2011/11/16 - Antonio Rohman Fernandez

Contents

1. Installation

Download the latest **core** package from http://www.phpcloudframework.com (zip file)
Unzip the core files on the server where your Apache webserver is installed. Is recommended that phpCloud's core is placed outside your webserver (**/var/www**), a common place to store it would be:

/etc/phpcloud

Is important to install several php packages to make full use of the framework:
Ubuntu > **aptitude install php-apc php-pear php5 php5-common php5-curl php5-dev php5-gd php5-mcrypt php5-memcached php5-mysql php5-pgsql**

php-apc is **optional** but required if you want to use APC as cache driver.
php5-memcached is **optional** but required if you want to use memcached as cache driver.
php5-mysql is **optional** but required if you want to use a MySQL Relational DB.
php5-pgsql is **optional** but required if you want to use a PostgreSQL Relational DB.

Is also important to activate Apache's **mod_rewrite** module or the MVC won't work properly.
Ubuntu > **a2enmode rewrite**
Ubuntu > **/etc/init.d/apache2 restart**

Download the **sample webapp** from http://www.phpcloudframework.com (zip file)
Unzip the sample files inside your webserver:

/var/www/mywebapp

Change ownership to the webserver's user:
Ubuntu> **chown - R www-data:www-data /var/www/mywebapp**

2. WebApp Configuration

Edit the main configuration file to point to your core's location and **edit** some important constants.

Ubuntu > **nano /var/www/mywebapp/config/config.php**

```
<?php
 /*
  * File: {APP}/config/config.php
  * - Define Constants for application configuration.
  *
  * phpCloud: PHP Framework for Cloud sites ( http://phpcloudframework.com )
  * Author: Antonio Rohman Fernandez ( rohman{at}mahalostudio.com )
  * Copyright 2010-2011, MahaloStudio. ( http://mahalostudio.com )
  *
  * Licensed under The MIT License ( http://www.opensource.org/licenses/mit-license.php )
```

```
 * Redistributions of files must retain the above copyright notice.
 */

/*
 * Folder where phpCloud core is stored; can be placed outside your webserver root.
 */
define('COREPATH', '/etc/phpcloud');

/*
 * Desired Application Name.
 */
define('APP_NAME', 'My Web Application');

/*
 * Indicate if the site is in Production or Development mode.
 * true = Production mode ( will log errors on file and redirect users to a 404 page )
 * false = Development mode ( will show errors on screen )
 */
define('PRODUCTION', false);

/*
 * Indicate if the site is in Maintenace mode.
 * false = Normal mode.
 * true = Maintenace mode ( will redirect users to a maintenace page )
 * to edit the contents of the default maintenance page '/error/maintenance', edit the file:
{APP}/views/Error/maintenance.php
 */
define('MAINTENANCE', false);
define('MAINTENANCE_PAGE', '/error/maintenance');

/*
 * Application's Default Layouts.
 * defaults: 'default' and 'error'
 * new layouts can be stored at {APP}/layouts
 */
define('DEFAULT_LAYOUT', 'default');
define('DEFAULT_LAYOUT_ERROR', 'error');

/*
 * Desired controller/action for Homepage [ http://yourdomain.com/ ]
 * format: '/controller/action'
 * to edit the contents of the default home page '/page/home', edit the file: {APP}/views/Page/home.php
 */
define('HOMEPAGE', '/page/home');

/*
 * phpCloud Error Pages for Production and Development.
 * format: '/controller/action'
 * to edit the contents of the default production error page '/error/404', edit the file: {APP}/views/Error/404.php
 * to edit the contents of the default development error page '/error/dev', edit the file:
{APP}/views/Error/dev.php
```

```
 */
 define('ERRORPAGE_PRODUCTION', '/error/404');
 define('ERRORPAGE_DEVELOPMENT', '/error/dev');

 /*
  * Special URL Conversions can be defined here. Use a JSON array here.
  * - patterns: regular expresions to match strings to be replaced.
  * - replacements: what will be replaced.
  * Example:
  *  Pattern: "/^@/" <-- regular expression to match a string starting with '@'
  *  Replacement: "/users/profile/"
  *  Result: http://phpcloudframework.com/@rohman => http://phpcloudframework.com/users/profile/rohman
  */
 define('URL_CONVERSION', false);
 define('URL_CONVERSION_RULES', '[{"pattern":"/^@/","replacement":"/user/profile/"}]');

 /*
  * Security string to encrypt session/cached data.
  * IS **EXTREMELY** IMPORTANT THAT YOU MODIFY THIS CONSTANT.
  * For more security use a complex string containing letters [a-zA-Z], numbers [0-9] and other characters like
'$%_='.
  * Once you decide your SALT, don't change it or will be imposible to retrieve encrypted data.
  */
 define('SALT', 's0m3+|2anD0m-S7r1n6');

 /*
  * Indicate where you want to save the sessions.
  * To use session type "cache" you need to have the "cache" component.
  * - SESSION_DEFAULT_TYPE can be:
  *     'php' -> Save session data in the php's $_SESSION[].
  *     'cache' -> Save session data in cache [ disk, apc, memcached ]
  * - SESSION_EXPIRATION: default number of minutes for data to expire. Default: 1440 ( 1 day ) ['cache' type
ONLY]
  */
 define('SESSION_DEFAULT_TYPE', 'php');
 define('SESSION_EXPIRATION', 1440);

 /*
  * Indicate if you want to use Cache mechanisms and options.
  * new Memcached Servers connections can be defined at {APP}/config/memcached.php
  * - CACHE_DEFAULT_TYPE can be:
  *     'disk' -> Save cached data in the server's hard drive.
  *     'apc' -> Save cached data with APC extension [ Another PHP Cache ]
  *     'memcached' -> Save cached data in memory using Memcached.
  * - CACHE_SIZE: maximum size of the cache. Default: 1000000 ( 1GB ) ['disk' cache ONLY]
  * - CACHE_PURGE: delete cache files that has not been ACCESSED in X minutes. Default: 1440 ( 1 day ) ['disk'
cache ONLY]
  * - CACHE_EXPIRATION: default number of minutes for data to expire.
  */
 define('CACHE', false);
 define('CACHE_DEFAULT_TYPE', 'disk');
```

```
 define('CACHE_SIZE', 1000000);
 define('CACHE_PURGE', 1440);
 define('CACHE_EXPIRATION', 30);

/*
 * Application's Default SQL Database.
 * set which database is your default database.
 * new database connections can be defined at {APP}/config/db.php
 * ** this constant is OPTIONAL, but can be useful if you have only 1 DB or a main DB **
 */
 define('SQL_DEFAULT_DB', 'main');

/*
 * Application's Default Riak Cluster.
 * set which Riak Cluster is your default.
 * new Riak Clusters can be defined at {APP}/config/riak.php
 * ** this constant is OPTIONAL, but can be useful if you have only 1 Cluster or a main Cluster **
 */
 define('RIAK_DEFAULT_CLUSTER', 'main');
 define('RIAK_ENCRYPTION', false);

/*
 * Application's Default Language.
 * default: en_us ( American English )
 * new language packs can be stored at {APP}/localization
 * ** this constant is OPTIONAL, but can be useful for OPTIONAL modules like localization **
 */
 define('DEFAULT_LANGUAGE', 'en_us');
```

Modified lines in {APP}/config/config.php

Tip: If you want to make use of cache, url conversion, etc... or even put the site on **maintenance** mode set them to '**true**'.

To make use of SQL Relational Databases, **edit** the db configuration file.

Ubuntu > **nano /var/www/mywebapp/config/db.php**

```
<?php
 /*
  * File: {APP}/config/db.php
  * - Define DataBase Connections.
  *
  * phpCloud: PHP Framework for Cloud sites ( http://phpcloudframework.com )
  * Author: Antonio Rohman Fernandez ( rohman{at}mahalostudio.com )
  * Copyright 2010-2011, MahaloStudio. ( http://mahalostudio.com )
  *
  * Licensed under The MIT License ( http://www.opensource.org/licenses/mit-license.php )
  * Redistributions of files must retain the above copyright notice.
  */
```

```
class DBConfig {

 /*
  * - description: {optional} text to specify what is the DB for.
  * - driver: type of DB server.
  *     'mysql' -> MySQL
  *     'mysqli' -> MySQL improved
  *     'pgsql' -> PostgreSQL
  *     'mssql' -> MS SQL Server
  *     'oracle' -> Oracle
  * - host: IP or hostname to connect to the database server.
  * - port: {optional} port to connect to the database server.
  * - schema: {optional} schema where your database belongs. Default = 'public' [ONLY pgsql]
  * - database: name of the database we want to use in our database server.
  * - username: username to connect to the db.
  * - password: password to connect to the db.
  * - encoding: {optional} specify which character encoding to use.
  */

 /*
  * Databases for Development mode.
  * If left empty, $production databases will be used instead.
  */
  var $development = array();

 /*
  * Databases for Production mode.
  */
  var $production = array(
    'main' => array(
      'description' => "Main PostgreSQL DB",
      'driver' => 'pgsql',
      'host' => '127.0.0.1',
      'schema' => 'public',
      'database' => 'PGdb',
      'username' => 'PGdbUser',
      'password' => 'PGdbPassword'
    ),
    'another' => array(
      'description' => "Another DB, this time MySQL",
      'driver' => 'mysql',
      'host' => '127.0.0.1',
      'database' => 'MYdb',
      'username' => 'MYdbUser',
      'password' => 'MYdbPassword'
    )
  );

} /* Class: DBConfig */
```

Modified lines in {APP}/config/db.php

Tip: We can specify different databases for both Production and Development. If Development array is left empty, Production databases will be used instead for both modes.
Can be set at config.php => **define('PRODUCTION', true/false);**

To make use of NoSQL **RIAK** Databases, **edit** the riak configuration file.

Ubuntu > **nano /var/www/mywebapp/config/riak.php**

```
<?php
 /*
  * File: {APP}/config/riak.php
  * - Define Riak Connections.
  *
  * phpCloud: PHP Framework for Cloud sites ( http://phpcloudframework.com )
  * Author: Antonio Rohman Fernandez ( rohman{at}mahalostudio.com )
  * Copyright 2010-2011, MahaloStudio. ( http://mahalostudio.com )
  *
  * Licensed under The MIT License ( http://www.opensource.org/licenses/mit-license.php )
  * Redistributions of files must retain the above copyright notice.
  */

class RiakConfig {

 /*
  * - description: {optional} text to specify what is the Riak Cluster for.
  * - version: {optional} current installed version and local path to installable package in the server.
  * - host: IP or hostname to connect to the Riak server.
  * - port: {optional} port to connect to the Riak server. Default: 8098
  * - riak: {optional} Riak's entrypoint for requests. Default 'riak'
  * - mapred: {optional} Riak's entrypoint for MapReduce jobs. Default: 'mapred'
  * - ping: {optional} Riak's entrypoint for Ping requests. Default: 'ping'
  */

 /*
  * Riak Clusters for Development mode.
  * If left empty, $production databases will be used instead.
  */
  var $development = array();

 /*
  * Riak Clusters for Production mode.
  */
  var $production = array(
    'main' => array(
      'description' => "Main Riak Cluster",
      'version' => '/sources/riak_1.0.0-1_amd64.deb',
      'host' => 'http://85.10.209.11',
      'port' => 8098,
```

```
    'riak' => 'riak',
    'mapred' => 'mapred',
    'ping' => 'ping'
  )
 );

} /* Class: RiakConfig */
```

Modified lines in {APP}/config/riak.php

Tip: If you obfuscated RIAK's entry urls on RIAK server's configuration file, change the 'riak' and 'mapred' parameters.

To make use of **Memcached**, **edit** the memcached configuration file.

Ubuntu > **nano /var/www/mywebapp/config/memcached.php**

```
<?php
 /*
  * File: {APP}/config/memcached.php
  * - Define Memcached Servers Connections.
  *
  * phpCloud: PHP Framework for Cloud sites ( http://phpcloudframework.com )
  * Author: Antonio Rohman Fernandez ( rohman{at}mahalostudio.com )
  * Copyright 2010-2011, MahaloStudio. ( http://mahalostudio.com )
  *
  * Licensed under The MIT License ( http://www.opensource.org/licenses/mit-license.php )
  * Redistributions of files must retain the above copyright notice.
  */

class MemcachedConfig {

 /*
  * Memcached Servers for Development mode.
  * If left empty, $production servers will be used instead.
  */
 var $development = array();

 /*
  * Memcached Servers for Production mode.
  */
 var $production = array(
   array('127.0.0.1', 11211, 80), /* IP, PORT, PRIORITY TO BE SELECTED {optional} -> 100 [ 100% ] */
   array('xxx.xxx.xxx.xxx', 11211, 20)
 );

} /* Class: MemcachedConfig */
```

Modified lines in {APP}/config/memcached.php

Tip: Pay attention that is an array of arrays and selection priority % is optional.

3. Using the MVC Framework

phpCloud Framework is an MVC (Model-View-Controller) PHP5 Framework that divides data, logic and visualization.

Folder Structure

The web application is organized in the following way:
/config ==========> Configuration files are stored here.
/controllers ==========> Controllers are stored here. i.e: *UserController.php*
/custom ==========> PHP code that may be used in different views are stored here. i.e: *a menubar*
/layouts ==========> HTML Layouts are stored here. i.e: *ajax.php*
/localization ==========> Localization language packs are stored here into folders.
...../en_us ======> i.e: *American English (en_us)* language pack.
...../es =========> i.e: *Spanish (es)* language pack.
/models ==========> Models are stored here. i.e: *UserModel.php*
/tmp ==========> Temporal data placeholder. Make user webserver (*www-data*) have **write access** to it.
...../cache ==========> When using '*disk*' cache, data will be stored here.
...../logs ==========> When using '*production*' mode, PHP errors will be logged here.
/views ==========> Views are stored here into folders.
...../Error ==========> Default error pages, pay attention to *404.php*, *dev.php* and *maintenance.php*.
...../Page ==========> Static pages can be stored here.
...../User ==========> i.e: *User* views for UserController.php
/webroot ==========> Main entrypoint for our web application. Put files you want to display on views here.
...../css ==========> CSS files go here.
...../img ==========> Images go here.
...../js ==========> JavaScript files go here

Models

Models take care of the data, how is constructed, relationships, validations, saving, querying, etc...

See the example below to understand how models work, this *PhotoModel* example is for data stored in a **SQL** Relational Database like PostgreSQL, MySQL, etc... Pay attention to the areas marked in **violet**.

Ubuntu > **nano /var/www/mywebapp/models/PhotoModel.php**

```
<?php
 /*
  * File: {APP}/models/PhotoModel.php
  * - Sample Model to see how phpCloud works.
  *
  * phpCloud: PHP Framework for Cloud sites ( http://phpcloudframework.com )
  * Author: Antonio Rohman Fernandez ( rohman{at}mahalostudio.com )
  * Copyright 2010-2011, MahaloStudio. ( http://mahalostudio.com )
  *
  * Licensed under The MIT License ( http://www.opensource.org/licenses/mit-license.php )
  * Redistributions of files must retain the above copyright notice.
```

```
*/

require(COREPATH.DS.'models'.DS.'sqlmodel.class.php');

class PhotoModel extends SQLModel {

 /*
  * Database Connection for this Model.
  * this variable is OPTIONAL, if removed, Constant SQL_DEFAULT_DB will be used as default.
  */
  protected $_dbSettings = "main";

 /*
  * Database Table for this Model. ( Mandatory variable )
  * specify in which table is the data stored.
  */
  protected $_dbTable = "photos";

 /*
  * Database ID field for this Model.
  * specify which is the ID field for this Model+Table.
  * this variable is OPTIONAL, if removed, "id" will be used as default.
  */
  protected $_dbID = "id";

  public $hasOne = array(
     "Exif" => array(
        "table" => "exifdata",
        "relationship" => array("id" => "photo_id")
     )
  );

  public $hasMany = array(
     "People" => array(
        "table" => "peopleinphotos",
        "relationship" => array("id" => "photo_id"),
        "belongsTo" => array(
            "Person" => array(
               "table" => "people",
               "relationship" => array("people_id" => "id")
           )
        )
     )
  );

  public $belongsTo = array(
     "Photographer" => array(
        "table" => "photographers",
        "relationship" => array("photographer_id" => "id")
     ),
     "Camera" => array(
```

```
        "table" => "cameras",
        "relationship" => array("camera_id" => "id")
    )
  );

  protected $_validate = array(
    /* Rules for field [filename] */
    'filename' => array(
      'unique' => array(
        'rule' => 'isUnique',
        'message' => 'The Filename must be unique, you are trying to upload a duplicated photo'
      ),
      'maxlenght' => array(
        'rule' => 'between',
        'options' => array('min' => 1, 'max' => 24),
        'message' => 'The Filename must have between 1 to 24 characters'
      )
    ),
    /* Rules for field [photo] */
    'photo' => array(
      'lalala' => array(
        'rule' => 'extension',
        'options' => array('extensions' => array("jpg", "png")),
        'message' => 'Only JPG and PNG files accepted'
      ),
      'lalela' => array(
        'rule' => 'mimetype',
        'options' => array('types' => array("image/jpg", "image/jpeg", "image/png")),
        'message' => 'The mimetype of the photo is incorrect'
      ),
      'lalele' => array(
        'rule' => 'filesize',
        'options' => array('value' => 30000),
        'message' => 'The size of the file exceeds 30K'
      )
    )
  );

} /* Class: PhotoModel */
```

Example SQL model in {APP}/models/PhotoModel.php

Now let's understand the code:

1. Naming and Database Selection

```
require(COREPATH.DS.'models'.DS.'sqlmodel.class.php');
class PhotoModel extends SQLModel { ... }
```

As said before, this example model is meant to be stored in a SQL database, so we need to include **core**'s "sqlmodel.class.php" and need to extend its **SQLModel** class. Models has to be named capitalized like **P**hoto**M**odel.

2. Database Setup

```
protected $_dbSettings = 'main';
protected $_dbTable = 'photos';
protected $_dbID = 'id';
```

Remember that in **{APP}/config/db.php** we specified different databases and that we named our PostgreSQL database as 'main'? **$_dbSettings** variable is used to tell the web application which of those databases to use for this model, so it knows that it needs to save/query data from our PostgreSQL. **$_dbTable** variable is used to specify the database table and **$_dbID** is to specify the field that contains the primary key for the table, as normally is "id" this variable is optional, if not set, the web application will set it to "id" automatically.

3. SQL Relationships

```
public $hasOne = array(...);
public $hasMany = array(...);
public $belongsTo = array(...);
```

Is important to note that SQL Relationships are only available for SQL Models. NoSQL databases like RIAK won't have this option.

SQL database are built upon Relationships on data... A photo **has many** people on it and at the same time it **belongs to** a photographer and a camera. Being more specific, all photos **has one** Exif data provided by the camera on the time of taking the photo. If we would store all this data into the database we can easily imagine in our heads the following tables:

- table: photos
- table: people
- table: peopleinphotos (table for 1 to many photo<->people)
- table: photographers
- table: cameras
- table: exifdata

also we could easily think of the following queries:

- SELECT * FROM photos WHERE camera = 'CANON-EOS-450D';
- SELECT * FROM photos WHERE photographer = 'Miwa';
- SELECT * FROM exifdata WHERE photo_id = 'DSC-0001';
- SELECT * FROM peopleinphotos WHERE photo_id = 'DSC-0001';

This kind of relationships can be set up and manage automatically in the Models. Lets take as example the relationships in our PhotoModel. We will get something like this:

$this->PhotoModel->find('DSC-0001');

['**Photo**'] => Array (

```
['id'] => 'DSC-0001',
['label'] => 'Wedding in Hawaii',
['photographer_id'] => 'Miwa',
['camera_id'] => 'CANON-EOS-450D',
['timestamp'] => '2010-08-29 12:25:23',
['belongsTo'] => Array (
   ['Photographer'] => Array (
      ['id'] => 'Miwa',
      ['agency'] => 'Watabe Weddings'
   ),
   ['Camera'] => Array (
      ['id'] => 'CANON-EOS-450D',
      ['brand'] => 'CANON',
      ['year'] => 2009
   )
),
['hasOne'] = Array (
   ['Exif'] = Array (
      ['id'] = 1515415,
      ['photo_id'] = 'DSC-0001',
      ['data'] => 'Exif data generated by camera....'
   )
),
['hasMany'] => Array (
   ['People'] => Array (
      ['people_id'] => 1,
      ['belongsTo'] => Array (
         ['People'] => Array (
            ['id'] => 1,
            ['name'] => 'Antonio Rohman Fernandez',
            ['age'] => 30,
            ['nationality'] => 'Spanish'
         )
      ),
      ['people_id'] => 2,
      ['belongsTo'] => Array (
         ['Person'] => Array (
            ['id'] => 2,
            ['name'] => 'Yichin Lee',
            ['age'] => 28,
            ['nationality'] => 'Taiwanese'
         )
      )
   )
)
```

As you can see, only querying the photo ID give us not only the photo information but also all the relationships we set up in the model saving us a lot of time making queries.

4. Validations

```
protected $_validate = array(...);
```

Validating data is a very important thing to have into consideration if you want to have a safe application and avoid hackers to break your code or steal sensitive information. Validating data that comes from HTML <Form> is crucial and kind of waste of time, so the Model is here to help you out. We can set several validation rules in the Model and make him validate data any time we need with simply doing something like: **$this->Photo->validates($data);** Easy right? Not only is easy, is also convenient and a life saver!

If you are using the HTML helper to generate the HTML <Form>, the controller will receive all the data inside an array called **$this->data** so we can validate it with the Model easily like the example below:

```
<?=$html->startForm('Photo', array('action' => 'upload', 'enctype' => 'multipart/form-data'));?>
 <div><?=$html->input('text', 'Photo.filename', array('label' => 'Filename: '));?></div>
 <div><?=$html->input('file', 'Photo.photo', array('label' => 'Photo: '));?></div>
<?=$html->endForm('Upload');?>
```

This will be received in the **PhotoController** into the **upload()** function like:

```
$this->data => Array (
   ['Photo'] => Array (
      ['filename'] = 'DSC-0002.JPG',
      ['photo'] => Array (
         ['extension'] => 'jpg',
         ['mimetype'] => 'image/jpg',
         ['filesize'] => 50000,
         ['some_other_data'] = '...'
      )
   )
)
```

As you may see from our example validation rules... the file size of the uploaded photo is 50K and we allow 30K maximum, so this one should be marked as an error... the rest of validations are fine. To validate the data in the controller we would do like this:

```
function upload() {
   if ($this->Photo->validates($this->data)) {
      /* Data is validated, we can safely save it into the database */
      $this->Photo->save($this->data);
   } else {
      /* Data is wrong! we send back the errors to the View */
      $this->set('errors', $this->Photo->invalidFields);
   }
}
```

Setting up Validations is pretty simple, we just need to build validation arrays in the following matter, adding the **name of the field** to validate ($this->data['User']['**email**']), a **description** for the rule, a **validation rule** from the list below and a **message** to tell the user that we did something wrong. There are also some validation rules that

require **extra parameters** like the "between" rule.

```
'email' => array(
   'validMail' => array(
      'rule' => 'isMail',
      'message' => 'Please, provide a Valid Email address!'
   ),
   'max128Characters' => array(
      'rule' => 'between',
      'options' => array('min' => 1, 'max' => 128),
      'message' => 'The Email address must have between 1 to 128 characters.'
   )
)
```

The list of Validations we can use in our Models is as follows:

- **notEmpty** => Won't allow empty values.

```
   array (
      'rule' => 'notEmpty',
      'message' => 'This field can not be empty'
   )
```

- **lettersOnly** => Only allow a-z characters.

```
   array (
      'rule' => 'letersOnly',
      'message' => 'This field can only have A-Z characters'
   )
```

- **isAlphanumeric** => Only allow a-z0-9 characters.

```
   array (
      'rule' => 'isAlphanumeric',
      'message' => 'This field can only have letters and numbers'
   )
```

- **isNumeric** => Only allow numeric values.

```
   array (
      'rule' => 'isNumeric',
      'message' => 'This field only allow numeric values'
   )
```

- **isBoolean** => Only allow TRUE/FALSE.

```
   array (
      'rule' => 'isBoolean',
      'message' => 'This field only allow true/false'
   )
```

- **isTime** => Checks if is a valid Time.

```
   array (
      'rule' => 'isTime',
      'message' => 'This field only allow a valid Time'
   )
```

- **inList** => Only allow values set in a list.

```
    array (
        'rule' => 'inList',
        'options' => array('values' => array('male', 'female')),
        'message' => 'This field only accept values: male and female'
    )

- extension => Only allow file extensions set in a list.
    array (
        'rule' => 'extension',
        'options' => array('extensions' => array('jpg', 'png', 'gif')),
        'message' => 'This field only accept file extensions: jpg, png and gif'
    )

- mimetype => Only allow file mimetypes set in a list.
    array (
        'rule' => 'mimetype',
        'options' => array("image/jpg", "image/jpeg", "image/png", "image/gif")),
        'message' => 'This field only accept file mimetypes: image/jpg, image/jpeg, image/png and image/gif'
    )

- filesize => Only allow filesizes under a maximum value.
    array (
        'rule' => 'filesize',
        'options' => array('value' => 30000),
        'message' => 'This field only accept files with size under 30K'
    )

- comparison =>Does numerical or string length validations.
    array (
        'rule' => 'comparison',
        'options' => array('operator' => ">=", 'value' => 18),
        'message' => 'This field only accept ages 18 or more'
    )

- between => Does numerical or string length validations between a range.
    array (
        'rule' => 'between',
        'options' => array('min' => 1, 'max' => 128),
        'message' => 'This field only accept 1 to 128 characters'
    )

- isMail => Only allow valid Mail addresses.
    array (
        'rule' => 'isMail',
        'message' => 'This field only accept valid Email Addresses'
    )

- isURL => Only allow valid URLs.
    array (
        'rule' => 'isURL',
        'message' => 'This field only accept valid URLs'
    )
```

- **isIP** => Only allow valid IP addresses (IPv4 and IPv6)

```
array (
    'rule' => 'isIP',
    'message' => 'This field only accept valid IPs'
)
```

- **isChecked** => Only allows a checkbox that is checked. (i.e: "Accept terms and conditions")

```
array (
    'rule' => 'isChecked',
    'message' => 'Please, accept the terms and conditions'
)
```

- **regEx** => Let's you build your own validation regular expression.

```
array (
    'rule' => 'regEx',
    'options' => array('pattern' => '/^R/'),
    'message' => Only names starting with "R"'
)
```

- **isUnique** => Check that the value doesn't exist already (**only on SQL Models**)

```
array (
    'rule' => 'between',
    'message' => 'This entry already exists in the database'
)
```

Now lets take a look at a **NoSQL RIAK Database Model**... Pay attention to the areas marked in **violet**.

Ubuntu > **nano /var/www/mywebapp/models/DomainModel.php**

```
<?php
 /*
  * File: {APP}/models/DomainModel.php
  *
  * phpCloud: PHP Framework for Cloud sites ( http://phpcloudframework.com )
  * Author: Antonio Rohman Fernandez ( rohman{at}mahalostudio.com )
  * Copyright 2010-2011, MahaloStudio. ( http://mahalostudio.com )
  *
  * Licensed under The MIT License ( http://www.opensource.org/licenses/mit-license.php )
  * Redistributions of files must retain the above copyright notice.
  */

  require_once(COREPATH.DS.'models'.DS.'riakmodel.class.php');

  class DomainModel extends RiakModel {

   /*
    * Riak Cluser Connection for this Model.
    * this variable is OPTIONAL, if removed, Constant RIAK_DEFAULT_CLUSTER will be used as default.
    */
```

```
    protected $_cluster = 'main';

  /*
    * Riak Bucket for this Model. ( Mandatory variable )
    * specify in which bucket is the data stored.
    * in Riak, a Bucket is something similar to a 'table' in relational databases.
    */
    protected $_bucket = 'domains';

   /*
    * Riak Key field for this Model.
    * specify which is the field that will be used as 'key' for this Model.
    * this variable is OPTIONAL, if removed, 'id' will be used as default.
    */
    protected $_key = 'id';

   /*
    * Select if you want to encrypt your bucket.
    * this variable is OPTIONAL, if removed, 'false' will be used as default or the value of RIAK_ENCRYPTION
    * ** CAUTION: Don't change the value of this variable if data has been already inserted into the bucket,
phpCloudFramework will not be able to find it! **
    */
    protected $_encrypt = false;

    protected $_validate = array(
     /* Rules for field [host] */
     'host' => array(
      'not_null' => array(
       'rule' => 'notEmpty',
       'message' => 'error_Host_Empty'
      ),
      'must_be_a_url' => array(
       'rule' => 'isURL',
       'message' => 'error_URL_Invalid'
      ),
      'max_256_chars' => array(
       'rule' => 'comparison',
       'options' => array('operator' => "<=", 'value' => 256),
       'message' => 'error_256Chars'
      )
     ),
     /* Rules for field [frequency] */
     'frequency' => array(
      'accepted_values' => array(
       'rule' => 'inList',
       'options' => array('values' => array(720, 1440)),
       'message' => 'error_Invalid_Frequency'
      )
     )
    );
```

```
} /* Class: DomainModel */
```

Example RIAK model in {APP}/models/DomainModel.php

Now let's see the differences:

1. RIAK Database Selection

```
require(COREPATH.DS.'models'.DS.'riakmodel.class.php');
class DomainModel extends RiakModel { ... }
```

At this time, phpCloud Framework can **ONLY** support RIAK as NoSQL database, but drivers for other NoSQL databases like Cassandra could be built and use in the same way:

```
require(COREPATH.DS.'models'.DS.'cassandramodel.class.php');
class AnotherModel extends CassandraModel { ... }
```

2. RIAK Database Setup

```
protected $_cluster = 'main';
protected $_bucket = 'domains';
protected $_key = 'id';
protected $_encrypt = false;
```

Remember that in **{APP}/config/riak.php** we specified different clusters and that we named our Main cluster as 'main'? **$_cluster** variable is used to tell the web application which of those clusters to use for this model, so it knows that it needs to save/query data there. NoSQL databases are schema-free, they don't have Tables and Rows like SQL databases do... NoSQL databases are Key->Value datastores and save those keys containing the values as a JSON objects into "virtual tables" called "buckets" so, **$_bucket** variable is used to specify the bucket to store data and **$_key** is to specify the field that will become the key for the data, "$_key" is optional, if not set, the web application will set it to "id" automatically.

What does that mean!? Imagine the example below, imagine a SQL table called "users" to store this 3 fields:

```
$userData['User']['id'] = 'Rohman';
$userData['User']['name'] = 'Antonio Rohman Fernandez';
$userData['User']['age'] = 30;
$this->User->save($userData);
```

In RIAK it would be the same code as above, but would be translated to:
POST {"id":"Rohman","name":"Antonio Rohman Fernandez","age":"30"} > http://{RIAK}:8098/riak/users/Rohman

Bucket = users
Key = Rohman
Value = {"id":"Rohman","name":"Antonio Rohman Fernandez","age":"30"}

The variable **$_encrypt** is for encrypting the buckets and keys on saving, but is not recommended to use it.

3. RIAK SQL Relationships

```
echo 'you had been tricked!';
```

Wait! What? Shouldn't NoSQL database don't have SQL Relationships? Yeah! That's the point... you **can not** use RIAK Models in the same way you use SQL Models... so, $hasOne, $hasMany and $belongsTo are not available.

4.RIAK Validations

```
protected $_validate = array(...);
```

Validations works exactly the same in RIAK as it does in SQL... only difference is that rule '**isUnique**' is not available.

Controllers

Controllers take care of the programming, logic, functions and data manipulation.

See the example below to understand how controller work. Pay attention to the areas marked in **violet**.

Ubuntu > **nano /var/www/mywebapp/controllers/PhotoController.php**

```
<?php
 /*
  * File: {APP}/controllers/PhotoController.php
  *
  * phpCloud: PHP Framework for Cloud sites ( http://phpcloudframework.com )
  * Author: Antonio Rohman Fernandez ( rohman{at}mahalostudio.com )
  * Copyright 2010-2011, MahaloStudio. ( http://mahalostudio.com )
  *
  * Licensed under The MIT License ( http://www.opensource.org/licenses/mit-license.php )
  * Redistributions of files must retain the above copyright notice.
  */

 class PhotoController extends Controller {

  /*
   * Add Models, Helpers and Components here.
   */
  public $models = array('photo');
  public $helpers = array('html', 'localization', 'jquery');
  public $components = array('localization', 'cache', 'session');

  public function upload() {
    if (isset($this->data)) {
      if ($this->Photo->validates($this->data)) {
        $this->data['Photo']['id'] = null;
        $this->data['Photo']['owner'] = $this->session->read('Account.User');
        $this->Photo->save($this->data);
        $this->set('success', true);
```

```
    } else {
      $this->set('errors', $this->Photo->invalidFields);
    }
   }
  } /* function: upload() */

  public function display($photo_id = null) {
   $photoData = $this->Photo->find($photo_id);
   If (is_array($photoData)) {
    $this->set('photo', $photoData);
   } else {
    $this->phpCloud->redirect('/error/404');
   }
  } /* function: display() */

  public function ajaxtest() {
   $this->setLayout('ajax');
   $this->set('foo', 'bar');
  } /* function: ajaxtest() */

 } /* Class: PhotoController */
```

Example controller in {APP}/controllers/PhotoController.php

Now let's understand the code:

1. Models, Helpers and Components

```
  public $models = array('photo', 'camera', 'photographer');
  public $helpers = array('html', 'localization', 'jquery');
  public $components = array('localization', 'cache', 'session');
```

In the controllers we can specify which elements to use in both the Controller and its Views. **$models** and **$components** are Controller-variables (that means that we use them in the Controller's code) and **$helpers** is a View-variable (that means that we use them in the View's code).

The **$models** variable loads the Models we want to use, i mean, from which tables/buckets we want to query/save data... In this example we are only loading the *photo* model, but we could be loading several models. We will be able to access this model as **$this->Photo** throughout all the Controller, to query, validate, store, delete...

The **$components** variable loads optional packages like Session, Cache, Localization or any self made component. If you want to store something in the Session, you could do it like **$this->Session->write('User.name', 'Rohman');**

The **$helpers** variable loads optional packages that can be used in the Views to help writing code faster. We **can not** use the helpers in the controller, but for example, we could use **<?=$html->CSS(array('mycssfile', 'anothercss'));?>** in the HTML View to include 2 CSS files instead of having to write:

<link rel="stylesheet" type="text/css" href="/css/mycssfile.css" />
<link rel="stylesheet" type="text/css" href="/css/anothercss.css" />

2. Controller's Functions

```
public function upload() {...}
public function display($photo_id = null) {...}
```

To understand the MVC architecture is really important to know what the Controller's functions are and how are they rendered as Views. For example, if you point your browser to http://mywebapp/**photo**/**upload**, you will be actually calling the **upload()** function of the **photo** Controller. And a view **/views/Photo/upload.php** will show on screen the HTML and data provided by the Controller. Parameters can be passed to the functions in the following way http://mywebapp/**photo**/**display**/**DSC-0001**. If you check the display() function in the Controller, you will easily guess that $photo_id will be equal to 'DSC-0001'. So we can query this photo in the **Controller** through the **Model** and display it in the **View**.

3.Setting Data for the Views

```
$this->set('success', true);
$this->set('errors', $this->Photo->invalidFields);
$this->set('photo', $photoData);
```

The Controller can query, prepare, transform data in the logic part of the application and send back the data ready to be printed on screen by the View. Also we can set boolean variables to let the View know if some action has been successfully executed, if there are errors, etc... actually... we can send anything we want to the view: variables, messages, booleans, arrays, objects... however, views should be limited as much as possible to just display data... the Controller should be the one doing the coding part.

In this example we could generate a view (**/views/User/upload.php**) like:

```
<? if ($success): ?>
 <div>The photo has been uploaded successfully!</div>
<? else: ?>
 <div>The photo couldn't be uploaded, please check the following errors:</div>
 <div>
 <? foreach ($errors as $key => $message): ?>
  <div><?=$message?></div>
 <? endforeach; ?>
 </div>
<? endif; ?>
```

4.Global phpCloud Object

```
$this->phpCloud->redirect('/error/404');
```

Controllers also have the ability to use framework's global variable **$this->phpCloud** to perform some special actions like redirecting to other controller/function, encrypting, etc. The following functions are available:

- **redirect($url);** => Redirects the user to another page.
- **errorMessage($message);** => Sets an error message to show in **/error/dev** (not for Production mode)
- **componentObject($class);** => Loads a Component On-Demand
- **loadHelper($class);** => Loads a helper On-Demand
- **encrypt($data);** => Encrypts data using the defined **SALT** in {APP}/config/config.php

- **decrypt($data);** => Decrypts data using the defined **SALT** in {APP}/config/config.php

5.Layouts

```
$this->setLayout('ajax');
```

Layouts are HTML templates in where our views will be rendered inside. Layouts are stored in **{APP}/layouts** and set in the controller for the view to be rendered in the selected layout. If no layout is selected, the default layour will be used. The default layout is **default.php** (the default layout can be changed in **{APP}/config/config.php**). Layouts are useful to simplify view's HTML code... allowing you not to repeat code, no need to write the HTML headers and footers all the time, etc... Is also useful to display different kinds of data... in this way you can easily have a template for public pages (not logged in) and another for private pages (after logging in), or to render different kinds of data... you can have a RSS layout that converts your data in RSS type XML page... or an ajax template that won't re-create the headers/footers/design of the normal pages, etc... You can create as many layouts as you need placing them in **{APP}/layouts** and loading them in the controller like **$this->setLayout();**

Views

Views take care of the final representation in HTML/CSS/JS.

See the example below to understand how views work. Pay attention to the areas marked in **violet**.

Ubuntu > **nano /var/www/mywebapp/views/User/login.php**

```
<div class="floatLeft" style="width: 350px;">
 <div class="spacer"></div>
<? if (isset($error)): ?>
 <div style="height: 30px;">
  <div id="errorMsg" class="error size11"><?=$localization->getKey('error_Login');?></div>
  <script type="text/javascript">setTimeout("jQuery('#errorMsg').fadeOut('1000');", 4000);</script>
 </div>
<? else :?>
 <div style="height: 30px;"></div>
<? endif; ?>
<?=$html->startForm('User', array('action' => 'login', 'enctype' => 'multipart/form-data'));?>
 <div class="floatLeft size11" style="padding-left: 5px; text-align: right;">
  <div>
   <?=$html->input('text', 'User.username', array('maxlength' => 100, 'size' => 30, 'class' => 'formInput', 'label' =>
$localization->getKey('label_username')));?>
  </div>
  <div class="spacer"></div>
  <div>
   <?=$html->input('password', 'User.password', array('maxlength' => 100, 'size' => 30, 'class' => 'formInput',
'label' => $localization->getKey('label_password')));?>
  </div>
  <div><?=$html->link($localization->getKey('forgot_password'), '/user/reset');?></div>
 </div>
 <div class="floatLeft" style="padding-left: 10px;">
  <?=$html->input('image', 'Form.loginButton', array('src' => '/img/sym-search-arrow.gif', 'title' => $localization-
```

```
>getKey('login')));?>
 </div>
 <div class="clear"></div>
 </form>
</div>
<div style="float: left;">
 <?=$html->image('banner_welcome.png')?>
 <div>
  <div style="position: absolute; width: 636px; text-align: center; top: 260px;">
   <a href="/keysurvey"><img src="/img/apply_button.png"></a>
  </div>
 </div>
</div>
<div class="clear"></div>
```

Example controller in {APP}/views/User/login.php

Now let's understand the code:

1.Simply HTML

```
<div class="floatLeft" style="width: 350px;">
<script type="text/javascript">setTimeout("jQuery('#errorMsg').fadeOut('1000');", 4000);</script>
```

Yes, you are right! The views are just HTML pages that we render inside a Layout (template) and you can write any CSS, JavaScript, AJAX, HTML, etc... as you want! However, no need to write all the header, body, etc... that should be done by the layout, and the View should have just the necessary code to display what we want.

2.PHP and Controller's variables

```
<? if ($error): ?>
```

As explained in the previous section, the controller is able to send data to the view **$this->set('error', true);** to display data or generate some logic in the View.

3.Using Helpers

```
<?=$localization->getKey('error_Login');?>
<?=$html->startForm('User', array('action' => 'login', 'enctype' => 'multipart/form-data'));?>
<a href="/keysurvey"><img src="/img/apply_button.png"></a>
```

Helpers are optional and are set up in the Controller, but are great to save us writing long lines of code, but if prefer, you can just write normal HTML like **<img src="/img/apply_button.png">** instead of **<?=$html->link($html->image('apply_button.png'), '/keysurvey')?>**.

4. Using the Models

As we learned before, at this moment there are 2 different kinds of Models (SQL and RIAK). Their usage is fairly similar, except for special limitations and features... but in general, they can be used in the exact same way.

Inserting Data

Both SQL and RIAK save data in the same way, however there are slightly differences. Let's see how to add a new Domain for user "rohman" in a SQL database:

```
$domainData = array();
$domainData['Domain']['id'] = ''; /* NOTE: we can specify a manual ID, if not set, database will take care of it */
$domainData['Domain']['hostname'] = 'www.mahalostudio.com';
$domainData['Domain']['owner'] = 'rohman';
$this->Domain->save($domainData);
```

In a SQL world we can imagine that 'id' will be an **auto_increment** or a **serial** generated by the database, and that owner is a foreign key to users table.

> SELECT * FROM **users**;

ID	Fullname	Age	Nationality
rohman	Antonio Rohman Fernandez	30	Spanish

> SELECT * FROM **domains** WHERE owner = 'rohman';

ID	Hostname	Owner
1	www.mahalostudio.com	rohman

Now let's see how to do the same but for RIAK:

```
$domainData = array();
$domainData['Domain']['id'] = UUID::mint(4)->string; /* NOTE: to avoid collisions, is better to use UUIDs as Keys */
$domainData['Domain']['hostname'] = 'www.mahalostudio.com';
$domainData['Domain']['owner'] = 'rohman';
$domainData['Domain']['RIAK']['indexes'] = array('x-riak-index-owner_bin: rohman');
$this->Domain->save($domainData);
```

As you can see, is almost the same, but in RIAK it doesn't exist foreign Keys and SQL Relationships, so we have to build **indexes** in order to simulate that behavior.

You can set up as many indexes as queries you want to make, however, you can only query 1 index at a time... is not possible to query multiple indexes (maybe in a future)... also is important to notice the "binary" or "integer" modes of the index **owner_bin** means is a binary/text index, **date_int** would mean a numeric index.

Tip: we can do range queries on numeric indexes! **date_int/2011/2012** (would give all year's result!)

The relationships in RIAK would look something like this:

> GET http://{RIAK_SERVER}:8098/riak/**users**/rohman

```
{"id":"rohman","fullname":"Antonio Rohman Fernandez","age":"30","nationality":"Spanish"}
```

> GET http://{RIAK_SERVER}:8098/**buckets**/**domains**/**index**/**owner_bin**/rohman

```
{"keys":["ddfca9ea-dd35-4e38-520eeebead70"]}
```

> GET http://{RIAK_SERVER}:8098/riak/**domains**/ddfca9ea-dd35-4e38-520eeebead70

```
{"id":"ddfca9ea-dd35-4e38-520eeebead70", "hostname":"www.mahalostudio.com", "owner":"rohman",
"RIAK":{"indexes":["x-riak-index-owner_bin: rohman"]}}
```

Updating Data

Updating Data is as easy as Inserting data... In the **SQL Model** is just the same code, just if the ID exists, it will update the data... and if the ID doesn't exist, it will create a new entry. Let's see the code for SQL:

```
$domainData = array();
$domainData['Domain']['id'] = 1;
$domainData['Domain']['hostname'] = 'www.pupcloud.com';
$domainData['Domain']['owner'] = 'rohman';
$this->Domain->save($domainData);
```

As you can see, is just the same as before but we need to provide the **ID** for the item to be updated and **ONLY** the fields to be updated (the 'owner' field doesn't need to be present... we can remove it).

In RIAK, however, is a bit different... we need to provide the **ID** but also the previous data or it will be lost:

```
$domainData = $this->Domain->find('ddfca9ea-dd35-4e38-520eeebead70');
$domainData['Domain']['hostname'] = 'www.pupcloud.com';
$this->Domain->save($domainData);
```

As you can see, we are using the Model's **find()** function to get the Domain's data and indexes, updating the fields we want to update and saving the data. Easy, right?

Finding Data

The Model's **find()** function is one of the most powerful functions in the framework, and even it tries to work as close as possible in both SQL and RIAK, it has many differences (advance modes)... however, they are easy to use.

SQL: $userData = $this->**User**->**find**('rohman');
> SELECT * FROM **users**;

ID	Fullname	Age	Nationality
rohman	Antonio Rohman Fernandez	30	Spanish

RIAK: $userData = $this->**User**->**find**('rohman');
> GET http://{RIAK_SERVER}:8098/riak/**users**/rohman

{"id":"rohman","fullname":"Antonio Rohman Fernandez","age":"30","nationality":"Spanish"}

So far, so good! both SQL and RIAK do the same and receive the same:

```
$userData = Array (
   ['User'] => Array (
      ['id'] => 'rohman',
      ['fullname'] => 'Antonio Rohman Fernandez',
      ['age'] => '30',
      ['nationality'] => 'Spanish'
   )
)
```

One difference between RIAK and SQL is that in RIAK we need to store the indexes when saving objects... if we do the same for the **domain** object, we will have the following differences:

SQL: $domainData = $this->**Domain**->**find**(1);
> SELECT * FROM **domains** WHERE id = 1;

ID	Hostname	Owner
1	www.mahalostudio.com	rohman

RIAK: $domainData = $this->**Domain**->**find**('ddfca9ea-dd35-4e38-520eeebead70');
> GET http://{RIAK_SERVER}:8098/riak/**domains**/ddfca9ea-dd35-4e38-520eeebead70

{"id":"ddfca9ea-dd35-4e38-520eeebead70", "hostname":"www.mahalostudio.com", "owner":"rohman", "RIAK":{"indexes":["x-riak-index-owner_bin: rohman"]}}

We will find the indexes in a "RIAK" array inside the Domain's object:

```
$domainData = Array (
   ['Domain'] => Array (
      ['id'] => 1,
      ['hostname'] => 'www.mahalostudio.com',
      ['owner'] => 'rohman'
   )
)
```

```
$domainData = Array (
   ['Domain'] => Array (
      ['id'] => 1,
      ['hostname'] => 'www.mahalostudio.com',
      ['owner'] => 'rohman',
      ['RIAK'] => Array (
         ['indexes'] => Array (
            [0] => x-riak-index-owner_bin: rohman
         )
      )
   )
)
```

There is one more difference to take into consideration here, the SQL Relationships **hasOne**, **hasMany** and

belongsTo that we can set up in the SQLModel and that are not available in RIAK... so, in the example above, if we have the Relationship **Domain belongsTo User**, we will receive the following:

```
$domainData = Array (
   ['Domain'] => Array (
      ['id'] => 1,
      ['hostname'] => 'www.mahalostudio.com',
      ['owner'] => 'rohman',
      ['belongsTo'] => Array (
         ['User'] = Array (
            ['id'] => 'rohman',
            ['fullname'] => 'Antonio Rohman Fernandez',
            ['age'] => '30',
            ['nationality'] => 'Spanish'
         )
      )
   )
)
```

```
$domainData = Array (
   ['Domain'] => Array (
      ['id'] => 1,
      ['hostname'] => 'www.mahalostudio.com',
      ['owner'] => 'rohman',
      ['RIAK'] => Array (
         ['indexes'] => Array (
            [0] => x-riak-index-owner_bin: rohman
         )
      )
   )
)
```

Ok, cool! but... how to query **ALL** domains that belongs to Rohman? SQL and RIAK are pretty different here... lets see an example:

SQL: $domains = $this->**Domain**->**find**('**all**', array(**'conditions' => array('owner' => 'rohman')**));
> SELECT * FROM **domains** WHERE owner = 'rohman';

ID	Hostname	Owner
1	www.mahalostudio.com	rohman
2	www.pupcloud.com	rohman

RIAK: $domains = $this->**Domain**->**find**('**index**', array(**'owner_bin' => 'rohman'**));
> GET http://{RIAK_SERVER}:8098/buckets/**domains**/**index**/**owner_bin**/rohman

```
{"keys":["ddfca9ea-dd35-4e38-520eeebead70","d9fjdf4-d9g3-s4hs-4mf3k2sng4h"]}
```

```
$domains = Array (
   [0] =>  Array (
      ['Domain'] => Array (
         ['id'] => 1,
         ['hostname'] => 'www.mahalostudio.com',
         ['owner'] => 'rohman'
      )
   ),
   [1] =>  Array (
      ['Domain'] => Array (
         ['id'] => 2,
         ['hostname'] => 'www.pupcloud.com',
         ['owner'] => 'rohman'
      )
   )
)
```

```
$domains = Array (
   ['Domain'] => Array (
      ['_keys'] => Array (
         [0] => ddfca93a-dd35-4e38-520eeebead70,
         [1] => xd9fjdf4-d9g3-s4hs-4mf3k2sng4h
      )
   )
)
```

As you can see, the response is pretty different... while SQL gives all domains and data fields for us to use immediately, RIAK will only provide us the keys of the objects, if we want to provide data from the objects, we will have to use **$this->Domain->find($key);**.

So... RIAK model doesn't support **find('all')**? That's not right, in fact, we can use find('all') in RIAK to trigger a MapReduce job that will end up giving the same data as the SQL Model... **HOWEVER!**... you may want to avoid doing it... MapReduce is extremely costly and you may only want to use that as part of a back-end scheduled job rather than an On-Demand query in the UI... so the use of indexes is recommended.

Tip: As tipped before, numeric ranges can be queried in numeric indexes... this is how:

RIAK: $scans = $this->**Scan**->**find**('**index**', array(**'scanned_int' => array('20111101','20111130')**));
> GET http://{RIAK_SERVER}:8098/buckets/**scans**/**index**/**scanned_int**/20111101/20111130

```
{"keys":["21cc4f07-6374-42eb-ac73-cf4b4788dbd7","f9fjdmc-43d6-98dh-fm3jlk8f12s"]}
```

This example will give us all scans done between 2011/11/01 and 2011/01/31 (they way you save the timestamps is totally up to you... those could be UNIX timestamps instead of more-readable YYYYMMDD.

There are plenty of options that can be used and also can be combined to create more complex queries in **find()** for SQL Models... let's explore them:

- **fields** => Provides only a subset of fields

SQL: $users = $this->**User**->**find**('**all**', array(**'fields' => array('id', 'fullname', 'age')**));
> SELECT **id, fullname, age** FROM **users**;

- **conditions** => Provides rows that matches certain conditions

SQL: $users = $this->**User**->**find**('**all**', array(**'conditions' => array('id' => 'rohman')**));
> SELECT * FROM **users** WHERE **id = 'rohman'**;

We can go even further with the conditions array and do more complex stuff:

```
SQL: $users = $this->User->find('all', array('conditions' => array('age' => array('value' = array(20, 30),
                                                                                  'operator' = 'BETWEEN',
                                                                                  'type' => PDO::PARAM_INT
                                                                                 ),
                                                                    'nationality' => array('value' => 'Spanish',
                                                                                           'condition' => 'OR',
                                                                                          )
                                                                   )
                                                    )
                                        );
> SELECT * FROM users WHERE (age BETWEEN 20 AND 30) OR (nationality = 'Spanish');
```

The available **keywords** and **options** for '**conditions**' are:
- **condition**: **AND, &&, OR, ||**
- **operator**: **LIKE, NOT LIKE, IN, NOT IN, IS NULL, IS NOT NULL, BETWEEN, NOT BETWEEN, =, <, >, !=, <>, <=, etc...**
- **type**: PDO type checking for more security: **PDO::PARAM_INT, PDO::PARAM_STR, PDO::PARAM_BOOL, etc...**
- **value**: a value to compare or an array of values for range/list comparison.

- **order** => Return an ordered result-set

SQL: $users = $this->**User**->**find**('**all**', array(**'order' => 'User.fullname ASC'**));
> SELECT * FROM **users** **ORDER BY fullname ASC**;

- **limit** => Return a limited result-set

SQL: $users = $this->**User**->**find**('**all**', array(**'limit' => 10**));
> SELECT * FROM **users** **LIMIT 10**;

- **page** => Provides pagination on a limited result-set

SQL: $users = $this->**User**->**find**('**all**', array(**'limit' => 10, 'page' => 2**));
> SELECT * FROM **users** **LIMIT 10 PAGE 2**;

- **relationships** => Can set relationships on-the-fly instead of using the ones defined in the Model!

```
SQL: $userData = $this->User->find('rohman', array('relationships' =>  array('hasMany' => array('Domain' =>
                                                    array('table' = 'domains', 'relationship' => array('id' => 'owner'))
                                                                                                    )
                                                                                )
                                                    )
                                        );
> SELECT * FROM users WHERE id = 'rohman'; // But will also return all domains associated to 'rohman'
```

Tip: The entity **'relationship' => array('id' => 'owner')** is equivalent to **user.id = domain.owner**.

- **extended** => This **disables ALL relationships** made in the Model. In this case it will return **only the users**.

SQL: $users = $this->**User**->**find**('**all**', array(**'extended' => false**));
> SELECT * FROM **users**; // No relationships returned, neither the ones defined in the Model

- **cache** => Tries to read from the cache before asking the database.

$userData = $this->**User**->**find**('**rohman**', array(**'cache' => 5**));
> If the data exists in the cache, it will read it from there, if not, will get the data from database, and save it in the cache for **5 minutes**.

$userData = $this->**User**->**find**('**rohman**', array(**'cache' => array('type' => 'memcached', 'expiration' => 2)**));
> This time we will use **memcached** instead of the default cache, and save the data for only 2 minutes.

- **indexes** => Will return **numeric indexes** along with fieldnames

SQL: $userData = $this->User->find('rohman', array('indexes' => true));

```
$userData = Array (
   ['User'] => Array (
      [0] => 'rohman',
      ['id'] => 'rohman',
      [1] => 'Antonio Rohman Fernandez',
      ['fullname'] => 'Antonio Rohman Fernandez',
      [2] => '30',
      ['age'] => '30',
      [3] => 'Spanish'
      ['nationality'] => 'Spanish'
   )
)
```

There are several ways we can call the **find()** function:

- Querying the ID field => $this->User->find('rohman');
- Querying a non-ID field => $this->User->find(**array('nationality' => 'Spanish')**);
- Querying all rows for a condition (or no condition) => $this->User->find('**all**');
- Querying **neighbors** (will be explained later)
- Custom **queries** (will be explained later)

What are neighbors? In a list, neighbors are the elements at the immediate left and right of the array. Imagine a photo album, where you display a selected photo and have navigation arrows to go to **Previous** and **Next** photos. That can easily done using find('neighbors');

SQL: $navigation = $this->Photo->find('neighbors', array('element' => array('filename' => 'DSC-0002'),
'order' => 'filename ASC'
)
);

```
$navigation = Array (
   ['Previous'] => Array (
      ['id'] => 1,
      ['filename'] => 'DSC-0001',
   ),
   ['Next'] => Array (
      ['id'] => 3,
      ['filename'] => 'DSC-0003',
   )
)
```

This will return all data for **Previous** and **Next** photos.

For other kinds of queries (like counting, joins, etc...), you can do traditional SQL queries with the **query()** function, for example:

SQL: $data = $this->**User**->**query**('SELECT count(id) FROM users');

Deleting Data

Deleting Data is very simple, and both SQL and RIAK works in the same exact way:

$this->**User**->**delete**('rohman');

This delete function only deletes 1 row at a time providing the ID, for other types of deletion we can use the **query()** function:

SQL: $this->**User**->**query**('DELETE FROM users WHERE expired = true');

Tip: Deleting using query() function is only available in SQL Models.

5. Components

Components add extra functionality to the Controllers. To use a component, place it inside the **components** folder of your phpCloud's core installation.

/etc/phpcloud/components

At this moment there are 3 components developed (localization, cache and session) but more components could be created by yourself or downloaded from http://www.phpcloudframework.com. To be able to use components you have to set it up in your controllers like **public $components = array('localization', 'cache', 'session');**

Cache

The cache component (**/etc/phpcloud/components/cache.class.php**) is used to store and retrieve data from the cache, this component is normally tied to your models' queries so you can avoid flooding your databases with requests, but it can also be used by the session component or independently.

We already learned how to use the Cache in the Model's **find()** function:

$userData = $this->**User**->**find**('**rohman**', array(**'cache' => array('type' => 'memcached', 'expiration' => 2)**));

This example will use **memcached** as cache driver and store the queries' data for **2 minutes**. If no 'type' and 'expiration' is set, it will read the default values from **{APP}/config/config.php**:

define('CACHE', true); <- Important to set the cache as true or won't work!
define('CACHE_DEFAULT_TYPE', 'disk'); <- The default type of cache to use
define('CACHE_SIZE', 1000000); <- 1GB is the default space reserved for cache data in 'disk' mode

define('CACHE_PURGE', 1440); <- data not accessed for 1440 minutes (1 day) will be purged automatically
define('CACHE_EXPIRATION', 30); <- 30 minutes is the default expiration time

Supported types of cache are '**disk**', '**apc**', '**memcached**'

- **disk** => Stores the data into files in **{APP}/tmp/cache**

- **apc** => Uses PHP's cache system.
To be able to use apc we need to install the extension:
Ubuntu > **aptitude install php-apc**

- **memcached** => Uses Memcached.
To be able to use memcached, we need to install it and its php extension:
Ubuntu > **aptitude install memcached php5-memcached**

To set some data into the cache, we can do it in the controller as follows:

```
$this->cache->set('a_key', 'some data to store', array('type' => 'memcached', 'expiration' => 5));
```

We need to specify a key name and the data we want to store, the options array is optional (always). In the same way, to read some data from the cache, we would do as follows:

```
$data = $this->cache->get('a_key', array('type' => 'memcached'));
```

And if we want to delete the data we simply do:

```
$this->cache->delete('a_key', array('type' => 'memcached'));
```

Session

The session component (**/etc/phpcloud/components/session.class.php**) is used to store data in the sessions. Sessions can be stored in the default PHP session object or using the cache component (in which then we will have: disk, apc and memcached to choose from). It works pretty similar to the cache component:

```
$this->session->write('a_key', 'data to save in the session');
```

If no options defined, it will take default values from **{APP}/config/config.php**:

```
define('SESSION_DEFAULT_TYPE', 'php');
define('SESSION_EXPIRATION', 1440);
```

```
$this->session->write('a_key', 'data_to_store', array('type' => 'cache', 'cachetype' => 'disk', 'expiration' => '2', 'owner' => 'rohman'));
```

When using the cache to store the sessions, we will be required to provide an '**owner**' attribute to add an extra layer of protection, only the right owner (user id? or something that determines the user) will be able to read it.

To **read** the data from the session, we will do as follows:

$this->**session**->**read**('a_key', array(**'type' => 'cache', 'cachetype' => 'disk', 'expiration' => '2', 'owner' => 'rohman'**));

In the same way, we will **delete** data as:

$this->**session**->**delete**('a_key', array(**'type' => 'cache', 'cachetype' => 'disk', 'expiration' => '2', 'owner' => 'rohman'**));

Additionally, we can **destroy** the full session for a user:

$ this->**session**->**destroy**(array(**'type' => 'cache', 'cachetype' => 'disk', 'expiration' => '2', 'owner' => 'rohman'**)) ;

Localization

The localization component (**/etc/phpcloud/components/localization.class.php**) is used to provide multi-language support to the application. We can set up a default language in **{APP}/config/config.php**:

define('DEFAULT_LANGUAGE', 'en_us'); <- American English

Language packs are created in **{APP}/localization** using languages code like 'en_us' for American English... if we want to have our application in 2 languages (American English and Spanish) we would create the following files:

{APP}/localization/**en_us**/languagepack.php
/rev20111120.php
/**es**/languagepack.php
/rev20111120.php

In languagepack.php we will find the following:

Ubuntu > **nano /var/www/mywebapp/localization/en_us/languagepack.php**

```
<?php
 /*
  * File: {APP}/localization/en_us/languangepack.php
  * - American English localization strings.
  *
  * phpCloud: PHP Framework for Cloud sites ( http://phpcloudframework.com )
  * Author: Antonio Rohman Fernandez ( rohman{at}mahalostudio.com )
  * Copyright 2010-2011, MahaloStudio. ( http://mahalostudio.com )
  *
  * Licensed under The MIT License ( http://www.opensource.org/licenses/mit-license.php )
  * Redistributions of files must retain the above copyright notice.
  */

 /*
  * Variables to identify the language pack.
```

```
 * These are optional, but usefull to know in which pack you are.
 */
 $localizationCode = "en_us";
 $localizationDetails = "American English";

 $localizationKeys = array();
/*
 * Test key for localization. {username} is a variable.
 * example -> $localizationKeys['welcome'] = "Hello {username}, Welcome to phpCloud!";
 * Called from a Controller with the Component: $this->localization->getKey("welcome", array("username" =>
$yourusername));
 * Called from a View with the Helper: $localization->getKey("welcome", array("username" => $yourusername));
 */
 require('rev20111120.php'); /* Include Language Keys from 2011/11/20 until next translation revision */
```

Example LanguagePack in {APP}/localization/en_us/languagepack.php

The languagepack.php is simply a wrapper to organize the translation revisions of the application, this is helpful to keep track of which languages has been translated and until which extend... imagine that our application has 10 available languages and we send the texts to a translator on 2011/11/30, then we can start a new revision to store new keys that will be translated later, so in our languagepack would look like:

require('rev20111120.php'); <- we sent this pack for translation on 2011/11/30.
require('rev20111130.php'); <- we start a new pack for further development and later translation.

In this way is very easy to track which languages are up to date and which keys are new, etc...

Ubuntu > **nano /var/www/mywebapp/localization/en_us/rev20111120.php**

```
<?php
 /*
  * File: {APP}/localization/en_us/rev20111120.php
  * - American English localization strings.
  * - Language Keys from 2011/11/20.
  *
  * phpCloud: PHP Framework for Cloud sites ( http://phpcloudframework.com )
  * Author: Antonio Rohman Fernandez ( rohman{at}mahalostudio.com )
  * Copyright 2010-2011, MahaloStudio. ( http://mahalostudio.com )
  *
  * Licensed under The MIT License ( http://www.opensource.org/licenses/mit-license.php )
  * Redistributions of files must retain the above copyright notice.
  */

 $localizationKeys['login'] = "Login";
 $localizationKeys['welcome'] = "Hello {username}, today is {date}.";
```

Example LanguagePack in {APP}/localization/en_us/rev20111120.php

The localization is handled through **keys** and **messages**, and the messages can include **variables** to provide dynamic messages. So how to use the localization keys we define in the language packs? As simply as follows:

```
echo $this->localization->getKey('login');
```

This will return '**Login**' as it shows in the '**en_us**' default language pack... so... how to provide messages in different languages? We can set the language pack to use at any time:

```
$this->localization->setLanguage('es'); <- will set the locale to Spanish, so all messages from now will be in Spanish.
echo $this->localization->getKey('login');
echo $this->localization->getKey('welcome', array('username' => 'Rohman', date => date('Y-m-d')));
```

So what's with the variables? different languages has different way to place nouns, verbs, etc... so variables are a great way to place dynamic data in your strings... let's see the 'welcome' message in English and Japanese:

```
$this->localization->setLanguage('jp'); <- switch to Japanese locale
echo $this->localization->getKey('welcome', array('username' => 'Rohman', date => date('Y-m-d')));
$this->localization->setLanguage('en_us'); <- switch back to American English locale
echo $this->localization->getKey('welcome', array('username' => 'Rohman', date => date('Y-m-d')));
```

This will give the following output:

```
Rohman さん、今日は 2011-11-22 です。
Hello Rohman, today is 2011-11-22.
```

As you can see, they are placed differently, but in our localization component is very easy to do so.