Inside Formulize: a Developers Guide

August 28, 2012

Introduction

Formulize is a complex piece of software, that has grown organically over many years. This document provides an overview the general flow of execution through the code, and of some of the common coding techniques used in Formulize. It also documents how to create your own element types. The intent is to provide a leg up for developers wishing to work with the Formulize code and contribute to the project. This information may also be useful to developers creating advanced applications.

Please post any and all questions and feedback to the Formulize support forums at www.freeformsolutions.ca/formulize.

The Path Through the Code: how Formulize renders a page

Formulize is a web-based software written in PHP. It resides on a web server, and communicates with a client (web browser) through the http protocol. The major pattern in the operation of Formulize is to have an entire page rendered, with decisions about what should appear and not appear getting made by consulting the database, the objects and methods of the XOOPS / ImpressCMS core, and the GET and POST information passed to Formulize from the client.

Some important core classes and objects you'll see in the code that you should know about:

$xoopsUser - the user object that represents the currently logged in user

$gperm_handler - the class for checking user permissions

$config_handler - the class used to get configuration options for Formulize

There are many internal functions that are used in Formulize. Most are in the **include/functions.php** file. Others are in the various ***display.php** files (ie: **include/formdisplay.php, calendardisplay.php, elementdisplay.php**, etc).

This overview is very high level and is not a line by line, or function by function view of the execution of the code. That would be practically incomprehensible. This is an overview that hopefully provides a general understanding of the major steps involved in the execution of a page in Formulize, so you know where to look to see how something is happening. If you have any questions or would like anything clarified in more detail, please post on the support forums at www.freeformsolutions.ca/formulize.

index.php

This is the file that starts every Formulize page load. Normally, it invokes the core mainfile.php, which sets up the user's session and various other "housekeeping." Then it goes on to call Formulize's own initialize.php file, where execution continues.

If a $formulize_screen_id is set, then it will not call the mainfile.php, and instead will proceed directly to initialize.php. This is because we assume that if $formulize_screen_id is already set, then someone is working with Formulize in PHP code, rather than making a request via the URL. We assume they know what they're doing and will have called mainfile.php already.

initialize.php

This file does a lot of basic checks and gathering information, and then hands off execution to whichever fundamental action is being carried out: either the display of a list of entries, or the display of a form, or the display of a specific screen.

One of the important initial things it does is a basic security check to see if the user has permission to view the current form.

Then it calls include/readelements.php which is the single file that handles all the reading of data submitted from a form on the prior pageload. readelements.php is fundamental. It is the only way that data gets written to the database, outside of someone writing specific PHP code to do it. It will read all data from all forms, and write it accordingly to the correct tables and fields.

Once readelements.php is finished, then a few conditions are checked. First, if a screen is requested, then execution hands off to that screen. What this means in practice is that if the screen is a list of entries, then that screen will internally call include/entriesdisplay.php, and if it is a multipage version of a form, it will call include/formdisplaypages.php, etc. Screens will look up configuration settings, and then call the underlying API function with the appropriate parameters. The screen object will be passed in to, and is often consulted as the page is built.

If no screen is rendered, then the permissions of the current user are checked, and if they have any right to see a list of entries, then include/entriesdisplay.php is called. Otherwise, include/formdisplay.php is called.

formdisplay.php

If a form is being displayed, then the displayForm function takes over execution of the page. It is located in the include/formdisplay.php file. Inside that function, there is a lot of initial code that checks what form was requested, what are the user's permissions, what entry is being displayed, if any, etc.

Eventually, the form is begun, around line 687. One of two classes will be used, depending on how the function was called. The bottom line though, is that a $form object is created, and all the further operations are basically around adding elements to that object, in accordance with the principles of the original XOOPS form class.

The text for the header of the form is generated first. Then the compileElements function is called, to add in all the elements that should appear on the form (it might be just a subset of all the elements on the form). It uses the displayElement function, which in turn uses the elementrenderer class to create the form elements that get added to the $form object. Then, any "subforms" are added, if they haven't been drawn in already through subform elements. And lastly, the "proxy user" box is added, that is the box that lets you specify if the data is being submitted on behalf of another user (or lets you update the owner/creator if this is an existing entry).

The row of buttons at the bottom of the form then gets added, and that row may have a Printable view button, a Save button and/or an All Done button.

The very end of the drawing of the form is the creation of the "saving" message, the animated image that appears when the user clicks the save button. It is in a hidden div that becomes visible when the save button is clicked.

entriesdisplay.php

If initialize.php determined that a list of entries was going to be drawn, then execution would be handed off to the displayEntries function inside the include/entriesdisplay.php file.

This function starts by handling certain actions from the previous pageload, if they were requested, such as saving new saved views, deleting views, deleting entries, etc. It does some bookkeeping, like checking permissions, identifying what screen has been requested if any, and loading all the parameters for any views that need to be loaded.

It calls the generateViews function around line 499 to determine what views should be in the current view drop down list, if any.

It also eventually determines if an entry has been "clicked on", ie: normally this would be a click on the magnifying glass icon in the left margin, and if so, it hands off execution to the displayForm function, or to another page/screen as if the configuration of this list of entries specifies a particular screen to use for the display of entries.

If a form itself does not need to be displayed, then execution of the list carries on, first with the processing of any custom buttons.

Then the actual list of data to display is gathered from the database, around line 689. This querying actually takes place in the include/extract.php file, which is an abstract SQL query generator that can take all the form ids, search terms, framework settings, etc, into account and return the data that should appear on the screen.

Once the data has been returned, the drawInterface function is called, and that function sets up all the buttons and other UI things that appear above the normal list of entries. It will also draw in the top template instead, if one is specified.

Once the interface is drawn in, then the drawEntries function is called, and that function actually creates the main body part of the page, including presenting any entries that are supposed to be shown to the user.

The drawEntries function starts out with lots of initializing and checking of values. Then the scrolling box for the entries is setup around line 1325. Then the calculations are performed and drawn on screen if there are calculations in effect.

If no calculations are in effect, then the normal list of entries is drawn, starting with a call to drawHeaders (if it's a list of entries screen that's being rendered, then this, and many other steps, might be skipped, depending on the configuration options for the screen).

The next thing drawn in are all the quicksearch boxes, with a call to the drawSearches function. Then lastly, the data is looped through, and drawn into a table on the screen...unless a custom list template is in effect. Then each entry in the data set is sent through the custom list template and drawn on the screen according to that template instead of the default markup.

Finally, once the drawEntries function is finished, the bottom template is drawn in, if one has been specified.

formdisplaypages.php

This file handles drawing of multipage forms. It is really an alternate wrapper for the displayForm function.

Inside this file, the function displayFormPages simply keeps track of which page the user is viewing. The set of available pages is specified through the parameters that are passed to this function, and normally a multipage screen invokes this function, and passes in a list of elements on pages that corresponds to whatever the creator of the screen entered when they made the multipage screen. Those screen settings become the basis for the execution of the displayFormPages function.

First, the function figures out what page the user is on. Then it draws in the Thank You page, if that is the page the user is on. Otherwise, it calls drawPageNav around line 375. That creates all the forward and backwards buttons and the dropdown list to jump between pages. After that, it calls displayForm, and passes the list of elements that should appear on this page to displayForm, so only those elements are included in the resulting form. Lastly, more navigation is drawn in, if the user is not on the Thank You page.

Major Classes, Their Properties and Methods

Following the pattern of other $xoopsObject classes, most Formulize classes have a main object class with a constructor and nothing else, and a companion "handler" class that is used to work with the main object. The forms and elements classes all behave this way.

The exception is the data class, which is used to interact with the data that has been submitted to a form. Because the underlying thing the data class works with is not actually a xoopsObject, the data class is structured differently.

The Forms Class

The forms class is contained in /modules/formulize/class/forms.php

Example of using the forms class

```
// get an object based on form 12 and extract its captions
$fid = 12;
$form_handler = xoops_getmodulehandler('forms', 'formulize');
$formObject = $form_handler->get($fid);
$captions = $formObject->getVar('elementCaptions');
foreach($captions as $id=>$thisCaption) {
  print "Caption for element $id is $thisCaption <br>";
}
```

Properties of form objects

id_form - the id number of the form

title - the text name of the form

lockedform – a 1 or 0 indicating if the form is “locked” which means it can't be altered by anyone through the admin UI.

tableform - the name of the datatable this form refers to, if this form is really a reference to a non-formulize datatable, otherwise for normal forms this is blank

single - a flag indicating how many entries per form are allowed. Possible values are: 'on' for one-entry-per-user forms, 'group' for one-entry-per-group forms, or '', ie: nothing, for more-than-one-entry-per-group forms.

elements - an array of all the element ids for the elements that belong to this form. Keys are element ids, and values are element ids. Elements will be in the order they should have according to their own 'order' property.

elementCaptions - an array of all the element captions for the elements that belong to this form. Keys are element ids, and values are captions.

elementColheads - an array of all the element column headings for the elements that belong to this form. Keys are element ids, and values are column headings.

elementHandles - an array of all the element data handles for the elements that belong to this form. Keys are element ids, and values are element data handles.

encryptedElements - an array of all the elements in the form where the data is encrypted in the database. Keys element ids, and values are element data handles.

elementTypes - an array of all the element types for the elements that belong to this form. Keys are element ids, and values are element types. Possible values are:

subform - a subform UI element
text - single line textbox
textarea - multiline textbox
areamodif - text for display (two cells)
ib - text for display (one cell spanning the form)
select - selectbox
checkbox - checkbox series
radio - radio button series
yn - simple yes/no radio buttons
date - date select box
grid - a tabular grid of elements
derived - a value derived from other values in the same entry
??? - custom elements (including the file upload type) will specify their own value

views - an array of all the ids of the saved views for this form, keys are sequential starting at 0, and values are the saved view ids.

viewNames - an array of the names of all the saved views for this form, keys are sequential and in the same order as the views property, the values are the names of the views.

viewFrids - an array of the framework id, if any, that is in effect for this view, keys are sequential and in the same order as the views property, the values are the ids of the frameworks, or '', ie: nothing.

viewPublished - an array of boolean flags, indicating if the view is published, keys are sequential and in the same order as the views property, the values are either true or false, indicating if the view is published or not.

filterSettings - an array that contains the group-specific filter settings that limit what entries certain groups of users can see in this form.

headerlist - the default columns that should appear when the list of entries is drawn, if there is no saved view in effect (in which case the saved view's columns would be used)

defaultform - the ID of the default form screen

defaultlist - the ID of the default list screen

menutext - the text used to list this form in the menu. “Use the form's title” will signify that the title of the form should be used (this is the default). An empty value will mean the form has no menu entry.

form_handle - the handle for the form, used as part of the table name in the database

store_revisions - a 1 or 0 indicating whether revision history is being kept for this form or not.

Some methods of the form handler class

get($fid) - takes a form id and returns a form object based on that form

getFormsByFramework($framework_object_of_frid) - takes a framework object, or a relationship id number and returns an array of form objects based on the forms in that framework.

getFormsByApplication($application_object_or_id, $returnIds) - takes an application object or id number, and returns an array of form objects based on the forms in that application. If $returnIds is true, then it will return IDs instead of form objects. $returnIds defaults to false.

There are several others methods, but they are really supporting internal uses. Read the file to know more.

The Data Handler Class

The data handler class is contained in /modules/formulize/class/data.php

Example of using the data handler class:

```
// check if an entry belongs to a certain group
// and if so, return the metadata for that entry
include_once XOOPS_ROOT_PATH . "/modules/formulize/class/data.php";
$fid = 12;
$entry_id = 241;
$targetGroup = 6;
$data_handler = new formulizeDataHandler($fid);
$owner_groups = $data_handler->getEntryOwnerGroups($entry_id);
if(in_array($targetGroup, $owner_groups)) {
  $metaData = $data_handler->getEntryMeta($entry_id);
  print "Creation date/time: ".$metaData[0] . "<br>";
  print "Modification date/time: ".$metaData[1] . "<br>";
  print "Created by: ".$metaData[2] . "<br>";
  print "Last Modified by: ".$metaData[3] . "<br>";
}
```

Some methods of the data handler class:

The data handler is always invoked by passing it a form id. All subsequent calls to that data handler object will operate on that form only. You can have multiple data handler objects working at the same time, each attached to a different form. For this reason, the form id is never used as a parameter in the methods below.

The data handler is primarily used for small scale interactions with the data in forms, ie: getting specific values to answer specific questions that affect how the page should be drawn. It does not perform the gathering of data for display in lists of entries, which is a task delegated to the include/extract.php file and the getData function.

deleteEntries($entry_ids) - takes an id or an array of ids, and deletes all those entries

entryExists($entry_id) - returns true or false depending on whether the entry id exists or not

getEntryMeta($entry_id, $updateCache) - takes an entry id and returns an array with its metadata. The array contains the following information in order: creation date/time, modification date/time, user id of the creator of the entry, user id of the user who last modified. The optional $updateCache parameter will force a query to be made on the database, otherwise, the last gathered value for this metadata will be used. It is important to update the cache if a data writing operation has happened since the last time the getEntryMeta method was used on this entry.

getAllUsersForEntries($entry_ids, $scope_uids) - takes an array of entry ids as input, and returns an array of all the user ids of the creators of those entries. The optional scope_uids parameter can be used to limit the possible user ids that should be returned.

elementHasValueInEntry($entry_id, $element) - takes an entry id and either an element id, element handle, or element object as input. It then returns true or false depending whether there is a value for that element in the specified entry.

getElementValueInEntry($entry_id, $element, $scope_uids, $scope_groups) - takes an entry id, and either an element id, element handle, or element object as input. Returns the value for the specified element in the specified entry, or false if no value was found. Optionally can use either an array of user ids, or an array of group ids, but not both, to limit the search for a value to entries created by those users, or owned by those groups.

getAllEntriesForUsers($user_ids, $scope_uids, $scope_groups) - takes a user id or array of user ids as input, and returns the entry ids of the entries that user has created. Optionally can use either an array of user ids, or an array of group ids, but not both, to limit the search for a value to entries created by those users, or owned by those groups.

getFirstEntryForGroups($group_ids) - takes a group id or array of ids and returns the entry id of the first entry that was created that belongs to that group or groups.

getFirstEntryForUsers($user_ids, $scope_uids) - takes a user id or an array of user ids as input, and returns the first entry id that was created by that user or group of users. Optionally can take a second array of user ids which will be used to limit the search.

findFirstEntryWithValue($element, $value) - takes an element id, or element handle, or element object as input, plus a value to search for, and returns the first entry id that has that value for that element.

findAllEntriesWithValue($element, $value, $scope_uids, $scope_groups, $operator) - takes an element id, or element handle, or element object as input, plus a value to search for, and returns all the entry ids that have that value for that element. Optionally can use either an array of user ids, or an array of group ids, but not both, to limit the search for a value to entries created by those users, or owned by those groups. Optionally can take an $operator parameter, to specify the comparison operator to use in the search. Default operator is = (equals).

findAllValuesForEntries($handle, $entry_ids) - takes an element's data handle, and an entry id or an array of entry ids, and returns all the values that those entries have for that element.

findAllValuesForField($handle, $sort) - takes an element handle, and optional sort order (ASC or DESC) and returns an array where the keys are the entry ids and the values are the raw database values for that element.

getEntryOwnerGroups($entry_id) - returns an array of the group ids that are flagged as owners of the entry, or false if the query failed.

writeEntry - this method is used to write an entry to the database, however the best API method for writing values to the database is the formulize_writeEntry function which is designed for outside interaction, and maintains entry ownership information.

The Elements Class

The forms class is contained in /modules/formulize/class/elements.php

Example of using the elements class

```
// get the object for element for 42 and print out its caption
// and display setting
$element_id = 42;
$element_handler = xoops_getmodulehandler('elements', 'formulize');
$elementObject = $element_handler->get($element_id);
$displayValue = $elementObject->getVar('ele_display');
if(is_numeric($displayValue)) { // 1 or 0 means display to all or none
  $displayExplanation = $displayValue ? " all groups " : " no groups ";
} else { // comma separated list of group ids, trim first and last comma
  $displayExplanation = trim($displayValue, ",");
}
print "Element number $element_id has the caption: ";
print $elementObject->getVar('ele_caption');
print " and it will display for these groups: $displayExplanation";
```

Properties of element objects

id_form - the id number of the form

ele_id - the id number of the element

ele_type - the type of the element, same types are allowed as for the elementTypes property of the form object

ele_caption - the caption that will appear beside the element on the form

ele_desc - the descriptive text that will appear below the caption on the form. This is an optional value and might be blank.

ele_colhead - the column heading that will be used at the top of columns in the list of entries. This is an optional value and might be blank.

ele_handle - the data handle that is used to refer to the form in code, and which is also the field name of this element in the datatable for this element's form. Data handles default to be the same as the element id, but can be overridden at the application creator's discretion.

ele_order - the order number assigned to this element, controlling where it appears relative to other elements on the form.

ele_req - a 1 or 0 indicating if the element is required or not.

ele_uitext - an array containing the values for checkbox, radio button or selectbox elements that should appear on screen to the user. Only present for elements where the | character has been used to distinguish between what values go in the database and what text appears to the user. If present, it will have keys that match the database values for each option in the question, and values that contain the text the corresponds to the database values. Example: a dropdown list with three options, and when the options were created, the following text was entered:

```
5|five
10|ten
20|twenty
```

The ele_uitext array would be:

```
$ele_uitext[5]="five";
$ele_uitext[10]="ten";
$ele_uitext[20]="twenty";
```

ele_delim - a field only used for checkboxes and radio buttons to record the delimiter to use between items in the series. Possible values are:

'br' - means a line break
'space' - means whitespace
[some user specified value] - which will be used as the delimiter, ie: some custom HTML. Historically this has usually been used to create more whitespace, ie: &nbsp;&nbsp;&nbsp;

ele_forcehidden - 1 or 0, a flag indicating whether we include this as a hidden element when users who can't view this element are viewing the form (this is useful for setting certain defaults that must be set when the form is first saved, regardless of the user's ability to see this element).

ele_private - 1 or 0, a flag indicating whether this element is considered private or not. Private elements can only be viewed by users with the view_private_elements permission on the form.

ele_display - whether this element should be displayed or not. Possible values: 1 or 0 for display to all or display to none, or a comma separated list of group ids corresponding to the groups that should see this element. The group list starts and ends with a comma too: ,4,6,7,9, (that's so we can do a search for ,X, and find X in the string.)

ele_disabled - whether this element should be disabled for some groups or not. Same possible values as for ele_display. Note that for an element to be disabled, all a user's groups must be specified in the list; if a user is a member of a group that is not listed here, then the element will not be disabled for them.

ele_encrypt - 1 or 0, indicating if the element has its data encrypted in the database or not

ele_filtersettings - an array containing the filter conditions that determine if this element should be displayed or not as part of a particular entry

ele_value

ele_value is a special property of the element object. It is usually an array, but its contents, and the number of keys in the array, vary according to what type of element the object is. The contents of ele_value for a textbox are different from the contents of ele_value for a selectbox.

> We believe this technique, originally pioneered in the Liase module for XOOPS, was designed to make it possible to use the same class for all elements, without having certain properties of the class that were only used for some types of elements. With Formulize, we have expanded the elements class in some ways that break this convention (ie: with ele_delim and ele_uitext that only apply to certain types of elements). Nonetheless, the use of ele_value does let developers add properties to certain elements without having to alter the constructor method for the object, or the methods that interact with the database. Instead, new values can simply be tacked onto the end of the ele_value array if new properties are required for a certain type of object.

Contents of each key in the ele_value array, for each type of element

ele_value keys for **text** elements (plain textboxes):

0 - width of the texbox
1 - max length of text that can be entered into the box
2 - default text to appear in the box
3 - 1 or 0 indicating whether the box accepts only numbers (1) or not (0)
4 - associated element - the ID number of an element that this box is associated with. If the text in this box matches a value in the associated element, then on list of entries screens the text from this box will be clickable and will link to the entry where a match was found.
5 - the number of decimals allowed, if this is a numbers only box
6 - the prefix for showing before numbers (usually a currency symbol, but could be anything)
7 - the decimal symbol to use
8 - the thousands separator to use
9 - 1 or 0 indicating whether a unique value is required for this element or not

ele_value keys for **textarea** elements (multiline textboxes):

0 - default text
1 - number of rows (default is 5)
2 - number of columns (default is 35)
3 - associated element - just like for regular textboxes

ele_value keys for **areamodif** elements (text for display in two cells):

0 - text for the second cell (caption is used for first cell)

ele_value keys for **ib** elements (text for display, one cell that spans the form):

0 - HTML contents for the cell
1 - CSS class for the cell

ele_value keys for **checkbox** elements:

Each key in the ele_value array contains the text/value of each checkbox in the series. Each value in the ele_value array contains the checked/unchecked status of that box, either a 1 or 0. Example:

$ele_value['This is box one'] = 0;
$ele_value['This is box two'] = 1;

In that example, 'This is box two' is checked and the other box is not checked.

Pay special attention to the fact it's the *key* and not the value that stores the actual text/value of the checkbox. The value of each item in the array is the checked/unchecked flag.

ele_value keys for **radio** elements:

Exact same as checkbox (but only one value can be set to 1, ie: checked, of course)

ele_value keys for **yes/no radio button** elements:

This element uses an associative array:
$ele_value['_YES'] - 1 or 0 for the default value of the Yes option
$ele_value['_NO'] - 1 or 0 for the default value of the No option

ele_value key for **date** elements:

0 - the date being stored, or "", ie: blank, for no date. Dates are in YYYY-mm-dd format, ie: 2007-06-25. When no date is specified, then the form will default to show YYYY-mm-dd in the form, if the formulize core datebox patch is applied, or if you are using the Standalone version of Formulize.

ele_value keys for **derived** elements:

0 - the block of code used to derive the value
1 - the number of decimals allowed, if this formula produces numbers
2 - the prefix for showing before numbers (usually a currency symbol, but could be anything)
3 - the decimal symbol to use
4 - the thousands separator to use

ele_value keys for **select** elements (the most complex ele_value possibility):

0 - number of rows in the box

1 - 1 or 0, a flag indicating whether the box accepts multiple selections or not (this setting only matters for boxes with more than one row)

2 - either an array just like the checkbox version of ele_value. But there are some complications for selectboxes since they have so many options.

First, if the first key of this array is either {FULLNAMES} or {USERNAMES} then that means this selectbox will be a dynamically generated list of users.

Second, if this is a selectbox that is linked to another element in order to get its options, then key 2 is not an array. Instead, it's a string that looks like this:

[form id]#*=:*[element handle]

For example, 12#*=:*42 would mean the selectbox was linked to the element with handle 42 in form 12. Note that element handles default to be the same as the element ids, but can be overridden by the application creator.

3 - Only has an effect for linked select boxes, or a selectbox that is based on usernames or fullnames. This is a string of group ids, separated by commas, indicating which groups this element's values should be limited to (ie: only entries created by these groups should be included in the dynamically generated list). This string does NOT have a leading and trailing comma. 'all' means use all groups.

4 - 1 or 0, indicating whether a selection of groups should be further limited to only the groups the current user is a member of or not.

5 - an array of arrays, specifying the filter conditions that apply. Each array contains three other arrays, in this format: array(0=>$elements, 1=>$ops, 2=>$terms). $elements, $ops and $terms are each arrays, that are constructed in parallel, and describe the element, operator and search term that are used to filter the values.

6 - 1 or 0, indicating whether a user must be a member of all the groups in a selection, as specified with key 4. If 4 is 1 (ie: limit to groups the current user is a member of), and this item is 1 then only users who are members of all the selected groups that the current user is also a member of, will be included. If 4 is 0 and 6 is 1, then only users who are members of all the selected groups will be included.

7 - 1 or 0 indicating whether the values should be clickable when they appear in lists of entries. If clickable, then they will take the user to the source entry for the value, in the source form.

8 - 1 or 0 indicating whether the element should be treated as an autocomplete box or not

9 - 0, 1, 2 or 3 indicating restrictions on how many times each option can be picked. 0 is no limit, 1 is each option can be chosen one time only, 2 is one time only per user, 3 is one time only per group.

10 - the element ID of the element in the source form that should be used to supply the value shown in lists. By default it will be 'none' which indicates that the element that supplies the options should be used. But with this feature, users can have options from one element, and values displayed in lists coming from a different element in the source form.

11 - the element ID of the element in the source form that should be used to supply the value exported in spreadsheets. Like 10, but for cases where the user wants something different from the actual options used in spreadsheets. Defaults to 'none' which will use the element specified by key 10.

12 - the element ID of the element in the source form that should be used to sort the options. Default is 'none' which results in an alphanumeric sort.

13 - the entry IDs of the entries in the source form that should supply the default values for this element when blank forms are being rendered.

14 - 1 or 0 indicating whether 1) the defaults should be used any time the element is rendered with no pre-existing value (including existing entries where the value is simply blank/empty), or 0) the defaults should only apply to first time entries

ele_value keys for **subform** elements:

0 - the form id of the form being used as a subform
1 - an array of the elements from that form that are to appear in the subform UI
2 - The number of blank default entries in the subform that should be displayed to the user when the form first loads (a big convenience so the user doesn't have to add that many to the form causing a reload before they can enter their info)
3 - 1 or 0 indicating whether the View button should be drawn as part of each row
4 - 1 or 0 indicating whether column headings (0) or captions (1) should be used as the title for each column of elements
5 - 1 or 0 indicating whether new entries created in the subform should be flagged as created by the current user (0) or the creator of the main form entry (1)
6 - 1 or 0 indicating whether the "Add x Entries" UI should be shown at the bottom of the subform interface
7 - The conditions that should be used, if any, for controlling which entries in the subform are shown in this instance of the subform UI. Conditions are enforced upon new entries created through this instance.
8 - a flag, either 'row' or 'form', that indicates whether the full form should be used for each entry in this subform interface, or the traditional UI with one entry per row

ele_value keys for **grid** elements:

0 - the heading option for this grid: 'caption' to use the element's caption, 'form' to use the form's caption, or 'none' to have no caption.
1 - a comma separated string of the captions for each row of the grid
2 - a comma separated string of the captions for each column of the grid
3 - 'horizontal' or 'vertical' indicating the direction of the background shading for this grid
4 - a number indicating the element ID of the element which should appear in the upper left corner of the grid. Elements are drawn into the grid in sequence until there are no elements left to draw in the grid.
5 - a 1 or '' indicating whether the heading for this grid should appear above it, or to the left side like the caption for regular elements. 0 is not used as the negative value, for backwards compatibility with grids created before this feature was added to Formulize.

Methods of the element handler class:

get($element_id_or_handle) - takes an element id number or handle and returns an element object based on that element. Assumes numeric values are ids and non numeric values are handles.

getObjects2($criteria, $fid, $id_as_key) - takes a xoopsCriteria object, and a form id, and returns an array of all the elements that match the criteria. Optionally, the id_as_key boolean can be set to true, to cause the returned array to use the element id numbers as the keys of the array.

Creating your own custom elements

The original, basic form elements in Formulize have their respective bits of logic spread out in different parts of Formulize. However, new elements, such as the file upload element, are based on a new element class, that standardizes and centralizes all the logic for handling elements, into a single file. You can make a copy of the dummyElement-example.php file and start modifying from there to make your own custom element if you need one in your application.

Naming conventions for custom elements

Each custom element file must be named typeElement.php, where type is a unique value you're using as the identifier of your new type of element. ie: provinceListElement.php

So, assuming that's the name of your file, then within the file, the class itself must be named formulizeProvinceListElement and the handler must be named formulizeProvinceListElementHandler.

The file must be placed in the /modules/formulize/class/ folder. Additionally, you need to create an admin template, to control the Options tab for this element type. The name of that file should be element_type_provinceList.html for example. It must be placed in the /modules/formulize/templates/admin/ folder.

Additionally, you may need to update the Formulize module under the Modules area of the administration control panel in your website, before the new template will be recognized. Whether you need to do this or not depends on whether your site's general preferences are set to always process new/changed templates or not.

Properties of custom elements

Custom elements have six properties. They are:

name – this is the name that gets displayed on screen along side the names of other elements

hasData – true/false, set to false if this is a **non-data** element, such as text-for-display or a subform. If users will be storing data in this element, then set this to true.

needsDataType – true/false, set to true if users can select/override the data type for this element. Set to false if you're going to force a specific data type in the database for this element type.

overrideDataType – if needsDataType is false, then use this to specify what the datatype should be. Use MySQL data types. If the user can specify a type, then set this to "".

adminCanMakeRequired – true/false, set to true if the administrators can flag this as a required element or not when they add it to a form.

alwaysValidateInputs – true/false, set to true if you want a custom validation function to always be run for this element. If you turn this on, then you should set adminCanMakeRequired to false, since the custom validation function supersedes any simple check to see that the element has a value or not.

Methods of custom elements

adminPrepare($element)

This method is used to prepare any data that should be sent to the admin-side template. You will probably want to create some UI in the admin side so people can set options for your element. So you will need to send values to the template in order for users to see what options are currently set, etc.

In this method, you will have access to the $element object, so you can inspect the $ele_value array for this element like this:

```
$ele_value = $element->getVar('ele_value');
```

You must return an array from this method, and the keys of the array will be available as template variables, and the values in the array will be the values in the template.

You will also need to create a template file. It should follow the same naming convention, ie: element_type_provinceList.html You must place this file in the modules/formulize/templates/admin/ folder. This template file will be used to create the Options tab for this element type.

In the template, you can use <{$content.key}> to refer to a key from the array you prepared. You can use <{$content.ele_value[0]}> to refer to any specific item from the ele_value array for this element.

You can also name form elements according to the following convention: elements-ele_value[0], elements-ele_value[1], etc. The values passed back for those elements from the admin form, will be automatically assigned to the ele_value property of the element, with the numeric keys you specified. You can pick up the values in the method and do any special logic that may be required.

However, if your element is simple, then you might not need to actually send any custom keys/values to the template at all. Simply relying on the automatic handling of ele_value keys will be enough for many cases. This method simply lets you interact with the template if you require something extra.

adminSave($element, $ele_value)

This method exists so you can do any special analysis and other actions that might be required as part of saving the element. You may not need to do anything at all if your element is simple enough. But if you need to interpret values that the user has selected in the Options tab, then you would do it in this method.

You have the element object available with all its properties. You also have the values from $ele_value available as returned from the form. You would also have $_POST of course, and other superglobals.

You can use the setVar method to assign a value to a property of the element if you need to manually. See dummyElement-example.php for an example of this. Also, if you changed any values from what they would appear as on screen to the user when they clicked Save, then you should return true from this method. That will cause the admin page to reload and display the newly changed value.

loadValue($value, $ele_value, $element)

Use this method to populate an element with the right data so it can be rendered correctly and show the value the user has selected previously, if any.

By default, for new entries, where users have not selected any values for elements, the $ele_value array is passed through the rendering method as-is, with no modifications. When displaying an existing entry, ele_value needs to be modified in the right way, so that when the element is rendered, it shows the value the user last selected for this element.

$value is the value from the database for this element, if any. Based on what you find there, you can alter $ele_value accordingly. For example, if the render method creates a textbox with the contents of $ele_value[1] as the default value, then in this method, you need to take $value and assign it to $ele_value[1] so the element renders what the user selected from the database. You need to return $ele_value from this method.

render($ele_value, $caption, $markupName, $isDisabled, $element, $entry_id)

This is the method that actually creates the form element. It must return a form element based on the XOOPS form classes. The simplest element in the XOOPS form classes is the xoopsFormLabel, which simply takes a caption parameter, and another string. You can make up whatever markup you want for that second parameter, and it will be printed on screen as is. So if you want to make up your own markup and bypass most of the XOOPS form classes, you can do it pretty easily by using the label class.

In this method you will have $ele_value (which may have been previously modified by the loadValue method) as well as the $caption for the element, the $markupName which is the name that must be used as the name attribute of the form element. The $isDisabled flag will be true/false indicating if the element should appear disabled or not. Plus you have the $element object as usual, and the $entry_id of the record that is currently being displayed. This will be “new” for new entries.

generateValidationCode($caption, $markupName, $element)

Use this method to create custom javascript that should run when this element is required (or all the time if you set the element properties so that validation runs all the time).

This method must return an array, and each line is one line of javascript code (yeah, really). Use $markupName to refer to the element, it should be the ID of the element in the markup. If you return false in that javascript, then the form will not submit.

prepareDataForSaving($value, $element)

Once the user has submitted the form, then you will get the value in this method, and you can do any interpretation that needs to be done. Return the value that you want written to the database. In many cases, it might be the same as was received from the user. Besides the $value from the user, you have access to the $element here too.

You can perform security checks here, but Formulize will also wrap everything in mysql_real_escape_string itself, so it's not critical to do that here, though it can't hurt of course.

You can return {WRITEASNULL} if you want a null value written to the database.

afterSavingLogic($value, $element_id, $entry_id)

This method will run after the data has been saved. In some cases, you might need to do something after that has happened. The most common reason for this is if you have to do some work when new entries are saved, but only after an entry id has been assigned to the new entry.

If you want this method to run, you must set this value in the $GLOBALS array, during the prepareDataForSaving method:

```
$GLOBALS['formulize_afterSavingLogicRequired']['elementId'] = type;
```

ElementId must be the numeric element ID, and type must be the unique type name you're using according to the naming convention for this element (ie: provinceList according to the example above).

When this method runs, you will have access to the $value that was just saved in the database, the $element_id number, and the $entry_id number that was just saved.

prepareDataForDataset($value, $handle, $entry_id)

Use this method to do any analysis or preparation when data is retrieved from the database for displaying on screen in a list, or handing out through the API. If you are storing literal values in the database, then you can just return the $value. If however you need to do some interpretation, you can do it here and return whatever you want.

You can use $handle to get the full element object (using the element_handler as shown above in the element class documentation).

prepareLiteralTextForDB($value, $element)

In cases where your database values are different from the literal values that users would select or type in, then you will need to use this method for translating the literal values to something that compares to the values in the database.

For example, if your element stores provinces as numbers in the database, but displays them as names to users, then use this method to convert names to numbers. This is important for when people use your element when specifying conditions and other configuration options, where they would type in a literal value, and that value has to be compared to something in the database.

The work of this method would presumably be very similar or the same as **prepareDataForSaving**, however that won't always be the case. It depends on how your element is rendered. Your element might show names to the user, but send back numbers with the form submission. In that case, you might do no interpretation in prepareDataForSaving, but you would have to do interpretation here. The two methods exist separately so you can account for whatever nuances are required for your element.

formatDataForList($value, $handle, $entry_id)

This method takes a value from a dataset, and potentially alters it for display on screen. The $value will already have been passed through prepareDataForDataset. If there are further adjustments to make for when this data is displayed as part of a list of entries, such as constructing a link out of it, or something else, this is the place to do it.

There are a few additional properties that are defined in this method:

> **clickable** – true/false, set to true if you want Formulize to automatically convert URLs to clickable links for you
>
> **striphtml** – true/false, set to true if you want Formulize to automatically strip HTML tags out of the $value, as a security precaution
>
> **length** – set to a number, this will be the maximum number of characters displayed in the list, beyond this characters will be cut off and a … appended to the end of the $value

This method must pass its return value through `parent::formatDataForList()` in order for the automatic features to kick in. See dummyElement-example.php for an example.