

Publishing FileMaker Container Data With the FileMaker API for PHP

by Tim Dietrich

Last month I discussed a couple of techniques that I use to optimize Web applications built using the FileMaker API for PHP. This month, I'll discuss a similar topic: Publishing data stored in container fields. I'll show you the standard way to publish images that are stored in container fields, and then show you a more optimal way to do so.

This article assumes that you have a working knowledge of FileMaker container fields and the various ways that you can insert data into them. It also makes the assumption that data has actually been embedded into the container fields, as opposed to their contents being stored as references.

In addition, I'm making the assumption that you do not want the FileMaker database users to have to upload images to a Web server. Instead, the goal is to maintain the images in FileMaker itself, and therefore, simply storing URLs is not an option.

And finally, this article contains a lot of code. I've done my best to comment everything so that you can follow along, and even if you don't understand all of the PHP code, I believe that at the very least you'll get a good understanding of what's going on.

Publishing Images - The Standard Method

Publishing images that are stored in FileMaker container fields is a multi-step process. First, you obtain a link to the container's data, and you do so using the getField method. For example, suppose we have a FileMaker record object named \$employee with a container field named Photo. Echoing the value returned by the getField method will return a URL similar to this:

```
/fmi/xml/cnt/data.jpg?-db=TESTDB&-lay=PHP%20-%20Data&-recid=234&-field=Photo(1)
```

Note that the container URL is automatically generated by FileMaker, and can be obtained very easily using the getField method.

This URL is helpful, but for several reasons (chief among them being security) you cannot use them as the source attribute of an HTML image tag. Instead, you need a second PHP template that takes the container URL, gets the data in the container from the database, and returns the data to the browser. Here is a PHP template (fm_container_v1.php) that does just that...

```

<?php

// If no container_url was specified...
if (!isset($_GET['container_url'])) {
    die ("No container URL was specified.");
}

// Include the FileMaker API for PHP.
require_once("FileMaker.php");

// Database connection information.
include ("../protected_data/database_settings.php");

// Create a connection to the database.
$fm = new FileMaker ();
$fm->setProperty('hostspec', FM_HOSTSPEC);
$fm->setProperty('database', FM_FILE);
$fm->setProperty('username', FM_USERNAME);
$fm->setProperty('password', FM_PASSWORD);

// Get the data in the container.
$container_data = $fm->getContainerData($_GET['container_url']);

// If FileMaker encountered an error...
if (FileMaker::isError($container_data)) {
    echo "Error Code: " . $container_data->getCode() . "<br>";
    echo "Error Message: " . $container_data->getMessage() . "<br>";
    die;
}

// Return the container data.
echo $container_data;

?>

```

The code includes plenty of comments, so I think you'll be able to follow it easily. Note that the database connection info is included in a separate file. Also note that we're using the `getContainerData` method to actually get the data out of the container.

The next step is to construct a URL for use as the source attribute of an HTML image tag. For example, the PHP code to render the image tag might look like this:

```

echo "<img src=\"fm_container_v1.php?container_url=\" . urlencode($employee-
>getField("Photo")) . "\">";

```

I've used the PHP `urlencode` function to encode the container URL and pass it to the `fm_container_v1.php` template. Don't forget this important (and often overlooked) step. Without URL encoding, the value of the `container_url` URL variable that `fm_container_v1.php` will receive will be truncated and incorrect.

At this point, we've been able to successfully publish an image stored in a container field to the Web. So far, so good.

However, from a performance standpoint, this method isn't ideal. Suppose that you have a Web page that lists 25 employees along with their photos, with the employee data and photos stored in a FileMaker database. When that page is loaded, FileMaker will need to respond to not just one request (the request for the list of employees), but to 26 requests. Remember that we make an initial request for the employee data, and then 25 more requests for each individual employee's photo. That's a lot of requests for a single page.

To make matters worse, there is no guarantee that the photos stored in the container fields are optimized. Suppose that a handful of the photos are large (1Mb or so). That means that now, in addition to processing 26 requests, the volume of data that FileMaker needs to return is also quite large.

Therefore, it won't be long before performance of both the database and Web server will begin to deteriorate.

Publishing Images - The Optimized Method

A more optimal way to publish images that are stored in FileMaker container fields is to cache them. This technique involves storing the contents of the containers in such a way that they can be used repeatedly, and without needing to request them from the database every time that they are displayed.

First, we need to setup a location on the Web server where we can store (cache) the images. This needs to be a directory that PHP has the permission to write to. The steps involved in doing this are different depending on the operating system that your Web server uses. In a Windows environment, you create a directory, and then give the Internet Guest Account "Write" permission on it. For complete instructions on how to do this with your server's operating system, I suggest searching for "php write permissions" using your favorite search engine. (Note that you only want to give PHP write permission on the directory that the images will be stored in. There's no need to give it write permission on the directory that your PHP files are stored in -- and in fact, for security reasons, you want to avoid doing that accidentally.)

Next, we need to modify the "fm_container_v1.php" template so that it caches the container data. Here is an updated version of the template (which I'll refer to as fm_container_v2.php).

```
<?php
```

```
    // If no container_url was specified...  
    if (! isset($_GET['container_url'])) {
```

```

        die ("No container URL was specified.");
    }

// Specify the path to the container cache directory.
$container_cache = "C:\inetpub\container_cache";

// Derive a filename from the URL...

    // Split the container_url into two parts (the path and the arguments).
    $url = explode("?", $_GET['container_url']);

    // Get the file extension from the path portion of the container_url.
    $file_extension = pathinfo($url[0], PATHINFO_EXTENSION);

    // Convert the string of arguments into an associative array.
    parse_str(str_replace("amp;", "&", $url[1]), $url_args);

    // Convert the elements of the array into a filename.
    $file_name = "";
    foreach ($url_args as $key => $value) {
        $file_name .= $value . "_";
    }
    $file_name = ereg_replace('[^A-Za-z0-9.],', '_', $file_name);
    $file_name .= "." . $file_extension;

// If the file is not already in the cache...
if (! file_exists("$container_cache/$file_name")) {

    // Include the FileMaker API for PHP.
    require_once("FileMaker.php");

    // Database connection information.
    include ("database_settings.php");

    // Create a connection to the database.
    $fm = new FileMaker ();
    $fm->setProperty('hostspec', FM_HOSTSPEC);
    $fm->setProperty('database', FM_FILE);
    $fm->setProperty('username', FM_USERNAME);
    $fm->setProperty('password', FM_PASSWORD);

    // Get the data in the container.
    $container_data = $fm->getContainerData($_GET['container_url']);

    // If FileMaker encountered an error...
    if (FileMaker::isError($container_data)) {
        echo "Error Code: " . $container_data->getCode() . "<br>";
        echo "Error Message: " . $container_data->getMessage() . "<br>";
        die;
    } else {

```

```

        // Save the container's contents in the cache.
        file_put_contents("$container_cache/$file_name", $container_data);
    }

} else {
    // Get the container's contents from the cache.
    $container_data = file_get_contents("$container_cache/$file_name");
}

// Return the container data.
echo $container_data;

```

?>

Once again, this code includes plenty of comments, making it easy to follow along. However, here are a few notes that might help to clarify what it does:

- We're assigning the location of the cache directory that we created earlier to a variable called `$container_cache`. In this case, I'm using "C:\inetpub\container_cache" but your directory's location will likely be different.
- We're deriving a filename based on the value of the container's URL. The code here is pretty ugly, but it does the job, and returns a filename that looks something like this: `Employees_PHP_Layout_234_Photo_1__.jpg`. Note that the name consists of the database name, the layout that the container is on, the record number, and the field name, as well as the correct file extension.
- Using the filename, we check to see if we already have that file in the cache. If not, we make a request for the data from FileMaker, and save it in a file in the cache. Otherwise, we read the file's contents, and then echo the data.

With this caching technique, we only ever need to retrieve the data from FileMaker once. This greatly reduces the load on the server, and results in much better performance and response times for the PHP application's users.

Next Steps

We've covered two techniques for publishing container data using the FileMaker API for PHP, and at this point you've probably got a good handle on what is involved.

In a future edition of this newsletter, I will take these techniques a step farther, and discuss:

- A technique for removing files from the file cache. You'll need to do this when the data in the container changes. With our current version of the code, if that were to happen, the cached file would not accurately reflect what is in the database.
- Techniques for publishing other types of data stored in containers, including PDF files, MP3 files, and more.

Until then, happy coding!

Tim Dietrich, is the Founder and Lead Developer of Xgravity, a Richmond, Virginia-based information technology firm that designs and develops custom database and Web solutions using FileMaker Pro. Tim has been using FileMaker since 1992, and is a FileMaker 9 Certified Developer. Learn more about Tim and Xgravity at: <http://www.xgravity.net>