detail in connection with the FlattenInnerArray() function, above.

>> The last function in this group (it appears just before the SQL functions section of this document), DoFXAction() is a one stop shop for the actions which can be accomplished by the other actions in this group, and its use is highly recommended.

FMDelete ($returnDataSet = false, $returnData = 'basic', $useInnerArray = true)

This function deletes the record specified by an AddDBParam() call using the special '-recid' parameter documented above. Since a record is being deleted, data ***is not*** returned by default. However, should you wish to access URL or error information, simply calling FMDelete() as shown below will return basic query information (i.e. no record data.)

```
$InstanceName->AddDBParam('-recid', $currentRecord);
$InstanceName->FMDelete(true);
```

FMDup ($returnDataSet = true, $returnData = 'full', $useInnerArray = true)

FMDup is new to FX.php version 3.x and is only available for FileMaker 7 data sources. As might be expected, this function duplicates a specified record. As is the case with FMDelete(), FMDup() requires an AddDBParam() call that sets the special '-recid' parameter -- the record corresponding to the RecordID passed, will be the one duplicated. Data is returned by default.

```
$InstanceName = new FX('127.0.0.1', 591, 'FMPro7');
$InstanceName->SetDBData('Customer_List.fp5', 'web_view');
$InstanceName->AddDBParam('-recid', $selectedRecord);
```

```
$ReturnedData = $InstanceName->FMDup();                    // stores the new record in
$ReturnedData
```

FMEdit ($returnDataSet = true, $returnData = 'full', $useInnerArray = true)

The function used to update the contents of a given record. FMEdit() requires an AddDBParam() call that sets the special '-recid' parameter, as well. Data ***is*** returned by default for this function (after all, you want to make sure the correct changes were made, right?)

```
$InstanceName = new FX('127.0.0.1');
$InstanceName->SetDBData('Customer_List.fp5', 'web_view');
$InstanceName->AddDBParam('-recid', $currentRecord);
$InstanceName->AddDBParam('FirstName', $firstName);
$InstanceName->AddDBParam('LastName', $lastName);
$InstanceName->AddDBParam('Title', $title);
$InstanceName->AddDBParam('Address', $address);
$InstanceName->AddDBParam('Office', 'Austin, TX');
$InstanceName->AddDBParam('CSR', $currentRep);
$ReturnedData = $InstanceName->FMEdit();                   // stores the updated record
in $ReturnedData
```

FMFind ($returnDataSet = true, $returnData = 'full', $useInnerArray = true)

Performs a find in FileMaker Pro as specified by the parameters previously passed. By default, this function ***will*** return data -- a group of records who size was specified in SetDBData().

```
$InstanceName = new FX('127.0.0.1');
$InstanceName->SetDBData('Customer_List.fp5', 'web_view');
$InstanceName->AddDBParam('FirstName', $firstName);
$InstanceName->AddDBParam('LastName', $lastName, 'eq');
$InstanceName->AddDBParam('Office', 'Austin, TX');
$ReturnedData = $InstanceName->FMFind();                   // stores the found
set in $ReturnedData
```

FMFindAll ($returnDataSet = true, $returnData = 'full', $useInnerArray = true)

Equivalent to "Show All Records" in FileMaker Pro. Data ***is*** returned by default. Only the number of records specified by the **$groupSize** parameter of SetDBData() will be returned. However, the remainder of the records can be returned by using the FMSkipRecords() function as documented above.

```
$InstanceName = new FX('127.0.0.1');
$InstanceName->SetDBData('CDList', 'web_view', 'All');
$ReturnedData = $InstanceName->FMFindAll();                    // stores all records in
$ReturnedData
```

FMFindAny ($returnDataSet = true, $returnData = 'full', $useInnerArray = true)

Returns a random record. By default data ***is*** returned.

```
$InstanceName = new FX('127.0.0.1');
$InstanceName->SetDBData('CDList', 'web_view');
$ReturnedData = $InstanceName->FMFindAny();          // stores a random record in
$ReturnedData
```

FMFindQuery ($returnDataSet = true, $returnData = 'full', $useInnerArray = true)

Submits a search request using multiple find records and omit records requests (extending and constraining found sets). All search requests are enclosed in parentheses. When performing an AND search request, enclose all requests in a single set of parentheses separated by commas (e.g. (q1,q2,q3). Make sure there are NO spaces between the commas. OR requests are separated by a semicolon (e.g. (q1);(q2)). Prefix a request with an exclamation mark to make it an OMIT search request (e.g. !(q1)). Search request are performed in the order they are specified using the '-query' parameter.

```
// Not a complete query, FX() and SetDBData() are missing...
// Performs a search for all dogs that have a name of 'Spot' AND breed is Shiba Inu OR
Corgi
$InstanceName->AddDBParam('-query', '(q1,q2);(q3)');
$InstanceName->AddDBParam('-q1', 'PetName');
$InstanceName->AddDBParam('-q1.value', 'Spot');
$InstanceName->AddDBParam('-q2', 'Breed');
$InstanceName->AddDBParam('-q2.value', 'Shiba Inu');
$InstanceName->AddDBParam('-q3', 'Breed');
$InstanceName->AddDBParam('-q3.value', 'Corgi');
$ReturnedData = $InstanceName->FMFindQuery();    // stores the found set in
$ReturnedData
```

FMNew ($returnDataSet = true, $returnData = 'full', $useInnerArray = true)

Creates a new record in FileMaker pro with fields populated with the values specified by instances of the AddDBParam() function. By default, data ***is*** returned -- the record just created.

```
// Not a complete query, FX() and SetDBData() are missing...
$InstanceName->AddDBParam('FirstName', $firstName);
$InstanceName->AddDBParam('LastName', $lastName);
$InstanceName->AddDBParam('Office', 'Austin, TX');
$ReturnedData = $InstanceName->FMNew();                    // stores the new record
in $ReturnedData
```

FMView ($returnDataSet = true, $returnData = 'full', $useInnerArray = true)

This function is somewhat different from the others in this group in that it returns information about the layout, not a certain set of records. Data is returned by default, and I can see little use for this function if **$returnDataSet** is set to 'false'. If there are fields that are formatted as drop-down menus ***on the current layout*** (as specified by SetDBData()), the following example will display a drop-down on the resulting web page for each one on the specified layout in the database:

```
$InstanceName = new FX('127.0.0.1');
$InstanceName->SetDBData('CDList', 'web_view');
$ReturnedData = $InstanceName->FMView();        // stores layout information in
$ReturnedData
foreach ($ReturnedData['valueLists'] as $key => $value) {
    foreach ($value as $key1 => $value1) {
        $valuelistData[$key] .= "\t<option value=\"$value1\">$value1</option>\n";
    }
    echo "<select name=\"$key\">\n";
    echo $valuelistData[$key];
    echo "</select>\n";
    echo "<br />\n";
}
```

FMDBNames ($returnDataSet = true, $returnData = 'full', $useInnerArray = true)

FMDBNames() is similar to FMView() in that it returns information about the FileMaker environment on the current server, rather than a group of records. Unlike every other action function, the SetDBData() function ***should not be called*** in conjunction with this function. When called, FMDBNames() returns the names of all databases that are open and shared via the Web Companion on the target machine. Data is returned by default. As above, setting **$returnDataSet** to false would be of little practical use. A query to display the currently active databases available via web companion might look like this:

```
$serverIPAddress = '127.0.0.1';
$InstanceName = new FX($serverIPAddress);
$ReturnedData = $InstanceName->FMDBNames();        // stores layout information in
$ReturnedData
```

```
echo "<b>Databases Open on $serverIPAddress </b><br />\n";
foreach ($ReturnedData['data'] as $value) {
    echo $value['DATABASE_NAME'][0] . "<br />\n";
}
```

FMLayoutNames ($returnDataSet = true, $returnData = 'full', $useInnerArray = true)

Like the two preceding functions, FMLayoutNames() returns information about the FileMaker environment on the current server, rather than a group of records. Like them, data is returned by default. And like them, setting **$returnDataSet** to false would be of little practical use. However, unlike every other action function, the SetDBData() function should be called, ***but no layout should be specified***. FMLayoutNames() returns the names of all available layouts in the database specified by the SetDBData() function. A query to display the available layouts in a database might look like this:

```
$serverIPAddress = '127.0.0.1';
$currentDatabase = 'My_Database.fm5';
$InstanceName = new FX($serverIPAddress);
$InstanceName->SetDBData($currentDatabase);
$ReturnedData = $InstanceName->FMLayoutNames();
echo "<b>Layouts in $currentDatabase </b><br />\n";
foreach ($ReturnedData['data'] as $value) {
    echo $value['LAYOUT_NAME'][0] . "<br />\n";
}
```

FMScriptNames ($returnDataSet = true, $returnData = 'full', $useInnerArray = true)

FMScriptNames() is another function which returns information about a database, rather than a record set. As is the case with the other informational functions, data is returned by default, and setting the function's **$returnDataSet** parameter to false would be of little practical use. This function is like FMLayoutNames() in that **no layout is specified**. Rather, the function returns the names of all available scripts for the specified database. An example query to display the available scripts in a database follows:

```
$serverIPAddress = '127.0.0.1';
$currentDatabase = 'My_Database.fp7';   // FMScriptNames() is only available in FM7
$InstanceName = new FX($serverIPAddress, 80, 'FMPro7');
$InstanceName->SetDBData($currentDatabase);
$ReturnedData = $InstanceName->FMScriptNames();
echo "<b>Scripts in $currentDatabase </b><br />\n";
foreach ($ReturnedData['data'] as $value) {
    echo $value['SCRIPT_NAME'][0] . "<br />\n";
}
```

DoFXAction
($currentAction,$returnDataSet=true,$useInnerArray=false,$returnType='object')

DoFXAction() is intended to streamline FX.php queries for advanced users. It works much the same as the other functions in this group, with a couple of exceptions. First, note that this function adds an additional, required function: **$currentAction**. Also, the default values of **$useInnerArray** and **$returnType** are different than its group-mates.

>> $currentAction is a required parameter for DoFXAction() which specifies which type of action (find, edit, delete, etc.) is to be performed with the current query. In the latest versions, this is simply a string which may have one of the following values: 'delete', 'duplicate', 'update', 'perform_find', 'show_all', 'show_any', 'new', 'view_layout_objects', 'view_database_names', 'view_layout_names', or 'view_script_names'.

Output Control

Until version 4.*, FX.php has 3 different return types, "full", "basic" and "object". From version 5.0, some more types are added. In this section, they are explained.

FlattenInnerArray()

DoFXAction or FM* methods have the **$useInnerArray** parameter and it's default value is false. If you call this **FlattenInnerArray** method, it affects the same as you set false to the **$useInnerArray** parameter.

RemainAsArray ($rArray1,$rArray2=NULL,...,$rArray12=NULL)

This method let FX.php to output new type result. Single value fields are not array, just one value. Multiple value fields such as inside of portal are array alone with one portal record is gathering one array. You should set the **$useInnerArray** parameter to false to work this method. So you should call FlattenInnerArray() if you want to omit some parameters.

This is example. The database has 5 tables and they are associated as below:

Table: person [**TO**: person_to] [**Layout**: person_layout]
[Relation: person_to.id = contact_to.person_id]
[Relation: person_to.id = history_to.person_id]

Table: contact [**TO**: contact_to] [Portal of layout "person_layout"]
[Relation: contact_to.person_id = person_to.id]
[Relation: contact_to.kind = contact_kind_person.id]

[Relation: contact_to.way = contact_way_person.id]
Table: contact_kind [**TO**: contact_kind_person]
[Relation: contact_kind_person.id = contact_to.kind]
Table: contact_way [**TO**: contact_way_person]
[Relation: contact_way_person.id = contact_to.way]

Table: history [**TO**: history_to] [Portal of layout "person_layout"]
[Relation: history_to.person_id = person_to.id]

The top level TO "person_to" is assigned the layout "person_layout". The portal coming from TO "history_to" has some fields but they belong the table "history". In this case, you should specify the portal's TO name to single parameter of RemainAsArray method. The other portal of TO "contact_to" has fields coming from different TOs. These are "contact_to", "contact_kind_person" and "contact_kind_way". In that case, you should put into one array these TO names as like array('contact_to','contact_way_person','contact_kind_person'), and specify as a parameter of RemainAsArray method.

Here is the sample to get data from above layout. Please pay attention to the RemainAsArray method. It has two parameters and later one is array. The array parameter of the method could have multiple elements. The first elements should be the TOC name (the portal name). The later elements will be included to the first element in the result. In other words, array parameter in RemainAsArray method will be grouped.

```
require_once( 'FX/FX.php' );
$fx = new FX('127.0.0.1', '80', 'FMPro7', 'HTTP');
$fx->setCharacterEncoding( 'UTF-8' );
$fx->setDBUserPass( 'web', 'password' );
$fx->setDBData( 'TestDB', 'person_layout', 100 );
$fx->FlattenInnerArray();// Same as setting false to3rd parameter of DoFxAction
$fx->RemainAsArray( 'history_to',
array('contact_to','contact_way_person','contact_kind_person') );
$result = $fx->DoFxAction( FX_ACTION_FIND );
echo '<pre>';
var_export($result);
echo '</pre>';
```

You can get the following result. You don't require [0] element accessing for single field, and could access as an array for each portal record.

```
  array (
    '1.69' =>
    array (             // The first record is staring here. It is relevant to the record on
layout.
      '-recid' => '1',          // -recid and -modid fields are existing in array for one record.
      '-modid' => '69',
```

```
        'id' => '1',
        'name' => 'Masayuki Nii',     // These are real fields.
        'address' => 'Ginza1',
        'mail' => 'msyk@msyk.net',
        'contact_to' =>           // One portal is starting here. The key is the portal name.
        array (
          0 =>
          array (         // The portal's first row. This row contains several TOs.
            'contact_to::id' => '181',                // These field's TO is the same as portal
name,
            'contact_to::person_id' => '1',  // Each field name starts with TO name.
            'contact_to::description' => '',
            'contact_to::datetime' => '',
            'contact_to::summary' => 'quit',
            'contact_to::important' => '',
            'contact_kind_person::name' => '',
            'contact_way_person::name' => 'Others',     // Another TO in this portal.
          ),
          1 =>
          array ( … ),      // Second portal record.
          2 =>
          array ( … ),
        ),
        'history_to' =>         // Another portal is starting here.
        array (
          0 =>
          array (
            'history_to::id' => '3',
            'history_to::person_id' => '1',
            'history_to::startdate' => '01/03/2002',
            'history_to::enddate' => '01/31/2003',
            'history_to::description' => 'company3',
          ),
          1 =>
          array ( ... ),
          2 =>
          array ( …   ),
        ),
      ),
      '20.26' =>
      array ( …. ),
      ….
```

Support for The Surogate-pair Character

FileMaker Server 11 has the bug for surrogate-pair character(the code is over &hFFFF).

It has FMS 10 also, and I guess all ancient products have it. If any surrogate-pair character is including in the database, FX.php reported error and could get nothing. The XML parser in FX.php must stop with illegal character. FX.php ver.5 prevents the surrogate-pair bug both reading and writing.

On reading from FMS, FX.php detects wrong encoded surogate-pair characters in UTF-8 mode, and correct it to right encoding. If FMS fixes it, FX.php will work fine.

On writing to FMS, FX.php proceeds kind of tricky way. If the string contains surrogate-pair characters, FX.php adds the same number SPACEs character after the string. Although it doesn't have any side effect on FMS 11, the feature FMS might be fixed the bug. In that case, you can cancel the patch part of surrogate-pair writing fixing to describe below:

```
define( 'SURROGATE_INPUT_PATCH_DISABLED', true);
```

ObjectiveFX

This class is experimental on version 5.0.

You could access the field data as like a property of record object. It's just storing the result of FX.php and any property access is tried to be a field data possibly. If the layout includes any portals, it requires to call RemainAsArray method before accessing FileMaker Server.

The sample program with comments is here.

```
require_once( 'FX/FX.php' );
require_once( 'FX/ObjectiveFX.php' );     // Including class definitions
$fx = new FX('127.0.0.1', '80', 'FMPro7', 'HTTP');
$fx->setCharacterEncoding( 'UTF-8' );
$fx->setDBUserPass( 'web', 'password' );
$fx->setDBData( 'TestDB', 'person_layout', 100 );
$fx->FlattenInnerArray();      // Require FlattenInnerArray and RemainAsArray
$fx->RemainAsArray( 'history_to',
array('contact_to','contact_way_person','contact_kind_person') );
$result = $fx->FMFind( );

$records = $result['object']->getRecords();
                                // FX.php supports 'object' key in its result array.
foreach($records as $oneRecord) {
    echo "<p>{$oneRecord->name} [{$oneRecord->mail}]</p><ul>";
                                                                    // name and
mail are field names.
    $portal1 = $oneRecord->contact_to;     // Portal name can access as an array.
```

```
    foreach($portal1 as $portalRecord) {     // Repeating for portal.
        echo "<li>{$portalRecord->contact_way_person__name} - ",
            "{$portalRecord->contact_to__summary} ",
            "[{$portalRecord->contact_to__datetime}]</li>";
    }
    echo"</ul>";
    try {
        echo $oneRecord->no_such_a_field;
    } catch( Exception $ex ) {
        echo "<p>Get Exception=", $ex->getMessage(), '</p>';
    }
}
```

The result is here. Each field in the result will be shown, and non-existing field will throw an exception.

```
    Masayuki Nii [msyk@msyk.net]          // name and email fields are shown.

     - Indirect - Called [02/13/2012 15:00:00]          // Data in portal
     - Indirect - Get email [01/13/2012 15:00:00]
     - Others - See on Web [12/13/2011 15:00:00]

    Get Exception=The field name 'no_such_a_field' that you specified doesn't exist.
            :
    (Repeating alone with each record.)
```

At first, the source "ObjectiveFX.php" should be included. You should call FX.php's query command. The method RemainAsArray is required. The result will include the index 'object' so it's another requirements. You can get the array of the record from the getRecords method. Each element is kind of a RECORD. You can get the field value by using RECORD->FIELD_NAME style referencing. If you specify the field name that doesn't exist in the layout, you will get exception.

As for the portal, you can get the value with the same way. The portal name property returns the array of records. Each record has the field value as the property. But you should change the field name because some letters doesn't support as the property name on PHP. Currently, I put the code to replace from :: to __. If your portal name (TO name) is contact_to::summary, it should be written "contact_to__summary". I think it should replace for another kind of letters too, I will include in a feature.

I guess you worry about performance issue of ObjectiveFX. It's very little as far as appropriate size of data. More over you don't need to fear. If you don't include "ObjectiveFX.php" file, the data of key 'object' doesn't exist.

SQL Functions

The following functions are only used when using FX.php to connect to a SQL data source.

PerformSQLQuery ($SQLQuery, $returnDataSet = true, $useInnerArray = false, $returnType = 'object')

FX.php is capable of creating very basic SQL statements from the parameters passed via AddDBParam(), AddSortParam(), etc. However, users who need to retrieve data from SQL data sources for whatever reason may find this limiting. To this end, PerformSQLQuery() allows an SQL query to be specified. The other parameters ($returnDataSet, $useInnerArray, and $returnType are discussed in detail above.)

>> $SQLQuery should contain the SQL statement to be executed. Documentation of SQL is beyond the scope of this document, and varies somewhat from one data-source to another.

SetDataKey ($keyField, $modifyField = '', $separator = '.')

In FileMaker, each record has a built-in unique record identifier: RecordID; as well as a built-in means of tracking modification order: ModID. Although many SQL databases have columns which perform the same functions, the name of these columns will likely vary from one database to another. SetDataKey() is intended to aid those who for whatever reason are moving an FX.php/FileMaker solution to an FX.php/SQL solution achieve feature parity. (The key used for each element in the data array from a FileMaker query is the RecordID, a period, and then the ModID.)

>> The **$keyField** parameter should contain the name of the field whose contents should be used as they keys of the data array for the current query.

>> $modifyField is is an optional parameter which can be used to further identify each returned row. The contents of this parameter should be the name of the field in the SQL database whose contents should be used for this purpose. In the case of ModID, the FileMaker equivalent, a common use would be to insure that a user did not overwrite a modification made by another active user.

>> The final parameter, **$separator**, contains the character that should be used to separate the data from the **$keyField** and the **$modifyField**. The default character is a period, as is used for the data returned from FileMaker queries.

SetSelectColumns ($columnList)

This function is designed to aid those who prefer to let FX.php create most portions of

their SQL queries for them, but need only specific columns of data returned. As its name implies, SetSelectColumns() is ***only*** relevant when performing an SQL SELECT statement. FX.php will use the list of columns specified to determine the returned columns, rather than returning all columns.

>> The **$columnList** parameter should contain a comma separated list of columns which should be returned by the current query. For example, your code might contain lines like the following:

```
$myColumns = "first_name, last_name, address1, address2, city, state, zip,
employee_id";
$InstanceName->SetSelectColumns($myColumns);
```

SQLFuzzyKeyLogicOn ($logicSwitch = false)

The SetDataKey() function, above, is used to specify which field's content should be used as the key for the returned data array. SQLFuzzyKeyLogicOn() tells FX.php to make it's best guess as to the field to use.

>> The only parameter of SQLFuzzyKeyLogicOn, **$logicSwitch,** is optional. If set to 'true', FX.php will attempt to determine a field whose contents would best work as keys for the returned data array (which has one element per row/record.)

UseGenericKeys ($genericKeys = true)

By default, FX.php uses the keys of the records returned as the indices for its returned data array. While this parallels FileMaker behavior, this may not be desirable in some cases. When UseGenericKeys() is called with $genericKeys equal to true, FX.php will return its data set with integer indices (i.e. 0, 1, 2, etc.)

>> The lone parameter, **$genericKeys**, should be set to true in order to use integer keys, and false to use records' keys as indices.

Additional Reading

For more information about FX.php (including the latest version of this document):

http://www.iviking.org/

For information about getting started with PHP and FX.php you might try one of these books:

Advanced FileMaker Pro 6 Web Development
http://www.amazon.com/exec/obidos/ASIN/1556228600/ivikingorg-20

FX.php for FileMaker
http://www.fmwebschool.com/php.htm

Open Source Technology Expanding FileMaker
http://www.fmwebschool.com/opensource.htm

For more information on FileMaker 7, Custom Web Publishing, and XML:

http://www.filemaker.com/downloads/pdf/fmsa7_custom_web_guide.pdf

For more information on FileMaker 5/6 and XML:

http://www.filemaker.com/downloads/pdf/xml_Chapter_7.pdf

For more information on the parameters accepted by the FileMaker 5/6 Web Companion:

http://www.filemaker.com/downloads/pdf/xml_Appendix_B.pdf

For additional information on the error codes returned the FileMaker 5/6 Web Companion:

http://www.filemaker.com/downloads/pdf/xml_Appendix_C.pdf

For additional information about PHP, see the online manual at:

http://www.php.net/manual/en/

or look for books on PHP or FileMaker at your local bookseller.