PCG Framework

Introduction to the phpCodeGenie Framework

by Nilesh Dosooye

10/20/2004

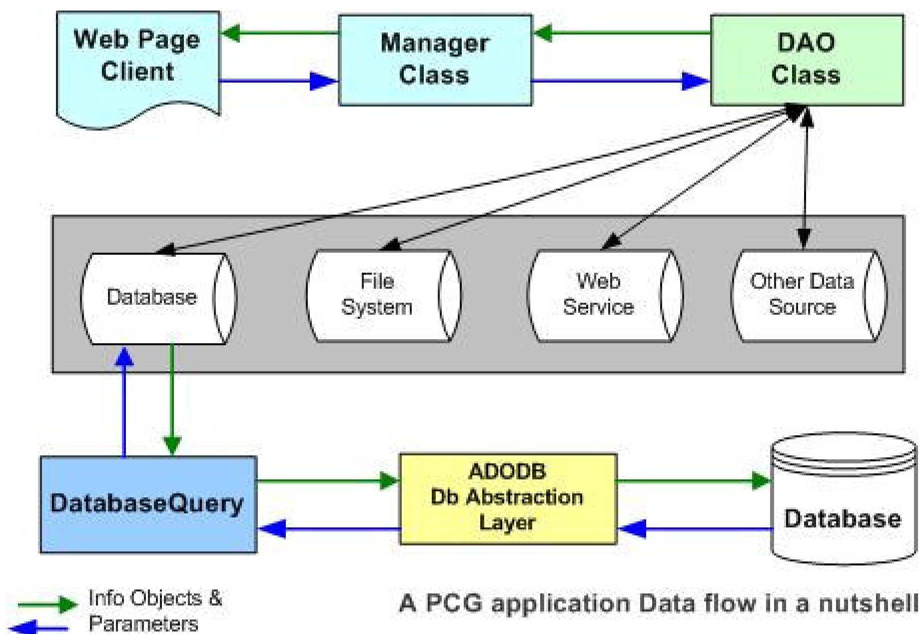
This document is licensed under the GNU GPL License

Introduction

Given any database schema, tables within, list of fields and properties, one ought to be able to build any kind of database backed applications in any programming language. The basic operations that a database backed application will have are usually the same - Create Records, Retrieve Records, Update Records, Delete Records (CRUD).

phpCodeGenie attempts to generate these CRUD code so that programmers have a base they can work on. The core database access code and scripts are usually the same for most applications. Rather than spending a lot of time doing these common code, we can spend our time on the business logic of our applications and let PCG do the boring code for you.

The PCG Framework aims at building object oriented php applications which are extensible, portable and scalable. This document will attempt to describe the different components in a PCG application.
The diagram below shows the PCG Framework in a nutshell.



A PCG application Data flow in a nutshell

Different components of your generated PCG application

The initial core code for your application can be generated by phpCodeGenie from your database and tables therein. phpCodeGenie will generate the files for your PCG application and save them to disk.

After phpCodeGenie generates the code for your PCG framework application, your initial directory structure will look like this.

PCG Application Directory Structure

```
Application Root
   |
   |----- app
   |          |------common
   |          |------lib
   |          |------tableClasses
   |
   |----- config
   |
   |----- include
   |
   |----- web
             |------common
             |------forms
```

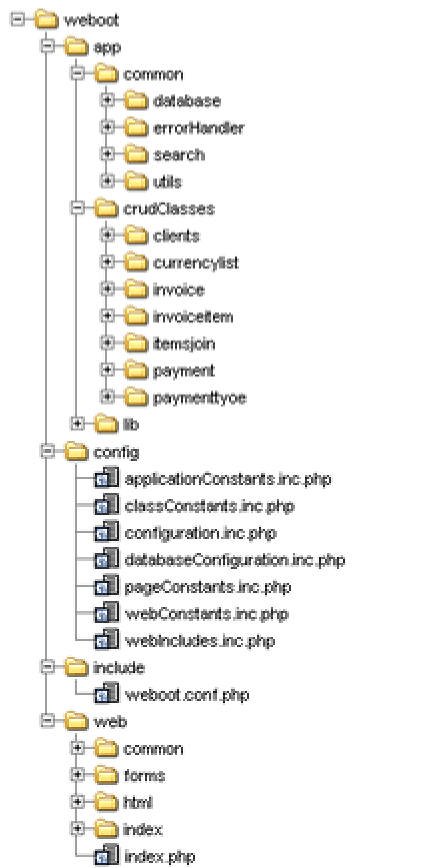
The 4 top level directories are app, config, include, web

app – Application Layer contains all the business and programming logic of your application. Files in the app directory are not accessed directly by the web clients from a browser. Clients from the web presentation layer make calls to code in this layer to be able to use them.

config – Configuration Layer contains all the configuration and constants definition for your application. It is very crucial and is the glue that ties the different components of a PCG application together.

include – this directory contains the main configuration file for your application. If you can use .htaccess to make this in the include path of your application its great or otherwise, you will need to copy the file here to the main php Include directory.

web – Presentation layer, contains all the User Interface sections of your application. Programs here are clients of the application layer of your application. They instantiate classes from the application layer to perform the business logic of your application and then output the resulting data on your web browser.

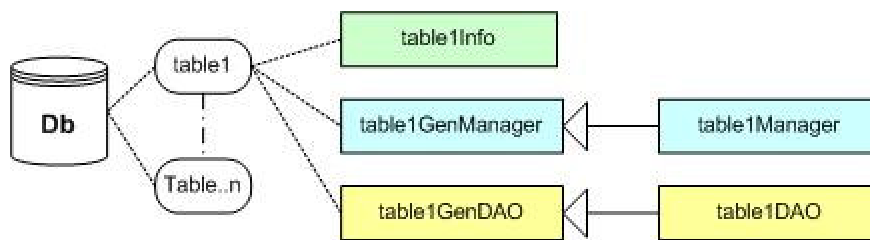


The above diagram is a snapshot of the directory structure generated by phpCodeGenie on initial code generation.

Application Layer Components

The application layer is the brains of your entire application. All the logic happens in here. There are many classes which interact together to make the application work. We will look into some of the crucial classes that a PCG application consists of.

The generator will create a directory called crudClasses under the app folder. This directory contains the code that performs CRUD operations for your application. For each table in your database, there will be a corresponding folder under the crudClasses directory, having the same name as the table. In each 'tableName' folder, there will be 5 files, each containing the code for a php class. These classes are the info, genDAO, DAO, genManager and Manager classes. For example for table called table1, we will have table1Info, table1GenDAO, table1GenManager, table1Manager and table1DAO.



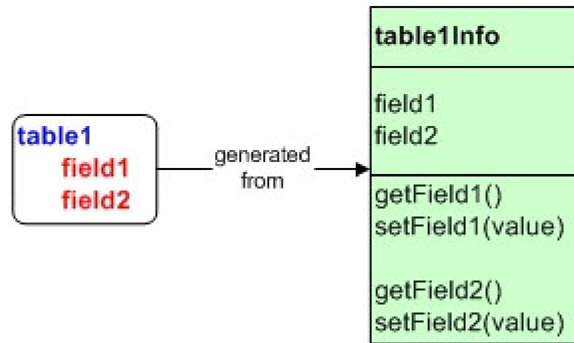
We will now look at each of these CRUD classes in details.

Info Class

An Info class is a php class that maps the fields in a database table, to attributes in a class (info class). Each table in a database has a corresponding Info class. Each of its field is represented in that Info class by a class attribute. For each of the latter, there are getter and setter methods defined for them. These attributes should only be accessed via their

respective getters and setters. These info classes are basically data containers. Their sole role is to carry data around. They do not have any business logic built into them.

The naming convention for the Info classes is ***tableName*****Info**.



The diagram above, shows the basic structure for an info class.

Info classes should NOT be manually changed, as it is intended to be regenerated if your table structure changes. You should be able to regenerate this class anytime from your database without affecting your running application.

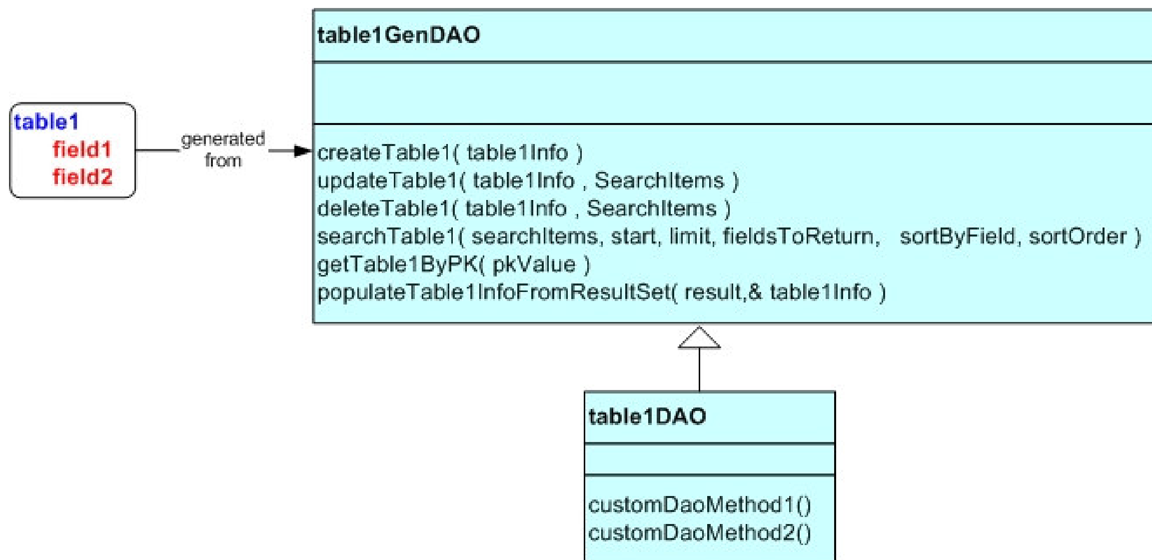
GenDAO and DAO Class

A DAO is defined as a Data Access Object. The GenDAO and DAO classes do exactly that - they access and manipulate data. The DAO is a subclass of the GenDAO class. The GenDAO class's role is to communicate with data sources that the application may need to send or retrieve data from. It will use other components to access the data sources. In a PCG database driven application, the GenDAO will access the database via the ***databaseQuery*** class which is a database abstraction layer class. Therefore, the latter can communicate seamlessly with many different databases. The generated GenDao class has code to communicate with a database as dataSource. If you want to change the implementation of how GenDao retrieves data or change the data source that it retrieve from (e.g flat file, a web service, or any other kind of data source), you can override all the methods of the GenDAO class in the DAO subclass. Your entire application will

only be instantiating new DAO classes. So, whichever methods you override in DAO, will be used instead of the super class methods.

The DAO class is the class where you will put custom DAO methods not available in the generic generated Code i.e. if you know of a better optimized query that you want to run straight on the database without passing through the PCG generated code framework, you will add a method for that in the DAO class. Any methods that need to access the Data Sources and that are not present in the application framework generated GenDAO, need to be put in the DAO class. The goal of this is to allow for regeneration of the GenDAO class anytime without losing custom code.

Naming conventions for these classes are ***tableName*****GenDAO** and ***tableName*****DAO**.



The diagram above shows the basic structure of the GenDAO and DAO classes.

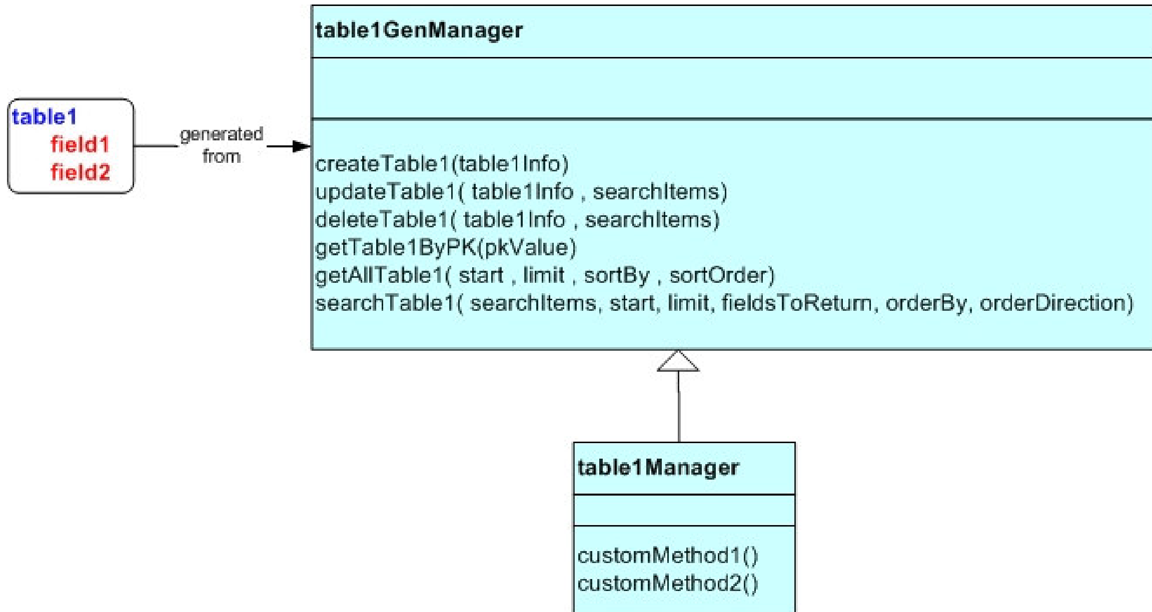
The GENDAO should NOT be manually changed, as it is intended to be regenerated if your table structure changes. Put all your custom methods in the DAO Class.

GenManager and Manager Class

The GenManager and Manager classes are facades for the front end pages to communicate with the DAO classes. Clients of the DAO (e.g. web pages) can only use the latter through the Manager classes. e.g. a page that needs to get a list of all rows in a table, will make a request to the Manager class which will in turn, request that information from the DAO classes. The advantage of this implementation is that if we decide to change the implementation of our DAO e.g. change the dataSource or use a commercial DAO package, we will not need to change the clients making requests to that layer. The clients just interact with the Manager classes and the Manager can instantiate whichever DAO package you might want to use, without having to change anything in the clients.

The basic generated GenManager class contains all basic CRUD operations on a table which is available for any client to use. If something is not available to you in the basic generated code, you can extend the GenManager class with added functionality by adding your custom code in the Manager class. You can override any of the methods in the GenManager class by rewriting them in the Manager subclass. Clients deal only with the Manager class. So, whichever methods you override in the latter will be used.

Naming conventions for these classes are ***tableName*****GenManager** and ***tableName*****Manager**.



The above diagram shows a basic structure for the GenManager and Manager classes

The GenManager class should NOT be manually changed, as it is intended to be regenerated anytime if your table structure changes. Put all your custom code in the Manager Class.

Include Directory

Main Application Configuration File

database.conf.php

Each PCG application has a main configuration file that is included in every file in the application. The default name for the configuration file would be ***databaseName*.conf.php** , but when generating the entire codebase for your application, you can specify a different configuration file name and this file will be included in all of your other code in the PCG framework.

This configuration file contains crucial information about your PCG generated application. Without this file, your PCG generated application will not work. The configuration file defines the path where your PCG generated application is installed, the URL to access it, the operating system used and the database connection settings. If you are moving your application from one server to another with different path, URL, database connections, operating systems, the only thing that you will need to change is this configuration file and your application will be all ready to be used.

This file needs to be in the phpInclude directory. Every file in your PCG generated application will include this configuration file

```
include_once("databaseName.conf.php");
```

as the first line of code. If you do not have access to the phpInclude directory, another way to make it seen from all your files is using an .htaccess file in the root of your application with the following directive in it

```
php_value include_path /path/to/includeFile/
```

Configuration Component

Constants File

'**define**' being a global action in php applications, once a constant is defined, it has global scope and can be used throughout an application, in functions, classes and pages. The PCG framework takes advantage of this global definition.

classConstants.inc.php

PHP does not have a package or import mechanisms. This limitation causes some inconveniences. To be able to use a class defined in another file, one has to 'include' or 'require' that file before instantiating that class. To include the file, one has to specify the full absolute path to that file. Problem arise when such an application has to be moved from one system to another where the paths may not be the same, making the developer having to scan all files to change the paths. To avoid that, in the PCG framework, the full absolute path to all classes existing in the system are defined in the ClassConstants.inc.php file.

Each class is defined in a similar format as below

e.g.
```
define("CLASS_EXAMPLE_CODE ",SITE_PATH.FILE_SEPARATOR."exampleCode.class.php");
```

after having defined your class in this file, a client can just use

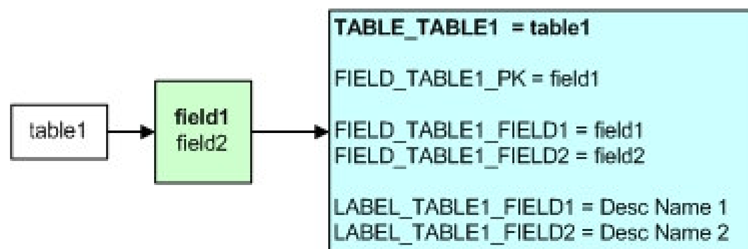
```
include_once(CLASS_EXAMPLE_CODE);
```

to include (import) and use that class.

For each new class that you add to the application, please add a definition line for it in this file.

databaseConstants.inc.php

The DatabaseConstants is a file mapping all tables that are being used in the application and the fields within to defined constants. The constants follow a strict naming convention. All tables are defined as - **TABLE_*tableName*** (e.g. TABLE_USER) and fields are defined as **FIELD_*tableName_fieldName*** (e.g. FIELD_USER_USERNAME). The primary key for each table is also defined in this constants file as **FIELD_*tableName*_PK**. For each Field, there is also another constant defined which is prefixed by LABEL_. This constant is used to store a more descriptive name for that particular field, i.e. the text that you want the user to see instead of the actual fieldname . Often times, what you name a field name in the database is not exactly what you want your users to see. So the LABEL_ constant is used in the User Interface to display headers or descriptive texts for these database fields. It makes it very easy to change the user interface field display text by changing it in this file. Also, it might be useful for localization of fieldnames in your application user Interface.



Any code in the PCG generated application which needs to reference to a table or fieldname does so by using the constants in this file.

pageConstants.inc.php

The same problem that we mentioned in the classConstants section about dependency of application on the system it resides, due to path and URL changes when moving to another system, also applies to URL addresses. All forms in your application will post to a script within the application. Sometimes, due to multi level directories, its difficult to reference to these scripts only by their location on the server. To avoid this problem, for any link or action post, we want to use the full URL addresses.

Naming convention for constants in this file is **PAGE_*pageName***. An example of a page definition would be.

```
define("PAGE_MY_PAGE",URL_ADDRESS."/myPage.php");
```

e.g if we have a form which posts to a script called processSomething.php found two directories above itself. Usually you would have it as

```
<form name = "thisForm" method="POST" action = "../../processSomething.php">
```

We do not want to have such kind of dependency on the directory structure. What we will do in a PCG application to avoid this is we will define the full URL for the process.php script in this page Constants file.

```
define("PAGE_PROCESS_SOMETHING",URL_ADDRESS."/dir1/dir2/processSomething.php");
```

Then in our form, we will use that full url as

```
<form name = "thisForm" method="POST" action="<? echo PAGE_PROCESS_SOMETHING;
?>">
```

The same idea applies to other elements referenced from the front end, e.g images, stylesheets etc.. instead of calling them directly by their relative path, define them by their full URL address in this file and then use these constants instead.

As the root URL is always defined in our main application configuration file, moving the application to another server will only entail changing that root URL in the main configuration file.

webIncludes.inc.php

Same as with the previous section, sometimes in your pages, you want to include another script or html page. So instead of referencing these includes by the relative paths, we define the full path to all these files you want to include in here and then use the defined constants in our code.

The naming convention for the Includes is **INC_*includeName***

An example definition in the file would be

```
define("INC_LOGIN_FORM",WEB_PATH.FILE_SEPARATOR."loginForm.php");
```

then in your code you would just use

```
include_once(INC_LOGIN_FORM);
```

The major difference between the WebIncludes and PageConstants is that the includes work with server paths and the pages work with URL addresses.

applicationConstants.inc.php

Application Constants store constants that are for your entire application. Some examples of the constants in this file are ApplicationName, Version, adminEmailAddress, errorHandling defaults. The default applicationConstants file that is built is as follows :-

```
define("TRUE_PARM","y");
define("FALSE_PARM","n");

define("MAX_UPLOAD_SIZE","50000");

define("APPLICATION_NAME","GATECOCO PCG");
```

```
define("APPLICATION_VERSION","{APPLICATION_VERSION}");
define("APPLICATION_ADMIN_EMAIL","{ADMINSITRATOR_EMAIL}");
define("APPLICATION_FROM_EMAIL","{FROM_EMAIL}");


define("ERROR_HANDLER_SEND_ADMIN_EMAIL_ON_ERROR",true);
define("ERROR_HANDLER_DISPLAY_ERROR_TO_USER",true);
define("ERROR_HANDLER_QUIT_PROGRAM_ON_ERROR",true);
define("ERROR_HANDLER_LOG_TO_FILE",true);
define("ERROR_HANDLER_LOG_FILE","error.txt");
```

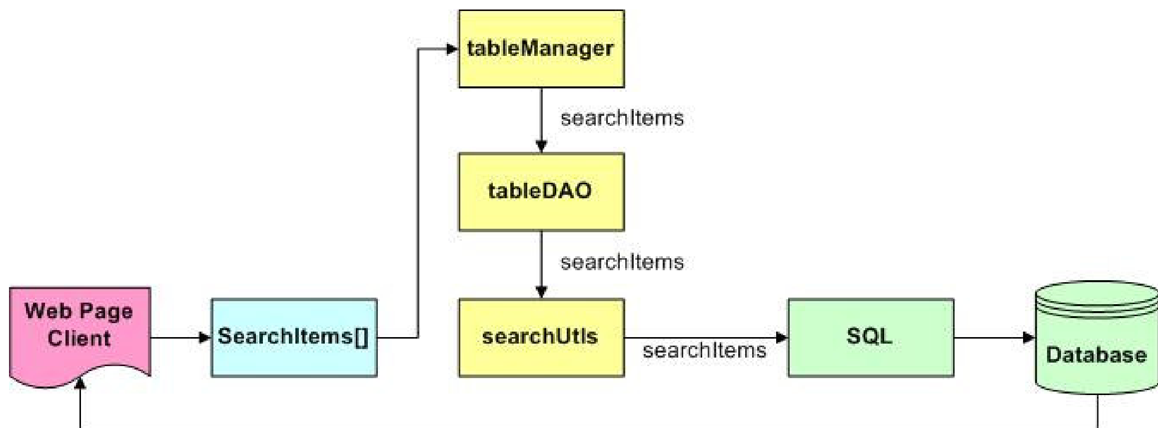
webConstants.inc.php

The webConstants file contains constants that are used in your front end web pages. Example of some of these constants are commonHeader, clientIpAddress, etc.. The default generated WebConstants.inc.php file has the following.

```
define("WEB_COMMON_COMPONENT",WEB_PATH.FILE_SEPARATOR."common");
define("PAGE_HEADER",WEB_COMMON_COMPONENT.FILE_SEPARATOR."commonHeader.php");
define("PAGE_FOOTER",WEB_COMMON_COMPONENT.FILE_SEPARATOR."commonFooter.php");

define("URL_STYLE_SHEET",URL_ADDRESS.WEB_SEPARATOR."common/styleSheet/style.css"
);
define("URL_COMMON_JAVA_SCRIPT",URL_ADDRESS.WEB_SEPARATOR."common/javascript/com
monJavaScript.js");
define("URL_IMAGE_FOLDER",URL_ADDRESS.WEB_SEPARATOR."images");

define("URL_JAVASCRIPT_VALIDATOR",URL_ADDRESS.WEB_SEPARATOR."common/javascript/f
ormValidator/javascriptFormValidator.js");

$today = date("Y-m-d");
$dateTime = date("Y-m-d h:i:s");
$userIpAddress=getenv("remote_addr");

define("DEFAULT_ROWS_PER_PAGE","50");
define("DEFAULT_SEARCH_CONDITION"," and ");

define("ROW_COLOR1","#FFFFFF");
define("ROW_COLOR2","#FFCCFF");
```

Retrieving Data from a PCG Generated application

One of the main functions of any application is to retrieve data. In the PCG framework, I have tried to designed an easy way to search through the databases without the user having to write any SQL statement. In a PCG application, we search using SearchItems.



The basic workflow for searching Data is as shown the diagram above. From our pages, we build searchItems which we pass to the Application Layer. In the applicaton Layer, the DAO classes pass the SearchItems to the searchUtils class which builds the proper SQL based on the parameters set in the SearchItems. Then the SQL is executed on the database and rows returned to the Web Page Client.

What is a SearchItem ?

A SearchItem is a search subunit. i.e if you had an SQL statement, a search item would be one element from your WHERE clause. A search will typically contain many searchItems and each of these searchItems can be joined by AND or ORs.

SearchItem Parameters

When building a new SearchItem, the constructor signature is as follows

searchItem($searchField="",$searchValues="",$searchCondition="",$outerOperand="",$innerOperand="",$prefix="",$suffix="")

As you can see from the above constructor method, all parameters passed to Searchitem are optional. If you do not put anything in a searchItem and pass it to the DAO layer, the latter will just return all fields with no conditions.

Here is an explanation of the fields in the SearchItem

searchField – This is the database fieldname that you want to search against

searchValues – This field holds either one unique value that you want to search or an array of values you want to search against the searchField in this searchItem.

searchCondition – The condition that you want to search the SearchValue(s) against the searchField.

outerOperand – The operand that will separate the different searchItems (AND or OR). .e.g SearchItem1 AND searchItem2 OR searchItem3

innerOperand – If you have multiple searchValues, the operand that you want to separate them (AND or OR) e.g. ((SearchField1 = searchValue1) OR (SearchField1 = searchValue2))

prefix – Any text in prefix will be put in front of that searchItem in the resulting where clause. I use this usually for "("

suffix - Any text in suffix will be put just after that searchItem in the resulting where clause. I use it usually for ")"

I think that a better way to explain what the searchItem are, is through examples. Let's assume we have a table called User and four fields called Id, Name, Skills and Age. In each of the following examples, we'll try to build searchItems to solve the needed information and show the SQL built.

Example 1 : Find user whose Id is 1

```
$thisSearchItem = new searchItem(FIELD_USER_ID,”1”);
```

Resulting SQL : Select * from users where (**id=’1’)**

<u>Example 2 </u>: Find User whose name is Nilesh and who is more than 18 years old

```
$thisSearchItem = new searchItem(FIELD_USER_NAME,”nilesh”);
$thisSearchItem = new searchItem(FIELD_USER_AGE,”18”,”>”,” AND “);
```

Resulting SQL : Select * from users where **(name=’nilesh’) AND (age > ‘18’)**

<u>Example 3 </u>: Find users (whose name is either nilesh or nil) and ((who have skills in either php and any kind of sql db and are not lazy) or (are greater than 50 years old)).

```
$namesToSearchArray[] = “nilesh”;
$namesToSearchArray[] = “nil”;
$thisSearchItems[] = new searchItem(FIELD_USER_NAME,$namesToSearchArray,””,””,” OR “);
$thisSearchItems[] = new searchItem(FIELD_USER_SKILLS,”php”,” = “,” AND “,””,”((“);
$thisSearchItems[] = new searchItem(FIELD_USER_SKILLS,”%sql%”,” like “,” AND “);
$thisSearchItems[] = new searchItem(FIELD_USER_SKILLS,”lazy”,” != “,” AND “,””,”“,”)”);
$thisSearchItems[] = new searchItem(FIELD_USER_AGE,”18”,” > “,” AND “,””,”“,”)”);
```

Resulting SQL : Select * from tableName where **(name = ’nilesh’ or name = ’nil’) and (((skills = ‘php’) and (skills like ‘%sql%’) and (skills != ‘lazy’)) or (age>50))**

In brief, to search or retrieve data from a PCG application, you just build the searchItems that you need, from the Request elements and pass it on the tableManager which will process the Searchitems, build the SQL and then retrieve the rows based on that.

The best way to get used to it is to use it and try on different kind of searches with it.

Note : searchItems are used to retrieve data from one Table only. You might use multiple searchItems on different managers to perform joins if you wish though.

databaseQuery class – Write your own custom SQLs

The DatabaseQuery is the cental place where all SQL queries are executed. The DAO classes are the only entities that communicate with the databaseQuery class. Currently in all the generated code, all the GenDAO instantiate the databaseQuery class in order to execute the SQLs.

PCG generated code can only execute searches in one single Table. If you need to execute a custom SQL not available from the genDAO classes or by bullding searchItems, you can do so using the databaseQuery class. Usually for joins, reportings, groupings etc.. you might need to write your own query. An example of how to do your own custom query is shown below.

```
<?
    $sql = "select field1,field2 from table1 t1, table2 t2 where t1.id=t2.id";
    $thisDatabaseQuery = new databaseQuery();
    $resultSet = $thisDatabaseQuery->executeDirectQuery($query);

            while (!$resultSet->EOF)
            {
                    $thisField1 = $resultSet->fields[field1];
                    $thisField2 = $resultSet->fields[field2];

                    echo "Field 1 →".$thisField1;
                    echo "Field 2 →".$thisField2;

                    $resultSet->MoveNext();

            } // end while

?>
```

The $resultSet will return you a resultSet Object and you can get retrieve the fields in it by using $resultSet->fields['fieldName'];

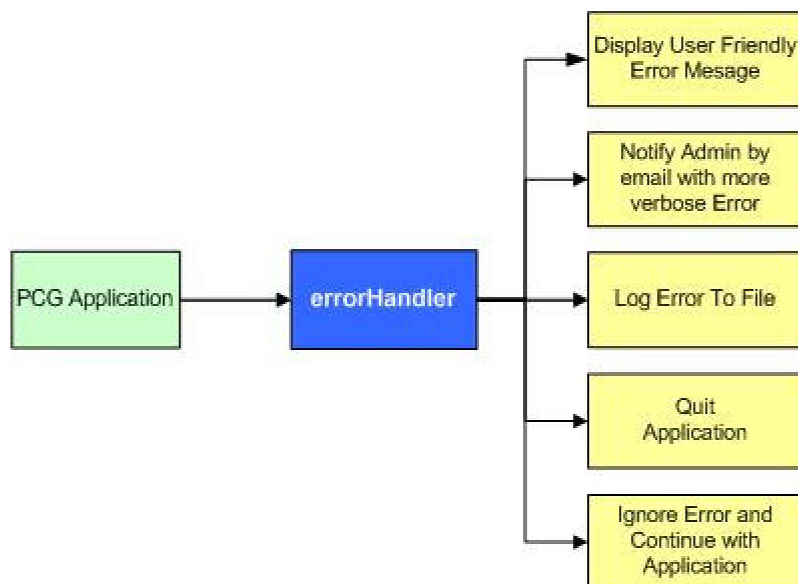
When doing reports or statistical queries, I always use custom queries for speed and more efficiency. For all these queries, I use database query to get the results. An example of a report query would be.

```
<?
    $sql = "select count(productId) as numberOfProducts , categoryName from
table1 t1 group by categoryId order by categoryName";

    $thisDatabaseQuery = new databaseQuery();
    $resultSet = $thisDatabaseQuery->executeDirectQuery($query);

            while (!$resultSet->EOF)
            {
                    $numberOfProducts = $resultSet->fields[numberOfProducts];
                    $category = $resultSet->fields[categoryName];
                    echo $categoryName." has ".$numberOfProducts."<br>";
                    $resultSet->MoveNext();

            } // end while

?>
```

Error Handling in your PCG Generated Application

PCG applications handle errors that occur in the Application layer using the PCG errorHandler class. The error handler takes different parameters to define its behavior in case of error. The different parameters it can take are

userErrorMessage – This is the error the user will see on the page if error happens.
programErrorMessage – This is the error message that the developer will see in email or LogFile. Put meaningful error message here to help developer debug program later on. Usually sql errors, exceptions etc..
errorPage – Put the name of the page you are at in this Attribute
quitProgram – if this flag is set to true. The errorHandler will quit the program at that point by issuing an 'exit' command.
emailAdmin – if this flag is set to true, the email address set in Application Constants as APPLICATION_ADMIN_EMAILwill receive email about the error
LogToFile – if this flag is set to true, errorhandler will log the error to file.
LogFile – this is file to logto. This file is set in application Constants.



All these values have a default value in the ApplicationConstants file. Please set these default as the default behavior of your PCG generated application in case of error. In case by case basis, your code will override these values and handle the errors accordingly.

Example usage of errorHandler

```
// if Error occured in Application, handle error.
$thisError = new errorHandler();
$thisError->setUserErrorMessage("Error ! This is the text user sees.");
$thisError->setProgramErrorMessage("Error ! This is the text developer sees");
$thisError->setErrorPage($_SERVER['PHP_SELF']);

$thisError->setQuitProgram(true);
$thisError->setEmailAdmin(true);
$thisError->setLogToFile(true);

$thisError->handleError();
```

Getting data from HTML FORM to a PCG application

User input in a web application is done via HTML forms. PHP provides the $_POST, $_GET and $_REQUEST global variables to retrieve data from html forms directly.

Retrieving the data directly from these global variables could cause a security threat in your application though. One can never trust user input. There can be malicious users trying to do Cross Site Scripting (XSS) or SQL injections hack by messing with the data. Therefore, one need to be parse the data and make sure it is safe before passing it to your application. An example of an XSS attack would be :
e.g Let say you have a text field element, which you allow user to input data and then your display the input data, the user could put something like ***<script> (for ;;) { alert('hacked'); </script>*** *in the input text and on displaying the input data, your page would go in a infinite loop asking the user to press on a javascript alert box forever.*

Another common problem with user input is magic_quotes is turned on or off. When magic_quotes is turned on in your php, some characters are automatically escaped. While you might have it enabled on your server, you are not sure that all servers you are going to deploy your application to, will have the same settings. So you need to make sure that, you check for whether its enabled or not on the server that your application runs on and escape your user data accordingly. It would be a big hassle to check for whether magic quotes is on or off each time you get a a user input.

PCG solves these two above problems by using the **requestUtils** class to get form elements. By using a separate class to do the form retrieval, it puts that action in one

central place and easy to change. So in that class, you can add whatever sanity check you need to make. Currently, the **requestUtils::getRequestObject($elementName)** method ltrim and rtrim your input data, strip the html tags (this setting is configurable in *applicationConstants.inc.php*, by setting the **ALLOW_HTML_TAGS_IN_POST_GET_REQUESTS** constant to true or false) and if magic_quotes is not enabled, addslashes to the data before returning to the client.

Usage example

Let's say you have a form

HTML FORM

```
<FORM NAME="nileshForm" METHOD="POST" ACTION="test.php">
UserInput : <input type="text" name="userInput" value="">
</FORM>
```

Instead of using <? $thisUserInput = $_POST['userInput']; ?> to get that form element, we will use the code below. It will perform the same action as using the $_POST, but it will also do sanity check on the data input by user. You can add your own additional sanity checks to the getRequestObject method if you need to in one central place.

PHP SCRIPT

```
<?
	$thisUserInput = requestUtils::getRequestObject('userInput');
?>
```

textEncrypter - Encrypting Data between scripts (requests)

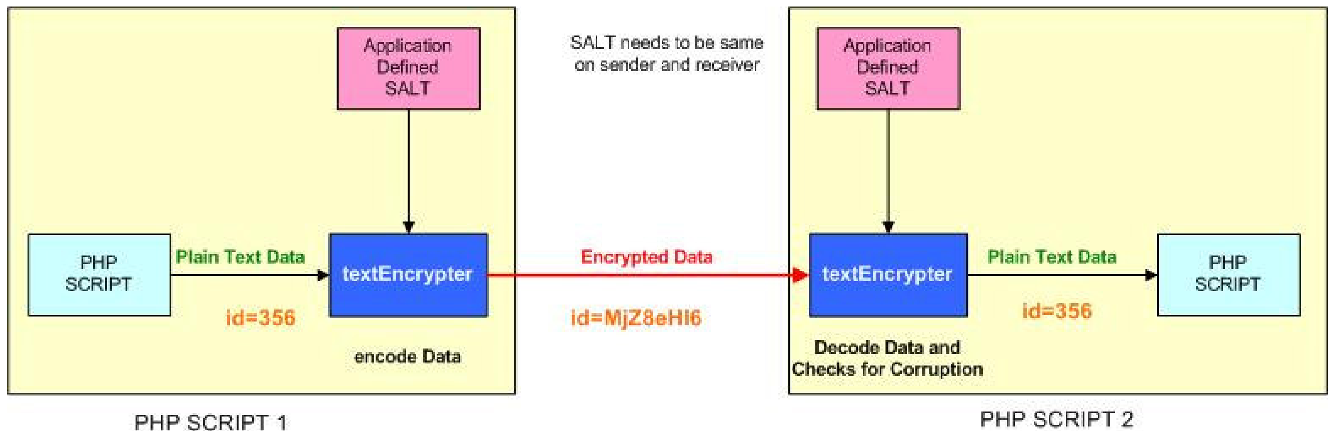
Often times, you have to pass data from one script to another in your web application. The means of passing data from one script to another in the same application would be either though the SESSION, REQUEST or COOKIE. While the session can only be modified by your application code, REQUEST and COOKIE can be modified by the user, if they know how it works.

e.g Lets say I have a script that shows the profile of a user when I pass the the userId to it and I pass that userId by a GET string thru the URL

http://www.someapplication.com/viewUserProfile.php?**userId=43254**

As the userId is in the get String and visible to all users, malicious users, may try to substitute that id for other id and be able to flood or get data they are not supposed to. So one needs a way to prevent the users from changing that data, but at the same time allow your application to pass it from script to script.

PCG uses the textEncrypyter utility class to achieve that. The textEncrypter encrypts the data while transmitting it. Before sending data to another script, the first script encodes it using the textEncrypter and the receiving script needs to decode it using the textEncrypter class. The encryption uses a SALT to encrypt the data. The SALT is like a key. You can put whatever secret word you want for it. On encryption and Decryption, the key need to be the same, otherwise, the textEcrypter will stop the application.



The diagram above shows how the text Encrypter works. Lets say have a script that prints a link to PHP SCRIPT 2 and appends '**id=356**' to that link. To avoid users being able to change the id, PCG encode the '**356**' using the textEncrypter class, such that the resulting get String will be something like '**id= MjZ8eHI6**'. If users try to change the value of that Id in the Get String, on decoding that value, the textEncrypter will detect that the SALT is not the same and stop the application.

Example Code

Php Script 1

```
<?
   include_once(CLASS_TEXT_ENCRYPTER);
   $thisTextEncrypter = new TextEncrypter();
   $id = "356";
   $encryptedId = $thisTextEncrypter->encode($id);
?>
<a href="phpScript2.php?id=<? echo $encryptedId; ?>">Link to Script 2</a>
```

Php Script 2

```
<?
    include_once(CLASS_TEXT_ENCRYPTER);
   $thisTextEncrypter = new TextEncrypter();
   $encryptedId = $_REQUEST['id'];
   $id = $thisTextEncrypter->decode($id);
   $userProfile = $userManager->getUserById($id);
?>
```

Quick Questions and Answers

Each time fatal error occurs, I want my PCG application to send me email about it. Where can I tell PCG my email address?

Edit these lines in the applicationConstants.inc.php file

```
define("APPLICATION_ADMIN_EMAIL","{ADMINSITRATOR_EMAIL}");
define("APPLICATION_FROM_EMAIL","{FROM_EMAIL}");
```

I want to see all SQLs that are being executed in my PCG application, so that I can debug an error. How can make my application print out all the SQL it executes?

Edit these lines in the applicationConstants.inc.php file

```
define("DEBUG_PRINT_SQL",false);
define("DEBUG_PRINT_COUNT_SQL",false);
```

I do not want my users to be able to input HTML tags in my application form inputs. How do I enable/disable that?

Edit these lines in the applicationConstants.inc.php file

```
define("ALLOW_HTML_TAGS_IN_POST_GET_REQUESTS",false);
```

I want to know how much time each page takes to execute. How can I do that?

Edit these lines in the applicationConstants.inc.php file

```
define("HAVE_PAGE_TIMER",true);
```

Useful Links

phpCodeGenie Project Page – http://phpcodegenie.sourceforge.net

PHP website – http://www.php.net/

phpClasses – http://www.phpclasses.org/

phpPatterns – http://www.phppatterns.com/

phpArchitect – http://www.phparch.com/