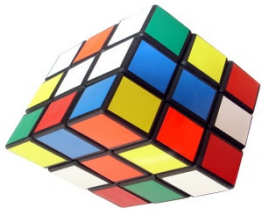
Cut to the Chase Series
More Walk – Less Talk

zenyan webapp framework



Native API's

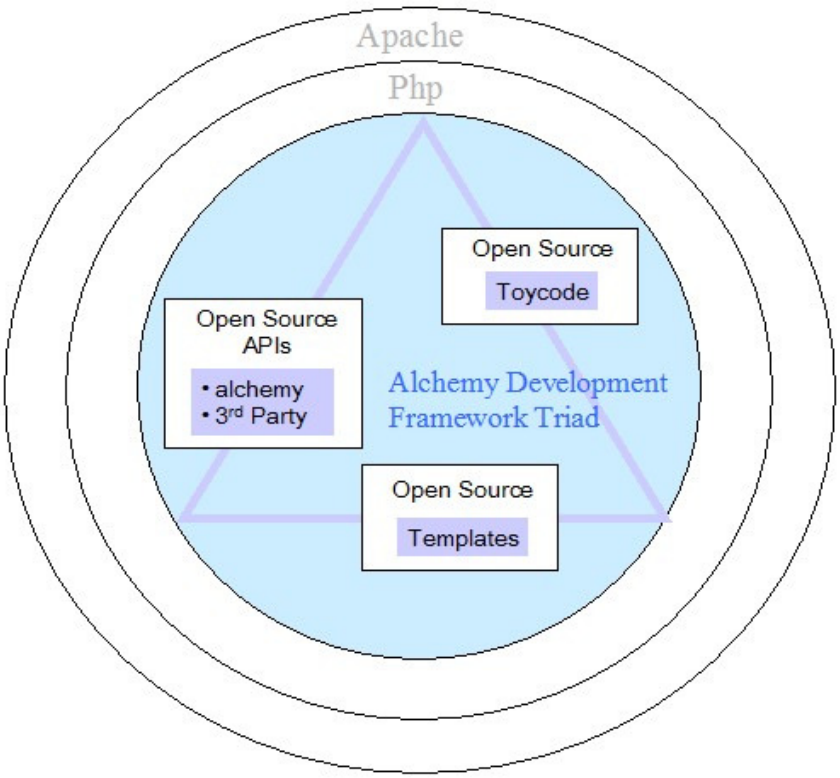
Documentation

zenyan webapp framework – Native API's

Table of Contents

The zenyan Native API's

The zenyan api's are part of the zenyan development triad.



The real bread-n-butter that these api's bring to the table is in the following two areas:

- Efficient and easy access to data
- Flexible Integration with User Interface controls

Location and other Foundational Information

The Native zenyan and 3rd party api's live under the zenyan root directory in a directory name "eLib". There are subdirectories under the "eLib" directory, but these pertain to 3rd party api's and as such will not be covered.

IMPORTANT NOTE: Regarding API functions, only those functions that are meant to be directly accessed by the programmer using the api are going to be covered. If you have an interest in looking further at the code under the hood, by all means do so.

Data Access API's (o_dbiomyp and o_dbiox.php)

As zenyan is a web development framework that is specific to a given implementation of a webserver I am only including the following two api's. There are facets of the zenyan infrastructure I have morphed into various other endeavors where multiple connections to MySQL databases residing on different servers was required. I am not including these as at present time I feel they are out of scope to the current implementation of this framework.

The real value behind the data access api's is the theology of "you tell me the data you want, and also the way you want it formatted (within the confines of what is available) and I will also do the data formatting work for you as well".

In general these API's strives to simplify data access within the program code itself by trying to simplify the CRUD activities into two general functions: read data and write data.

In order to make the data access api functions generic across various DBMS's this code is written using object oriented programming techniques.

Another generalized goal of these API's are to put the power of data access into the hands of the specific SQL language you are using.

Local MySQL (o_dbiomyp.php)

This api facilitates direct access to a MySQL database without the need for an ODBC connection.

Functions

dbread()

The purpose of this function is to simplify the task of reading data from a database through simplification into a single function.

Syntax

```
dbread(~~sql_select_statement~~",~~data_return_type_argument~~")
```

Input Arguments:

- The sql select statement you wish to run ...and...
- the method by which you want the data returned to you

Returns: A formatted recordset whose type you specify

Input Arg	RS Return Type	Return Struct	Notes
delim	Plural	Array	(default) data only
delim2	Plural	Array	data, first elem = col name
delim3	Plural	String	delimited, newline as row separator
delim4	Plural	Array	2D Array, data only
trrow	Plural	Array	used for grid control, returns html table tr stub

hash	Singleton*	Hash	Key/Value hash key=col val=data

* One row in the returned recordset.

Example

```
$rs1c = $cls1->dbread($query,"trrow");
```

Additional Information

The 'trrow' return type is designed to specifically return a recordset that is bound to the UI grid control.

dbwrite()

The purpose of this function is to simplify the task of writing data to a database through simplification into a single function. You pass in the applicable SQL update, delete, or insert statement (including any wrapping transaction code) and let the api take care of the rest.

Syntax

```
dbwrite(~~sql_update_insert_or_delete_statement~~)
```

Example

```
$ret = $cls1->dbwrite($query);
```

Test Harness

useclass_o_dbiomyp.php provides a minimal means of testing connectivity to the local MySQL database where zenyan is implemented. This test presumes you or someone has correctly installed and configured both the MySQL and the zenyan data access configuration parameters. These are covered in other documents.

Generic ODBC (o_dbiox.php)

This is the generic ODBC api. It presumes you have a working ODBC connection. It offers the same functions found in the o_dbiomyp api. The plumbing is just a little bit different.

Pseudo Recordset (dicers.php)

The whole purpose of dicers (pronounced dice rs) is to take an existing recordset and to slice and dice in into a different output format, and/or to create a new pseudo recordset with less columns. By doing this we can save a trip back to the database assuming we already have the data we need.

Change Delimiter

Accepts a delimited recordset and returns a delimited recordset with a different delimiter.

Syntax

```
change_delimiter(~~delimited_recordset~~,~~input_delim~~,~~desired_delim~~)
```

Example

```
$rs = change_delimiter($rs1a,"|",",");
```

Convert Array to String

Accepts a recordset of type "array" and returns a delimited recordset. This function is typically called to avoid in-line coding when binding to UI controls like the datagrid.

Syntax

```
convert_array_toString(~~row_seperator~~,~~input_recordset~~)
```

Example

```
$cstr = convert_array_toString("<br />",$rs1a);
```

Column Dicer

This takes a delimited recordset and returns a delimited recordset that contains a subset of the columns of the original recordset. This function is most useful when using the same recordset for the grid control and for generating graphs and charts.

Syntax

```
delimColDicer(~~delimited_recordset~~,~~column_list~~,~~input_delimiter~~)
```

Example

```
$diced_rs = delimColDicer($rs1a,"0,1,4,9","|")
```

Test Harness

test_dicers.php

Application Library

This is a fairly skeletal library for routines that support the application interface that are not covered in other api libraries. You should add your reusable function into this library. Feel free to modify this library, including removing functions as you see fit for your application.

Validate SQL Date

This is a very simple and incomplete function to check for a valid sql server date. The intended use is not to check user input, rather to validate sql date after date transformation (if required) prior to passing argument into an SQL Server dml statement.

Syntax

```
$status = validSQLServerDate(~~sql_date~~)
```

Example

```
$status = validSQLServerDate($dte)
```

Generate a GUID

Creates a 21 character (default) GUID.

Syntax

```
$generatedGUIDstring = generateGUID()
```

Example

```
$ generatedGUIDstring = generateGUID()
```

Create a Synchronized List UI Control

Though we have swung most of our UI framework to JQuery we still have some notable legacy holdouts (where we do not have equivalent functionality). This is one of the holdouts. This control produces and returns a synchronized list code stub consisting of a parent list box and a child list box.

Input Arguments

$delimr - delimited array. row delimiter in array is currently |.

$plist - row offset that contains data for the parent list (note 0=1)

$clist - row offset that contains data for the child list (note 0=1)

Returns: An array with two items

$jsSyncListCode - Javascript code needed for synchronized list

$syncListCodeStub - HTML code stub for your form

Syntax

```
$codestub =   getSyncList(~~delim_recordset~~,~~parent_array~~,~~child_array~~)
```

Example

```
$codestub = getSyncList($delimrs,$plist,$clist
```

Change Delimiter

Accepts an array containing whose indices contain delimited strings, loops through it and changes the delimiter and returns the array. Hard coded currently for input delimiter "|" which is the zenyan default delimiter.

Syntax

```
cngDelim(~~array_of_delim_strings~~,~~desired_delim~~)
```

Example

```
cngDelim($rs,"~!~")
```

Get Page Name

Strips the path and returns the web page name. This function can be used for security to prevent IP spoofing.

Syntax

```
$<var> = getPageName()
```

Example

```
$pgnm = getPageName()
```

File Library

Fairly skeletal library for file system access. It is highly recommended that you only allow public (meaning end user and development users of this api) file system access to a sandbox area.

Does File Exist?

Check to see if a file exists

checkFileExists($filspec)

```
$status = checkFileExists(~~filespec~~)
```

Example

```
$status = checkFileExists($filspec)
```

"y" = yes; ""=no

File Slicer

Accepts a fully qualified file name and returns a hash of the component parts

```
$fileHash = filespecSlicer(~~filespec~~)
```

Example

```
$fileHash = filespecSlicer($filespec)
```

$fileHash['path'] = file path

$fileHash['prefix'] = file name

$fileHash['extension'] = file extension

$fileHash['filename'] = file name + file extension

$fileHash['filespec'] = full file name including path and extension

Get Contents of a File

Accepts a file name and returns its contents

Syntax

```
$filecontents = getFileContents(~~filespec~~)
```

Example

```
$filecontents = getFileContents($txtfil)
```

Create a File

Accepts a file specification and file contents and creates a file. Note: this function creates a new file. It will not replace or overwrite a file that already exists with the same name.

Syntax

```
$status = createFile($filecontents,$filnm)
```

Example

```
$status = createFile($filecontents,$filnm)
```

Status returned will be a string ...

Successful = "file " + filename +" created "

or

Failed = "could not create file"

Parse File Return Array

Takes a file and returns an array. One element per line.

Syntax

```
$<array> = readFileReturnArray($filspec)
```

Example

```
$retarr = readFileReturnArray($filspec)
```

Return Hash of all files in a given directory

Takes a file takes a directory path and returns a hash of all files within it optionally filtered by an extension list.

Syntax

```
$<file hash> = get_files_hash($dir,$extlst)
```

Example

```
$hash = get_files_hash($mydir,$extensionarraylist)
```

Returns array of all files and directories in a given path

Accepts a path and returns a complete listing of files and directories/subdiectories.

Syntax

```
$<array> = get_files($dir)
```

Example

```
$retarr = get_files($dir)
```

Get Base Directories

Returns an array of all subdirectories under a given base directory.

Syntax

```
$<array> = get_base_dirs($dir)
```

Example

```
$subdirarr = get_base_dirs($dir)
```

Copy a File

Make a copy of a file. Return values: 1=successful, 0=failed. $src is the file to be copied.

Syntax

```
$<status> = copyfile($src,$tgt)
```

Example

```
$retv = copyfile($src,$tgt)
```

Append File

Appends data to end of file.

Syntax

```
appendfile($ln,$fil)
```

Example

```
appendfile($ln,$fil)
```

Purge File Contents

Delete the contents of a file.

Syntax

```
purgefileContents($fil)
```

Example

```
purgefileContents($fil)
```

CK Editor Functions

The following is a list of functions used by the CK Editor which has been integrated into zenyan.

Functions

```
oFileMU ($fs)
oFileMUss ($fs)
oFile ($fs)
svFile ($fs, $fcntnt)
clsFileSave ($fs)
clsFileNoSave ($fs)
```

Logging Subsystem

The logging subsystem is inherent to zenyan. Of course you can choose not to use it in any way shape or form. You can also choose to use application level logging to a level that would make George Orwell proud, meaning you can choose to log every thing an end user does. You can even log full program execution. Of course, implementing what you wish to log is entirely up to you.

Runtime Logging

This subsystem is used to log desired application runtime activity. Out of the box it logs to a text file but can easily be morphed to log to a database table. Keep in mind this is an application level log. You also have Php logs and Apache logs and MySQL logs as part of the zenyan infrastructure.

Syntax

```
eAppLog(~~data_to_log~~)
```

Example

```
eAppLog($dWrite)
```

The log is appended to a file (hardcoded path within the function). What gets logged is the string you pass into the function pre-pended with the current date/time and appended with a newline.

Debug Logging

I typically use the debug log to troubleshoot problems that are intermittent where you have a general idea where the problem might be. It is also useful for logging problems your think might be end user related or that you think may pertain to data. For normal development time debugging I prefer to use the dynamic logging mechanism within the program itself that pipes to stdout as it is much more efficient.

Syntax

```
eDebugLog(~~data_to_log~~)
```

Example

```
eDebugLog($dWrite)
```

The debug log is eLog\debuglog.txt. What gets logged is what you specify

Extending/Modifying zenyan API Guidelines

You should feel free to add new functions to the appropriate libraries as you need them. Unless specifically addressed below it is my recommendation that you do not modify existing api library code as there can be certain framework dependencies that you can break. When you add new functions I recommend you pre-pend your function name with the string *xx_* where *xx* are your initials.

Data Access API's

Do not modify any existing data access functions. You can add new functions to the library and modify the calling conditional within the library.

Pseudo Recordset

Do not modify any existing data access functions. You can add new functions to the library.

Application Library

You can add new functions to the library. Do not modify any functions pertaining to GUID or Synchronized List.

File Library

Do not modify any existing data access functions. You can add new functions to the library.

Logging Subsystem

You can add new functions to the library. It is also acceptable to rewrite the body of these functions to write to a logging database that you create.