FMCakeMix

Use FileMaker in an MVC web development framework.

User Guide

Table of Contents

Introduction

FMCakeMix is a FileMaker datasource driver for the CakePHP MVC framework. FMCakeMix enables FileMaker databases to integrate into Cake as if they were native SQL based sources, allowing for rapid development of FileMaker based web solutions in a modern web solution framework.

This guide covers the basics of using the FMCakeMix driver within the CakePHP. The examples included gloss over some important implementation and security details.

To get more familiar with CakePHP visit: http://cakephp.org/

Installation

CakePHP

Download and follow the installation instructions from the cake website http://cakephp.org/.

FX.php

FX.php is PHP class created by Chris Hansen to speak with FileMaker via XML. The FMCakeMix driver uses fx.php to send queries to FileMaker and is necessary for the driver's functionality. Install FX.php by downloading the files from http://www.iviking.org/FX.php/ and placing the FX.php, FX_Error.php, FX_Constants.php, and image_proxy.php files at the root of the yourcakeinstall/vendors folder.

FileMaker

Because the driver uses XML to communicate with FileMaker, your FileMaker solutions must be hosted on a version of FileMaker Server that supports web publishing and xml access. See the FileMaker Server documentation for instructions on enabling these features.

FMCakeMix

Install the dbo_fmcakemix.php file into yourcakeinstall/app/models/datasources/dbo, you'll likely have to create the dbo directory in the datasources folder.

Define Your Database Connection

Database connections are defined within app/config/database.php. Below we've defined our default connection to use the fmcakemix driver and provided the necessary details for cake's connection manager to connect with our filemaker database. If your models refer to multiple FileMaker database files, don't worry we will override this setting when defining our model.

```
var $default = array(
        'driver' => 'fmcakemix',
        'persistent' => false,
        'dataSourceType' => 'FMPro7',
        'scheme' => 'http',
        'port' => 80,
        'host' => '127.0.0.1',
        'login' => 'myUserName',
        'password' => 'myPassword',
        'database' => 'FMServer_Sample',
        'prefix' => '',
);
```

Define Your Model

Define your model as you normally would within CakePHP, though certain features may not be available, refer to the Known Limitations section for more details. In addition to the standard model attributes of name, useDbConfig, and primaryKey, we'll also want to tell Cake to associate our model with a default FileMaker layout using the defaultLayout attribute and define a fmDatabaseName for the FileMaker file where our layout lives.

Relations are defined through the hasMany, hasOne, belongsTo, and hasAndBelongsToMany attributes. Currently the driver only supports hasMany and belongsTo relations. There are essentially two options you have when working with related data in FileMaker; either use relationships defined within Cake or leverage FileMaker's ability to relate and retrieve data through portals. Remember when retrieving data through a Cake defined relationship you're actually making a new call for every related model, this could have a negative impact on performance.

The fmTOtranslations attribute allows you to associate related portal data that may be returned by the model's layout to a Cake model we may have defined elsewhere within our application. Here we're associating any data returned from a portal of commentsTO, the name of a FileMaker table occurrence, to a Comments model.

```
class Book extends AppModel {

        var $name = 'Book';
        var $useDbConfig = 'default';
        var $primaryKey = 'ID';

        // FMCakeMix specific attributes
```

```
	var $defualtLayout = 'web_books_general';
	var $fmDatabaseName = 'FMServer_Sample';

	// Optionally assign related models
	var $hasMany = array(
		'Comment' => array(
			'foreignKey' => '_fk_book_id'
		),
		'History' => array(
			'foreignKey' => '_fk_book_id'
		)
	);

	// Optionally provide translations of related FileMaker table occurrences
	// that may be returned through FileMaker portals into Cake model names
	var $fmTOtranslations = array(
		'CommentsTO' => 'Comment'
	);

	// Optionally provide validation criteria for our model
	var $validate = array(
		'Title' => array(
			'rule' => 'notEmpty'
		),
		'Author' => array(
			'rule' => 'notEmpty'
		)
	);
}
```

Controller Examples

Controllers are where all the action is. Because we've defined our connection details in our database config file and our model details for all the relevant models we want to use, we can now concentrate on the logic of our application and interacting with our models. Below we'll cover the basics for creating, reading, deleting, and updating data within our FileMaker database.

Create

save

A basic add method for our controller. Here we're taking information passed from a form, cake takes care of this, from $this->data and calling two model methods to save this data to a new record in FileMaker. First we call *create* to prepare the model to save a new record and then we call *save* passing our form data as the parameter. It's important to note that cake will continue to treat fields with certain names as special, such as fields named *created* or *modified* which will get populated with the appropriate timestamps automatically.

```
function add() {
	if (!empty($this->data)) {
		$this->Book->create();
		if ($this->Book->save($this->data)) {
			$this->Session->setFlash(__('The Book has been saved', true));
			$this->redirect(array('action'=>'index'));
		} else {
```

```
                        $this->Session->setFlash(__('The Book could not be saved. Please, try
again.', true));
                }
        }
}
```

saveAll

The saveAll model method will allow us to save multiple models at a time. When using the saveAll method always pass the option atomic is false to tell Cake not to attempt a transactional save to our database.

```
$_data = array(
        'Comment' => array(
                array(
                        '_fk_article_id' => $this->Book[‘ID’],
                        'body' => 'New Comment'
                ),
                array(
                        '_fk_article_id' => $this->Book[‘ID’],
                        'body' => 'Another Comment'
                )
        )
);
$this->Comment->create();
$this->Comment->saveAll($_data['Comment'], array('atomic' => FALSE));
```

Read

find

In the example below a basic search function has been implemented. Here we collect a query for a recipe title and perform a find request for recipes containing this title and with a published value of 1. The returned result is then sent to the view using the *set* method.

```
function search() {
        $query = $this->data['Recipe']['title'];

        $recipes = $this->Recipe->find('all', array(
                'conditions' => array(
                        'title’ => $query,
                        ‘published’ => ‘=’.1
                )
        ));

        $this->set('recipes', $recipes);
}
```

paginate

The paginate method works with a paginate helper in the view to create a paginated list of records. Here we set the index method of the controller to return a paginated list of our books. Setting the recursive attribute of the Book model to 0 will prevent any queries for related model data.

```
var $paginate = array('limit' => 10, 'page' => 1);

function index() {
        $this->Book->recursive = 0;
        $this->set('books', $this->paginate('Book'));
}
```

Delete

del, remove

The *del* method and its alias *remove* will delete a single record from your database. FileMaker requires that we send the internal recid of the record we wish to delete with every delete request. A recid is returned as one of the fields in the returned data set whenever we return record data, such as after a find command. Additionally the recid is saved to the model id attribute which leaves the model referencing the record returned on the last query, this is especially useful after a create action. Note however that this is a departure from a CakePHP standard that assumes the primaryKey id will be stored in this attribute.

In the example below the find sets the model id attribute so that when calling the *del* method FileMaker is passed the appropriate recid of the record to be deleted.

```
delete() {

	$this->Book->find('first', array(
		'conditions' => array(
			'Book.ID' => 48
		),
		'recursive' => 0
	));

	$model->del()
}
```

deleteAll

Here's a more functional example of how you might implement a delete method. Here we pass the recid of the record to delete and provide some user feedback to the view. Instead of using the *del* method we use *deleteAll* to be explicit about the record we wish to delete.

```
function delete($recid = null) {
	if (!$recid) {
		$this->Session->setFlash(__('Invalid id for Book', true));
		$this->redirect(array('action'=>'index'));
	}
	if ($this->Book->deleteAll(array('-recid' => $recid), false)) {
		$this->Session->setFlash(__('Book deleted', true));
		$this->redirect(array('action'=>'index'));
	} else {
		$this->Session->setFlash(__('Book could not be deleted', true));
		$this->redirect(array('action'=>'index'));
	}
}
```

Update

save

An update works much like a create and uses the same *save* model method, but instead we pass along the FileMaker required recid of the record we wish to edit. In this example the recid is included in the passed form data, implemented as a hidden input.

```
function edit($id = null) {
	if (!$id && empty($this->data)) {
		$this->Session->setFlash(__('Invalid Book', true));
	}
	if (!empty($this->data)) {
```

```
                if ($this->Book->save($this->data)) {
                        $this->Session->setFlash(__('The Book has been saved', true));
                        $this->redirect(array('action'=>'index'));
                } else {
                        $this->Session->setFlash(__('The Book could not be saved.', true));
                }
        }
        if (empty($this->data)) {
                $this->data = $this->Book->read(null, $id);
        }
}
```

View Examples

automagic fields

Because FMCakeMix is able to provide basic schema information about the fields in your model, Cake is able to make intelligent choices when performing certain tasks with your data such as creating the appropriate input types when building a form.

```
<?php echo $form->create('Book');?>
        <fieldset>
                <legend><?php __('Edit Book');?></legend>
        <?php
                echo $form->input('Title');
                echo $form->input('Author');
                echo $form->input('Publisher');
                echo $form->input('Status');
                echo $form->input('Description', array('type' => 'textarea'));
                echo $form->input('Quantity in Stock');
                echo $form->input('Number of Pages');
                echo $form->hidden('-recid');
        ?>
        </fieldset>
<?php echo $form->end('Submit');?>
```

paginate

An example that implements the paginated index method of our controller.

```
<?php
echo $paginator->counter(array(
'format' => __('Page %page% of %pages%, showing %current% records out of %count% total, starting
on record %start%, ending on %end%', true)
));
?></p>
<table cellpadding="0" cellspacing="0">
<tr>
        <th><?php echo $paginator->sort('Title');?></th>
        <th><?php echo $paginator->sort('Author');?></th>
        <th><?php echo $paginator->sort('Publisher');?></th>
        <th class="actions"><?php __('Actions');?></th>
</tr>
<?php
$i = 0;
foreach ($books as $book):
        $class = null;
        if ($i++ % 2 == 0) {
                $class = ' class="altrow"';
        }
?>
```

```
        <tr<?php echo $class;?>>
                <td>
                        <?php echo $html->link($book['Book']['Title'], array('controller'=>
'books', 'action'=>'view', $book['Book']['ID'])); ?>
                </td>
                <td>
                        <?php echo $book['Book']['Author']; ?>
                </td>
                <td>
                        <?php echo $book['Book']['Publisher']; ?>
                </td>
                <td class="actions">
                        <?php echo $html->link(__('View', true), array('action'=>'view',
$book['Book']['ID'])); ?>
                        <?php echo $html->link(__('Edit', true), array('action'=>'edit',
$book['Book']['ID'])); ?>
                        <?php echo $html->link(__('Delete', true), array('action'=>'delete',
$book['Book']['-recid']), null, sprintf(__('Are you sure you want to delete %s?', true),
$book['Book']['Title'])); ?>
                </td>
        </tr>
<?php endforeach; ?>
</table>
<div class="paging">
        <?php echo $paginator->prev('<< '.__('previous', true), array(), null, ar-
ray('class'=>'disabled'));?>
 |      <?php echo $paginator->numbers();?>
        <?php echo $paginator->next(__('next', true).' >>', array(), null, array('class'=>'di-
sabled'));?>
</div>
```

Known Limitations

FileMaker

- Container Fields : container fields will supply a url string to the resource or a copy of the resource made by filemaker, but files can not be uploaded into container fields.

CakePHP Model

Attributes

- hasOne : currently no support for this relationship type
- hasAndBelongsToMany : currently no support for this relationship type

Methods

- deleteAll : only takes the condition that the -recid equals the recid of the record to delete and therefore does not support deleting many records at a time. Also, you must pass a boolean false as the second parameter of this request so that it does not attempt recursive deletion of related records
- save : the fields parameter, or white list of fields to save, does not work.

- saveAll : does not support database transactions and therefore the atomic option must be set to false