

5 Minutes to Better and Faster Web Apps: Reducing Database Requests With Caching

by Tim Dietrich

In previous articles that I've written for the FMWebSchool Newsletter, I've demonstrated some of the techniques that I use to optimize FileMaker-based Web applications. In this article, I'll review some of those techniques and demonstrate another powerful technique.

In general, there are two ways that you can optimize Web applications:

- You can reduce the number of requests that the application sends to FileMaker.
- You can reduce the volume of data that FileMaker returns to the application.

The technique that I am demonstrating this month helps you to reduce the number of requests that the Web application sends to FileMaker Server, and it does so using a concept known as caching. With caching, data is requested from the database and is stored on the Web server for future use, so that we don't need to make repeated requests for the data. This takes some of the load off of the database, and usually results in the application running a little faster.

I first introduced caching back in September, when I presented a method for caching data stored in non-container fields in session state. With this method, a cache is created for each user of the application. In cases where the data being cached is user-specific (for example, a user's profile after they log in), this is an ideal method for caching the data. In other cases, where the cached data is the same for all users (such as a list of product categories for an online store), this method still works, but isn't as efficient as it could be. Ideally, we want to store that type of data in a shared cache that is accessible by all users -- which rules out storing it in session state.

I also discussed caching in October, when I presented a technique for caching data stored in container fields. That technique involves storing container data (such as images) in files on the Web Server itself, and is helpful for two reasons: First, it provides a way to cache data that typically is quite burdensome for FileMaker to provide to the application. And second, the cache created by that technique can be shared by all users of the application.

The technique that I'm presenting this month is a hybrid of the caching techniques that I've discussed in the past. It allows us to create a shared cache of non-container data.

Below is some PHP code that demonstrates how the technique works. In this example, we're creating a shared cache that includes the names of product categories for an online store.

```
<?php
```

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

```

// If a cache file exists and can be read...
if ( $categories_cache = @file_get_contents ("cache/categories.txt") ) {

    // Load the class definition "FileMaker_Result" to avoid receiving the error:
    // "The script tried to execute a method or access a property of an incomplete object."
    require_once ("Result.php");

    // Unserialize the cached find result.
    $find_result = unserialize ($categories_cache);

} else {

    // Get the database connection information.
    require_once ("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);

    // Find categories that are online, and sort them by name.
    $find_request = $fm -> newFindCommand ("PHP - Categories");
    $find_request -> addFindCriterion ("Is_Online", "Yes");
    $find_request -> addSortRule ('Category_Name', 1);
    $find_result = $find_request -> execute();

    // If FileMaker encountered an error...
    if (FileMaker::isError ($find_result)) {
        echo "Error Code: " . $find_result -> getCode() . "<br>";
        echo "Error Message: " . $find_result -> getMessage() . "<br>";
        die;
    } else {
        // Save a serialized version of the find results in the cache.
        file_put_contents ("cache/categories.txt", serialize ($find_result));
    }

}

// Get the records from the find result.
$categories = $find_result->getRecords();

// Display the categories.
foreach ( $categories as $category ) {
    echo $category -> getField("Category_Name") . "<br>";
}

?>

```

The code has been thoroughly commented. However, let's review some important aspects of it.

First, we try to read the file that contains the cached data. If the file is read successfully, we have to do a few things to make it usable -- in particular, we have to unserialize the data. (I'll explain more about this in a moment.) Also note that we are explicitly including the

"FileMaker_Result" class definition, which is a part of the FileMaker API for PHP. We need to do this to avoid a somewhat complicated error that would otherwise be thrown involving incomplete objects. (If you want to see this for yourself, simply comment out the line that includes the "Result.php" file.)

If the file that contains the cached data cannot be read (most likely because the file hasn't been created yet), then we send a find request to FileMaker to get the data. We then save a serialized version of the find results in the cache file. We do this using PHP's "file_put_contents" command (which allows us to write a string to a file) and the "serialize" command (which allows us to easily store PHP values as strings without losing their type and structure). Note that we used the PHP "unserialize" function earlier, which converts the serialized string back into a PHP value.

One more note about this code: The cache files are being saved in a subdirectory called "cache." In order for the caching technique to work, PHP will need to have write permissions on that directory. 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 should avoid doing that.)

We now have several techniques for caching data -- one that creates a user-specific cache for non-container data, and two that can be create caches for container and non-container data that can be shared by all users.

Caching data can greatly reduce the number of requests that the application sends to FileMaker, and can significantly boost the performance of your application as a result. However, one challenge that caching presents is that the cache can get stale. Continuing with the example above (where we cached categories for an online store), suppose that the store administrator takes one of those categories offline. If the categories were cached prior to the category being taken offline, then the cache will still include that category.

In a future issue, I'll discuss ways to refresh the cache when it ages. I'll also demonstrate ways to reduce the volume of data that FileMaker returns to the Web application.

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>