

Inside Formulize: a Developers Guide

November 5, 2009



This document has been prepared by Freeform Solutions. It is licensed under a Creative Commons license (attribution – non-commercial – no derivatives).

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. 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.

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.

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 493:

```
$form = new XoopsThemeForm($title, 'formulize', "$currentURL", "post", true);
```

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.

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.

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 specified by the properties of the current screen (if it's a list of entries screen that is currently being rendered).

If a form itself does not need to be displayed, then execution carries on with the processing of any custom buttons (if it's a list of entries screen that is being rendered).

Then the actual list of data to display is gathered from the database, around line 669. This 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 a list of entries screen is in effect.

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 1279. 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 screen is being drawn, and it has a custom list template. Then each entry in the data set is sent through the custom list template and drawn on the screen that way.

Finally, once the drawEntries function is finished, the bottom template is drawn in, if this is a list of entries screen that has a bottom template.

formdisplaypages.php

This file handles drawing of multipage forms. It is really an alternate wrapper for the displayForm function, since the displayForm function can be used to draw only certain elements of a form, rather than the whole form.

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.

After figuring out what page the user is on, this function then draws the "thank you page" if necessary (if the user has finished the last page of the multipage form).

Otherwise, it calls drawPageNav around line 337. 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.

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.

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

tableform - the name of the datatable this form refers to, if this form is a reference to a 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.

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

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.

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 framework id number and returns an array of form objects based on the forms in that framework.

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";
```


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_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 textbox
- 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

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.

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.

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)

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.

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.

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.

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.

Anatomy of an Element

Elements in Formulize are not currently well encapsulated. There are bits of element-related logic in several places. This should be changed from a development point of view, to make it easier to create and add elements. However, for the time being, here is where the different bits of element-related logic exist.

Creating Elements/Admin Controls for Elements

The file **admin/elements.php** does most of the work displaying the element properties page where you type the name for an element and set all its options. This file handles all the generic stuff that every element shares, like a name, the "display" setting for which groups get to see the element, etc.

Each kind of element also has a separate file of its own, such as **admin/ele_radio.php** for radio buttons. These files contain all the unique options that relate only to that element. For example, default values for textboxes, the list of options for radio buttons, etc. All these element-specific options are packaged up in an array called **\$ele_value** when the admin form gets submitted. \$ele_value will have a varying number of keys depending on the kind of element. These all correspond exactly with the information above about the ele_value property of element objects.

When the admin form is read after submission, and the element is updated (or created if it's a new element), the file **admin/elements_save.php** is where all the logic happens that reads the values submitted from the admin form and saves them to the database.

So, to add a new property to an element, you need to add appropriate options to the element admin form, in admin/elements.php if this option applies to all elements, or admin/ele_[element type].php if it only applies to one type of element.

If you are adding something to a specific type of element, make sure your new option gets the value \$ele_value[X] when it is submitted, where X is the next highest key available. For example, textboxes currently use keys 0 through 8 in \$ele_value, so a new textbox-specific option should use key 9.

You will need to modify `admin/elements_save.php` to listen for your new option, and apply it to the element object properly so the object includes that value when it is saved.

If the property you're adding is for all elements, then you will also need to modify the element class itself. If the property you're adding is specific to one kind of element, and you are just adding a key to `$ele_value`, then you do not need to modify the element class.

The Element Class

The element class will need to be modified any time someone adds a property that affects all elements. A change to the "ele_value" property, such as adding another element-specific option, *will not require a change to the element class*.

The file **`class/elements.php`** is where the element class is defined. It follows the convention laid out for XOOBS objects. The properties are all defined at the top, and then a handler class is written that takes care of all basic operations on the element.

To add a property, it will need to go at the top with the other properties, in the constructor. The property will also need to be added to the insert method. Take care when modifying the insert method, since it uses the `sprintf` function to sanitize the values going into the database, so you have to make sure you are adding the database fields and the reference to the value, in the right place. Generally, following the format of all the other properties will keep you on the right path.

The Database Itself

All elements have their properties stored in the database. If you are adding a property to the element class, then you will need to update the database schema for elements. If you are just adding a value to the `ele_value` property for a specific kind of element, *you do not need to update the database schema*.

To update the database schema, you need to change the **`sql/mysql.sql`** file and add a field to the definition for the **`formulize`** table. That's it. Make sure the name of the field matches exactly with the name of the property you defined in the element class!

Also, in `admin/formindex.php`, there is a database patching function, and an Alter Table statement should be added there to add this new field. This is a trickier part of the process, don't be shy to post on the forums for pointers if you need help finding your way around the database patching process.

Reading Values Users Have Saved in the Database

When users look at an existing entry in a form, the existing values for that entry need to be called up. This is handled in a function called **`loadValue`** in the **`include/formdisplay.php`** file.

The `loadValue` function basically takes the element and takes the existing value from the database, and replaces the normal default value for the element with the value that is in the database already. The process for doing this varies from element type to element type. If you need to interrupt or change this process, you would do so in the `loadValue` function.

Reading Element Properties and Rendering Elements

Elements are all rendered by the **class/elementrenderer.php** file. In this file, all the properties of the element, including element-specific properties in `ele_value`, are interpreted and the element is displayed to the user.

So if you modify an element to add a property, this is where you need to read that property and do something with it. You can use the standard `getVar` method to read an existing property, and when it comes to `ele_value`, you generally need to look at a specific key in that property to find what you're looking for. See the documentation on element objects above for full details of all properties.

Each element ends up as an element object as part of the XOOPS form class system. In general though, if you are modifying an element, you should not have to know much about the XOOPS form classes. You can usually just look at the value for a property and do some logic with it, and then alter some value that gets passed in to the form classes.

Reading Values that Users Submit Through Elements

When users submit values in elements in a form, they all get read by the **include/readelements.php** file. That file should pretty much be a black box, as far as any work you might do with elements is concerned. However, the `readelements.php` file does call an important function called **prepDataForWrite** in the **include/functions.php** file. The `prepDataForWrite` function does have important logic that you might need to alter, if you are changing the meaning of values that the user submits through the form.

Some values that users submit get passed in to the database more or less directly, such as values in a textbox. There is some important sanitization that happens to them, but there is no interpretation. Whereas some values, like the values that get submitted for a selectbox, especially a linked selectbox, do not have a literal meaning that we can just put straight in the database. Some values need to be interpreted and then another value gets put in the database to represent what the user has selected.

So if you are working with what users submit in the form, and need to interrupt the process between the form submission and writing values to the database, look in `prepDataForWrite`.

Making Elements into Independent Objects

If you've read the rest of the details about elements carefully, and looked at the files, you will notice a pattern in how elements are handled in Formulize. There are several places in the page loading and submission process, where a switch statement gets called, and based on the element type, different logic gets executed. This is true in `admin/elements.php`, `admin/elements_save.php`, the `loadValue` function, `class/elementrenderer.php`, and the `prepDataForWrite` function. All those individual bits of code that relate to each type of element, could be collected together into a single class for each type of element, and all those places throughout the page loading process could be modified to call standard methods on the element object(s). It would be really nice if the base element class could be extended so all these different types of elements could share some common characteristics, such as `ele_caption`, `ele_required`, etc. This is a major bit of rearchitecting that would be a good way for someone to get to know their way around Formulize better, if they wanted to dig into the code more.