Medusa / Meduza

Container class - generic container for database objects:

The Container class is an abstract class which can be extended to create classes which are specific to a table on a database server. In this way you can incorporate the default properties and useful methods for these types of objects easily.

For instance, suppose you have a table on a database called “customers” and which contains records for each individual customer identified by an integer database key or id. Thus you can create a class for a customer and use it with these statements:

```
class Customer extends Container {
      public $database = “companyDB”;
      public $table = “customers”;
      public $connection = null;
}
...
$id = 123;                                  // int id for the customer
$customer = new Customer;                   // create the object
$customer->fetch($id);                      // read a record into it
...
or
$customer = new Customer($id);              // create the object & fetch record
...
$customer->name = “Acme Widgets, Inc.”;     // change the name property
...
$customer->store();                         // store the record into the DB
```

Properties:

The Container class, and classes you extend from the Container class, have these public properties:

```
(string) $table         : name of the DB table
(string) $database      : name of the database
(string) $connection    : name of the connection (leave blank)
(object) $db            : handle to low-level Databoss interface
(array)  $properties    : list of property(column) names for a record
```

Methods:

constructor:

```
$obj = new {classname}([$id=null])
```

determine if object contains a valid record:

```
if ($obj->valid()) …
```

produce text breakout of the object:

```
echo $obj->exhibit()
```

purge all properties(columns) to the default values:

```
$obj->purge()
```

merge another object or hash into this object:

```
$obj->merge(&$ref)
```

fetch a record from the database into this object:

```
$obj->fetch($key[,$key...])
```

store this object into the database:

```
$obj->store()
```

update the db record from this object (using SQL update ..):

```
$updatedCount = $obj->update()
```

write the db record from this object (using SQL replace ..):

```
$keyWritten = $obj->write()
```

delete record for this object (or another key):

```
$obj->delete([$key])
```

truncate all records from the table and reset auto-increment value:

```
$obj->truncate()
```

export this table's data:

```
$obj->export([xml/csv/TXT][,where clause])
```

Data class - methods to work on data |

Convert a string as dbColumnName to “Db Column Name”:

```
$displayName = Data::toDisplayName((string)name)
```

Convert a phone number to displayable text, i.e 9999999999 to (999) 999-9999:

```
$phoneNumber = Data::toDisplayPhone((string)phone,[usa=true])
```

Convert an ASCII string to a hex representation

$hexString = Data::toHex(string)

Convert a hex string to an ASCII character string:

$string = Data::toAscii(hex)

Mask all but nn digits of a credit card number:

$maskedNumber = Data::maskCardNumber((string)number[,visible=4])

Output a string representation of a variable (simple/array/object/hash):

Data::breakout((mixed)&$thing)

Output a simple text exhibit display (simple/array/object/hash):

Data::exhibit((mixed)&$thing)

Return a string of specified length of random characters:

$random = Data::genRandomString(length)

Render a string with phone numbers that Skype will not try to make links with:

Data::skypeProof($text)

Return a string of no more than nn characters of another string:

$limitedString = Data::strLimit(string[,limit=40])

Same as the above but without the ... at the end:

$fixedLengthString = Data::max(string,[limit=40])

Convert a simple array to a string with “'” and “,” (for SQL):

$sqlString = Data::arrayToString(array)

Databoss class - low-level database interface

The Databoss class provides for a low-level interface to an (my)SQL server connection. Once you have the object ($db) you can invoke a variety of methods to manipulate the databases and tables on that connection. Every Container class object has a 'db' property which is a Databoss object for the connection where that table/record exists.

Databoss can also be used by itself to perform queries on a database producing a variety of returned data. At the very least Medusa-Databoss provides you with a way of accessing multiple database server connections of multiple types

(Oracle,MySQL,MSSQL) without regard to the connection parameters or implementation.

As an example, if I wanted to do a quick summary of customer records in the customer table resident in the 3usite database on the default connection, I would use this:

```
$db = new Databoss;
$customers = $db->fetchChoices("
      SELECT id,CONCAT(firstname,' ',lastname)
      FROM 3usite.customer LIMIT 50
      ");
foreach ($customers as $customer=>$name) { ... }
```

One-time constructor shorthand:

```
Databoss::db([dbname][,connection][,username][,password])->method...
example: $records = Databoss::db()->fetchValue("SELECT count(*)..);
```

Constructor:

```
$db = new Databoss([dbname][,connection][,username][,password])
"connection" is one of ("development","live" or "hlrdb"). If omitted, the
default connection for the server is used ("development" for beta or "live".
```

Focus the connection to one particular database for subsequent queries:

```
$db->focus($dbname)
```

Check for existence of a database & table:

```
if ( $db->exists($table=null,$dbname=null) ) ...
```

Return a reference to the structure (properties/defaults/types) for a table:

```
$structure = $db->&structure($table=null,$database=null)
```

Obtain a list of the properties/columns in a table:

```
$properties = $db->properties($table=null)
```

Log a query and the error string into the system log file:

```
$db->logErrors($query)
```

Return the last SQL error from the connection as a string:

```
$errorString = $db->error()
```

Escape a string for use in queries:

```
$escapedString = $db->escape($string)
```

Return the last inserted id for auto_increment:

```
$id = $db->lastInserId()
```

Perform a query and return result:

```
$result = $db->query(query)
```

Fetch a record from a database.table using key and return a hash:

```
$result = $db->fetch(table,key)
```

Fetch a single record and return a stdClass object with properties:

```
$object = $db->fetchRecord(query)
```

Fetch a single record and return a stdClass or specified class object:

```
$object = $db->fetchObject(query[,classname])
```

Fetch a single record as a simple sequential array:

```
$array = $db->fetchArray(query)
```

Fetch a single record and return an associative array (hash):

```
$hash = $db->fetchHash(query)
```

Fetch a single column from a single record and return a simple variable:

```
$value = $db->fetchValue(query)
```

Fetch a single column from multiple records and return an array of values:

```
$array = $db->fetchValues(query)
```

Fetch an array of arrays for multiple records:

```
$array = $db->fetchAllRecords(query)
```

Fetch an array of arrays for multiple records:

```
$array = $db->fetchArrays(query)
```

Fetch an array of hashes for multiple records:

```
$array = $db->fetchHashes(query)
```

Fetch an array of objects for multiple records (optionally of a specified class):

```
$array = $db->fetchAllRecordObjects(query[,class=null])
```

Same as the above method:

```
$array = $db->fetchObjects(query[,class=null])
```

Fetch a hash of name=>value pairs using a query that returns two columns:

```
$choices = $db->fetchChoices(query)
```

Store an object or hash into a database table:

```
$recordId = $db->store(table,&source)
```

Update an existing record using an object or hash:

```
$recordsUpdatedCount = $db->update(table,&source)
```

Write a new record from an object or hash:

```
$recordId = $db->write(table,&source)
```

Delete a database.table record from a database using a key:

```
$recordsDeleted = $db->delete(table,key)
```

Truncate a database.table table

```
$db->truncate(table)
```

Return a count of records in a table (optionally with a where clause):

```
$recordCount = $db->records(table[,where=null])
```

Return a hash of tableName=>recordCounts for the current database:

```
$statistics = $db->statistics()
```

Date class - properties & methods for dealing with date values

The Date class provides both class and object methods for dealing with dates in a uniform and objectified way. Where a method takes a date as a string, virtually any format can be used and is correctly interpreted. If omitted, the current date/time is used. Methods that take a format type for the return value can accept one of the following constants:

```
TEXTDATE        Mmmmmmmmmmmm dd, yyyy
LONGDATE        mm/dd/yyyy hh:mm(a/p) or yyyy-mm-dd hh:mm:ss
SHORTDATE       mm/dd/yyyy or yyyy-mm-dd
```

VERYSHORTDATE mm/dd/yy

Compare two Date objects and return <0, =0, or >0:

$result = Date::compare(first,second)

Return the difference in days between two Date objects:

$days = Date::difference(first,second)

Convert a date string to an internal (SQL) format date:

$sqlDate = Date::toInternal([date=null[,format=LONGDATE]])

Convert a date string to a datestamp—i.e. yyyymmddhhmmss:

$datestamp = Date::toDatestamp([date=now])

Convert a date string to an external format date:

$displayDate = Date::toExternal([date=now[,format=SHORTDATE]])

Convert a date string to a very short date (m/dd h:mm(a|p)):

$quickDate = Date::toQuick([date=now])

Obtain a timestamp from the current date/time as yyyy-mm-dd hh:mm:ss:

$timestamp = Date::timestamp()

Obtain the timestamp for the start of today as yyyy-mm-dd 00:00:00:

$today = Date::today([format=LONGDATE])

Obtain the maximum UNIX date as a timestamp ~ 2037-12-31 23:59:59

$maxDate = Date::forever()

Obtain the timestamp for right now:

$now = Date::now([format=LONGDATE])

Constructor:

$date = new Date([(string)date])

Determine if a Date object has a valid date in it:

$hasDate = $date->valid()

Set a Date object to a date & time from string:

```
$date->set([(string)date])
```

Get the days in the month contained by a Date object:

```
$days = $date->daysInMonth()
```

Express a Date object as an internal (SQL) date: yyyy-mm-dd hh:mm:ss

```
$sqlDate = $date->internal([format=LONGDATE])
```

Express a Date object as a datestamp yyyymmddhhmmss:

```
$datestamp = $date->datestamp()
```

Express a Date object as an external date (mm/dd/yy[yy][hh:mm (a/p)m):

```
$displayDate = $date->external([format=SHORTDATE])
```

Express a Date object as a quick date (m/dd h:mm(a|p)):

```
$quickDate = $date->quick()
```

Move a Date object by interval i.e. ([-]nn days or [-]nn hours, etc.):

```
$date->move((string)interval)
```

Apply an offset (-/+hh[mm]) to UTC to a Date object:

```
$date->applyUTCoffset((string)offset)
```

Return # seconds as elapsed time ([nn hours][nn minutes]nn seconds):

```
$elapsedTime = $date->elapsed(seconds)
```

Conclusion:

Medusa has other classes and methods within it, however, these are deemed to be the most important to realize some immediate benefit from using Medusa.