

FREEFORM
s o l u t i o n s



Using Formulize and Pageworks to Create Custom Applications

A Conceptual Overview and API Documentation

June 6, 2009



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

Table of Contents

Introduction.....	7
 How Pages Get Built.....	7
 Linking to Pages, Forms and Entries.....	8
 Frameworks.....	8
 No Smarty, except for the Calendars.....	8
 About Storing HTML in Forms.....	8
Creating a page in Pageworks.....	9
 A Basic Page.....	9
 Displaying a Form.....	9
Special Variables Passed By Forms.....	10
Publishing on another web site.....	11
Using Frameworks.....	12
 What Are Frameworks?.....	12
 Creating Frameworks.....	12
 Connecting to a Framework the Simple Way.....	13
 Scenarios for using Frameworks.....	13
 Permission Control and Who Sees What.....	16
 Some Uses are Still Experimental.....	16
 Frameworks with just one form.....	16
 Connecting to a Framework in Pageworks and Displaying Data	16
Displaying Pageworks Pages in Blocks.....	18
Advanced Manipulation of Data in a Form.....	19
Functions Summary.....	20

<u>displayForm.....</u>	<u>21</u>
<u>Syntax.....</u>	<u>21</u>
<u>Examples.....</u>	<u>21</u>
<u>Parameters.....</u>	<u>22</u>
<u>displayFormPages.....</u>	<u>25</u>
<u>Syntax.....</u>	<u>25</u>
<u>Example.....</u>	<u>25</u>
<u>Parameters.....</u>	<u>26</u>
<u>displayEntries.....</u>	<u>28</u>
<u>Syntax.....</u>	<u>28</u>
<u>Parameters.....</u>	<u>28</u>
<u>displayGrid.....</u>	<u>29</u>
<u>Syntax.....</u>	<u>29</u>
<u>Example.....</u>	<u>29</u>
<u>Parameters.....</u>	<u>30</u>
<u>displayCalendar.....</u>	<u>31</u>
<u>Syntax.....</u>	<u>31</u>
<u>Examples.....</u>	<u>31</u>
<u>Parameters.....</u>	<u>32</u>
<u>displayFilter.....</u>	<u>35</u>
<u>Syntax.....</u>	<u>35</u>
<u>Parameters.....</u>	<u>35</u>
<u>displayElement.....</u>	<u>36</u>
<u>Syntax.....</u>	<u>36</u>
<u>Examples.....</u>	<u>36</u>
<u>Parameters.....</u>	<u>37</u>

<u>displayElementSave.....</u>	<u>37</u>
<u>Syntax.....</u>	<u>37</u>
<u>Parameters.....</u>	<u>37</u>
<u>displayButton.....</u>	<u>38</u>
<u>Syntax.....</u>	<u>38</u>
<u>Examples.....</u>	<u>38</u>
<u>Parameters.....</u>	<u>39</u>
<u>displayCaption.....</u>	<u>40</u>
<u>Syntax.....</u>	<u>40</u>
<u>Parameters.....</u>	<u>40</u>
<u>displayDescription.....</u>	<u>40</u>
<u>Syntax.....</u>	<u>40</u>
<u>Parameters.....</u>	<u>40</u>
<u>formulize_getCalcs.....</u>	<u>41</u>
<u>Syntax.....</u>	<u>41</u>
<u>Parameters.....</u>	<u>42</u>
<u>formulize_writeEntry.....</u>	<u>42</u>
<u>Syntax.....</u>	<u>42</u>
<u>Parameters.....</u>	<u>43</u>
<u>getData.....</u>	<u>43</u>
<u>Syntax.....</u>	<u>43</u>
<u>Parameters.....</u>	<u>44</u>
<u>Output of getData.....</u>	<u>46</u>
<u>Functions for Gathering IDs from Entries in a Dataset, and Sorting Data</u>	<u>49</u>
<u>Data Parsing and Display Functions.....</u>	<u>51</u>

[Low Level Functions.....55](#)

Introduction

This document explains some of the different scenarios for using Freeform Solutions' Formulize and Pageworks modules. Formulize and Pageworks are tools for creating data-entry applications that model business processes, and they provide easy access to the recorded data for reporting and analysis purposes.

This document is a collection of concepts, examples and reference material that we hope provides a basic understanding of the tools available. We encourage you to post questions and feedback to the support forums at <http://www.freeformsolutions.ca/formulize>.

Creating applications involves two types of work. First there is setup and configuration of forms and frameworks of forms. Second, you can use the screen system in Formulize to customize how things are presented to users. See the Formulize readme for more information. However, in some cases that's not enough, you might need some simple PHP programming. That's where the API reference below comes in.

Often, a key part of building applications is knowing the ID numbers of forms, frameworks and their elements. These are often displayed in the administration areas next to the form or element, but you can also check the URLs in the administration screens for Formulize. When modifying a Framework, the "cf" parameter is the framework ID.

How Pages Get Built

Formulize is the form creation module and handles all interactions with forms and with data. Pageworks, on the other hand, is the optional interface building tool that gives you extra control over what users see on the screen and how they can interact with what they see. Creating an application is a matter of creating the necessary forms in Formulize, to act as data-entry screens and data containers. After that, the "Form Menu" block in Formulize may provide all the basic interaction with the forms that you need.

But if not, you can use Pageworks and create whatever specialized interfaces you require to either those forms or to the data that has been submitted through them, or both.

Pageworks is a very simple module really. All it allows you to do is create individual pages for your website, that are accessible through the following URL structure:

<http://www.yoursite.com/modules/pageworks/index.php?page=34>

That address will display page number 34. Pageworks lets you specify custom PHP code that should be executed when page 34 is accessed, thereby creating a page for the visitor to see. Pageworks also lets you specify certain specialized data containers, or Frameworks, that should be made available to the PHP code for that page. This allows you to easily refer to important data from within your custom code, without having to manually connect to the data source.

Linking to Pages, Forms and Entries

The normal way to integrate Pageworks pages into the navigation scheme for your site, is to use a custom menu module. We like iMenu and have included an updated release of that module along with Formulize and Pageworks.

Each form is also uniquely addressable through the URL, and even each entry in each form. This URL displays entry 132 from form 5. The "ve" parameter is optional; leaving it out displays a blank form:

<http://www.yoursite.com/modules/formulize/index.php?fid=5&ve=132>

Frameworks

FRAMEWORKS ARE OPTIONAL. ALL THE FUNCTIONS BELOW CAN BE USED ON STANDALONE FORMS. But they can also be used with Frameworks, and sometimes that's easier. For a beginner, the important thing to know about the Frameworks system is it lets you specify the relationship between forms, and lets you assign handles to each element in the forms to make it easier to work with the data in the forms (instead of having to refer to the entire caption for an element, which might be lengthy, or the ID number of the element which will be hard to remember).

No Smarty, except for the Calendars

With the exception of `displayCalendar`, the main functions described below each output HTML directly to the screen and do not make use of the XOOPS Smarty template engine. A future version may include Smarty equivalents of all the main functions, so it is easier to customize the appearance of the forms and entry lists to match your own site.

Why haven't we used templates? We have built these tools primarily for use on internal websites for medium and large sized organizations. The public does not access these sites, so building in fully customizable templates has never been a priority. If you require highly customized forms, you can use the `displayElement` function in a Pageworks page, along with your own HTML, to provide a completely customized appearance to the forms, but that's a lot more work than simply using the default form layout.

About Storing HTML in Forms

Formulize converts all HTML characters to their "htmlSpecialChars" equivalents, as a security precaution. This means that if you type HTML into a textbox, you cannot then get that HTML via the `getData` function and just output it to the screen. You can convert the HTML special characters to their regular equivalents using the XOOPS text sanitizer class, or the PHP function `html_entity_decode`. There is also an option on each Pageworks page to allow HTML characters from the DB onto the screen. If you select this option, Pageworks will do the conversion for you.

PART 1 – About creating pages and applications

Creating a page in Pageworks

For details about the various functions used in these examples, consult the reference sections of this document.

A Basic Page

Fundamentally, Pageworks is a container for PHP code that gets executed when the page is viewed. Here's how to make a very simple page in Pageworks:

1. Add a new page in Pageworks. Give it a name and title. Click Save.
2. In the Template Box, type this:

```
print "Hello World";
```
3. Click the Save button. Then on the main administration page for Pageworks, click the Permissions link at the top, and grant access to your new page, for the appropriate groups.

This page will simply display the words "Hello World" on the screen.

Displaying a Form

Suppose you want to have an activity log form, and you want users to be able to see a list of the entries, and the activity log form on the same page. If users click on an activity log entry in the list, then the form will reload with the details for that entry. This kind of interface might be useful if you don't want to present the full blown "list of entries" screen to the users.

First, let's assume you've created the form already, and it's form number 12 in your site. To display the form, do this:

1. Add a new page in Pageworks. Give it a name and title. Click Save.
2. In the Template Box, type this:

```
displayForm("12");
```
3. Click the Save button. Then on the main administration page for Pageworks, click the Permissions link at the top, and grant access to your new page, for the appropriate groups.

That will display the form on the screen, just the same as if someone went to the form via the Form Menu (note: you will need to have setup permissions within Formulize for accessing the form itself).

Next, we need to get a list of entries in the form. Change the contents of the Template Box on your Pageworks page to this:

```
// get all the entries in form 12
$data = getData("", "12");

// getCurrentURL is a special function available in Pageworks pages
$currentURL = getCurrentURL();
// "Activity Log" is the title of the form
// "87" used in the display function is the element ID of the form
// element where the name of activity is entered
foreach($data as $entry) {
    $sids = internalRecordIds($entry, "Activity Log");
    $sid = $sids[0];
    print "<p><a href=\"" . $currentURL . "&entry=$id>" .
display($entry, "87") . "</a></p>";
}

displayForm("12");
```

This page will now display a list of all the activities that have been entered in the form, and each one will be a clickable link. The destination of the link will be the current page, with an extra parameter in the URL: `entry=$id`, where the `$id` is the ID number of that activity's entry in the form.

Lastly, we need to make the page aware of the 'entry' parameter and cause it to display that entry in the form. Change the line:

```
displayForm("12");
```

To this:

```
displayForm("12", $_GET['entry']);
```

Now, if there is a value for 'entry' in the URL, it will be used to display that entry in the form.

Special Variables Passed By Forms

There are some special flags passed by Forms in the `$_POST` array which are useful when creating advanced applications in Pageworks. A couple of the most important are:

`$_POST['form_submitted']` – this is true if the user has just clicked the Save button. It is not present otherwise.

`$_POST['lastentry']` – this is present if the user has just clicked the All Done button, and will contain the ID number of the last entry that was saved, if any. It is not present at any other time.

Publishing on another web site

The `extract.php` file contains the entire data extraction layer for Formulize. It is a self contained file that you can include remotely from other websites, if you want to publish data from your Formulize database elsewhere, outside your XOOPS installation.

For instance, an organization might have a "contact us" section in their static public website, and they have contact information contained in profile forms within their XOOPS-based intranet. Using the code below, it is possible to refer to the `extract.php` file in the XOOPS site, from a page in the otherwise static website. Then you can use all the data display functions (discussed a few sections below) to format the data for public presentation.

In essence, this lets you create simple, custom CMS applications for other websites, where your XOOPS site and Formulize forms become the administration interface for managing the other websites. The code:

```
// *****
// THESE CONSTANTS MUST BE SPECIFIED IN ORDER TO CONNECT TO THE DATABASE
// *****

define('DBHOST', 'localhost');
define('DBUSER', 'dbuser');
define('DBPASS', 'password');
define('DBNAME', 'myXOOPSdb');

// physical path to the root of the XOOPS installation
// (can be relative to the current directory if the sites
// are on the same server)
// (may be possible to use a full URL, ie: http://www.mysite.com
// depending on server config)
define('PATH', '/www/mysite.com/http/');

define('LANG', 'English');

// *****
// CONNECTION FUNCTIONS FOLLOW
// *****

// when included from a non-XOOPS website, extract.php
// automatically connects to the DB specified with the
// constants above.
include PATH . "modules/formulize/include/extract.php";

// *****
// DISPLAY LOGIC FOLLOWS
// *****

// this start function is just a wrapper for getData
// it turns error reporting off and then back on again
function start($framework, $form, $filter="", $andor="AND") {
    error_reporting(0);
    $data = getData($framework, $form, $filter, $andor);
    error_reporting(E_ALL ^ E_NOTICE);
    return $data;
}
```

Using Frameworks

This section is relevant to users who need to relate data in multiple forms together for reporting or display purposes. Frameworks can also make it easier to use certain aspects of the functions described here, since all the functions are able to take framework handles as parameters.

If you don't need to work with multiple forms in special relationships, then skip this section.

What Are Frameworks?

A Framework is simply a series of forms that have some kind of defined relationship. In database terms, it's a join. For instance, in a project management system there might be a client form, and a project form. Each project belongs to one client. So there's a one-to-many relationship between entries in the client form and entries in the project form. Furthermore, there is likely a special field in the project form, a linked selectbox, which points to the client form and contains a list of client names. Linked selectboxes are added to forms just like regular selectboxes, in the administration screens of Formulize. The difference is that instead of the options for the box being typed in manually, a link to an element in another form is specified.

Frameworks are useful because sometimes you want to get a whole series of data (such as information about a client plus all their projects) and a Framework makes it easy to gather that information.

Creating Frameworks

You create Frameworks using the administration screens of Formulize, in the section called "Create or Modify a Form Framework". When you create a Framework you choose two forms and add them to the Framework, you can then specify what their relationship is (one-to-one, one-to-many) and what the key elements are that relate them to one another (in the example above, it would probably be something like "Client Name").

You also have the option of specifying whether the forms should be displayed together when users are entering data. If you specify that there is no "unified display" then the Framework is simply an advanced way of referring to data in the forms. If you specify that the forms should have a "unified display", then in addition to the data in the forms being related according to the options you specified above, the forms can appear on screen together, in their one-to-one or one-to-many relationship.

Once you've saved these options, you then need to click on each form name and type in a form handle. A form handle is just a short word or words that you use to refer to this form when you need to ask for data from it. The idea is that the handle is easier to use than the full form title. You may also want to specify custom handles for each form element. Formulize auto-generates unique handles for the form elements, but if you want meaningful ones (ie: "client" for the element called "Client Name:"), you will have to type them in yourself.

Handles must be unique within the Framework! No two elements, even if they are in different forms, can have the same handle.

Connecting to a Framework the Simple Way

Once you have a Framework, you can access it simply by putting the Framework ID and a form ID into the URL like this:

```
http://www.yoursite.com/modules/formulize/index.php?frid=3&fid=7
```

That will call up Framework number 3 with form number 7 as the primary form (mainform). To get the ID numbers, look in the URL of your browser when you are editing a Framework, for the Framework ID. The "cf" value is the Framework ID. Look in the URL when you are editing the elements in a form for the form ID. The "title" value is the form ID.

The primary form, or mainform, is simply the form that you are most interested in getting data from, with all other forms in the Framework being secondary. For instance, if you wanted to see client data plus all the projects each client has, then the client form would be the mainform. If you wanted to see a list of projects, including client data for each project, then the project form would be the mainform. For more information, see *One to Many Relationships and Changing the Main Form: Reversing the Point of View* on page 15.

Scenarios for using Frameworks

Unified Data Viewing

For instance, imagine a profile form that includes fields like name, age, etc. And imagine an activity log form that people use to record things they have done. The activity log might have fields like task name, description, etc.

The profile form would most likely be setup as a one-entry-per-user form. The activity log is obviously going to be a multiple-entry-per-user form (you would only have one profile for a person, but that person could enter many different activities). This means there's a one-to-many relationship between the forms.

Suppose you want to view all the activity logs someone has entered, plus their profile information. Perhaps you want to search for logs, based on information in a person's profile. This is where a Framework can provide a way to join the data in the forms so you can view it all together. One "entry" in the framework would include a person's profile data, plus data from all their activities. This way you don't have to manually compare and relate information in two views from two forms.

To setup such a Framework, you would select the two forms, specify the relationship as one-to-many, and specify that the link between the forms is the "user ID of the person who filled them in".

To call up the unified list of entries, you can use the simple way to access a Framework through the URL, described above. You could add a link to that URL to a custom menu module such as iMenu. You could also create a custom Pageworks page and use the displayEntries function.

Unified Data Entry – Subforms

Imagine a product list that includes fields like name, price, colour, etc. Imagine that in the data entry form for the product list, you want to include details about the parts that are used in making a product. There will be many parts for each product, and for each part you want to include the same information, such as part number, manufacturer, etc.

There is a one-to-many relationship between the products and the parts. If you create a parts list form in addition to the product list form, and then relate them in a Framework, and turn on "unified display", you will be able to have the parts list embedded into the product list form. Users who go to the product list form will be able to add information about one or more parts used to create the product. All that data will be accessible as a single unit of information; the product data plus all the data about all the parts will be bound together.

Once you have two forms related this way in a Framework, you can add the Subform element to the parent form. This lets you control where the special Subform features appear in the form. For instance, you might want the "parts list" info to appear right after the product name, so you would add the Subform element as the second element in the form.

You could then access the product list using the simple method described above, by just going to a URL with the parameters. You could add a link to that URL to a custom menu module such as iMenu. You could also create a custom Pageworks page and use the displayForm function to display just the form, or the displayEntries function to display the full list of products and part information together.

One to Many Relationships and Changing the Main Form: Reversing the Point of View

Each Framework is simply a description of the links between forms. You must specify the "main form", or the starting point, that you want to use each time you work with the Framework. In the case of the product list/parts list scenario above, the product list is the main form. Each "entry" in the framework would include the product info, plus info about all the parts related to that product. Here's how data from **two** entries in that Framework would be arranged:

```
Product 1 data    Part 1 data
                  Part 2 data
                  Part 3 data
Product 2 data    Part 4 data
                  Part 5 data
                  Part 6 data
```

Suppose you wanted to view that information, but have it arranged by part, so there would be **six** entries, looking like this:

```
Part 1 data      Product 1 data
Part 2 data      Product 1 data
Part 3 data      Product 1 data
Part 4 data      Product 2 data
Part 5 data      Product 2 data
Part 6 data      Product 2 data
```

Achieving this is trivially simple: when using the simple URL access method, just change the value of the "fid", or form ID. When using the displayEntries function, use the part list as the main form parameter instead of using the product list as the main form parameter.

Unified Data - Co-joined Forms

Suppose you have an inventory list. For each item in the inventory, there are certain common pieces of information, such as barcode, type of item, price, etc. But for each type of item, you want certain other fields to appear, and only for that type. For instance, you might only want to store information about the intended age groups for items that are toys. Essentially, part of the form should be common to all items in the inventory, and the other part of the form should be customized based on the kind of item this is.

You could create one common inventory form, and one custom inventory form for each type of item, and then create one Framework for each pairing. With the "unified display" option turned on, this would allow you to display "unique" inventory forms for each type of item, but all the common data would be stored in the single common form, while the data unique to each type would be stored in each type's form.

You could access each Framework using the simple, URL method described above. Or you could use a custom Pageworks page and the `displayForm` function. Note that the `$overrideValue` parameter could be useful for pre-setting the type of item in the common form. If you created another Framework that included all the pairings, you could then access that Framework via the URL, or in Pageworks using the `displayEntries` function, and you could then view a summary list of all the data about each inventory item, from all types.

Permission Control and Who Sees What

Regardless of how forms are included in a Framework, users still must have permission to view and use each form itself. The Framework is just a description of how the forms are related.

You can use this fact to create complex Frameworks with multiple forms, and have some forms available to all groups, and some forms available to only certain groups. This way, when different users interact with the Framework, they would see different forms being unified for display, or different data being brought together for viewing.

Some Uses are Still Experimental

Some of the fancier uses of the Frameworks are still experimental; not all scenarios have been tested extensively, so `displayForm` and `displayEntries` may give odd results in some cases. The general scenarios are all described here, but when you factor in the varying number of forms that can exist in a Framework, and the varying permissions that users can have in forms, the possibilities become quite numerous very quickly. Please pass on any oddities you find, using the forums and trackers in the Formulize area on dev.xoops.org.

Frameworks with just one form

Currently, a Framework must contain at least two forms. But sometimes you want to use just one form, but still use handles to refer to the form and elements, instead of having to figure out the ID numbers of elements. You can do this if you create a "dummy" form with one element and use that as the second form for any "one-form-frameworks" that you want to create.

Connecting to a Framework in Pageworks and Displaying Data

If you connect to a Framework the simple way (via the URL as described above), most of the time you will simply be interacting with the "list of entries" page, or screen. If your Framework has "unified display" turned on, then you may be entering data in forms that way.

But, you can also create customized displays of data in a Framework using Pageworks, in just the same way that you create custom displays of single form data. Here is a basic overview of how you would create a Pageworks page that connects to a Framework. The code below will simply output all the data in the Framework.

1. Create a Framework. Remember to create handles for all the form elements.
2. Add a new page in Pageworks. Give it a name and title. Click Save.
3. Pageworks pages can automatically connect to Frameworks, eliminating the need to manually use the `getData` function to retrieve information. You can still use the `getData` function to connect to a Framework if you want to though, and this can provide more flexibility, particularly if you need to dynamically create filters to control what data is being gathered.

To add a Framework, click Add a new Framework at the bottom of the page. Choose the Framework, choose the main form. Specify an Output Name. We'll assume you used the name "data". (Note, the UI here is not perfected, if you find the dropdown list options don't appear to be updating correctly, save the options you have selected, and the page will reload showing all the other options correctly.)

4. Click Save. Then click Back.
5. In the Template box, type this:

```
// use your own handles here!
$handles[0] = "name";
$handles[1] = "age";
$handles[2] = "location";
// and so on until you've specified all your handles

print "All my data:<br>";

foreach($data as $id=>$entry) {
    print "<br>Data for master ID: $id<br>";
    foreach($handles as $handle) {
        $values = display($entry, $handle);
        if(is_array($values)) {
            print "-";
            print_r($values);
            print "<br>";
        } else {
            print "-$values<br>";
        }
    }
}
```

6. Click the Save button. Then on the main administration page for Pageworks, click the Permissions link at the top, and grant access to your new page, for the appropriate groups.

That code will simply print all the data in your Framework on the screen. Hopefully this gives you a basic understanding of how Pageworks can be used to interface with your data. Since you have complete access to PHP here, you can do whatever conditions and checks you need to create complex behaviours and interfaces. For instance, you can access the `$xoopsUser` object from within a Pageworks page and do something like this:

```

global $xoopsUser;
$thisuser = $xoopsUser->getVar('uid')
foreach($data as $entry) {
    if(display($entry, "uid") == $thisuser) {
        // "uid" is a special handle that all datasets have.
        // It contains the user ID of the user who created
        // the entry.
        // You could do something interesting now, like display
        // some data and put in a link to another pageworks page,
        // and call displayForm on that page so the user can
        // update the data.
    }
}

```

The `displayForm` function in particular can be very useful for giving users the ability to update or change data that is summarized or highlighted in the Pageworks page. For example, as suggested in the code comments above, you could create a list of only certain data that pertains to a user, and provide update links beside each entry in the list. See `internalRecordIds` on page 49 for more information.

For some more details on all this, check out these forum threads:

http://www.freeformsolutions.ca/en/forums/modules/newbb/viewtopic.php?topic_id=1752
http://www.freeformsolutions.ca/en/forums/modules/newbb/viewtopic.php?topic_id=2513

For more details on the structure of the `$data` array and how to manipulate it, see the section below called `Output of getData` on page 46.

Displaying Pageworks Pages in Blocks

You can display a Pageworks page inside a block, if you want it to appear on a specific part of the page, or a specific kind of page in your site.

To do this, make a custom PHP block, and use this code for the contents of the block:

```

$page = 5;
include XOOPS_ROOT_PATH . "/modules/pageworks/index.php";

```

That will cause page number 5 to appear in that block.

You can also refer to a Pageworks page via a URL, if you need to include a pageworks page via some "web services" style system. The following URL will return the HTML output for the Pageworks page, without the XOOPS template surrounding it (the "block=1" part is the key feature):

<http://www.yoursite.com/modules/pageworks/index.php?page=5&block=1>

Advanced Manipulation of Data in a Form

If you are creating advanced applications and custom interfaces to forms, you may need to let users alter the values in certain entries in a form, but you don't want to make them go through the standard form interface. For instance, suppose you have created a Pageworks page that displays all the tasks in a to-do list. If your users need to be able to flag tasks as complete, you could give them a link to the details for each entry in the to-do list and ask them to go to the details, change the status of the task to "complete" and then save that entry in the form.

But wouldn't it be better if they could just click a button beside each task to flag it as complete?

This is what the `displayElement` and `displayButton` functions are for. The `displayElement` function renders a single element from a form on the screen. So if there is a yes/no radio button in a task list form that indicates whether a task is complete, you could embed that element directly in your Pageworks page, and users could update the status of that element for each task, without having to actually go to the details for each entry.

The `displayButton` function also lets users alter values in a form, but in a more controlled way. Using `displayButton`, you could put a button called "Done!" beside each task in the list. You could specify that when the button is clicked, the yes/no option for whether the task is complete should be changed to "yes".

With `displayButton` and `displayElement`, application designers can be freed from the constraints of the default form interface, and in fact they can create completely custom form layouts. You can make extensive use of these functions and never display a form to a user at all. With these advanced techniques, the forms can simply become containers for data.

For examples, see the specific write-ups for these functions below.

PART 2 – Reference Material

Functions Summary

There are several different functions available for use within a Pageworks page. These functions let you do many things with forms and data, including display forms or parts of forms, display the standard list of entries page, display calendars based on the data in a certain form or Framework, or even extract data from a certain form or Framework.

Functions for displaying forms, parts of forms, or entries in forms:

- **displayForm** – display a form, or only certain elements in the form
- **displayFormPages** – display a form with a multi-page layout including skip logic
- **displayEntries** – display the standard "list of entries" page with all its features
- **displayGrid** – display a grid of form elements with custom headings
- **displayCalendar** (and **displayFilter**) – display a calendar view of entries in a form

Functions for displaying form elements, or buttons that will modify values when clicked:

- **displayElement** – display a specific element (including with a value from an entry)
- **displayElementSave** – display a save button
- **displayButton** – display a custom button that will change an entry's value if clicked
- **displayCaption** – display the caption for a form element
- **displayDescription** – display the descriptive text for a form element

Functions for reading and writing in the database, and parsing that data:

- **formulize_getCalcs** – get the calculations that are part of a saved view
- **formulize_writeEntry** – write values to an entry in the database
- **getData** – gather a dataset from the database using search terms and filters
- **internalRecordIds** – get the ID numbers of certain entries within a dataset
- **resultSort** – sort a dataset
- **resultSortRelevance** – sort a dataset based on the prevalence of search terms

Functions for displaying data that has been extracted from the database:

- **display** – return a value from one element in one form in a dataset
- **displayPara** – return values from textboxes as HTML paragraphs
- **displayBR** – return values from textboxes as HTML with
 tags
- **displayList** – return values from textboxes as an HTML list
- **displayTogether** – return values with custom HTML joining each one

displayForm

Syntax

```
displayForm($formframe, $entry, $mainform, $done_dest,  
$button_text, $settings, $titleOverride, $overrideValue,  
$overrideMulti, $overrideSubMulti, $viewallforms)
```

This function draws a form on the screen. When someone fills in the form and saves the data, the form action reloads the current URL and saves any data that is posted from the elements in the form. The various parameters are below.

Examples

Display the form with ID number 4 on the screen:

```
displayForm("4");
```

Display the entry with ID number 245 in form 4 for editing:

```
displayForm("4", "245");
```

Display a Framework of forms on the screen, using the "profile form" as the main form of the Framework (ie: it is the first form drawn, with other forms included based on their relationship to the main form):

```
displayForm("Volunteer Data", "", "profile form");
```

Display the entry with ID 188 in the profile form from the Volunteer Data framework, for editing:

```
displayForm("Volunteer Data", "188", "profile form");
```

Display a form, and set it so that when the user clicks "All Done" they get taken to the front page of your XOOPS site:

```
displayForm("17", "", "", XOOPS_URL);
```

Display a form, and set it so that the "All Done" button reads "Back to the list":

```
displayForm("17", "", "", "", "Back to the list");
```

Display a form, and set it so that the "All Done" button reads "Back to the list", and when the user clicks that button, they will go to a pageworks page number 33:

```
displayForm("17", "", "", XOOPS_URL . "/modules/pageworks/index.php?page=33",  
"Back to the list");
```

Display a form with no "All Done" button at all (only a Save button will appear at the bottom):

```
displayForm("17", "", "", "", "{NOBUTTON}");
```

Display a form and force the default value of a selectbox to be "East Region":

```
displayForm("63", "", "", "", "", "", "", "", "East Region");
```

Display only certain elements in form 63, based on the ID numbers of those elements:

```
$ff['formframe'] = 63;  
$ff['elements'] = array(0=>87, 1=>88, 2=>89, 3=>90);  
displayForm($ff);
```

Parameters

\$formframe - string or number, or array - required

This is either the ID number of the form to draw, or the name of the Framework that you want to access, or the ID number of the Framework you want to access. It is required and cannot be omitted.

If it is an array, it must have two keys: 'formframe' which corresponds to the regular \$formframe variable above, and 'elements' which is itself an array of element IDs that should be displayed. Only those element IDs will be displayed, which allows you to show only part of a form at once.

\$entry - number - optional

This is the ID number of a specific entry which you want displayed in the form so the visitor can update that entry. By default this is empty.

\$mainform - string or number - optional

This is the name (handle) of the form you want to access within the Framework specified with \$formframe. You can also use the ID number of the form. By default this is empty. It is only necessary to specify this if \$formframe is a Framework name or ID.

`$done_dest` - string - optional

All forms have two buttons at the bottom. Save, and a Done button. When the user clicks Save, the form stores their information and reappears with either the information they have just entered, or in some cases with a blank form so they can fill in another entry. The Done button is meant to signify that the user is finished entering or updating their data and wants to go back to wherever they were before.

To accommodate the fact that the proper destination for the Done button could be just about anywhere, you can specify an URL with this string. If no URL is specified, then the current URL is used. By default this string is empty.

`$button_text` - string or array - optional

If specified, this string will be used instead of the default text for the Done button (which is "All Done"). By default this string is empty. You can pass the text "{NOBUTTON}" to this parameter, in order to remove the All Done button from the form.

If this is an array, then the first value (key 0) should be a string that will be used instead of the default text for the Done button. The second value (key 1) should be a string that will be used instead of the default text for the Save button.

If both buttons have the value "{NOBUTTON}" then the bottom row of the form will be omitted. This may be desirable if you are creating some other mechanism to submit the form in your Pageworks page.

`$settings` - array or string - optional

This is a special array used to pass information between various other functions and `displayForm`. By default this array is empty and you should specify it as "" if you need to use any of the subsequent parameters.

There is one special use for this parameter which you may wish to use in certain special cases. If you are calling `displayForm` in a custom Pageworks, but do not want the form to display again after the user clicks the Save button, then set this parameter to "{RETURNAFTERSAVE}". This will cause the data to be saved when the user clicks the Save button, but then the `displayForm` function will immediately return and you can continue with other logic in your Pageworks page.

`$titleOverride` - 1, 0 or "all" - optional

The "all" setting causes all the normal heading information in the form to be stripped out ("About this entry...", etc). Can be useful in conjunction with the `$formframe['elements']` option described above.

When displaying a framework of forms, containing forms that are in a one-to-one relationship with the unified display option turned on, then these forms will be drawn on screen as if they are one form: by default, the second and subsequent forms will be drawn without title bars. However, if this parameter is set to 1, then the second and subsequent

forms will be drawn with their title bars, identifying which form they are. By default this is set to zero, or empty.

\$overrideValue - string or array - optional

If there is a selectbox in the form, then you can set a default selection for that selectbox, other than the first value in the list (which is what is selected by default). To do this, specify a string, or an array of strings here. If there is a date box or boxes in the form, then you can specify a date to be used as the default (applies to all the date boxes). An array can contain mixed overrides, ie: one value could be for a selectbox, and another would be for a date, the system would sort them out (based on date overrides having to be in YYYY-mm-dd format).

Example:

```
$overrides[0] = "East Region";  
$overrides[1] = "2005-09-16";  
displayForm("63", "", "", "", "", "", "", "", $overrides);
```

This parameter only has an effect if no entry is being displayed, ie: it only affects the drawing of blank forms for new entries. By default, this is empty.

\$overrideMulti - 1 or 0 - optional

All forms have a particular setting regarding how many entries a user is allowed to have in the form. That setting can be either one per group, one per user or more than one per user. In the case of forms which are set at "more than one per user" then if there is no `$entry` specified, ie: the user is entering new data, then the form redraws blank after the user saves their data. This is so the user can carry on entering more entries with more data. However in some cases you may wish the form to redisplay the entry the user just made, even though the form allows the user more than one entry. To override the default behaviour and redisplay the entry the user just made, set this to 1.

Alternatively, if you have a form that is set for only one entry per user, in some cases you would rather the form redraw blank. Usually this is when anonymous users are working with the form. Setting this to 1 will also override the default "single-entry" form behaviour so they behave like multi-entry forms.

By default this is set to zero, or empty.

\$overrideSubMulti - 1 or 0 - optional

Just like `overrideMulti`, but used to override the default multi-entry form behaviour for subforms that are part of a framework that is being displayed.

\$viewallforms - 1 or 0 - optional

This is an override setting, which is used to force the display for all forms in a Framework, regardless of the user's permissions to see them.

displayFormPages

Syntax

```
displayFormPages($formframe, $entry, $mainform, $pages,  
$conditions, $introtext, $thankstext, $done_dest, $button_text,  
$settings, $overrideValue)
```

This function is used to display a form as a series of pages instead of as one long form. Each page has certain elements from the form on it, and each page can also have certain conditions attached to it, controlling whether it is displayed or not. The conditions are based on the answers to questions in the form. This allows you to create "skip-logic" typically found in surveys, where certain questions appear or don't appear depending on what the user answered to a previous question.

The syntax for this function's parameters is rather complicated, because of the amount of information you need to specify in order to control the behaviour of a multi-page form. It consists mostly of a series of arrays, including serialized arrays of arrays. There is an example below, with detailed explanations in the Parameters section following that.

Example

This is a two page form, where the second page is displayed only if the answer to element 147 on page 1 is "Yes".

```
$formframe = 22; // use form number 22 - not a framework  
$entry = ""; // show the form blank so a new entry can be saved  
$mainform = ""; // no mainform since we're not using a framework  
  
// create an array that describes which elements appear on which pages.  
// Note: this array just controls which elements appear on the page, while  
// the order that the elements appear in is still determined by the order  
// specified in the Formulize admin pages.  
$pages[1][] = 146;  
$pages[1][] = 147;  
$pages[1][] = 148;  
$pages[1][] = 149;  
$pages[2][] = 150;  
$pages[2][] = 151;  
$pages[2][] = 152;  
  
// setup a condition to control the display of page 2.  
// Conditions are made up of an array containing one or more sets of  
// elements, operators and terms that should be compared against the  
// answers in this form entry so far.  
$element[] = "147";  
$op[] = "=";  
$term[] = "Yes";  
$conditions[2] = array(0=>$element, 1=>$op, 2=>$term);  
  
displayFormPages($formframe, $entry, $mainform, $pages, $conditions);
```

Parameters

`$formframe` - string or number - required

This is either the ID number of the form to draw, or the name of the Framework that you want to access, or the ID number of the Framework you want to access. It is required and cannot be omitted.

`$entry` - number - optional

This is the ID number of a specific entry which you want displayed in the form so the visitor can update that entry. By default this is empty.

```
$formframe, $entry, $mainform, $pages, $conditions, $introtext, $thankstext, $done_dest, $button_text, $settings, $overrideValue
```

`$mainform` - string or number - optional

This is the name (handle) of the form you want to access within the Framework specified with `$formframe`. You can also use the ID number of the form. By default this is empty. It is only necessary to specify this if `$formframe` is a Framework name or ID.

`$pages` - array

This is a multidimensional array that describes which elements should appear on which pages. The first key is the page number that the element should appear on. The value for that key is another array containing the element IDs of all the elements that should appear on the page. The order of the elements doesn't matter; their visible order on the screen is controlled by their order setting in the Formulize admin screen. You can have an unlimited number of pages in a form.

You can also specify pages in your multi-page form that don't contain any elements. You may want a page of text to appear at a certain point and then more form pages to appear afterwards. You can do this by using the string "HTML" or the string "PHP" as the first "element" on a page. The second "element" is the text, either HTML or PHP, to be output to the page, ie:

```
$pages[4][] = "HTML";  
$pages[4][] = "<h1>This is an HTML page in the middle of my form</h1>";
```

`$conditions` - array - optional

This parameter contains all the conditions to apply to a page, to determine whether it should be displayed or not. You can have multiple conditions for a page, and every page in your form can have conditions (though it doesn't make sense to have a condition on the first page).

The first key in this array is the page number that the condition applies to. The value for that key is itself array, with three values in it. Each value is an array, each with an equal number of values. The first array is the elements for each condition, the second is the operators for each condition, and the third is the terms for each condition. So if the elements, operators and terms arrays each have two values, then that means there are two conditions for this page. The first values in each array make up the first condition, and the second values in each array make up the second condition.

The example above hopefully makes this clear. Keep in mind, the example above contains only one condition on one page.

`$introtext` - string - optional

This string should contain HTML that gets printed above the first page of the form. This HTML only appears above the first page.

`$thankstext` - string or array - optional (highly recommended)

This parameter can be a regular HTML string that will get printed on a final page, after the last specified page in the `$pages` array is completed by the user. Alternatively, this parameter can be an array with two values, just like the optional non-form pages. The first should be either "HTML" or "PHP" indicating whether the thank you text should be interpreted as HTML or PHP. The second value in the array should be the actual text, either HTML or PHP.

`$done_dest` - string - optional (highly recommended)

By default, when a user gets to the end of a multi-page form, they will be given a link to click to continue browsing the website. The default destination for this link is the first page of the form. In almost every case, another page in the website would be better than the first page of the form, but only the application developer knows what that page is. You should specify that page as the value of this parameter.

`$button_text` - string - optional

This parameter controls the text of the link on the thank you page which users see after completing the form. It has a generic default value, but something more specific would be better in most cases.

`$settings` - array - special

This is a special array used to pass information between various other functions and `displayFormPages`. By default this array is empty and you should specify it as "" if you need to use any of the subsequent parameters.

`$overrideValue` - string or array - optional

This value is passed through to the form and behaves just like the `displayForm` parameter of the same name.

displayEntries

Syntax

```
displayEntries($formframe, $mainform, $loadview, $loadonlyview,  
$viewallforms)
```

This function calls up the main interface to display lists of entries. This interface includes all reporting features as well, such as searches, sorts, calculations, exporting of data and saving of views.

Parameters

\$formframe - string or number - required

This is either the ID number of the form to draw, or the name of the Framework that you want to access, or the ID number of the Framework you want to access. It is required and cannot be omitted.

\$mainform - string or number - optional

This is the name (handle) of the form you want to access within the Framework specified with `$formframe`. You can also use the ID number of the form. By default this is empty. It is only necessary to specify this if `$formframe` is a Framework name or ID.

\$loadview - string or number - optional

This variable identifies a saved view (or saved report from the 1.5/1.6RC versions of the module) which will be loaded instead of the default view of the entries in this form.

`$loadview` can have the following kinds of values:

- "old_34" – where 34 is an ID number of a report in the old 1.5/1.6RC system.
- 12 – where 12 is the ID number of a saved view in the 2.0 beta system.
- "Name of a view" – where the string equals the name of a saved view in the 2.0 beta system. If more than one view share the same name, then the first one found will be used.

\$loadonlyview - 1 or 0 - optional

This parameter is intended for use with a loaded view. If this parameter is set, then none of the standard views are available in the Current View drop down list. This prevents users from changing the "scope" of the view to include other groups besides the ones that the loaded view uses.

\$viewallforms - 1 or 0 - optional

This is an override setting, which is used to force the display for all forms in a Framework, regardless of the user's permissions to see them.

displayGrid

Syntax

```
displayGrid($fid, $entry, $rowcaps, $colcaps, $title,  
$orientation, $startID, $finalCell, $finalRow)
```

This function displays a table on the screen, containing cells that each have one element from a form in them. It is used to provide an alternative way of displaying a part of a form on the screen, since the default layout of the displayForm function is not always appropriate.

Example

One of the most useful applications of the displayGrid function is to provide a spreadsheet-like series of textboxes. Suppose you have a form that asks for budget information. Suppose the form has textbox elements like this:

Salaries – Budgetted:

Salaries – Actual:

Marketing Costs – Budgetted:

Marketing Costs – Actual:

And so on...

Such a form would be difficult to read and fill in if displayed in the normal way. But it could be easily displayed as a grid using this function:

```
$rowcaps[0] = "Salaries";  
$rowcaps[1] = "Marketing Costs";  
// continue the $rowcaps array for each set of elements in the form  
$colcaps[0] = "Budgetted";  
$colcaps[1] = "Actual";  
displayGrid("27", "", $rowcaps, $colcaps);
```

This will now display the form something like this:

	Budgetted	Actual
Salaries	<input type="text"/>	<input type="text"/>
Marketing Costs	<input type="text"/>	<input type="text"/>

Parameters

\$fid - number. Required

The ID number of the form that you want to display.

\$entry - number. Optional

The ID number of the entry that should be displayed in the form for editing. By default, the form is displayed blank and a new entry is created when the form is saved.

If the form is a one-entry-per-user or one-entry-per-group form, then any existing entry will be displayed in the form if this parameter is left empty.

\$rowcaps - array. Required

This array contains the text that should be used in the left margin of each row. The table will have as many rows as there are values in this array.

\$colcaps - array. Required

This array contains the text that should be used in the heading row above each column. The table will have as many columns as there are values in this array.

\$title - string. Optional

If present, the text in this parameter will be used as the title above the form, instead of the actual name of the form.

\$orientation - "horizontal" or "vertical". Optional

This parameter controls the direction of the background shading in the table. Horizontal shading uses alternate colours for each row. Vertical shading uses alternate colours for each column. By default, this is set to "horizontal".

\$startID - number. Optional

By default, the grid starts with the first element in the form and continues adding elements to the form as long as there are columns and rows to draw in the table. You can use this parameter to specify a particular element to start the grid with, and then the elements will be drawn in from that point in the form onwards according to their order in the form.

This is useful to display only part of a form in a grid, if other elements of the form are better suited to displaying in other ways.

This can be particularly useful when used in conjunction with the option to specify `$formframe` as an array in the `displayForm` function. Doing so lets you have a very long form in Formulize, and you can then then display only certain parts of the form at a time.

\$finalCell - array. Optional

If this array is passed to the function, then the values in this array are drawn in the table cells at the end of each row. They are placed in their own column. The values should be valid HTML, and should not include the <td> and </td> tags.

This parameter is useful if you require some kind of summary information at the end of each row. If you query the form for the data in a particular entry prior to calling the displayGrid function, then you could parse the data and calculate totals or other information that would be useful to display in the final column.

This parameter must have numeric keys.

\$finalRow - string. Optional

If present, the value of this parameter is drawn in the table as a final row. The value should be valid HTML, and should not include the <tr> and </tr> tags. Based on the number of values in the \$colcaps array, you can determine how many cells you need to include in the row. Don't forget that there is one left margin column created for the row captions, and one right margin column created if the \$finalCell array is present.

displayCalendar

Syntax

```
displayCalendar($formframes, $mainforms, $viewHandles,  
$dateHandles, $filters, $viewPrefixes, $scopes, $hidden, $type,  
$start, $multiPageData)
```

This function displays a calendar on the screen, and populates the calendar with data from one or more forms and/or frameworks of forms. Currently only a large month view is supported (the month will take up most of the screen – not suitable for use in blocks).

Several of the main parameters that this function accepts must be set up as arrays that are in-synch with each other, ie: key 0 in each array corresponds to the first dataset, key 1 corresponds to the second dataset.

Examples

Display a calendar style view of the data in form 4. Suppose form 4 is a task tracking form which includes a field called "Deadline", that has ID number 21, and the form also includes a field called "Task Name", that has ID number 15. The Task name will be printed on the calendar in the box corresponding to the deadline

```
$form[0] = 4;  
$text[0] = 15;  
$date[0] = 21;  
displayCalendar($form, "", $text, $date);
```

Suppose form 4 has been added to a framework called "Task Info". The Deadline field has been given the handle "deadline" and the Task Name field has been given the handle "name". Suppose also that you only want tasks that have been assigned to "John Smith" to be shown on the calendar.

```
$framework[0] = "Task Info";
$mainform[0] = "task tracker form";
$text[0] = "name";
$date[0] = "deadline";
$filter[0] = "personname/**/John Smith";
displayCalendar($framework, $mainform, $text, $date, $filter);
```

(See the write up for `getData` for details about the filter syntax.)

Suppose that, instead of the name of the task being drawn on the screen, you want "Due Today! – " to be used as a prefix before the task name. Add one more parameter:

```
$prefix[0] = "Due Today! - ";
displayCalendar($framework, $mainform, $text, $date, $filter, $prefix);
```

Suppose you want to display meeting times and task due dates on the same calendar:

```
$framework[0] = "Task Info";
$mainform[0] = "task tracker form";
$text[0] = "name";
$date[0] = "deadline";
$prefix[0] = "Task Deadline: ";
$framework[1] = "Staff Info";
$mainform[1] = "meeting bookings";
$text[1] = "meeting";
$date[1] = "date";
$prefix[1] = "Meeting:";
displayCalendar($framework, $mainform, $text, $date, $filter, $prefix);
```

Note that you can use the same data source as an input to the calendar multiple times. You might want to do this if there are two different dates in a particular form or framework and you want both of them displayed on a calendar. ie: an order tracking form might have an order date and a ship date on it, so you would use the same form as the data source twice, but each time you would use different fields for the `dateHandle` and `viewHandle`.

Parameters

`$formframes` - array - required

This is an array of all the form IDs or framework names to use as data sources for this calendar. Required.

`$mainforms` - array - optional

This is an array of the main forms for each framework. If a given array key in `$formframes` refers to a Framework, then the corresponding array key in `$mainform` must be a valid main form for that Framework. Otherwise, it is not required.

\$viewHandles - array - required

This is an array of the element IDs (for a form) or the element handles (for a Framework) of the elements that are to be displayed on the calendar. ie: if you want task names displayed on the calendar, then pass in a reference to the textbox element labelled "Task Name". This array is required, and there must be a valid handle or ID for each dataset.

You can specify multiple handles for each dataset. For instance, you may want to include the name and the city on a calendar view of conference dates. If you specify an array of handles for a given dataset, then the values for all those handles will be used. Example:

```
$viewHandles[0][0] = "name";  
$viewhandles[0][1] = "city";
```

\$dateHandles - array - required

This is an array of the element IDs (for a form) or the element handles (for a Framework) of the elements that contain the dates on which each entry should be displayed. ie: if you want task names displayed on their due date, then pass in a reference to the datebox element labeled "Deadline". This array is required, and there must be a valid handle or ID for each dataset.

If you have two date fields in the underlying form, you can use one as the start date and one as the end date, which allows you to plot a date range on the calendar. To specify a start and end date instead of just a single date, use an array. The first value in the array is the handle of the start date, and the second value is the handle of the end date. Example:

```
$dateHandles[0][0] = "startdate";  
$dateHandles[0][1] = "enddate";
```

\$filters - array - optional

This is an array of filters to apply to the data extraction operation for each dataset, if any. The syntax of the filters is discussed on page 44. This array is optional, and if it is present, a filter is only required for each dataset that you want to apply a filter to.

\$viewPrefixes - array - optional

This is an array of text strings to append to the beginning of the text returned by the \$viewHandle for this each dataset. See the examples above for an illustration. This array is optional, and if it is present, a prefix is only required for each dataset that you want to apply a prefix to.

\$scopes - array - optional

This is an array of scopes to use when getting data to populate the calendar with. A scope limits the entries returned to only those that were made by users in a certain group or groups. Valid values are:

- "mine" – include only entries made by the current user.
- "group" – include all groups that the user is a member of.
- "all" – include all groups.
- a list of specific group IDs separated by commas and with commas at the front and back. ie: ,5,12,14,

This array is optional, and if it is present, a scope is only required for each dataset that you want to apply a scope to.

\$hidden - array - optional

This is an array of values that you want remembered when the page reloads. This is meant primarily for situations where you have other form elements manually created in a Pageworks page that calls `displayCalendar`. In that situation, when a user clicks on a part of the calendar interface, the state of the manually created form elements will be lost unless they are include in `$hidden`, and the manual code is programmed to look for them in `$_POST` on page loading.

\$type - string - optional

This is a string meant to indicate what sort of calendar to draw. Currently, only "month" is supported. Month is the default value, and this string is optional.

\$start - string - optional

This string is used to indicate the starting year and month for the calendar. The format of the string must be: YYYY-mm, ie: 2005-09. This string is optional. The default value is the current month.

\$multiPageData - array - optional

This parameter is used to integrate a multi-page form with a calendar. If a multi-page form is used with a calendar, then this array must contain four values, each corresponding to the following parameters for `displayFormPages`: `$pages`, `$conditions`, `$introtext`, `$thankstext`.

displayFilter

Syntax

```
displayFilter($page, $name, $filtername, $element, $overrides)
```

This function is used to put a dropdown list on the same Pageworks page as a calendar, and the dropdown list operates as a filter that limits the items drawn on the calendar to only ones that match the value selected in the list. The filtering is based on an element in the underlying form, which you specify as one of the parameters of the function. The values for that element become the options in the dropdown list.

For example, if you have an event calendar and each event is classified according to regions – north, south, east and west – and there is an element in the form which is used to specify the region, then you could use this function to create a filter with those four options in it and selecting east, for example, would result in only events that occur in the east region being displayed.

Currently, this function works only when tied to an underlying selectbox in the form.

Parameters

\$page - number - required

This must be the ID number of the Pageworks page where the displayFilter function is being used.

\$name - string - required

This is a name to be used as the name of the form that the filter creates for itself. It can be anything, as long as it doesn't conflict with any existing forms that are created in the Pageworks page.

\$filtername - string - required

This is a name to be used as the name of the filter itself within its form. It can be anything.

\$element - number - required

This must be the element ID number of the element on which the filter options should be based.

\$overrides - array - optional

This is an array containing a default value for the filter.

displayElement

Syntax

```
displayElement($framework, $element, $entry)
```

This function is used to display a particular element from a form on a Pageworks page. Each element displayed is tied back to a particular entry in the form, so the value displayed reflects the current state of that entry, and if the value is modified and saved, that entry in the form is updated. If this function is used in a Pageworks page, a "Save" button will appear at the bottom right of the page, allowing users to save any changes they may make to the state of any elements in the page.

This function is aware of the display settings for an element. So if you try to display an element to a user who is not a member of a group that is allowed to see the element, then the element will not display.

This function will return the string "rendered" if it successfully renders the element. It returns "not_allowed" if the user is not supposed to see the element, and "invalid_element" if you do not pass a valid \$element to it.

Also of note: there is a strange sounding configuration option for all form elements called "Include as a hidden element for users who can't see it". This has an effect when a new entry is being created. If an element is passed to displayElement, but the user is not allowed to see it, and the "Include as hidden..." option is on for the element, then the element will be included as a hidden element in your form, and the default value for the element will be submitted with the form. This is useful if you must set certain default values in the database for your application to function correctly. If the displayElement function renders an element this way, then it returns the string "hidden".

FYI: the displayGrid function makes use of the displayElement function to present the elements in the table. Consulting the source code for displayGrid may be instructive to an advanced developer.

Examples

Suppose you have a task list form and there is a yes/no question in the form indicating if the task is complete. That question has element number 43. You want that yes/no option to appear beside each entry in the task list that you draw on a particular Pageworks page.

```
$tasks = getData("Tasklist", "tasks"); // details on getData are below

foreach($tasks as $onetask) {
    // see below for details of the internalRecordIds function
    $ids = internalRecordIds($onetask, "tasks");
    print display($onetask, "name");
    displayElement("", 43, $ids[0]);
    print "<br>";
}
```

The output of the above code would be a list of tasks from the Tasklist framework, and beside each one would be element 43 from the form, showing the value of that element for that particular task. In this example, that is supposed to be a yes/no radio button indicating whether the task is complete or not.

Parameters

\$framework - string - optional

The name of a framework that you are working with. Specify a handle for any form element in the framework using the \$element param (see below).

This parameter can be omitted if you use a numeric element ID in the \$element parameter, or an object in the \$element parameter.

\$element - number, string or object - required

If a number, this parameter is treated as the element ID number of the element you want to display. If a string, it must be the handle of an element in the framework specified above. If an object, it must be the element object that you want rendered.

\$entry - number - optional

This parameter is the ID number of the entry that the value in this element should be based on. Optional. If omitted, then any value entered into the element and saved by the user will result in a new entry being created in the form.

displayElementSave

Syntax

```
displayElementSave($text, $style)
```

This function is used to manually place a Submit button on the page. By default, Pageworks will add a button labeled "Save" to the lower right corner of any page where the displayElement or displayGrid functions are used. However, you may wish to manually specify what such a button should be labeled and/or where it should appear on the page.

If so, then use the displayElementSave function to place the button on the page.

Parameters

\$text - string - optional

This is the text that will appear on the button. By default is it "Save".

\$style - string - optional

This text will be rendered as a CSS inline style for the button.

displayButton

Syntax

```
displayButton($text, $element, $value, $entry, $action, $type, $framework)
```

This function displays a clickable button, or text link, on a Pageworks page, and if the user clicks it then the value of a particular element in a particular entry in a form is altered. Like `displayElement`, this provides a way for users to interact with data in forms without actually working with the form itself. In this case, the users don't even interact with the actual element associated with the value.

This function does not work with linked selectbox elements.

Examples

Suppose you have task list, and you want to list the tasks, and give users a big button they can click to say the task is done. There is an element in the form which indicates whether the task is done or not, but rather than giving the user that element to work with, you just want to give them a button to click. The element in the form is number 43 and is a checkbox labeled "task completed".

```
// assume $task is the output of a previous getData function

foreach($task as $onetask) {
    // for details of the internalRecordIds function see below
    $ids = internalRecordIds($onetask, "task");
    print display($onetask, "name");
    displayButton("Done!", 43, "task completed", $ids[0], "replace");
    print "<br>";
}
```

Suppose you have list of activities and you want to let users sign up for them simply by clicking a button. You could create a form where each entry is an activity, and include a textarea box in the form where user's names will be entered. Then make a Pageworks page that lists the activities that have been entered, and use `displayButton` to make a button beside each activity and if a user clicks on the button, their name will be added to the textarea box.

```
// assume $activities is the output of a previous getData function

foreach($activities as $activity) {
    // for details of the internalRecordIds function see below
    $ids = internalRecordIds($onetask, "task");
    print display($activity, "name");
    displayButton("Sign Me Up!", 25, $xoopsUser->getVar('name'), $ids[0],
"append");
    print "<br>";
}
```

Parameters

\$text - string - required

This is the text that will appear on the button (or in the link)

\$element - number, string or object - required

If a number, this parameter is treated as the element ID number of the element where the value is going to be modified.

If a string, this parameter is treated as the framework handle of the element. (See the \$framework parameter below.)

If an object, this parameter must be the element object that you want the button to modify the value for.

\$value - string - required

This is the value that should be used to modify the existing value of the element.

\$entry - number - optional

This is the ID number of the entry where the value is going to be modified. Optional. If not specified, then a new entry is created with the value.

\$action - string - optional

Valid values are: replace, append or remove. This is an optional parameter which indicates the type of modification to perform. Replace overwrites the current value with the one specified for the \$value parameter here. Append adds \$value to the end of the current value for the element. Remove searches the current value of the element for \$value and if found, removes that text from the current value. The default is "replace".

\$type - string - optional.

This optional parameter controls whether the thing the user clicks is a button or an HTML link. There is no functional difference between them, it is purely aesthetic. The default is "button". To specify a link, use "link".

\$framework - string - optional

The name of a framework that you are working with. Specify a handle for any form element in the framework using the \$element param (see above).

displayCaption

Syntax

```
displayCaption($formframe, $element)
```

This function returns the caption for an element and nothing else. Useful with `displayElement` to provide custom control over a form's layout.

Parameters

\$formframe - string - optional

The name of a framework that you are working with. Specify a handle for any form element in the framework using the `$element` param (see below).

This parameter can be omitted if you use a numeric element ID in the `$element` parameter, or an object in the `$element` parameter.

\$element - number, string or object - required

If a number, this parameter is treated as the element ID number of the element you want to display. If a string, it must be the handle of an element in the framework specified above. If an object, it must be the element object that you want rendered.

displayDescription

Syntax

```
displayCaption($formframe, $element)
```

This function returns the description for an element and nothing else. Useful with `displayElement` and `displayCaption` to provide custom control over a form's layout.

Parameters

\$formframe - string - optional

The name of a framework that you are working with. Specify a handle for any form element in the framework using the `$element` param (see below).

This parameter can be omitted if you use a numeric element ID in the `$element` parameter, or an object in the `$element` parameter.

\$element - number, string or object - required

If a number, this parameter is treated as the element ID number of the element you want to display. If a string, it must be the handle of an element in the framework specified above. If an object, it must be the element object that you want rendered.

formulize_getCalcs

Syntax

```
formulize_getCalcs($formframe, $mainform, $savedView, $handle, $type, $grouping)
```

This function is used to retrieve the calculations that are in effect for a certain saved view. The raw HTML output for the calculations is returned as a multidimensional array with the following structure:

```
$resultArray
  [element handle]
    [calculation type]           - sum, min, max, count, avg or per
    [auto incrementing number] - one for each grouping in effect
    ['result' or 'grouping'] - 'result' is the HTML for the calculation
                              - 'grouping' is a comma separated list of
                                all the values that this calculation has
                                been grouped by
```

You can then iterate over the returned array to manually do something to display the calculations. This is useful for creating custom dashboards of the current state of data in forms. This example code will print a simple listing of all the calculations returned by the function:

```
// $calcs is the result of a call to
// the formulize_getCalcs function
function printCalcResult($calcs) {
    $element_handler = xoops_getmodulehandler('elements',
'formulize');
    foreach($calcs as $handle=>$thisCalcData) {
        $elementObject = $element_handler->get($handle); //
only works when handle and ele_id are the same
        $caption = $elementObject->getVar('ele_caption');
        print "<h3>$caption</h3>\n";
        foreach($thisCalcData as $type=>$results) {
            if(count($results)>1) {
                foreach($results as $result) {
                    print "<p><b>Grouped By: ".implode(", ",
$result['grouping'])."</b></p>\n";
                    print $result['result'];
                }
            }
            else {
                print $results[0]['result'];
            }
        }
    }
}
```

Parameters

\$formframe - string or number - required

This is either the ID number of the form to draw, or the name of the Framework that you want to access, or the ID number of the Framework you want to access. It is required and cannot be omitted.

\$mainform - string or number - optional

This is the name (handle) of the form you want to access within the Framework specified with \$formframe. You can also use the ID number of the form. By default this is empty. It is only necessary to specify this if \$formframe is a Framework name or ID.

\$savedView - string or number - required

This must be either the ID number of the saved view, or its name as typed when it was saved. This is the saved view that contains the calculations you are looking for.

\$handle - string or number - optional

If included, this value will be used to limit the calculations returned to only those performed on this element.

\$type - string or number - required

If included, this value will be used to limit the calculations returned to only those of this type.

\$grouping - string or number - required

If included, this value will be used to limit the calculations returned to only those with this value as one of the calculations grouping values. Note that this will match the value that the grouping is performed with, not the element that the grouping value comes from. For example, if you have a series of sum totals being grouped by a "province" element, then if you specify "Ontario" for this parameter, only the calculations where the province is "Ontario" will be returned by the function.

formulize_writeEntry

Syntax

```
formulize_writeEntry($values, $entry, $action, $proxyUser,  
$forceUpdate, $writeOwnerInfo)
```

This function is used to write data to an existing entry, or to create a new one.

Parameters

\$values - array

This is an array of the values to write to various elements in the form. The keys must be either the element ids that are being written to, or the element handles. The values must be the values to write to the element identified by the key. The elements must all be from the same form!

\$entry - number or string

The ID number of the entry being written to, or "new" for creating a new entry.

\$action - deprecated - no effect

\$proxyUser - number - optional

The ID number of the user who should be recorded as the creator of this entry, if this is a new entry, and the current user's ID should not be used.

\$forceUpdate - true or false - optional

By default this is false, but if it is set to true, then the query will be executed even on a GET request, when database updates are normally disallowed.

\$writeOwnerInfo - true or false - optional

By default this is true, but if set to false, then the entry ownership information for a new entry will not be written to the the entry_owner_groups database table.

getData

Syntax

```
getData($framework, $form, $filter, $andor, $scope)
```

This function extracts data from the specified form or Framework, using the specified filter and scope options. It returns a multidimensional array that contains all the found data in the form or Framework.

You do not have to use this function to get data into a Pageworks page! There is a graphical user interface in the Pageworks admin section which lets you specify what Frameworks to hook up to the page. That UI effectively specifies a series of getData operations to execute automatically every time the page loads. You can use this function to manually perform data extractions if you need to, but a lot of the time, that won't be necessary. See page 16 for more information about using the UI to connect to a Framework.

Parameters

\$framework - string or number - optional

This is the name of the Framework that you want to access, or the ID number of the Framework you want to access. It can be omitted if you are querying a form directly. By default is it empty.

\$form - string or number - required

This is the name of the form within the Framework you are accessing, or the ID number of the form. It is a required parameter and cannot be omitted.

\$filter - number or string or array - optional

If this is a number, then that number is treated as the ID of the record that you want to return. Only that record will be returned.

If this is a string, then it is assumed to be in the format described below, containing filtering options to apply to the query that extracts the data from the database. By default this is empty, no filter is applied.

This is formatted as a string (instead of an array which might seem more logical) in order to make it easier to work with filter parameters using the GET method if necessary. The format of the string is as follows:

```
handle1/**/term1][handle2/**/term2][etc...
```

The filtering is very simple, it only looks for a 'LIKE' match within the field.

The handles must either be the Framework handle of the element you want to filter on, or if you are querying a form directly, then you can use the ID number of the element. The full caption of the field you are filtering on can also be used if you are querying a form directly, but this is strongly NOT recommended if you are running a multi-language site (the language tags will screw up the filter parsing).

Example:

```
prov/**/ON][city/**/b
```

Assuming prov is the Framework handle for a field called Province, and city is the Framework handle of a field called City, then the filter above will return data pertaining to all cities in Ontario that have the letter b in their name.

Note that the][are only used to separate search terms, they do not precede or conclude the filter string.

Operators

You can specify particular operators to use for each part of the filter, if LIKE is too general for your needs. To do this, add on another term separated by `/**/`, for example:

```
prov/**/ON/**/!=
```

This filter will return data where the value of the prov field is not equal to ON.

Newest

There is a special kind of filter that is **valid only for a date field**. This is called the "Newest" filter and it is specified as follows:

```
HandleForDateField/**/term/**/newestX
```

Where X is a number indicating the number of records you want returned. This special filter simply involves the addition of the extra term "newest" plus the number.

The normal term is ignored if the special newest term is specified. **Note again: this is only valid for date fields.** ie: this will return the X number of records with the most recent dates in that field.

Example:

```
date/**/whatever/**/newest5
```

This example will return the records with the five most recent dates in the field identified by the handle Date.

Metadata Filters

There are five special field names that are reserved for metadata, ie: information about an entry. They are: uid, proxyid, creation_date, mod_date and creator_email. Uid is the ID of the user who created the entry. It never changes. Proxyid is the ID of the user who last modified the entry. Creation_date is the date the entry was created. Mod_date is the date the entry was last modified. Creator_email is the email address of the user who created the entry (taken from their account profile). You can use these four special field names as part of a filter string:

```
"uid/**/" . $xoopsUser->getVar('uid') . "/**/!="
```

That is a valid filter string which will return entries that were not created by the current user.

If \$filter is an array, then it can be structured as a series of filter strings, each with their own and/or setting. This technique is used to handle more complex boolean operations than can be handled by a single filter string with one common and/or value (which is set by the next parameter).

The format for the array is as follows:

`$filter[0][0]` ... this is the and/or setting for the first filter string. Valid values are "and" or "or".

`$filter[0][1]` ... this is the first filter string to use. It follows all the syntactic rules as described above.

`$filter[1][0]` ... this is the and/or setting for the second filter string.

`$filter[1][1]` ... this is the second filter string to use.

`$filter[2][0]` ... and so on....

\$andor - AND or OR - optional

This parameter specifies how to interpret multiple filters, ie: a filter string that contains more than one set of handles and terms separated by `]`. By default it is set to AND, and so *the overlapping* set of records found by all the filters will be returned. If it is set to OR, then all records found by all filters will be returned.

For example, in the example above, `prov/**/ON][city/**/b`, if `$andor` is set to OR, then all records where either one of "province contains ON" or "city contains letter b" will be returned. If `$andor` is set to AND or left at the default setting, then only records where both those conditions are true will be returned.

\$scope - string or array - optional

This parameter is used to filter records according to the group membership of users. This parameter is used extensively by the logic within the other main functions, but this should not need to be used by basic applications. By default this parameter is empty (entries by all users from all groups are returned).

The easiest way to use this parameter is to pass an array of group IDs. Those groups will be used as the scope.

A more complex, but valid way to use this parameter is to pass a specially formatted string:

`uid = X OR uid = Y OR uid = Z` and so on.

This of course puts the burden of determining the user ids on the application developer, but this is a useful approach if you want to limit the scope to a group of users that is smaller than any particular group in the website.

Output of getData

The most important thing to understand about `getData` is what it gives you. It is helpful to know the basics of the output array that it creates. Although the display functions discussed in the next section eliminate the need to refer to all the bits and pieces of the array, you won't be able to build sophisticated applications without understanding the structure of this array, and therefore how to traverse it.

The abstract description of this array would be something like this:

```
$array [master id][form handle][record id][field handle][value id] = value
```

The only values contained in the array are the actual pieces of information submitted through the form(s) that was queried. **The array simply adds a whole bunch of array keys to help identify where each value came from.**

Here's an example of real data. Assume the result array is called "data":

```
$data[0][profile][22][name][0] = "Geoff"
$data[0][profile][22][age][0] = "99"
$data[0][activity log][49][activity name][0] = "work on workshops "
$data[0][activity log][49][activity characteristics][0] = "hard"
$data[0][activity log][49][activity characteristics][1] = "boring"
$data[0][activity log][55][activity name][0] = "documentation"
$data[0][activity log][55][activity characteristics][0] = "hard"
$data[0][activity log][55][activity characteristics][1] = "literary"
$data[1][profile][16][name][0] = "Cory"
$data[1][profile][16][age][0] = "66"
$data[1][activity log][31][activity name][0] = "website audit"
$data[1][activity log][31][activity characteristics][0] = "time consuming"
$data[1][activity log][31][activity characteristics][1] = "challenging"
$data[1][activity log][31][activity characteristics][2] = "boring"
$data[1][activity log][38][activity name][0] = "site hacking"
$data[1][activity log][38][activity characteristics][0] = "challenging"
```

So, that is a data set with two results in it. Each result is made up of three separate entries in two different forms. One entry is from the profile form, and two entries are from the activity log form. The fact that there are only two activity log entries per main result is just chance. Obviously over time the two users would create several activity log entries and would probably not create the exact same number.

The first number identifies the main result that each "line" belongs to. The second value is the handle of the form that the data is taken from. The third number is the ID number of the entry in that form which this data is coming from. This is an important point.

Remember that the data in each form is stored sort of like this:

Activity Log Form:

```
31,activity name=CHR website audit
31,activity characteristics=time consuming, challenging, boring
38,activity name=LTS site hacking
38,activity characteristics=challenging
49,activity name=work on LTS workshops curriculum
49,activity characteristics=hard, boring
55,activity name=documentation
55,activity characteristics=hard, literary
```

Profile Form:

```
16,name=Cory  
16,age=66  
22,name=Geoff  
22,age=99
```

So that number in the third position represents the ID of the entry in the particular form that contains all the data related to the specific value on that line. To put it another way, if you want to get all the data about Cory from the profile form, then you need to query for ID 16 in the profile form. If you want to get all the data about Cory's activity on documentation, then you need to query for ID 55 in the activity log form.

Compare the result set above, with the description of the underlying data in the forms that produced it. Hopefully you will quickly see where all the information in the array keys comes from.

IMPORTANT POINT: the "internal" ids are always unique. No two entries in any forms will ever have the same internal id, so you don't have to worry about needing to check these or the data for validity.

The fourth value in the results array is the handle of the form element that the data was entered into. The fifth value is the id number of that value within that form element. In most cases, this is simply a zero, and there is only one value. But in some cases, for instance checkboxes, or multiple selectboxes, where users can pick more than one response for a form element, there would be multiple values. The hypothetical activity characteristics element in this example is like that. Suppose it is a series of checkboxes. That leads to a variable number of answers and so there are varying numbers of values related to that form element in the final results array.

IMPORTANT POINT: if you are working with a form directly, not through a Framework, then instead of the form and element handles used in the Framework, the IDs of the form elements, and the title of the form are used.

There are five pieces of metadata for each entry in the result set: uid, proxyid, creation_date, mod_date, creator_email.

They are accessible using the display functions (see below), just like any regular form element. Just use the names uid, proxyid, creation_date, mod_date or creator_email as the \$handle.

creation_uid is the ID of the user who created the entry (the owner of the entry). This never changes.

mod_uid is the ID of the user who last modified the entry.

creation_datetime is the creation date of the entry.

mod_datetime is the most recent modification date of the entry.

creator_email is the e-mail address associated with the account of the user who created the entry.

Prior to version 3.0, creation_uid was just uid, mod_uid was proxyid, creation_datetime was creation_date and mod_datetime was mod_date. These old handles still work in version 3.0, but are deprecated.

Functions for Gathering IDs from Entries in a Dataset, and Sorting Data

internalRecordIds(\$entry, \$formhandleOrId, \$id)

This important function takes the same \$entry and \$id parameters as the display function, but instead of an element handle, it uses a form handle (or title of the form if no framework is being used). It will return an array of all the IDs of the entries in that particular form which are part of the requested entry. For example, considering the sample data set above, this line:

```
$entry = $data[0];  
$ids = internalRecordIds($entry, "profile", 0);
```

Will return an array like this: [0] => 22.

You can then use the id or ids returned to drill down on a specific entry on a subsequent page. For instance, you could include "22" in the URL for a link to a "details" page that would provide a specially formatted view of all the data for that particular entry, in this case, someone's profile.

The \$formhandleOrId parameter is actually optional, and if omitted then an array will be returned where the keys are *all* the form handles in the dataset, and the value for each will be an array of the ids of the included entries from that form.

The \$formhandleOrId parameter can also be an array of handles, or form Ids, if you want to get the ids for entries in multiple forms at once. In that case, the keys will be all the form handles you specified, and the value for each will be an array of the ids of the included entries from that form. Specifying \$formhandleOrId as an array only makes sense if you're dealing with a Framework and there's more than one form making up the data in the dataset.

Example:

Pageworks page 1:

```
// this page will display a list of activity log entries, with the
// name of the person who made the entry linked to their profile
// form details.
foreach($activitydata as $entry) {
    $ids = internalRecordIds($entry, "profile");
    $id = $ids[0];
    print "<p>" . display($entry, "activityname") . ", done by <a href=" .
XOOPS_URL . "/modules/pageworks/index.php?page=2&id=$id>" . display($entry,
"name") . "</a></p>";
}
```

Pageworks page 2:

```
// this will display the profile form entry of the user who made
// the activity log entry that was clicked on in page 1

displayForm("Profile Plus Activities", $_GET['id'], "profile", XOOPS_URL .
"/modules/pageworks/index.php?page=1");

// no data is referred to or displayed on this page but the id parameter
// in the URL will act as a filter that will restrict any data extractions
// that do happen to only the data for this particular entry See note below:
```

IMPORTANT POINT: Pageworks will automatically look for a URL parameter called "id" (ie: Pageworks checks to see if `$_GET['id']` exists), and if so it will attempt to use that value as a filter in any Frameworks specified for that page using the admin side interface for doing so (ie: Frameworks you are querying with manual `getData` functions are not affected). This allows for convenient drilling down to details, since the output array for the page will contain only the data related to that particular entry. The filter will only be valid if it references an entry that is part of the Framework.

However, developers should be careful here, since abuse of the id param in the URL could result in users seeing entries that they would not normally be allowed to see. Appropriate security checks should be made, ie: checking the value of certain data that has been returned, and checking the group membership of the user to see if that group ought to see that data.

resultSort(\$data, \$handle, \$order, \$type)

This function takes an entire result set and sorts it according to the values in a particular field. *The sorted result set is returned.* So proper usage is like this:

```
$data = resultSort($data, "orderdate");
```

Parameters:

`$data` – the entire results set.

`$handle` – the element handle within the set that you want to sort by.

`$order` – Optional. Valid values are `SORT_ASC` and `SORT_DESC`. Default is `SORT_ASC`.

`$type` – Optional. Corresponds to the Type options available to the PHP sort functions (`SORT_NUMERIC`, `SORT_STRING`, `SORT_REGULAR`). Default is `SORT_REGULAR`.

resultSortRelevance (`$data`, `$handles`, `$terms`, `$weights`)

This function is used to sort a dataset based on certain search terms. Pass in the dataset, along with an array of handles (or element IDs if not using a framework) of elements in the dataset, and an array of search terms. Optionally, you can pass in a weighting array with numbers corresponding to each handle, indicating their relative importance.

The dataset is sorted by counting the number of occurrences of each search term in each specified element, and multiplying the number of occurrences by the optional weighting (1 is used if no weighting is specified). The entry in the dataset with the highest score takes first position and so on.

Example:

```
// get a list of products
$data = getData("Product List", "Products");

// Parse search terms from the user. This function does not use the terms
// in any database queries, so it is *not* an SQL injection risk to pass
// the user specified text straight to the function in this case.
$searchterms = str_replace(",", "", $_POST['searchterms']); // remove commas
$terms = explode(" ", $searchterms); // breakup into array

// specify the fields to search, using handles from the framework
$handles[] = "productname";
$handles[] = "productdescription";

// consider hits in the product name three times as valuable as hits
// in the description
$weights[] = 3;
$weights[] = 1;

$data = resultSortRelevance($data, $handles, $terms, $weights);
```

Data Parsing and Display Functions

There are several functions available for formatting and displaying data returned from the `getData` function. The examples below refer to the sample dataset above extensively, so it pays to be very familiar with it.

display(\$entry, \$handle, \$id, \$localid)

Returns the value(s) of an element handle within a given entry.

`$entry` – either an entire results set, or a single entry from a result set (see below). Required.

`$handle` – either the element handle of the field you want to access (if you're working with a Framework), or the ID of that element (if you're not working with a Framework). Required.

`$id` – if an entire results set is passed in with `$entry`, then this is the array address of the entry in the results set you want to access. Optional, but necessary if an entire results set is passed through `$entry`.

`$localid` – this parameter is used to access the values for just one specific entry within a form that makes up the entire entry being accessed (see below). Optional.

Examples:

The following are both valid, and equivalent:

```
$results = getData("staff", "profile");  
  
print display($results, 'name', 0); // prints the name from master result 0  
  
$entry = $results[0];  
print display($entry, 'name'); // prints the name from master result 0
```

The `localid` is used to display only values related to a certain entry in a certain form. For instance, in the hypothetical result set above, the following code would display the activity characteristics related to Geoff's first activity log entry:

```
$results = getData("staff", "profile");  
$entry = $results[0]  
$values = display($entry, 'activity characteristics', '', 49);
```

ie: the `$values` array would be: `[0]=>'hard' [1]=>'boring'`

Contrast with:

```
$entry = $results[0]  
$values = display($entry, 'activity characteristics');
```

Without the filter for internal id 49, `$values` would be:

`[0]=>'hard' [1]=>'boring' [2]=>'hard' [3]=>'literary'`

Note also that the `display` function can return an array, if there is more than one value corresponding to the element name specified. In the first example above, passing `'name'` results in the single name being returned, as a string. In the second example, `$values` becomes an array since there is more than one value for activity characteristics. Of course, this is a factor of the number of values assigned to a particular entry in a particular form element, not a factor of the form element itself. So although this example is very similar to the previous example, `$values` would not be an array:

```
$entry = $results[1]
$values = display($entry, 'activity characteristics', '', 38);
```

\$values would equal 'challenging'. This is because in the master result entry number 1 (as opposed to number 0), internal id 38 contains only one value for the activity characteristics form element.

displayPara(\$entry, \$handle, \$id)

This function is used to take the contents of a textarea box and formats it as a series of HTML paragraphs. ie: the following text:

```
This is a paragraph
This is another paragraph
This is yet another paragraph
```

```
This is a fourth paragraph
```

Would be returned like this:

```
<p>This is a paragraph</p>
<p>This is another paragraph</p>
<p>This is yet another paragraph</p>
<p>This is a fourth paragraph</p>
```

Note that blank lines in the textarea box are ignored.

The parameter syntax is exactly the same as for the display function; an entire result set plus master ID, or a single entry can be passed in.

There can be more than one instance of the requested form element in the entry requested, for instance suppose there is another field in the activity log form in the hypothetical results array above, a field called 'activity description'. Suppose 'activity description' is a text area box. We would expect that master id 0 would have two values generated from this box, one for internal id 49 and one for 55. Similarly, master id 1 would have two values, one for internal id 31 and 38. In that case, displayPara would return an array containing HTML formatted text like above, where each value in the array corresponds to one of the text areas in the master entry.

displayBR(\$entry, \$handle, \$id)

This function returns the contents of a text area box and formats it as HTML with simply
 tags between each line. ie: the following text:

```
This is a paragraph
This is another paragraph
This is yet another paragraph
```

```
This is a fourth paragraph
```

Would be returned like this:

```
This is a paragraph<br>
This is another paragraph<br>
This is yet another paragraph<br>
This is a fourth paragraph<br>
```

Note that blank lines in the textarea box are ignored.

The syntax is exactly the same as for the display function; an entire result set plus master ID, or a single entry can be passed.

There can be more than one instance of the requested form element in the entry requested, for instance suppose there is another field in the activity log form in the hypothetical results array above, a field called 'activity description'. Suppose 'activity description' is a text area box. We would expect that master id 0 would have two values generated from this box, one for internal id 49 and one for 55. Similarly, master id 1 would have two values, one for internal id 31 and 38. In that case, displayPara would return an array containing HTML formatted text like above, where each value in the array corresponds to one of the text areas in the master entry.

displayList(\$entry, \$handle, \$type, \$id, \$localid)

Like the other "display" functions, this one supports the same parameters, with the addition of the \$type parameter, which can be either "bulleted" or "numbered". Default is bulleted. Note that this function supports the \$localid parameter while displayBR and displayPara do not (yet).

This function returns the contents of a text area box and formats it as an HTML list. ie:

```
This is a paragraph
This is another paragraph

This is yet another paragraph
```

Would be returned like this:

```
<ul>
<li>This is a paragraph</li>
<li>This is another paragraph</li>
<li>This is yet another paragraph</li>
</ul>
```

Note that blank lines in the textarea box are ignored.

There can be more than one instance of the requested form element in the entry requested, for instance suppose there is another field in the activity log form in the hypothetical results array above, a field called 'activity description'. Suppose 'activity description' is a text area box. We would expect that master id 0 would have two values generated from this box, one for internal id 49 and one for 55. Similarly, master id 1 would have two values, one for internal id 31 and 38. In that case, displayPara would return an array containing HTML

formatted text like above, where each value in the array corresponds to one of the text areas in the master entry.

displayTogether(\$entry, \$handle, \$sep, \$id, \$localid)

This function is very similar to the other "junior" display functions, except it takes the value passed to \$sep and uses that as "glue" between all the values that are returned. It is intended to be used with form elements that allow multiple selections, such as checkboxes and multi-select boxes, rather than textboxes. It will take all the values returned by the standard display function and merge them into a single string with \$sep in between each one.

Low Level Functions

Within the Pageworks template, you may need to perform some other tasks related to forms and data. To assist, there are some other miscellaneous functions available, including the important ones listed below. Any function in the Formulize include/functions.php file is actually available within a Pageworks page, but most of those functions are only useful when called internally from displayForm or displayEntries.

getCurrentURL()

This function simply returns the current URL complete with all parameters.

q(\$query, \$keyfield)

This function takes a valid SQL query and returns the results in an array format. Intended for use with SELECT statements. *This function is very memory intensive, so be careful with queries that return thousands of records!*

Example:

```
$groups = q("SELECT groupid, name FROM xoops_groups");
```

\$groups would then be a two dimensional array, where the first dimension is a number indicating the record and the second dimension is one of the field names selected. ie:

```
$groups[0]['groupid'] = 1  
$groups[0]['name'] = Webmasters  
$groups[1]['groupid'] = 2  
$groups[1]['name'] = Registered Users
```

If you use the optional \$keyfield parameter, then the value of that field will be used as the key for each record returned. ie:

```
$groups = q("SELECT groupid, name FROM xoops_groups", "groupid");
```

Result:

```
$groups[1]['groupid'] = 1  
$groups[1]['name'] = Webmasters  
$groups[2]['groupid'] = 2  
$groups[2]['name'] = Registered Users
```

IMPORTANT POINT: this particular SELECT statement is hopefully easy to understand and makes a good example for the concept of the q function, but no one should actually be gathering group information this way! Use the XOOPS core classes, particularly the member handler class, to get this kind of information about groups and users.

printSmart(\$value, \$chars)

\$value – a text string. Required.

\$chars – the number of characters of the string you want displayed. After this number of characters, the rest of the string is replaced with "...". Optional. Default is 35.

This function returns the newly formatted string, it does not print the result to the screen.

```
writeElementValue($framework, $element, $entry, $value, $action,  
                  $prevValue)
```

This function is deprecated and should not be used any longer. Use `formulize_writeEntry` instead (described above). If you need help writing data to the database, please post in the support forums and ask about the data handler class, and/or read the `formulize/class/data.php` file.