Richard Carlton Consulting
1941 Rollingwood Drive • Fairfield, CA 94534
(707) 422-4053 • www.RCConsulting.com

Optimizing FX.PHP Websites with Cache_Lite
Using FileMaker Server Advanced 8 & iTools 8.2.2 (Apache 2.2)

Introduction

This document details the installation, configuration, and use of Cache_Lite with FX.PHP powered sites. Cache_Lite is a PHP extension that caches the results of PHP code, including queries made to a FileMaker database through FX.php. When Cache_Lite is properly installed, cached queries have no impact on FileMaker Server Advanced, and place only minimal load on PHP. These instructions are intended to assist a programmer in utilizing Cache_Lite to optimize the performance new or existing solutions. These instructions assume a server configured with Tenon's iTools 8.2.2 and FileMaker Server Advanced 8.0v4 running on Mac OS X 10.4 or later[1]. An intermediate to advanced level of proficiency in both PHP and FileMaker is required.

Installation

Cache_Lite is a PEAR package available at http://pear.php.net/package/Cache_Lite. To install Cache_Lite, simply open Terminal, and type,

```
sudo /Library/Tenon/PHP/bin/pear install Cache_Lite
```

You will be prompted for your password; just type it and press enter.

PHP

Using Cache_Lite can be exceptionally simple. Cache_Lite needs a few lines of setup code, an opening line, and a closing line. For example the PHP to connect to a database and display a database might look something like this:

```
<?PHP
include_once('FX/FX.php');
include_once('FX/server_data.php');
$showall = new FX($serverIP,$webCompanionPort, $groupSize = 5000 );
$showall -> SetDBData('test_database.fp7','test_layout');
$showall -> SetDBPassword($webPW,$webUN);
$showallResult = $showall -> FMFindall();

foreach ($showallResult['data'] as $record) {
  echo $record['title']['0'];
  echo $record['date']['0'];
  echo $record['Description']['0'];
```

[1] See http://www.rcconsulting.com/pdfs.html for instructions on how to install FMSA and iTools on OS X.

```
}
?>
```

This example will return all the records in the table specified by the query up to 5000 records. Depending on the amount of data in each field, such a request could take quite a while. So now we have a block of PHP code with an FX.php query that takes a fair amount of time to execute. If a page with this code is accessed once or twice a day it probably won't even be noticed. If code is part of a high-traffic page, you probably will have a problem with server load. Use the log analysis tool AWStats, which is included with Tenon's iTools package, to determine the pages that receive the most hits. Most solutions can be greatly optimized by caching the fewer number of pages that receive the greater number of hits. This is where Cache_Lite can really improve your site's performance.

Basic Optimization

For each PHP page that generates an unacceptable load on a web or database server, there are several things that need to be examined. Sometimes you will find a bug in your code that can be fixed, relieving some of the load on the server. There might be an unneeded find bogging down the page. Before you install a caching system in your PHP, you should take some basic measure to ensure that caching is really necessary.

Bugs are perhaps the most common problem with a high-load PHP page. Even pages that will eventually need caching usually have some room for increased performance. Look for queries that are not being used, perhaps added during development, but no longer required. Double-check your search criteria. Are you sure your query is returning only the records you need? Sorting is something else that can add additionally strain on a query. Do your records really need to be displayed in a given sort order? If you can remove a sort command, that will lighten the load caused by the PHP. If you must have the records sorted, pull them from the database unsorted, and then sort them using PHP. This is much faster than sorting as part of the query.

After you have checked your PHP, take any steps necessary to optimize the database the PHP page is pulling from. Each FX.PHP query should have its own layout. This layout should be entirely empty, except for the fields used in the query. This will ensure that only data required will be returned. Any additional fields can greatly reduce efficiency.

Configuring Cache_Lite

If, after the preceding optimization efforts, the PHP page still places too much load on the server, it is time to install Cache_Lite.

There are two primary segments of code required for Cache_Lite to work. Configuration is the first thing that must be addressed. The following block of code is suitable for most basic caching setups:

```
require_once('Cache/Lite/Output.php');
```

```
$cache_id = 'foo';
$cache_options = array(
    'cacheDir' => '/tmp/',
    'lifeTime' => 1200
);
$cache = new Cache_Lite_Output($cache_options);
```

The first line imports Cache_Lite so we can use the module. The next line creates a variable, $cache_id, that we will use to store a unique identifier for this cache. Each cache id should be specific to a set of results from a query. So if you have a product page, with a product number and a currency option your cache id would look something like this:

```
$cache_id = 'products'.$_GET['product'].$_GET['currency'];
```

This creates a cache unique to a particular product and currency type. This way a customer who requests product information in Dollars will not get a cache listed in Euros. This will also ensure that each product number has its own cache. Cache IDs are what ensure that products.php displays the appropriate data related to the records requested.

The next section in the configuration, called $cache_options, is an array that is used to store options to use with Cache_Lite. We have used only two options, however there are several more available[2]. (For a complete listing of available options for Cache_Lite, please see the documentation included with Cache_Lite.) The first option, 'cacheDir' specifies the directory in which the cache(s) will be created. The example uses /tmp, which is fine for most purposes. That directory is used by various programs to store temporary data, and therefore it is fine to keep our cache files there. The next option called 'lifeTime' determines how long a cache will be used before it expires and a new cache is created. This value is in seconds, so 60 is a minute, 3600 is an hour, etc. The final line that starts with $cache creates a new instance of the caching object for use in our PHP.

Use

Now that we have a properly configured Cache_Lite, all that is left to do is add two lines of code, one to start the caching, and one to end it. The line to start caching looks like this:

```
if(!($cache->start('page_id'))) {
```

The closing line is even shorter:

```
$cache->end();}
```

[2] http://pear.php.net/package/Cache_Lite/docs/1.7.2/apidoc/Cache_Lite/Cache_Lite.html#methodCache_Lite

To add this functionality to our earlier example, we will add the setup code, placing the starting line at the beginning of the main code[3] and the closing line at the end. The end result should look something like the code on the next page:

```
<?PHP

require_once('Cache/Lite/Output.php');
$cache_id = 'foo';
$cache_options = array(
    'cacheDir' => '/tmp/',
    'lifeTime' => 1200
);
if(!($cache->start('page_id'))) {

include_once('FX/FX.PHP');
include_once('FX/server_data.PHP');
$showall = new FX($serverIP,$webCompanionPort, $groupSize = 5000 );
$showall -> SetDBData('test_database.fp7','test_layout');
$showall -> SetDBPassword($webPW,$webUN);
$showallResult = $showall -> FMFindall();

foreach ($showallResult['data'] as $record) {
  echo $record['title']['0'];
  echo $record['date']['0'];
  echo $record['Description']['0'];
}
$cache->end();}
?>
```

This is a simple example of how to use Cache_Lite. For most purposes this kind of setup works just fine. However, in the next section we will examine some tricks that can help you get more out of Cache_Lite.

Cache Refresh

One of the first settings we covered during the setup for Cache_Lite was the cache lifetime. It can be difficult to strike the right balance with this option. By definition, the data we are caching is at least partially dynamic. If the lifetime is set too long, the data will not be updated in a timely fashion, if it is set too short, the advantages of caching can be largely neutralized. There are three main methods of updating cached data, and we will examine the advantages and disadvantages of each.

Cache Flush

[3] As close to the beginning of the file as possible. Generally this will be immediately following the Cache_Lite setup code.

The simplest way to guarantee that a cache gets refreshed is to delete the cache file. Just go into the Finder, Select the files in question, and drag them to the trash. This ensures that the next time the page is queried, a new cache based on fresh data is rebuilt. While this may seem somewhat simplistic, it can work very well in certain situations. Say you had a product database that was largely static, with most of the records updated more or less all at once. You could set your cache directory as an easy-to-access folder and just delete all the files in the folder. This solution works just fine if your product catalog is updated infrequently or irregularly. However, if you have a different update schedule, the next solution might be more suited to your needs.

Flush on Demand

The next system for managing cache refresh is to set up a mechanism for flushing individual caches, either directly from FileMaker or thorough a web page. Caches can usually be organized by an ID number or something similar. With just a few configuration changes, we can set up a system that will purge a set of caches when the PHP page is called with a particular option. The following code gives a good example of how such a system could be implemented:

```
require_once('Cache/Lite/Output.php');
$cache_id = 'foo';
$cache_options = array(
    'cacheDir' => '/tmp/'.$_GET['product'].'/',
    'lifeTime' => 1200
);

@mkdir($cache_options['cacheDir']);
$cache = new Cache_Lite_Output($cache_options);

if($_GET['flush']) {
$cache->clean();
}
if(!($cache->start($cache_id))) {

include_once('FX/FX.php');
include_once('FX/server_data.php');
$showall = new FX($serverIP,$webCompanionPort, $groupSize = 5000 );
$showall -> SetDBData('test_database.fp7','test_layout');
$showall -> SetDBPassword('password','username');
$showallResult = $showall -> FMFindall();

foreach ($showallResult['data'] as $record) {
  echo $record['title']['0'];
  echo $record['date']['0'];
  echo $record['Description']['0'];
}

$cache->end();

}
?>
```

If the page is called with "xyz.php?product=123&currency=USD&flush=1" the flush variable will cause all caches in the cache directory to be deleted. In this case, the cache directory is determined by the product number. Therefore, if the PHP page is called with a particular product number and the 'flush' option it will force the cache to be refreshed for that particular product, without disturbing caches for other products.

The easiest way to call the URL to flush the cache would be the Web Viewer in FileMaker 8.5. Just write a script to call that URL based on the product id of the current record. Attach the script to a button on the products layout, and you are set. If you are using FileMaker 8.0 or earlier, the implementation can be somewhat complicated. When a database is running on Mac OS X, you can use the AppleScript by calculation script step to call the URL, something like:

```
"do shell script \"curl http://xyz.com/xyz.php?product=" & productid &
"\&flush\""
```

When using the database on Windows, you can use the Open URL script step, although, that has the undesirable side effect of opening a new web browser window. There are other methods useable on Windows, but they are beyond the scope of this document.

Automatic Cache Refresh

Both of the previous solutions have their place, but they require user input. Wouldn't it be nice to have the whole thing run automatically? This requires coordination between both FileMaker and PHP. In FileMaker we will need to create three fields. The first field may be there already. We need a modification timestamp for each record. If you do not already have one, you may create a timestamp field and set it to auto-enter the modification time and not permit modification. The second field will be a calculation that will return the modification time of the record in Unix format. One thing that requires special attention is the time_zone_offset. This will change based on daylight saving time and the local on the machine. You could write an expression to automatically return the correct offset based on the time of year. However, due to uncertainty about the time daylight saving starts, it would be best to just mark it on your calendar to adjust the calculation when daylight savings time begins or ends. This can be done quickly by use of a flag field somewhere in your database. Then you can just "flip the switch" to alter the calculation across the board. The calculation should look something like this, based on a custom function from briandunning.com:

```
Let (
[current_timestamp = modification_time ;
time_zone_offset = GetAsNumber(case(flag_dlight;"-7";"-6")];

If (
Date ( (Month (current_timestamp)); (Day (current_timestamp)); Year
((current_timestamp)) )  ≥  (Date(1;1;1970))
and
Date ( (Month (current_timestamp)); (Day (current_timestamp)); Year
((current_timestamp)) ) ≤ (Date(7;8;2038));
```

```
(Seconds ( current_timestamp )) +
(Minute ( current_timestamp ) * 60) +
( (Hour (current_timestamp) - time_zone_offset) * 3600) +
((Date ( (Month (current_timestamp)); (Day (current_timestamp)); Year
((current_timestamp)) ) - (Date(1;1;1970))) * 86400)

;-1))
```

The final field, called umodtime_global, should be set to global storage and return the latest and greatest modification time in this table. Using the Max function, it is trivial:

```
Max ( umodtime )
```

Once we have made the necessary modifications to the FileMaker we will need to update our PHP. We will add a search at the beginning of the script to see if we need to build a new cache, or if we may use the one we already have. The following code illustrates how a PHP page can use the fields we created to decide whether to refresh the cache:

```
require_once('Cache/Lite/Output.php');
include_once('FX/FX.php');
include_once('FX/server_data.php');

$modquery = new FX($serverIP,$webCompanionPort,$groupSize = 1 );
$modquery -> SetDBData('test_database.fp7','test_layout');
$modquery -> SetDBPassword('password','username');
$modqueryResult = $modquery -> FMFind();
foreach ($modqueryResult['data'] as $modrecord) {
  $fmmodtime = $modrecord['umodtime_global']['0'];
}

$cache_id = 'foo';
$cache_options = array(
  'cacheDir' => '/tmp/'.$_GET['product'].'/',
  'lifeTime' => 1200
);
@mkdir($cache_options['cacheDir']);
$cache = new $Cache_Lite_Output($cache_options);
if($fmmodtime > $cache->lastModified() ) {
$cache->clean();
}
if(!($cache->start($cache_id))) {

$showall = new FX($serverIP,$webCompanionPort, $groupSize = 5000 );
$showall -> SetDBData('test_database.fp7','test_layout');
$showall -> SetDBPassword('password','username');
$showallResult = $showall -> FMFindall();

foreach ($showallResult['data'] as $record) {
  echo $record['title']['0'];
  echo $record['date']['0'];
  echo $record['Description']['0'];
}
$cache->end();
}
?>
```

These examples should be enough to get you started. If you need details beyond those provided in this document, consult the PHP manual[4] or Cache_Lite documentation[5]. Each implementation will be slightly different, with different caching configurations to fit different requirements.

Utilizing Cache_Lite in your PHP solutions will improve user experience by decreasing load time, and decrease hardware costs by increasing the number of hits a server can handle before its response time slows.

About Richard Carlton Consulting

Richard Carlton Consulting, Inc. is a solutions development company that utilizes FileMaker Pro to deploy mission critical business solutions. We service both small and large businesses as well as government agencies. We employ approximately 16 FileMaker and Web engineers, with varying skill sets, allowing us to develop complete solutions and support customer needs.

Richard Carlton has been in the business of writing custom software and contracting with businesses for computer consultation purposes since 1985. In 1990, Richard Carlton Consulting was created, and later was "recommissioned" as Richard Carlton Consulting, Inc.

Richard Carlton Consulting (RCC) provides customized database development services for business and government organizations. Our core competency is FileMaker Pro, where we hold FileMaker's highest certification. We are "Certified" as well as a "FileMaker Partner." We support FileMaker 3,4,5,6,7 and of course, FileMaker 8. We also utilize FileMaker Web Publishing and PHP to deploy FileMaker databases to the web. RCC has extensive experience in deploying web solutions with database connectivity. Our experience ranges from simple projects to complex e-commerce based systems.

[4] http://www.php.net/manual/en/
[5] http://pear.php.net/manual/en/package.caching.cache-lite.php