FileMaker for PHP Developers

by Jonathan Stark

FileMaker is a popular and powerful desktop database application toolkit. FileMaker, Inc. recently released a beta version of the FileMaker API for PHP, which allows PHP to more easily talk to the FileMaker Server Advanced product. Last month, author Jonathan Stark introduced some of the concepts behind the newly hatched API. In the concluding episode of this two-part series, he explains how FileMaker makes editing your database records a snap.

FileMaker is a workgroup productivity toolkit that was designed to allow knowledge workers to quickly and easily construct data management systems for themselves.

In the first part of the FileMaker for PHP Developers series, I introduced you to the basics of FileMaker development in the desktop environment, and explained how to leverage that development work to easily display your data on the Web using the FileMaker API for PHP. I covered the terminology used by the FileMaker API, introduced the list view and form view layouts, and explained how having the business logic embedded in the layout can be a surprisingly efficient approach in real-world applications. We then went on to look at views in more detail, and ended with a brief exploration of where (and why) FileMaker might be deployed to the greatest effect.

With much of the FileMaker desktop development basics behind us, we can focus more on the PHP side of things. This time around, you will learn two different techniques for updating your database records.

Updating a Single Record

Loyal php|architect readers will recall the view_products.php script, which was the first of the code listings from Part One of this series. If you recall, we used this script to display a searchable and sortable list of products from the ProductCatalog database, which is included with the FileMaker API for PHP download bundle in the form of ProductCatalog.fp7. To make the edit functions accessible from the demo scripts available to you, I have simply changed the view link beside each product listed on that page to an edit link by altering the line:
This new link will navigate to a form view of the clicked product. However, before we can look at the PHP code used to update the selected product record, we really need to take a peek at the corresponding layout in FileMaker Pro.

Figure 1 shows the FileMaker layout in Browse mode, which is also known as data entry mode. Notice that there are radio buttons applied to the Category field here, and there is also a drop-down list attached to the Manufacturer field. It’s obvious that editing these value lists will be a trivial task, even for novice FileMaker Pro users. The end user need only click the Edit... link at the bottom of the list to be presented with the Edit Value List dialog shown in Figure 2. You will see in a moment that this is very cool, because this simple action on the part of the user will trickle through to the Web without any changes being made to the PHP code.

The code that makes up edit_product.php is reproduced in full in Listing 1. For the sake of clarity, I have left out large chunks of form validation and the sanitization of user input. These are important concerns and relevant here, but a discussion of general form submission handling is outside the scope of this article. It is also notable that I left out much of the FileMaker error checking because it is very repetitive and does not serve to illustrate my point.

You will notice that the script is divided into four distinct sections. The first of these, initialization, opens with the definitions used for database connection. For reasons of security, any database credentials should be stored in a separate configuration file above the document root and included from there. Other items in the script initialization section include the require_once() call representing the FileMaker dependency, global variable initialization, and some code to check the status of the $_POST array to determine whether the form has yet to be processed. If the process_form element has been set, the result of the form processing will be displayed in the browser above the empty form; otherwise, the empty form alone will be displayed.

The second section is all about form display. It contains the function show_form(), which takes all its cues from the specified FileMaker layout. The fields that have value lists applied in FileMaker will be formatted appropriately in HTML, depending on the style type associated with the underlying field object. Note that everything here is completely dynamic, so that changes made to the
FileMaker layout or to the values lists on that layout will be reflected in the HTML page without any modification of the PHP code.

Thirdly, there is the form processing, which takes place, unsurprisingly, within the process_form() function. As with the show_form() function, process_form() bases all its logic on the FileMaker layout named at the beginning of the function; in this case, the chosen layout is edit_product. When the time comes for the record to be updated, PHP queries the layout object for the fields it contains, using $layout->getFields(). It then loops through the array of fields and matches the field names with those in the $_REQUEST superglobal array. On finding a match, it pulls the corresponding data out of the $_REQUEST array and updates the field value. Finally, it submits the change to the database. It is important you should be aware that there is a lot of validation missing from this area in particular, as mentioned earlier; a database should never be updated with raw user input in any real-life application.

With that out of the way, the final section of the script is dedicated to HTML rendering. Since this is a demo script, I chose to have the CSS style definitions inline rather than force an unnecessary listing upon you. Apart from that and the title, all we have here is a back link to view_products.php and the HTML content generated by show_form() and process_form(), if applicable.

**Updating a Group of Records**

Technically, it would be possible to update a group of records by simply expanding on the “single record update” technique, feeding the script an array of record IDs in a do..while loop. However, this would be less than optimal from the performance perspective, since a) it would require a call to the server for every single record and b) the data is transmitted as XML. A better option would
be to use PHP to call a FileMaker script that will do all the dirty work for you; and that's precisely why there are FileMaker scripts.

**FileMaker Scripts**

FileMaker Pro has a point-and-click scripting environment called *ScriptMaker*. This ScriptMaker allows you to create macros that can execute all sorts of useful commands with a great deal of ease. Normally, scripts are run by FileMaker Pro users, but they can be triggered by PHP as well. The coolest part is that you can send parameters to a FileMaker script via PHP, thereby customizing the behavior of that script on the fly. In this example, I am going to create a PHP page that will allow the user to select a Manufacturer, enter a Status and submit the form. The form will send the Manufacturer Name and Status to the Update Status script in FileMaker, passing all the data elements as arguments. The FileMaker script will then locate any Product records with a matching Manufacturer, and update the Status value accordingly.

Figure 3 is an illustration of the ScriptMaker environment in FileMaker Pro. Hopefully you can see from the image that it's quite simple to use. The area on the left contains a list of the available commands, and you can double click on any of these to move them into the text area on the right, which displays the script itself. Not all the commands that are made available in ScriptMaker are compatible with PHP, so I have activated the *Indicate web compatibility* checkbox; those of the command options that can't be used are grayed out as a result.

Figure 4 is the Update Status script itself. As you can see, it is very short, and in fact it only took about three minutes to write. It would have taken me much longer to write it in PHP and, as I mentioned earlier, the performance obtained in this way would have been less than wonderful.

Let's break down that Update Status script and see what it's made of.

---

**FIGURE 4**

Edit Script

View: all by category

Script Name: Update Status

- # Grab the incoming script parameter and break it into two values
  - Set Variable [$Manufacturer; Value:GetValue ( Get (ScriptParameter) ; 1 )]
  - Set Variable [$Status; Value:GetValue ( Get (ScriptParameter) ; 2 )]
- #
- # Find the records in question, if any
  - Enter Find Mode []
  - Set Field [Product:Manufacturer; $Manufacturer]
  - Perform Find []
- #
- # If we found any records, run a replace on them
  - If [Get (FoundCount )]
  - Replace Field Contents [No dialog; Product::Status; $Status]
  - Commit Records/Requests [No dialog]
- End If

Script Step Options

Move  Duplicate  Disable  Copy  Paste

- Indicate web compatibility
- Run script with full access privileges

Cancel  OK

---
Update Status
The first section in the Update Status script accepts the incoming script parameter, breaks it into two values, and stores the values in separate variables:

```
Set Variable [ $Manufacturer;
    Value:GetValue(Get(ScriptParameter); 1) ]
```

```
Set Variable [ $Status;
    Value:GetValue(Get(ScriptParameter); 2) ]
```

Technically speaking, a FileMaker script can only accept one parameter, and you should access that parameter value with the `Get(ScriptParameter)` function. You can get around the single parameter limitation, as shown here, by delimiting your values with returns and using the `GetValue()` function. `GetValue()` accepts an EOL-delimited list of values and a value number as parameters, and will return the value indicated by the number. If you think of the EOL-delimited list as an array, then `GetValue($Values; 2)` is equivalent to `$Values['2']` in PHP.

Now that we have the number of arguments we need to pass, the next thing is to find the `Product` records that are associated with the selected `Manufacturer` name. We do this by entering `Find` mode, inserting the selected `Manufacturer` name into the `Manufacturer` field, and performing the `Find` request. While we're there, notice that the `Product::` prefix in the `Set Field` step indicates that the `Manufacturer` field belongs to the `Product` table.

```
Enter Find Mode
Set Field [ Product::Manufacturer;
    $Manufacturer
]
Perform Find
```

At this point, we need to check to see whether our request matched any records. To do so, we open an `If` block and make our enquiry using the function `Get(FoundCount)`, which will return an integer. If the integer it returns happens to be 0, the `If` condition will evaluate to `FALSE` and the rest of the script will be skipped. If, however, the number of items is greater than 0, the `If` condition will evaluate to `TRUE`. This will trigger the execution of the `Replace` and `Commit Records/Requests` commands.

```
If [ Get(FoundCount) ]
    Replace Field Contents [ Product::$Status;
        Replace with calculation: $Status
    ]
End If
```

The call to `Replace` does just as you might expect—name-

**update_status.php**
With the FileMaker script in place, we can turn our attention to the PHP page that will call it: `update_status.php`, rendered here as Listing 2. As with the earlier code listing, I have left out much in the way of form validation and the sanitization of user input, so please tread with care when it comes to implementing this functionality yourself. There are five distinct sections in `update_status.php`, some of which match the sections in `edit_product.php` (Listing 1) and some of which are unique to this script. Thus, as before, we have the `initialization` stage making the decision about the nature of the HTML page content, depending on the stage of processing the script has reached. We meet, once again, the `form display` section containing the `show_form()` function, where the options for the `select` block in the `Manufacturer` field are pulled from the layout in FileMaker. Next up, there's something you haven't seen until now; `form validation`. In this instance, this is restricted to checking that the `Manufacturer` and `Status` fields contain some input, and ensuring that `$_POST['manufacturer']` doesn't contain an illegal hyphen or `$_POST['status']` any HTML tags. Again, this offers very little protection, and you will need tighter control over your user input data in any real-life application.

Next up is our old friend `form processing`, which is the home of the `process_form()` function. In this particular case, `process_form()` is the focus of the example, as it shows you how to go about sending a form submission to a FileMaker script. As an added bonus, you can see the syntax for sending multiple parameters in the line:

```
$script_param = "$_POST['manufacturer']."\n".$_POST['status'];
```

Remember, though, that "\n" is not valid syntax on every platform. Those of you who are running PHP 5.0.2 or newer will be able to use the built-in constant `PHP_EOL` here, but if you’re stuck with an older version of PHP you will need to create your own EOL constant to achieve portability.

Finally, there is the HTML template, which once again contains inline CSS style definitions, a title, and the absolute basic necessities to frame and decorate this dynamically rendered page.
```php
<?php

function showForm() { 
    // grab the record id sent in the url from list page or a post from this page
    $recid = (array_key_exists('recid', $_REQUEST)) ? htmlspecialchars($_REQUEST['recid']) : '';
    $htmlspecialchars($REQUEST['recid']) : ' ';

    # set the layout name for this page
    $layout_name = 'edit-product';

    # initialize our output var
    $html = ' ';

    # instantiate a new FileMaker object
    $fm = new FileMaker($fm_FILE, $fm_HOST, $fm_USER, $fm_PASS);

    # get the record by its id
    $record = $fm-getRecordById($layout_name, $recid);

    # get the fields from the layout as an array of objects
    $field_objects = $fm-getLayoutByField();

    # start compiling our form inputs
    $html = '<form action="' . $fm_HOST . '" method="post">;
    $html = '<input type="hidden" name="process_form" value="true" />

    $html = '<input type="hidden" name="recid" value="{$recid}" />

    $html = "<input type="radio" name="{$field_name_underscore}" value="{$value}" checked="checked" />
    "$value"<br />

    # close the open tags
    $html = "</td>
    $html = "</select>
    break;

    case 'CHECKBOX':
    # start compiling html for this checkbox set
    $html = "<select name="{$field_name_underscore}" value="{$value}" checked="checked" />
    $html = "<option>{$values}="{$value}"</option>
    $html = "</td>
    break;

    case 'RadioButtons':
    # start compiling html for this checkbox set
    $html = "<tr><td><input type="radio" name="{$field_name_underscore}" value="{$value}" checked="checked" />
    $html = "</td>
    break;

    default:
    # the remaining field style types (EDITTEXT and CALENDAR) are best represented as text inputs
    $html = "<input type="text" name="{$field_name_underscore}" value="{$value}"/>
    $html = "</td>
    break;
}
```

FILEMaker for PHP Developers: Part 2

# For security reasons, these lines should either be included from a config file above the document root, or possibly captured during a login and stored in the $SESSION superglobal array

# output the form control appropriate to the field style type
switch ($field_style_type) {
    # POPULIST
    # start compiling html for this select control
    $html = "$option[Selected]="$option" />
    foreach($values as $value) {
        $checked = ($field_value == $value) ? 'checked' : ' ';
        $html = "<input type="checkbox" name="{$field_name_underscore}" value="{$value}" {$checked} />
        $html = "</td>
        break;
    }
```

LISTING 1: Continued...

```
LISTING 2

```php
<?php
# For security reasons, these lines should either be included from a config file above the document root, or possibly captured during a login and stored in the SESSION superglobal array
define("FM_HOST", '127.0.0.1');
define("FM_FILE", "ProductCatalog.fp7");
define("FM_USER", 'admin');
define("FM_PASS", 'filemaker');

# include the FileMaker API for PHP
require_once ('FileMaker.php');

# handler for showing, validating, and processing the form
if (array_key_exists('process_form', $_POST)) {
  if (isset($script_result)) {
    return ('<p>'.$_POST['post_message'].'</p>');</
  } else {
    $script_result = array();
    if (isset($script_param)) {
      return ('<p>'.$_POST['get_message'].'</p>');</
    } else {
      return ('<p>'.$_POST['post_message'].'</p>');</
  }
} else {
  return ($_POST['post_message']);
}

# instantiate a new FileMaker object
$fm = new FileMaker(FM_FILE, FM_HOST, FM_USER, FM_PASS);  

# set a couple variables
$script_name = 'update_status';
$script_param = 'status';

# call the script with the parameter
$script_object = $fm->newPerformScriptCommand($script_param, $script_name, $script_param);

# run the script
$script_result = $script_object->execute();

# check for errors
if ($script_result->isError($script_param)) {
  return ('<p>'.$script_result->getMessage().' ('.error.'.$script_result->getCode().''));</
} else {
  return ($_POST['post_message']);
}

# create the html manufacturer options
$script_result = $script->get($script_param, array('status' => $script_name));
$script_result = $script_result->getScriptResult();
$script_result = $script_result->getErrors();
$script_result = $script_result->getOutput();

# create the html manufacturer options
$script_result = $script_result->getScriptResult();
$script_result = $script_result->getErrors();
$script_result = $script_result->getOutput();

# compile errors as html.
$error_list = array();
foreach ($script->getErrors() as $error) {
  $error_list[] = '<li>'.$error.'</li>';  
}

# insert the errors and manufacturer options into a form
$html = <<<HTML
<form action="[$_SERVER["PHP_SELF"]]" method="post">

  <select name="manufacturer">
    <option>-</option>
    <option value="1">Manufacturer 1</option>
    <option value="2">Manufacturer 2</option>
  </select>

  <input type="text" name="status" value="" />

  <input type="submit" name="submit" value="Continue" />

</form>

</html>
```

LISTING 2: Continued...

```php

# For security reasons, these lines should either be included from a config file above the document root, or possibly captured during a login and stored in the SESSION superglobal array
define("FM_HOST", '127.0.0.1');
define("FM_FILE", "ProductCatalog.fp7");
define("FM_USER", 'admin');
define("FM_PASS", 'filemaker');

# include the FileMaker API for PHP
require_once ('FileMaker.php');

# handler for showing, validating, and processing the form
if (array_key_exists('process_form', $_POST)) {
  if (isset($script_result)) {
    return ('<p>'.$_POST['post_message'].'</p>');</
  } else {
    $script_result = array();
    if (isset($script_param)) {
      return ('<p>'.$_POST['get_message'].'</p>');</
    } else {
      return ('<p>'.$_POST['post_message'].'</p>');</
  }
} else {
  return ($_POST['post_message']);
}

# instantiate a new FileMaker object
$fm = new FileMaker(FM_FILE, FM_HOST, FM_USER, FM_PASS);  

# set a couple variables
$script_name = 'update_status';
$script_param = 'status';

# call the script with the parameter
$script_object = $fm->newPerformScriptCommand($script_param, $script_name, $script_param);

# run the script
$script_result = $script_object->execute();

# check for errors
if ($script_result->isError($script_param)) {
  return ('<p>'.$script_result->getMessage().' ('.error.'.$script_result->getCode().''));</
} else {
  return ($_POST['post_message']);
}

# create the html manufacturer options
$script_result = $script_result->getScriptResult();
$script_result = $script_result->getErrors();
$script_result = $script_result->getOutput();

# compile errors as html.
$error_list = array();
foreach ($script->getErrors() as $error) {
  $error_list[] = '<li>'.$error.'</li>';  
}

# insert the errors and manufacturer options into a form
$html = <<<HTML
<form action="[$_SERVER["PHP_SELF"]]" method="post">

  <select name="manufacturer">
    <option>-</option>
    <option value="1">Manufacturer 1</option>
    <option value="2">Manufacturer 2</option>
  </select>

  <input type="text" name="status" value="" />

  <input type="submit" name="submit" value="Continue" />

</form>

</html>
```
Conclusion

I hope that this article has given you a taste for the rapid application development that is possible with FileMaker Pro, FileMaker Server Advanced, and the FileMaker API for PHP. No, FileMaker is never going to be an Oracle killer; but I can’t tell you the number of times I have seen a “temporary” FileMaker solution bridge the gap for someone who was waiting for a SQL solution that ultimately never materialized. If you would like to look at the API code, currently at public beta status, you can download the FileMaker for a SQL solution that ultimately never materialized.

If you would like to play around with this code, you will need a copy of FileMaker Pro, and you will also need FileMaker Server Advanced. Neither are available for free, but you can get limited versions of each by joining the FileMaker Solutions Alliance (FSA). There is an annual fee for FSA membership, but the amount of free software offered to members would more than offset the membership fee. Please visit http://www.filemaker.com/developers/join_fsa.html for more information about joining the FSA.

LISTING 1: Continued...

```php
Listing 1:

```