



SynApp2.org

# SynApp2 Customization

Web Application Generator Reference



## SynApp2 Customization

*SynApp2* is a generalized, reusable body of program code whose behavior varies predictably, at key points, according to prevailing conditions and applicable rules. The underlying framework provides many elements and behaviors that may be subtly influenced or controlled outright, by various means. The setup – *customization* – of the mechanisms is how the business logic of a *SynApp2* powered web applications is expressed.

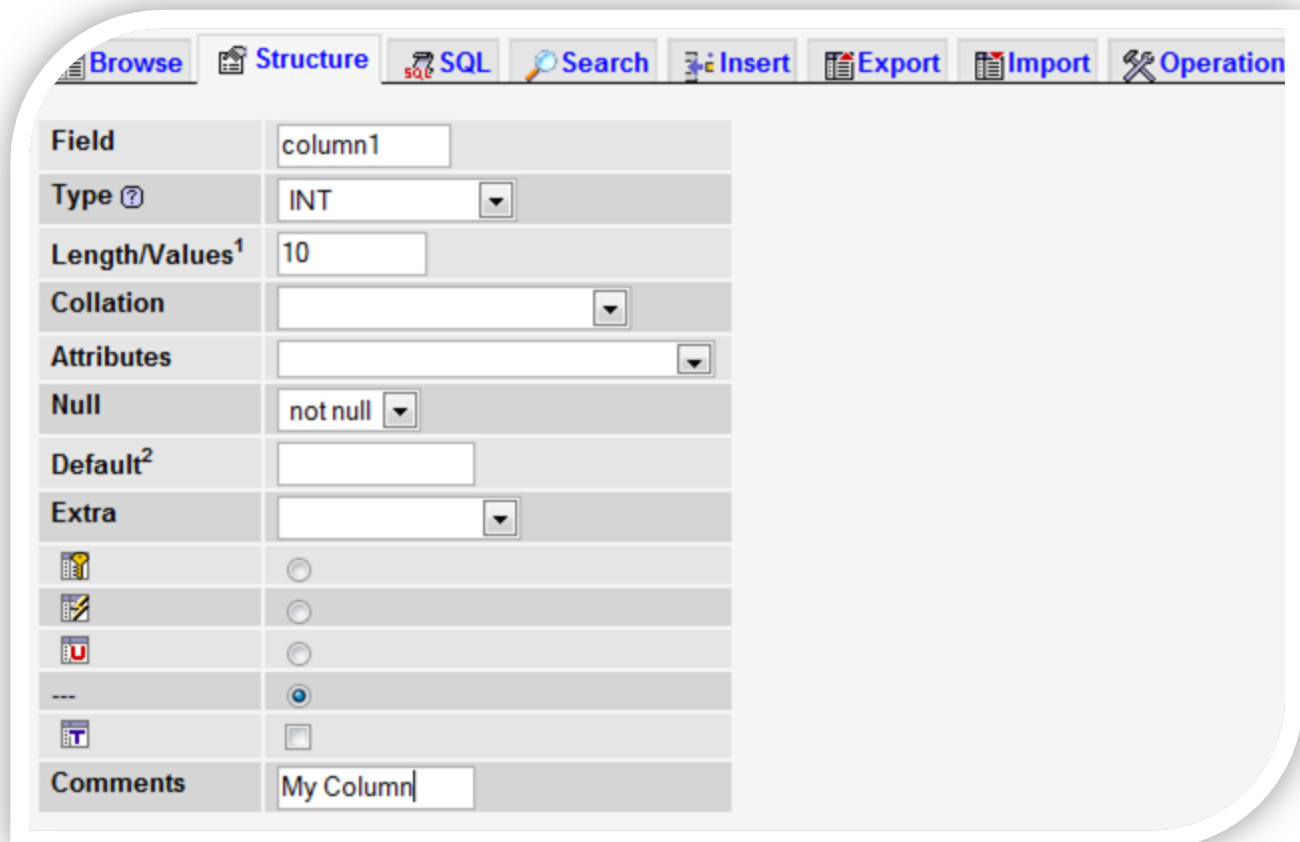
### Display Name

Perhaps the most common and basic customization, you'll likely make, will be to specify a legend, or *display name*, for columns where the table column name isn't suitable for the user interface (UI). In such cases, you may supply an alternative by entering a value as a comment for the field/column. The way you do this varies depending on the database engine involved.

For *MySQL*, at least, using an administration tool such as *phpMyAdmin* can make life easier. If, for some reason, you're using *MySQL* and you don't have *phpMyAdmin* – please don't waste another minute – get it. Download *phpMyAdmin* from <http://www.phpmyadmin.net>.

Assuming that you're using *phpMyAdmin* to manage your database and table structure, enter the display name into the **Comments** field as you `CREATE` or `ALTER` your tables.

Adding a table column with *phpMyAdmin*:



The screenshot shows the 'Structure' tab in phpMyAdmin. The 'Add New Field' form is displayed with the following fields and values:

| Field                      | Value                            |
|----------------------------|----------------------------------|
| Field                      | column1                          |
| Type ?                     | INT                              |
| Length/Values <sup>1</sup> | 10                               |
| Collation                  |                                  |
| Attributes                 |                                  |
| Null                       | not null                         |
| Default <sup>2</sup>       |                                  |
| Extra                      |                                  |
| Primary                    | <input type="radio"/>            |
| Index                      | <input type="radio"/>            |
| Unique                     | <input type="radio"/>            |
| --->                       | <input checked="" type="radio"/> |
| Fulltext                   | <input type="checkbox"/>         |
| Comments                   | My Column                        |

If you're using a command line interface to manage your database definitions, here are two possible ways to manipulate a field/column COMMENT.

```
mysql>ALTER TABLE table1 CHANGE column1 column1 <col def> COMMENT 'My Column';  
  
SQL>COMMENT ON COLUMN table1.column1 IS 'My Column';
```

If you do not customize a field/column display name, during PageGen, *SynApp2* will generate a display name, from the field/column name, by replacing underscores with spaces and then converting the first character of each word to upper case (e.g. `first_name` → `First Name`).

## COMMENT Mechanism

Beyond display names, the mechanism of using the field/column COMMENT to customize behavior, applies to *numeric range validation*, *column initialization*, *default values* and *TIMESTAMP columns*.

Specialized tokens and expressions may appear as comma separated values (CSV) in the field/column COMMENT. The display name must always come first. If there is no alternative display name, a leading comma must be present, followed by customization tokens or expressions.

```
Synopsis: <display name>, <token or expression>, <token or expression> ...
```

Using the field/column COMMENT, in this way, encapsulates at least a few important customization elements into the database definition itself. The customizations may even prove to be portable between different database engines. You'll find it quick and convenient to specify basic display customization and validation limits as you define the database structure.

## Numeric Range Validation

Use the COMMENT mechanism to cause *SynApp2* to trigger a clear and concise diagnostic message, prompting the user to enter a valid value, anytime a corresponding field violates a numeric limit as the *input* (*iform*) is submitted.

```
Synopsis: <display name>,min=<value>  
         <display name>,max=<value>  
  
Examples: Quantity, min=0.001, max=999999.999  
         Discount, min=0
```

Notice the comma between display name and the validation expression. The comma must be present whether or not you specify a display name.

*SynApp2* will automatically trigger a diagnostic message if a negative value is entered for an UNSIGNED value, or if a fractional value is submitted for a field/column designated as any flavor of INT. If your database engine supports the UNSIGNED attribute, using it might be preferable to using a limit of `min=0`.

## Form Field/Column Value Initialization

If your database engine supports specifying simple default values, often as part of the table definition, then the specified values will be used to initialize form field/column values, unless specialized initialization has been defined – as described below.

There are times when one or more values should be dynamically determined, returned to, and presented by the application UI during the interactive sequence of adding new records.

Use the `COMMENT` mechanism to call out specialized initialization for column values. Value generation occurs as the `input (iform)` is initialized - after the *Add* event, but prior to the pending `ACTION_INSERT` being carried out.

```
Synopsis: <display name>,auto=<function>  
         <display name>,init=<function>
```

```
Examples: By,auto=login_username()  
         Customer Since,init=year_today()
```

Notice the comma between display name and the key generation expression. The comma must be present whether or not you specify a display name.

The `auto` token will cause related `input` elements to be generated with the `READONLY` attribute.

Several pre-defined functions are available:

- `login_username()`
- `date_today()`
- `weekday_today()`
- `day_today()`,
- `month_today()`
- `year_today()`

The pre-defined function definitions appear in `_shared_/custom.inc.php`.

See Custom Initialization Functions, later in this document, for information about how to create your own functions.

Note that primary key (PK) values must be generated and applied by mechanisms implemented within your database engine. For *MySQL*, `AUTO_INCREMENT` is typically used. You may use any procedures and/or trigger support that your engine provides as long as it's automatic and transparent.

## Applying Default Values on Insert or Update

You can use the `COMMENT` mechanism and the `ON_EMPTY_VALUE_SET_DEFAULT` token to cause *SynApp2* to apply the specified field/column `DEFAULT` value, during a record `INSERT` or `UPDATE` operation, if the corresponding value is left empty (blank) as the *input (iform)* is submitted.

Admittedly, this is not obviously expected behavior for your end-user. And, if you don't like it – then don't use it. But – the behavior can be useful, and it doesn't require additional UI control elements to make it work. A reminder note as part of the display name or a user trained according to a policy of – “if in doubt, leave the field blank” – may be all that is needed to have a convenient and effective way to restore a default value.

Do not use the `ON_EMPTY_VALUE_SET_DEFAULT` token if the field/column, is the primary key (PK), a foreign key (FK), or in the case of *MySQL*, if the type is `TIMESTAMP`.

## Required Values

If your design requires users to enter a value for a field, use `NOT NULL` for the field/column when you `CREATE` or `ALTER` the table. If `NOT NULL` is specified, *SynApp2* will generate a diagnostic message prompting the user to enter a value, anytime a corresponding field remains empty (blank) as the *input (iform)* is submitted.

## TIMESTAMP Columns

Database field/columns defined as `TIMESTAMP` are treated as read-only on any *input (iform)* where they appear. Typically, there is only one `TIMESTAMP` field per table. How the value of the `TIMESTAMP` field value is determined varies according to the database engine.

*MySQL* provides `TIMESTAMP` properties that make it relatively easy to define a single field/column that will set the value when a record is inserted and/or updated. With a little more work, it's possible to define two columns, one that will reflect the `TIMESTAMP` of the record creation (insert) and another that will track the `TIMESTAMP` of the last update.

Define the `TIMESTAMP` column to track record creation as:

```
NOT NULL DEFAULT '0000-00-00 00:00:00'.
```

Define a second `TIMESTAMP` column to track last record update as:

```
NOT NULL DEFAULT CURRENT_TIMESTAMP
```

and with the attributes:

```
ON UPDATE CURRENT_TIMESTAMP.
```

*SynApp2* will detect and automatically set the value of all `TIMESTAMP` columns that do not have the `ON UPDATE` attribute. This behavior effectively tracks when a new record is inserted. *MySQL* handles the record update column according to the attribute.

The mechanisms for managing values for `TIMESTAMP` columns in other database engines, such as *Oracle*, will require other techniques. The column initialization mechanism, described above, can be used to set the value of a `TIMESTAMP` column used to *approximate* the time of

record creation. The value will reflect when the *input* (*iform*) is initialized and presented, not the moment of record insertion.

*Work is pending to provide means of managing not only `TIMESTAMP` values, but values that get computed and recorded when `on_update` or `on_create` events occur.*

## Include File Mechanism

The typical way to customize database query elements is with an include file that you create and maintain in the directory where your *SynApp2* generated application resides.

The name of the customization include file is `custom.inc.php`. An empty file is automatically created for you, when you generate the first page of your application. Put your application specific customizations in this file. *SynApp2* also creates a similar file, named `synapp2.inc.php`. This is where *SynApp2* stores customizations and state information that is manipulated via the *SynApp2* pages - Options and PageGen.

There is also a `custom.inc.php` file in the `_shared_` directory. Customizations that are intended to affect all applications, that share a *SynApp2* installation instance, should be done there. The local file is included *after* the shared file, so it is possible to override some things in the local file.

## Implementing Customization

Most customization will take the general form of PHP statements that initialize elements of an associative array. The array is a member of the class `custom` and has the name `m_data`. The customization data structure values are typically a string, or an array with named elements. See the file `_shared_/custom.php` for implementation details.

You can, of course, use any valid PHP expression(s) to construct these customization statements. Thus, imbedding something like `{$my_var}` within a customization value, or even *within a key expression*, is perfectly reasonable - it's PHP code. Be sure to use single and double quotes appropriately.

Use a plain-text editor to make changes to customization files.

Here is an example of a very simple `custom.inc.php`:

```
<?php
$this->m_data[APPID]['sample'][DATABASE] = 'census';
?>
```

The `custom.inc.php` file is also a place where you can define functions or anything else that you can use to support your customization objectives.

**IMPORTANT:** Customization files are included from within the body of a function and therefore any variables that you define in customization files, have limited scope. Functions referenced by *SynApp2* custom element initialization statements are called during instantiation of the `custom` object, and not when the customization elements are (later) accessed.

## Customization Effects

Some customizations affect the *output* of *SynApp2* PageGen, as well as having an impact at application run-time.

In general, customizations that affect whether or not a column of data gets produced will have an effect on the code written to the *SynApp2* generated web pages. Also, customizations that trigger a request/response cycle, when a record is selected, result in some supporting elements and/or code needing to be written to the applicable web page(s). If changes are made to these kinds of customizations, you should use PageGen to re-generate potentially impacted pages. Changes to your database structure, should also be followed with re-generation of related pages.

At run-time, an instance of a customization object presents an interface, to other objects during execution of the *SynApp2* server-side framework, which reflects values and behaviors for important logical abstractions, at key points, and in response to prevailing conditions. The customization object strategically parameterizes activity during all request/response cycles.

Study the methods for class `custom` in the file `_shared_/custom.php`. Also, search the other PHP source files, in the `_shared_` directory, to see where and how the `custom` class methods are invoked, while considering the potential effects.

## Interactive Customization

The *SynApp2* Options pages provide for interactive manipulation of many elements.

The screenshot shows the 'SynApp2 Options' window. At the top, there are tabs for 'KeyMap', 'Options', 'PageGen', 'Tools', 'Reports', and 'Logout'. Below the tabs, there are input fields for 'AppID (or database): census' and 'QueryID (or table): country'. A row of buttons includes 'Display', 'Macro', 'List Macro', 'List Macro Order', 'Extra', 'Order', 'Col Order', 'Detail Summary Cols', and 'Reporting'. The main content area is titled 'Options Form - Column Display Attributes' and contains a table with columns for 'Search Form', 'Display Form', 'Select Form', 'Input Form', 'Report Form', and 'Report Column'. The table has four rows: 'Country Name' (VARCHAR 40), 'Country Abbreviation' (CHAR 3), 'National Flag' (EXTRA), and 'All Columns'. Each row contains settings for 'Omit', 'Size', 'Align', 'Editor', 'Rows', and 'Format'. A 'Process Column Display Attributes' button is at the bottom left of the table. Below the table, there is a link 'Jump to: census.page Reports'. The status bar at the bottom says 'SynApp2 - Ready.'

|   | Search Form                             | Display Form                            | Select Form  | Input Form  | Report Form                             | Report Column   |
|---|---|---|--|---|---|---|
| <b>Country Name</b><br>country_name<br>VARCHAR 40             | Omit: No<br>Size: <input type="text"/>  | Omit: No<br>Size: <input type="text"/>  | Omit: No<br>Size: <input type="text"/><br>Align: -- default -- | Omit: No<br>Editor: -- default --<br>Size: <input type="text"/><br>Rows: <input type="text"/> | Omit: No<br>Size: <input type="text"/>  | Omit: No<br>Size: <input type="text"/><br>Align: -- default --<br>Format: <input type="text"/>  |
| <b>Country Abbreviation</b><br>country_abbreviation<br>CHAR 3 | Omit: No<br>Size: <input type="text"/>  | Omit: No<br>Size: <input type="text"/>  | Omit: No<br>Size: <input type="text"/><br>Align: -- default -- | Omit: No<br>Editor: -- default --<br>Size: <input type="text"/><br>Rows: <input type="text"/> | Omit: No<br>Size: <input type="text"/>  | Omit: No<br>Size: <input type="text"/><br>Align: -- default --<br>Format: <input type="text"/>  |
| <b>National Flag</b><br>national_flag<br>EXTRA                | Omit: Yes<br>Size: <input type="text"/> | Omit: Yes<br>Size: <input type="text"/> | Omit: No<br>Size: <input type="text"/><br>Align: Center        |   | Omit: Yes<br>Size: <input type="text"/> | Omit: Yes<br>Size: <input type="text"/><br>Align: -- default --<br>Format: <input type="text"/> |
| <b>All Columns</b>  |   |   | Rows: 5  |   |   |   |

Process Column Display Attributes

[Jump to: census.page Reports](#)

SynApp2 - Ready.

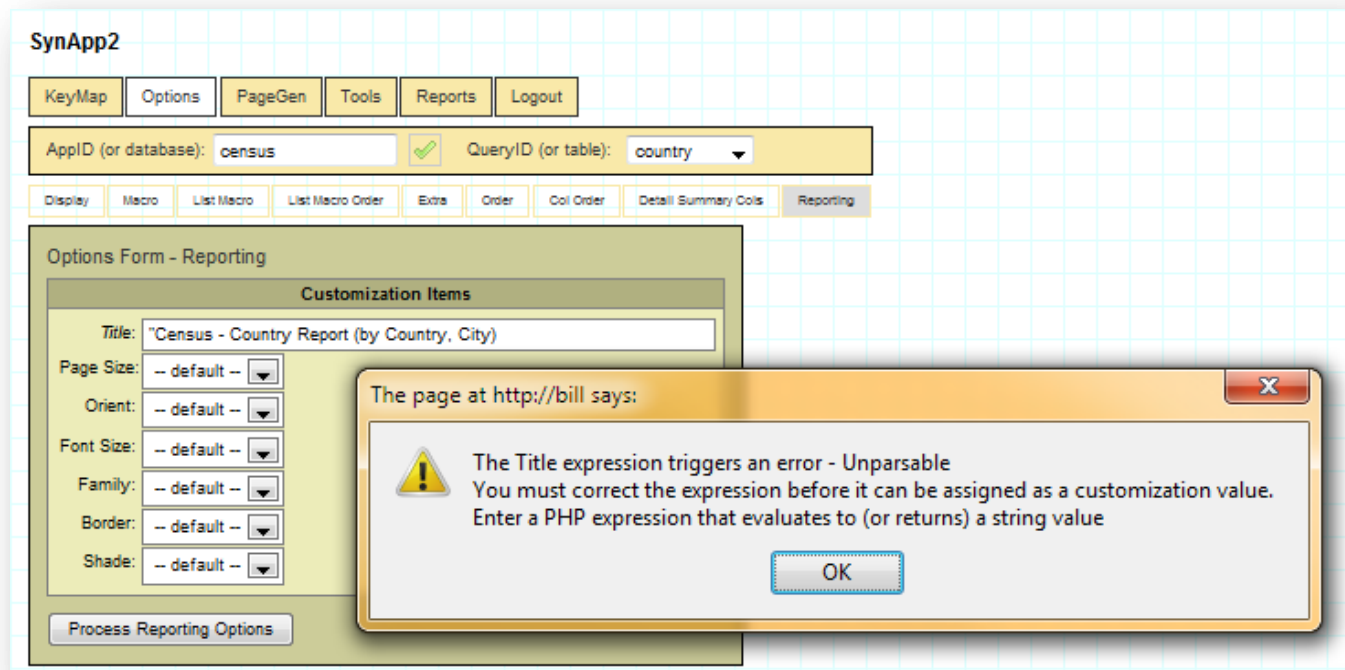
The customization values for the attributes on these pages are stored to the `synapp2.inc.php` file rather than the `custom.inc.php` file. In the discussions of customization elements, that that appear later in this document, you will see notations for attribute values that are managed from these pages. There is no need to create or edit them manually. *SynApp2* will take care of the details.

If any interactively definable element happens to appear in the `custom.inc.php` file, the corresponding GUI element will be disabled (grayed). To enable interactive manipulation, move the statement from `custom.inc.php` to `synapp2.inc.php`. Be sure that the statement appears inside the existing output delimiter tags: `<!--{inc}-->` `<!--{/inc}-->`, and on its own (single) line.

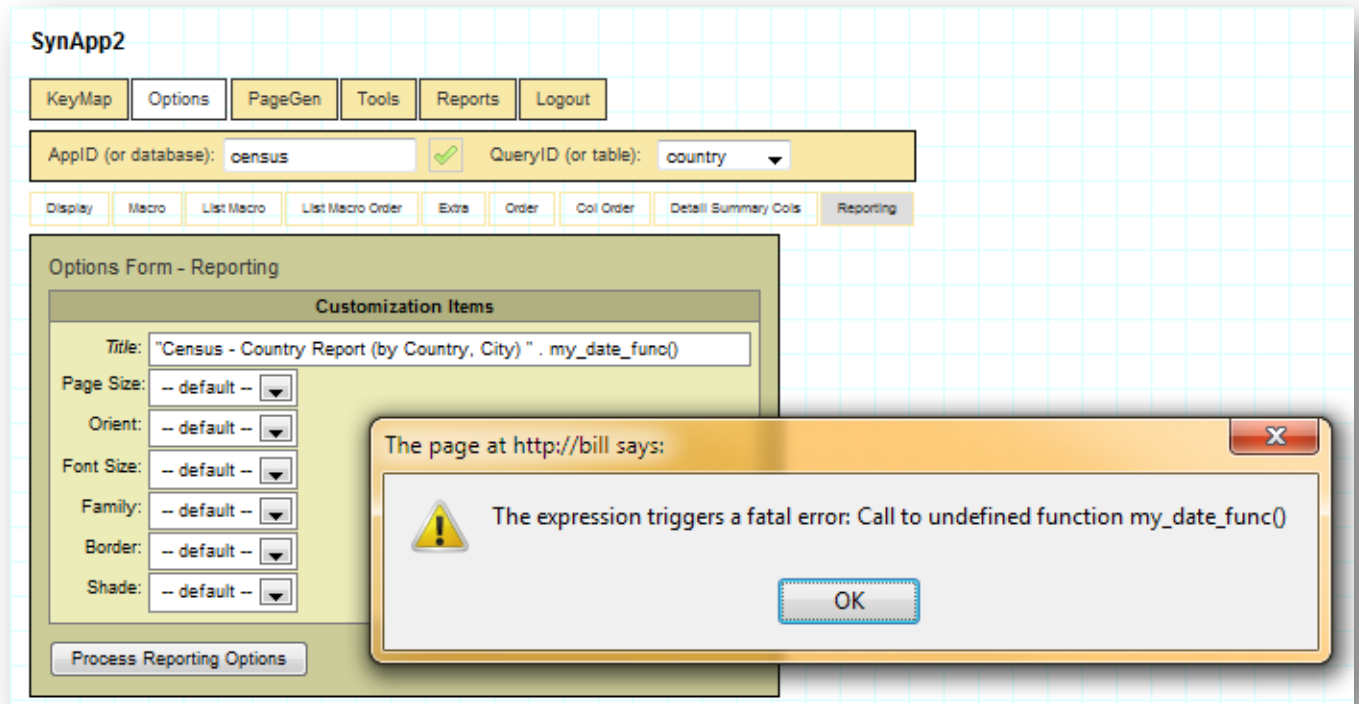
Display customizations that involve a Report Column, or Reporting Customization Items such as Title, Page Size, etc., are effective immediately. Customizations that affect query result rows, such as Order (i.e. `ORDER BY`) or produce values, such as Detail Summary Cols, are also effective immediately. All other changes – *that affect page layout* - will require that pages be regenerated. Use the *PageGen – Regenerate All* feature to quickly incorporate display options, for the various forms, into your application.

Many of the interactively customized elements are expressed as PHP code. It is essential that the code be, at the very least, syntactically correct, i.e., parsable.

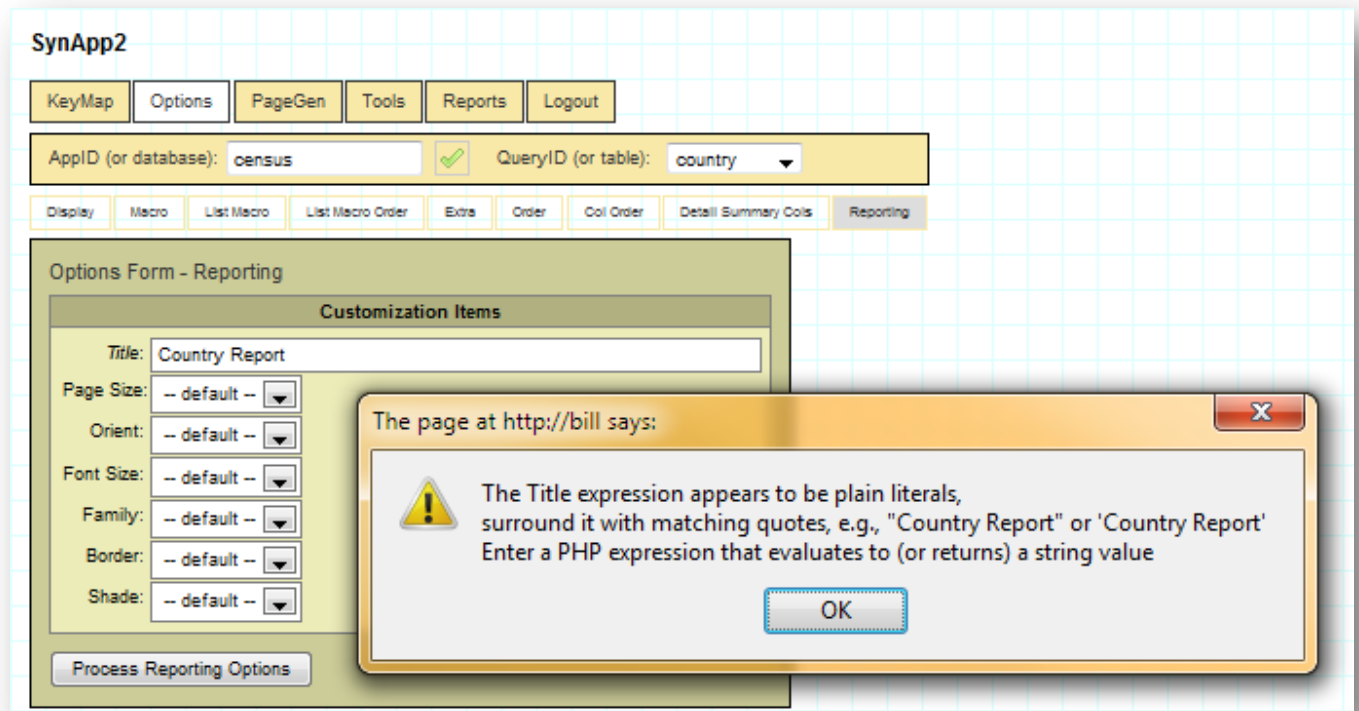
If *SynApp2* detects a problem, you will see an error dialog, similar to the following:



Reference to an undefined function produces a message similar to:



Literal values must be enclosed in quotes, or you'll see a message like:



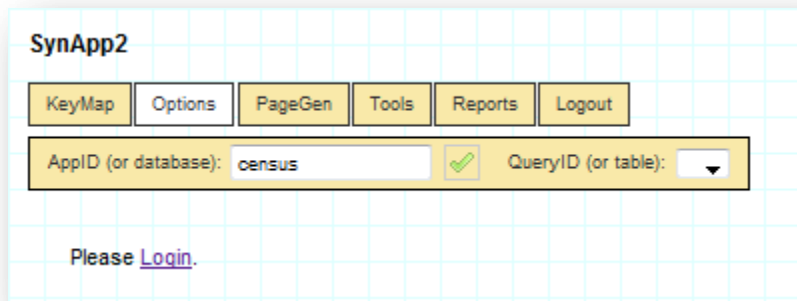
If by some means, you *do manage* to get invalid PHP code written to either `custom.inc.php` or `synapp2.inc.php` you may see – and only if browser popup windows are enabled – the *SynApp2* popup Message Window.

An error message display:

```
** XML Parsing Error **
[check SynApp2 response: must be well-formed and valid]
[check PHP configuration: display_errors={false,'0'}]
<br />
<b>Parse error</b>:  syntax error, unexpected $end, expecting T_VARIABLE or T_DOI
\synapp2.inc.php</b> on line <b>70</b><br />
```

And, anytime the XML response cannot be parsed, your login and authorization cannot be determined. You will see the login prompt, as this the default behavior for a page. But, if the message appears while you're editing customization files, it doesn't mean that you should login again.

The Please Login prompt:

The screenshot shows the SynApp2 web interface. At the top, there's a navigation bar with buttons for 'KeyMap', 'Options', 'PageGen', 'Tools', 'Reports', and 'Logout'. Below this is a form with two input fields: 'AppID (or database):' containing the text 'census' and a green checkmark icon, and 'QueryID (or table):' with a dropdown arrow. At the bottom of the form, it says 'Please Login.' The entire interface is set against a light blue grid background.

If you see the *SynApp2* popup Message Window displayed with an XML Parsing Error, and/or you see **Please Login** appear – after changing a customization element or expression – then there is almost certainly a problem with either or both of the `inc.php` files that must be corrected by manually editing.

In that case, open the file(s), find and correct the problem(s), and save the file(s) back to the server. Then, refresh/reload the page in your browser to resume your work.

Once the problem has been corrected and the page refreshed, the login prompt should go away – without having to login again.

## Custom Initialization Functions

When the built-in column value initialization functions aren't enough, you can make your own.

A sample function to generate a next order number appears below.

```
function order_number()
{
    $dbx = get_dbx();

    // TODO: reserve/lock pending order number
    return 1 + $dbx->get_query_value('select max(order_no) from orders');
}
```

Your function returns the column initialization value as a number or string.

Use the `COMMENT` mechanism, described earlier, to integrate your function:

```
Order Number,init=order_number()
```

You may define your custom function(s) directly in the local `custom.inc.php` file, but also consider using a PHP `include` or `require` statement to incorporate a separate file.

## Custom Validation Functions

All posted column variables are automatically validated against intrinsic rules for their data types (e.g. check for valid INT, DECIMAL, UNSIGNED, DATE, TIME, NOT NULL, etc.) and according to any optionally specified min/max limits. These tests are considered to be **primary** validation because all tests are carried out independently with regard to the validity or values of any other concurrently posted column variables.

More sophisticated primary validation tests can be defined and carried out for column variables representing such things as email addresses or telephone numbers.

When you need to validate a column variable value against the value or values of one or more concurrently posted column variables, **secondary** validation is appropriate.

The invocation of secondary validation functions are deferred until after primary validation of all concurrently posted column variables has been completed. Secondary validation functions are called only when the named column variable has satisfied primary validation. The validation state of all concurrently posted column variables are passed to a secondary validation function via the feedback parameter.

A sample customization entry and primary validation function appear below:

```
$this->m_data[APPID]['yourapp'][QID]['customer'][VALIDATOR]
[VALIDATOR_PRIMARY]['email'] = 'customer_email';

function customer_email($col_name, $col_value, $display_name, &$col_vars)
{
    $msg = '';

    if (!is_email($col_value))
```

```

{
    $msg = "Please enter a valid email address in the {$display_name} field.";
}

return $msg;
}

```

A sample customization entry and secondary validation function appear below:

```

$this->m_data[APPID]['yourapp'][QID]['schedule'][VALIDATOR]
[VALIDATOR_SECONDARY]['sched_end'] = 'sched_end';

function sched_end($col_name, $col_value, $display_name, &$col_vars, &$feedback)
{
    $msg = '';

    if (empty($feedback['sched_beg']) && $col_value <= $col_vars['sched_beg']['v'])
    {
        $msg = "{$display_name} must come after {$col_vars['sched_beg']['v']}";
    }

    return $msg;
}

```

The return value of a validation function is a string. An empty string indicates successful validation. Indicate validation failure with the exact message text you want to appear in the **IFORM** (Input Form) feedback label element bound to subject column variable.

Notice the check for the empty `$feedback` array element. This test establishes that the concurrently posted variable that the validity test of `$col_value` depends on, has passed *primary* validation.

It is not a safe practice to assess the *secondary* validation status of column variables based upon the `$feedback` array, because the order of individual variable validation is not guaranteed. In other words, any concurrently posted column variable, upon which a secondary validation of another column variable depends, should not also be subject to secondary validation.

You are not required to test or compare column variable values against one another in a validation function called within the secondary context. The implementation may be similar or identical to that of function designed for the primary context.

You may define your custom function(s) directly in the local `custom.inc.php` file, but also consider using a PHP `include` or `require` statement to incorporate a separate file.

## Custom Processing Functions

The power to do just about anything is available through the framework of the custom processing mechanism. You can accomplish extensive data manipulation and generate markup for the browser.

The typical pattern for employing custom processing starts with web page and JavaScript code that initiates a *SynApp2* request/response cycle using `MODE_ADHOC`. Use of this mode is not requirement. You may define custom processing functions that respond to any query ID (`QID`). If your function returns `true`, the current request/response cycle continues normally. A `false` return value circumvents additional action. The actual effect on program flow depends on the point, or *context*, the function invocation.

*As of SynApp2 version 0.1.7, there is only one point where a custom processing function can have an effect on program flow. Review `_shared_/action.php` and the code involving the class method `action::do_process()`.*

A suitable web page may be created by copying and then modifying an existing *SynApp2* generated page from the application directory where you are working and including the new page in the application navigation tabs. The processing web page is then directly customized with static markup including controls to provide parameters and to invoke processing, or markup for UI can be generated dynamically, or both.

A sample web page to support a custom processing function:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>
Temp Conversion
</title>
<link rel="stylesheet" type="text/css" href="../../_shared_/standard.css">
<script type="text/javascript" src="../../_shared_/synapp2.js"></script>

<script type="text/javascript">

set_appid('census');
set_pid('temp_conv');

function page_init()
{
    do_init();
    do_app_nav("id_app_nav");
    map_vkey_action('id_iform_temp', VKEY_ENTER, request_wrapper, true);
}

function request_wrapper()
{
    var cxlref = new cxl();

    if (cxlref)
    {
        cxlref.set_container("id_result");
        cxlref.set_filter("id_result", "id_iform_temp");
        cxlref.set_request_mode_adhoc();
    }
}
```

```

        cxlref.send_request("convert");
    }
}

</script>
</head>
<body id="id_body" onload="page_init();">

<div id="id_app_nav" class="class_app_nav"></div>

<div id="id_page_content" class="class_page_content">

<div class="class_layout_group_std">
    <h3>Temperature Conversion</h3><br>
    <div id="id_iform_temp" class="class_form_std">
        <label>Temperature F:</label>
        <input type="text" name="_process_temp_F"><br>
        <input value="Ok" type="button" onclick="request_wrapper();"><br>
        <div id="id_result"></div>
    </div>
</div>

<div id="id_page_msg" class="class_msg_std"></div>

</div>
</body>
</html>

```

Notice that the name attribute of the temperature input element has the magic "\_process\_" prefix (e.g. "\_process\_temp\_F").

Sample PHP code to integrate and implement a custom processing function:

```

<?php

$this->m_data[APPID]['census'][INCL][NAV] = 'temp_conv';
$this->m_data[APPID]['census'][QID]['convert'][PROCESS][PROCESS_MAIN] = 'F_to_C';

function F_to_C(&$args, &$action, &$adhoc_markup)
{
    $temp_F = sprintf("%.2f", $args['temp_F']);
    $temp_C = sprintf("%.2f", (5 / 9) * ($temp_F - 32));

    $adhoc_markup[] = "<h1>{$temp_F}F = {$temp_C}C</h1>";

    return true; // continue action
}

?>

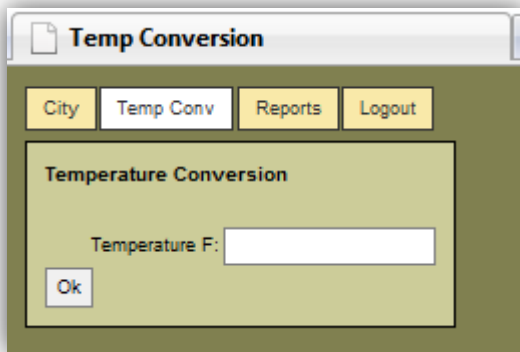
```

The temperature value passed to the function is retrieved from the \$args[ ] array using the input element name without the magic "\_process\_" prefix (e.g. 'temp\_F').

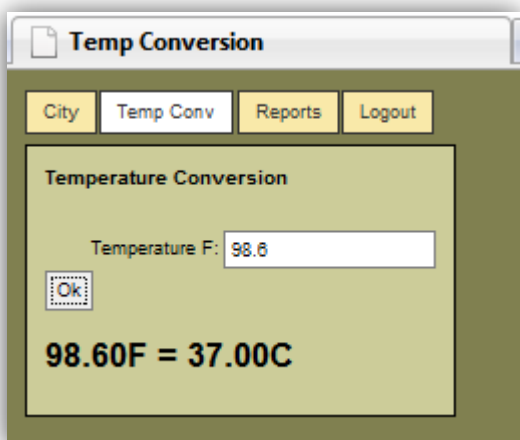
You may define your custom function(s) directly in the local custom.inc.php file, but also consider using a PHP include or require statement to incorporate a separate file.

You can try out the sample custom processing code by adding it to the *Census* application created by following the steps for *Part A* in the [SynApp2 Walk through No. 1](#) document.

1. Complete *Part A* of the walk through, at least as far as generating the `city.htm` page, as seen on page 9 of the walk through.
2. Create a new `temp_conv.htm` web page file in the `census` application directory. See page 9 of the walk through if you need help finding the `census` application files.
3. Copy and paste the sample web page code into your plain text editor and save as `temp_conv.htm`.
4. Open the `custom.inc.php` file for the `census` application with your plain text editor.
5. Copy, paste and save the sample PHP code – between but not including the opening `<?php` and closing `?>` lines - into that file, between the existing `<?php ?>` tags. Use care to make sure that the result is a well-formed PHP file.
6. Open the `temp_conv.htm` page in your browser: *(You may need to login.)*



7. Enter a temperature value and click Ok:



With a bit of imagination, you'll find that you can create custom processing functions that are as powerful as you need them to be, and integrate them with your applications in any way that makes sense.

## Custom Authorization and Record Filtering Functions

The power to control access to areas of application functionality and data, based on situational context, is provided through simple, yet effective, mechanisms. Common scenarios that occur in everyday business can be modeled and used to gate access to key areas – including: specific applications, pages, reports, and processing. Further control over interactive data retrieval and manipulation features is also available and easy to establish.

In the absence of custom authorization details, everything is accessible to logged in users. But, with a few line lines of code, you can tailor the exact level of feature and data availability any given user will see.

You may define your custom authorization entries and function(s) directly in the local `custom.inc.php` file, but also consider using a PHP `include` or `require` statement to incorporate a separate file.

Helper functions are defined and used to return important values for two key reasons – encapsulation and visibility. Functions are globally visible and are therefore may be referenced by expressions entered from the *SynApp2* Options pages.

Sample authorization entries and filtering functions:

```
$this->m_data[APPID]['timelog'][AUTH_PID]['resource'] = admin_user();
$this->m_data[APPID]['timelog'][AUTH RID]['resource'] = admin_user();

$this->m_data[APPID]['timelog'][QID]['resource'][AUTH_ADD] = '';
$this->m_data[APPID]['timelog'][QID]['resource'][AUTH_EDIT] = admin_user();
$this->m_data[APPID]['timelog'][QID]['resource'][AUTH_DELETE] = admin_user();

$this->m_data[APPID]['timelog'][QID]['activity'][AUTH_ADD] = admin_user();
$this->m_data[APPID]['timelog'][QID]['activity'][AUTH_EDIT] = admin_user();
$this->m_data[APPID]['timelog'][QID]['activity'][AUTH_DELETE] = admin_user();
```

```
function admin_user()
{
    return 'timelog_administrator';
}

$this->m_data[APPID]['timelog'][QID]['resource'][AUTH_RECS] = 'auth_recs_resource';

function auth_recs_resource()
{
    $auth_recs_expr = '';

    $username = login_username();

    if ($username != admin_user())
    {
        $auth_recs_expr = "username = '$username'";
    }

    return $auth_recs_expr;
}
```

```

$this->m_data[APPID]['timelog'][QID]['timelog'][AUTH_RECS] =
'auth_recs_timelog';

$this->m_data[APPID]['timelog'][QID]['timelog_(by_resource)'][AUTH_RECS] =
'auth_recs_timelog';

$this->m_data[APPID]['timelog'][QID]['timelog_(by_activity)'][AUTH_RECS] =
'auth_recs_timelog';

function auth_recs_timelog()
{
    $auth_recs_expr = '';

    $username = login_username();

    if ($username != admin_user())
    {
        $id_resource = get_dbx()->get_query_value("select id from resource
                                                    where username = '$username'");

        $auth_recs_expr = "id_resource = '$id_resource'";
    }

    return $auth_recs_expr;
}

```

An AUTH\_RECS function should return an SQL term that, if not empty, will be incorporated into the WHERE clause of complete SQL statements, generated by the *SynApp2* framework, when the QID(s) you specify with customization entries are active.

To prevent selection of records, *you must return a valid expression*. In a case where, for example, the logged in user should not be authorized to see any records, then return an expression that cannot be satisfied, e.g. `id_resource = ''`, when incorporated into the WHERE clause of the controlling SQL statement.

Your AUTH\_RECS function(s) can access context information in many ways. The `login_username()` function, seen above, is only one example. The globally visible instance of the `action` object, established to handle every exchange cycle, has methods available to monitor many situational details, including primary and foreign key values of selected records.

Methods like `action::get_pk()`, `action::get_pk_values()`, and `action::get_fk_constraint()` return details about the records currently selected along the *nodes of interaction* of the active *page flow*. Other methods can provide details about the exchange action, mode, and submitted column (i.e. form) variables. Add the statement – `global $g_action;` – to the body of any customization function that needs to reference the `action` object.

Any combination of information you can gather from the framework objects and queries of the data records, may be used to establish context and provide any details needed to produce the precise SQL terms returned by your functions. Related examples and design patterns are provided elsewhere. This text is but an introduction and a tickler to get you thinking about the possibilities.

During development, consider instrumenting your functions with calls to the `add_debug_msg()` function (`_shared_/util.php`) to emit debugging information. Use the function to monitor the state of variables via the *SynApp2 Message Window*. Use it much as you would the PHP variable handling functions `var_dump()` or `print_r()`, but with the results formatted to work within the AJAX exchange cycle of the *SynApp2* framework.

The customization entries and functions, appearing above, can be exercised with the following *MySQL* table definitions:

```
-- example database: timelog --

CREATE TABLE IF NOT EXISTS `activity` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `descr` varchar(40) NOT NULL COMMENT 'Description',
  `code` char(16) DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `descr` (`descr`)
);

CREATE TABLE IF NOT EXISTS `resource` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(24) NOT NULL,
  `username` varchar(48) DEFAULT NULL,
  `password` varchar(48) DEFAULT NULL,
  `password_action` enum('no_change','encrypt') DEFAULT 'no_change',
  PRIMARY KEY (`id`),
  UNIQUE KEY `name` (`name`),
  UNIQUE KEY `username` (`username`)
);

CREATE TABLE IF NOT EXISTS `timelog` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `id_resource` int(11) NOT NULL,
  `id_activity` int(11) NOT NULL,
  `log_date` date NOT NULL COMMENT ',init=date_today()',
  `log_hours` decimal(6,1) NOT NULL DEFAULT '8.0' COMMENT ',min=0.1,max=24.0',
  PRIMARY KEY (`id`)
);
```

## Custom Application Username and Password Validation Functions

The following discussion and example applies only to *SynApp2* installations configured to work with *MySQL*. Work has already begun to support alternative access models for database engines such as *Oracle*. A *sandbox* model will make similar user management features available for one or more applications running from a single *Oracle* schema.

Building on the earlier discussion about authorization and record filtering, we can easily add a few more customization entries and functions to the `custom.inc.php` file (for the *timelog* example), that will support user management from within the application. The techniques can be adapted for any application and specific database design.

Sample authorization entries and validation functions:

```
$this->m_data[APPID]['timelog'][QID]['users'][TABLE] = "resource";

$this->m_data[APPID]['timelog'][AUTH_PID]['users'] = admin_user();
$this->m_data[APPID]['timelog'][AUTH_RID]['users'] = admin_user();

$this->m_data[APPID]['timelog'][QID]['users'][AUTH_ADD] = admin_user();
$this->m_data[APPID]['timelog'][QID]['users'][AUTH_EDIT] = admin_user();
$this->m_data[APPID]['timelog'][QID]['users'][AUTH_DELETE] = admin_user();
```

```
$this->m_data[APPID]['timelog'][QID]['users'][VALIDATOR][VALIDATOR_PRIMARY]
['username'] = 'validate_username_timelog';

function validate_username_timelog($col_name, $col_value,
                                   $display_name, &$col_vars)
{
    global $g_action;

    $msg = '';

    if (!empty($col_value))
    {
        if (preg_match('/[^\a-z0-9_\.\-\@]+\i', $col_value) ||
            $col_value != escape_sql_term($col_value))
        {
            $msg = "{$display_name} contains disallowed characters";
        }
        else
        {
            $update_filter = $g_action->is_update() ?
                (" and id != '" . $g_action->get_pk() . "'" ) :
                '';

            if (get_dbx()->get_query_value("select count(*) from resource where
                                           { $col_name } =
                                           '{ $col_value }'{$update_filter}"))
            {
                $msg = "{$display_name} already exists";
            }
        }
    }

    return $msg;
}
```

```

$this->m_data[APPID]['timelog'][QID]['users'][VALIDATOR][VALIDATOR_SECONDARY]
['password'] = 'validate_password_timelog';

function validate_password_timelog($col_name, $col_value,
                                   $display_name, &$col_vars)
{
    global $g_action;

    $msg = '';

    if (!empty($col_value) &&
        ($password_action = &$col_vars['password_action']['v']))
    {
        if ($col_value != escape_sql_term($col_value))
        {
            $msg = "{$display_name} contains disallowed characters";
        }
        else
        {
            if (preg_match('/encrypt/i', $password_action))
            {
                $password_function = 'password';

                $password_encrypted = get_dbx()->get_query_value(
                    "select {$password_function}('{ $col_value}')");

                if (!empty($password_encrypted))
                {
                    $col_vars[$col_name]['v'] = $password_encrypted;
                    $password_action = 'no_change'; // MAGIC:
                }
                else
                {
                    $msg = "Unable to process {$display_name} value:
                        {$password_function}('{ $col_value}')";
                }
            }
        }
    }

    return $msg;
}

```

In order to employ the application login model, you would need an entry in `_config_/access.inc.php`. to map the logical name of the database to the database and table where the usernames and passwords are stored.

It doesn't technically matter where the entry appears, but it naturally goes with and below the section marked with the comments: `// see access_app::auth_connect()`

Your `access.inc.php` file would need to incorporate the following line:

```

$this->m_config[MAP_AUTH_INTERFACE_DEF]['timelog'] = new auth_interface_def('timelog', 'resource');

```

The entry specifies that user login and authentication for the `timelog` database/application will occur against the `timelog.resource` table. And, in the absence of explicit definitions, the default username and password field names and encryption function are used – e.g. `resource.username`, `resource.password` and the MySQL function `password()`.

Realistically, additional customization elements are needed to make the GUI for the *timelog* example look and operate as a finished application. You may download the complete source code for the application from: <http://www.synapp2.org/documentation/timelog.MySQL.zip>

There are some login/authentication behaviors that are worth noting:

It matters where you login.

- If you successfully login via the welcome page of a *SynApp2* generated application, that will provide authorization to access to that application, but not other generated applications, or to the *SynApp2 Web Application Generator* application.
- If you successfully login via the *SynApp2* welcome page (`synapp2.htm`), you will automatically be promoted to have access to all *SynApp2* generated applications unless there is/are specific `AUTH_` customizations to prevent that.

In order to move back and forth (using the convenient breadcrumb links) between the *SynApp2 Web Application Generator* application and a generated application, while developing and testing with the `access_app` model and username-based `AUTH_RECS` customization, it will be important to have matching login usernames in both the `synapp2.users` table and the configured application authentication table – that is – if you don't want to logout and login in again, each time you want to switch your application focus. It is, however, perfectly reasonable to do that.

You could also reference the `synapp2.users` table from your username-based `AUTH_RECS` customization, but that would frustrate simplicity and portability, especially when deploying to a shared server, due to logical vs. physical database names. It's doable, but you'd have to be careful to make sure it all worked as you would expect, and without having to modify configurations or deviate from *SynApp2* conventions.

Two matching usernames means two passwords, but they don't have to be the same. This would allow you to set up an administrative username for the generated application and a matching username in `synapp2.users`. Different passwords would guard against a person logging in as the administrative user of the generated application from having access to the *SynApp2 Web Application Generator* application.

Another alternative, to setting up matching usernames, would be to open multiple browser instances – each with a separate `session_id` – logged in to the appropriate application. You could then switch back and forth between them without repeated logins. But, if you regenerate a page that is currently visible in another window, you may have to refresh your browser display to see changes.

## Customizing Data Views

Your *SynApp2* applications can be quickly empowered with alternate perspectives of the data for interactive manipulation, processing, reporting and export. Use the *SynApp2* Tools – QueryID page to map named views to a basis table. Each secondary QueryID (i.e. QID) can be used to generate an interactive page and/or report.

The data visualization virtues of spreadsheet [pivot tables](#) become available in nice, neat PDF reports. The *timelog* example uses multiple QueryID's to report *timelog* by resource or by activity. The reports process the same data, but sort and group the rows differently. This kind of power makes it easy to visualize the answers to different business questions such as: What activates was each person (resource) working on over a given period, or, How much time (and money) is being spent on various activities over a given period. The answers come from exactly the same data, presented in a way that makes it easy to answer useful questions.

The *SynApp2* Tools – QueryID page used to create view mappings:

The screenshot shows the SynApp2 web interface. At the top, there's a navigation bar with buttons: KeyMap, Options, PageGen, Tools, Reports, and Logout. Below this is a yellow bar with a text input for 'AppID (or database):' containing the value 'timelog' and a green checkmark icon. Underneath are three tabs: Users, QueryID (selected), and Structure. The main content area is titled 'Tools Form - Secondary QueryID (table) Mapping'. It contains a table with two columns: 'Basis QueryID (or table)' and 'Secondary QueryID (table)'. The 'Basis QueryID' column has a list box with items: 'activity (activity)', 'cursors (cursors)', 'resource (resource) [1]', and 'timelog (timelog) [2]'. The 'Secondary QueryID' column has a text input with a '>>' button and a 'Delete' button. The text input contains 'timelog\_(by\_resource) (timelog)' and 'timelog\_(by\_activity) (timelog)'. The 'QID TABLE' label is on the left side of the table.

The mappings cause entries to be generated into the `synapp2.inc.php` file of the active AppID (or database).

```
$this->m_data[APPID]['timelog'][QID]['users'][TABLE] = "resource";  
$this->m_data[APPID]['timelog'][QID]['timelog_(by_resource)'][TABLE] = 'timelog';  
$this->m_data[APPID]['timelog'][QID]['timelog_(by_activity)'][TABLE] = 'timelog';
```

The literal values of the Secondary QueryID will automatically appear in the application and report navigation tabs and as filenames, if PageGen is used to generate corresponding pages or reports.

As soon as you've mapped a new QueryID, you can use the Options pages to interactively customize Display attributes, column expressions and reporting details exactly as you would for any primary (i.e. *natural*) QueryID (or table).

A SynApp2 generated Report Form for a secondary QueryID – Timelog (by Resource):

Timelog Users Reports Logout

Activity Resource Timelog (by Resource) Timelog (by Activity) Users

Report Form - Timelog (by Resource)

Resource:

Activity:

Log Date >= :

<= :

Download Export: No

Timelog (by Resource)

Fill in values and then click Report

A neatly formatted report to answer: “What activities are people working on?”

Timelog Users Reports Logout

Activity Resource Timelog (by Resource) Timelog (by Activity) Users

1 / 1 100% Find

Timelog Report (by Resource, Log Date, Activity) - 10/18/2010

| Resource | Log Date   | Activity                              | Log Hours |
|----------|------------|---------------------------------------|-----------|
| Mary Ann | 2010-09-07 | Supervision - Document Translation    | 1.2       |
|          |            |                                       | 1.2       |
| Richard  | 2010-09-06 | Engineering - Application Development | 4.0       |
|          | 2010-09-06 | Time Off - Holiday                    | 4.0       |
|          | 2010-09-07 | Engineering - Application Development | 8.0       |
|          |            |                                       | 16.0      |
|          |            |                                       | 17.2      |

A SynApp2 generated Report Form for a secondary QueryID – Timelog (by Activity):

Report Form - Timelog (by Activity)

Resource:

Activity:

Code:

Log Date >= :

<= :

Download Export: No

**Timelog (by Activity)**

Fill in values and then click Report

A neatly formatted report detailing: “How time is being spent on each activity?”

Timelog Report (by Activity, Log Date, Resource) - 10/18/2010

| Code   | Activity                              | Log Date   | Resource | Log Hours |
|--------|---------------------------------------|------------|----------|-----------|
| 961200 | Engineering - Application Development | 2010-09-06 | Richard  | 4.0       |
|        |                                       | 2010-09-07 | Richard  | 8.0       |
|        |                                       |            |          | 12.0      |
| 961101 | Supervision - Document Translation    | 2010-09-07 | Mary Ann | 1.2       |
|        |                                       |            |          | 1.2       |
| 950002 | Time Off - Holiday                    | 2010-09-06 | Richard  | 4.0       |
|        |                                       |            |          | 4.0       |
|        |                                       |            |          | 17.2      |

These are simple, yet practical, examples of how multiple data views can be used to answer different questions.

There's lots of power here. By mapping secondary QueryID's and customizing ORDER, EXTRA, COL\_ORDER and DETAIL\_SUMMARY\_COL expressions for generated reports, you can provide valuable data analysis tools.

This is **THE BIG TAKEAWAY**: Make sure you understand what multiple data views – that are quick and easy to create with SynApp2 - can mean for you and your clients.

## Drag and Drop Functions

Event driven drag and drop can be added to your *SynApp2* applications. The mechanism is capable of managing and reflecting dynamic constraints with regard to what item or items can be dragged and where or upon which target elements the drag items may be dropped.

The behavioral implementation is completely up to you. It can be as simple as repositioning an element, to affecting records in the application database tables.

```
function on_dragBegin(drag_item, dx, dy) {} // OPTIONAL: implement this method for
drag/drop

function on_target_dragBegin(drag_target) {} // OPTIONAL: implement this method for
drag/drop (eg. element.style.backgroundColor = 'white';)

function on_target_dragEnd(drag_target) {} // OPTIONAL: implement this method for
drag/drop (eg. element.style.backgroundColor = 'transparent';)

function on_drag(drag_item, dx, dy) {} // OPTIONAL: implement this method for
drag/drop

function on_dragEnd(drag_item, drag_target, dx, dy) {} // IMPORTANT: implement this
method for drag/drop
```

*At a minimum, you should implement `on_dragEnd()`.* The implementation of this function could trigger a custom process request/response cycle, or be handled entirely within the browser local scripting, or some combination of the two. Enter or include your event handler code in your web page file. Use appropriate `<script>` elements.

The `drag_item` and `drag_target` parameters are references to *HTML DOM objects*. The delta x and y position change values, in pixels, at the time the event was triggered, are passed as `dx` and `dy`.

A *SynApp2* JavaScript helper function can be useful in determining the *effective* position of HTML DOM objects:

```
var pos = get_pos(<HTML DOM object reference>);

var x = pos.left;
var y = pos.top;
```

The position data returned by `get_pos()` is the absolute position offset from the upper left of the browser *viewport*. The topic of browser window geometry is beyond the scope of this document. A comprehensive reference is *JavaScript: The Definitive Guide, Fifth Edition* by David Flanagan.

Drag item elements are established by supplying a CSV *string* or simple *array* object to the function `set_drag_items(idds)`. The values passed must be the `id` attribute values of any HTML DOM objects that are allowed to be dragged. This can be accomplished by either a direct call from the page script, or by a request/response cycle triggered by a page `onload` event (typically from `page_init()`), user interaction, or some other source.

You may optionally establish drag target elements using the same techniques as for drag items. A request/response cycle triggered by a *dragBegin* event can produce the set of drop targets that are valid for the item being dragged.

Sample JavaScript to support dynamic drag and drop:

```
QID_SCHEDULE = 'schedule';

function rmi_schedule()
{
    var cxlref = new cxl();

    if (cxlref)
    {
        cxlref.set_container('id_schedule_result');
        cxlref.set_filter('id_schedule_result', 'id__iform__schedule');
        cxlref.set_request_mode_adhoc();
        cxlref.send_request(QID_SCHEDULE);
    }
}
```

```
function on_dragBegin(drag_item, dx, dy)
{
    set_extra_arg(QID_SCHEDULE, '_process_schedule_get_tid', drag_item.id);

    rmi_schedule();

    set_extra_arg(QID_SCHEDULE, '_process_schedule_get_tid', '');
}

function on_drag(drag_item, dx, dy)
{
    drag_item.style.color = 'blue';
}

function on_target_dragBegin(drag_target)
{
    drag_target.style.backgroundColor = 'white';
}

function on_target_dragEnd(drag_target)
{
    drag_target.style.backgroundColor = 'transparent';
}

function on_dragEnd(drag_item, drag_target, dx, dy)
{
    var target_id = drag_target ? drag_target.id : '';

    set_extra_arg(QID_SCHEDULE, '_process_schedule_item_id', drag_item.id);
    set_extra_arg(QID_SCHEDULE, '_process_schedule_target_id', target_id);

    rmi_schedule();

    set_extra_arg(QID_SCHEDULE, '_process_schedule_item_id', '');
    set_extra_arg(QID_SCHEDULE, '_process_schedule_target_id', '');
}
```

*Notice the SynApp2 function `set_extra_arg(qid, name, value)`. The name-value pairs persist as long as the containing page is displayed. They will be submitted with any subsequent requests generated having the same `qid` value. It is good practice to clear the extra arguments as soon as a request is triggered. Notice also, the magic `'_process_'` prefix.*

Whenever a SynApp2 request/response cycle is triggered, drag item or drag target id values, corresponding to HTML DOM objects of the requesting page, may be returned via data members of the markup class. See `_shared/markup.php`.

The class data members, `markup::m_IDD` and `markup::m_TDD`, are PHP array objects that serve as output buffers used to compose a portion of the XML response. The `m_IDD` member is for drag item id values and `m_TDD` is for drag target id values.

*As of version 0.1.7 alpha, a formal interface to the markup class data members and the class instance that contains them has not been established. Use the PHP `global $g_markup;` statement to access the markup class instance supporting the request/response cycle.*

A sample custom processing function with PHP code to generate **drag target** id values:

```
$this->m_data[APPID][<appid>][QID]['schedule'][PROCESS][PROCESS_MAIN] = 'schedule';

function schedule(&$argv, &$action, &$adhoc_markup)
{
    if (!empty($argv['schedule_get_tid']))
    {
        $drag_item_id = $argv['schedule_get_tid'];

        $csv_target_ids = get_csv_target_ids($drag_item_id); // NOTE: your function

        if (!empty($csv_target_ids))
        {
            global $g_markup;
            $g_markup->m_TDD[] = $csv_target_ids; // NOTE: empty square braces
        }
    }
}
```

When drag target id values are returned in an exchange response, the function `on_target_dragBegin(drag_target)`, *if implemented*, will be called, in turn, for each target element. Use these notifications to update the UI to reflect valid drop targets for the item being dragged. Similar calls to `on_target_dragEnd(drag_target)` are made when a drag item is dropped. You are not required to implement either of these functions.

A custom processing function can generate **drag item** id values. The significant difference from the example above is that the `markup::m_IDD` member is used.

```
global $g_markup;
$g_markup->m_IDD[] = $csv_item_ids; // NOTE: empty square braces
```

It is also possible *and reasonable* to return (X)HTML markup along with drag item or drag target id values. Update the `$adhoc_markup[]` array object with well-formed (X)HTML code. The result will replace the *innerHTML* of the SynApp2 request container element.

## Identifiers

There are various defined values - *identifiers* - that serve as keys to the customization data structure.

The identifiers are:

- APPID
- QID
- AUTH\_APPID
- AUTH\_QID
- AUTH\_PID
- AUTH\_RID
- AUTH\_RECS
- AUTH\_ADD
- AUTH\_EDIT
- AUTH\_DELETE
- AUTH\_PROCESS
- DATABASE
- TABLE
- KEYMAP
- ORDER
- MACRO
- LIST\_MACRO
- LIST\_MACRO\_ORDER
- LIST\_MACRO\_FILTER
- EXTRA
- OMIT
- INCL
- TAB\_ORDER
- QUERY
- ~~SUBQUERY~~ *DEPRECATED - use EXTRA*
- FETCH
- TITLE
- ~~LEGEND~~
- LIMIT\_ROWS
- COL\_OMIT
- COL\_SIZE
- COL\_ALIGN
- COL\_FORMAT
- COL\_EDITOR
- COL\_EDITOR\_ROWS
- ~~COL\_ATTRS~~
- COL\_ORDER

## Identifiers - *continued*

- RPT\_PAGE\_SIZE
- RPT\_PAGE\_ORIENTATION
- RPT\_FONT\_SIZE
- RPT\_FONT\_FAMILY
- RPT\_CELL\_BORDER
- RPT\_ROW\_SHADE
- RPT\_ROW\_SIZE
- DETAIL\_SUMMARY\_COLS
- EXPORT\_TYPE
- VALIDATOR
- PROCESS
- MEDIA\_TYPE
- TEMP\_LOC
- MEDIA\_LOC

## Sub-Identifiers

There are definitions for several *sub-identifiers* that are used to name elements for customization values that are conceptually regarded as objects.

The sub-identifiers are:

- IFORM
- DFORM
- SFORM
- TFORM
- AFORM
- FFORM
- RFORM
- NAV
- RPT
- A\_ID
- A\_HREF
- A\_TEXT
- TABLE\_NAME
- COL\_NAME
- JOIN\_TABLE
- JOIN\_COL
- LOC\_ANY
- LOC\_TEXT
- LOC\_IMAGE
- LOC\_VIDEO

## Attribute Value Identifiers

There are definitions for several identifiers that are used to enumerate values for customization of specific attributes.

The attribute value identifiers for various on/off customization elements are:

- *SWITCH\_ON*
- *SWITCH\_OFF*

The attribute value identifiers for *COL\_ALIGN* are:

- *ALIGN\_L*
- *ALIGN\_J*
- *ALIGN\_C*
- *ALIGN\_R*

The attribute value identifiers for *COL\_EDITOR* are:

- *EDITOR\_INPUT*
- *EDITOR\_TEXTAREA*

The attribute value identifiers for PDF report (*RPT*) customization elements are:

- *PAGE\_ORIENTATION\_PORTRAIT*
- *PAGE\_ORIENTATION\_LANDSCAPE*
- *PAGE\_SIZE\_LETTER*
- *PAGE\_SIZE\_LEGAL*
- *PAGE\_SIZE\_A3*
- *PAGE\_SIZE\_A4*
- *PAGE\_SIZE\_A5*
- *FONT\_FAMILY\_ARIAL*
- *FONT\_FAMILY\_COURIER*
- *FONT\_FAMILY\_HELVETICA*
- *FONT\_FAMILY\_TIMES*

The attribute value identifiers for *MEDIA\_TYPE* are:

- *TYPE\_ANY*
- *TYPE\_TEXT*
- *TYPE\_VIDEO*
- *TYPE\_IMAGE*

The attribute value identifiers for *EXPORT\_TYPE* are:

- *EXPORT\_TYPE\_TXT\_TAB*
- ~~*EXPORT\_TYPE\_TXT\_CSV*~~
- *EXPORT\_TYPE\_ODS*

## Context Identifiers

There are definitions for several context identifiers that are used to map custom functions into the program flow of the *SynApp2* request/response cycle.

The context identifiers for validation and process functions are:

- *VALIDATOR\_PRIMARY*
- *VALIDATOR\_SECONDARY*
- *PROCESS\_MAIN*

## Identifier Reference

*Note that the syntax of SQL statements or fragments in the examples may differ slightly from the syntax you might actually use, depending upon the database engine.*

### APPID

**Purpose:** Serves as a kind of namespace by defining a scope, within which, all customizations for a given application appear

**Synopsis:** `$this->m_data[APPID][<appid>] <key expression> = value`

**Example:** `$this->m_data[APPID]['app'][QID]['table1'][ORDER] = 'table1.column1';`

**Discussion:** The APPID identifier is always used in conjunction with other identifiers and appears in all of customization expressions covered by the **Identifier Reference** section.

Typically, the value associated with APPID is synonymous with a database name.

### QID

**Purpose:** Refine the scope of customizations within the context of the application APPID and support grouping according to purpose

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>] <key expression> = <value>;`

**Example:** `$this->m_data[APPID]['app'][QID]['table1'][ORDER] = 'table1.column1';`

**Discussion:** The QID identifier is always used in conjunction with other identifiers and appears in most of customization expressions covered by the **Identifier Reference** section.

Typically, the value associated with QID is synonymous with a table name.

### AUTH\_APPID

**Purpose:** Restrict an entire SynApp2 application to a specific set of users

**Synopsis:** `$this->m_data[APPID][<appid>][AUTH_APPID] = <usernames CSV>;`

**Example:** `this->m_data[APPID]['payroll'][AUTH_APPID] = 'dick, jane';`

**Discussion:** none

#### AUTH\_QID

**Purpose:** Restrict use of a QID to a specific set of users

**Synopsis:** `$this->m_data[APPID][<appid>][AUTH_QID][<qid>] = <usernames CSV>;`

**Example:** `this->m_data[APPID]['payroll'][AUTH_QID]['table1'] = 'dick, jane';`

**Discussion:** This provides an extra layer of safety, by preventing unauthorized query invocation regardless of how a request originates.

#### AUTH\_PID

**Purpose:** Restrict use of any SynApp2 page to a specific set of users

**Synopsis:** `$this->m_data[APPID][<appid>][AUTH_PID][<pid>] = <usernames CSV>;`

**Example:** `$this->m_data[APPID]['projtrack'][AUTH_PID]['process'] = 'dick, jane';`

**Discussion:** Also, combine [AUTH\_PID] with [INCL][NAV], to add navigation tabs, and then selectively authorize [i.e. display] those tabs only for the listed usernames.

#### AUTH\_RID

**Purpose:** Restrict use of any SynApp2 report to a specific set of users

**Synopsis:** `$this->m_data[APPID][<appid>][AUTH_RID][<rid>] = <usernames CSV>;`

**Example:** `$this->m_data[APPID]['projtrack'][AUTH_RID]['activity'] = 'tom, dick, harry';`

**Discussion:** Also, combine [AUTH\_RID] with [INCL][RPT], to add report tabs, and then selectively authorize [i.e. display] those tabs only for the listed usernames.

#### AUTH\_RECS

**Purpose:** Restrict a QID to a subset of records based on situation

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][AUTH_RECS] = <function name>;`

**Example:** `$this->m_data[APPID]['payroll'][QID]['pay'][AUTH_RECS] = "restrict to paymaster or manager direct reports";`

**Discussion:** The entry provides the name of a function that returns an empty string or an SQL term that will precisely define which records may be accessed by the active QID.

Consider the following pseudo-code:

```
function restrict_to_paymaster_or_manager_direct_reports()  
{
```

```

    $auth_recs_expr = '';

    // if the logged in user IS NOT the paymaster
    //     get the manager id, if any, for the login_username()
    //     $auth_recs_expr = "employee.id_current_manager =
                            '$logged_in_mgr_id'";

    return $auth_recs_expr;
}

```

To prevent selection of records, **you must return a valid expression**. In a case where, for the above example, the logged in user is not found to be a manager, then return an expression that cannot be satisfied, e.g. `employee.id_current_manager = ''`, when incorporated into the WHERE clause of the controlling SQL statement generated by the SynApp2 framework.

AUTH\_ADD,  
AUTH\_EDIT,  
AUTH\_DELETE

**Purpose:** Restrict database record manipulation to a specific set of users

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][AUTH_RECS] = <filter expression>;`

**Example:** `$this->m_data[APPID]['projtrack'][QID]['worker'][AUTH_ADD] = $admin_users;`

`$this->m_data[APPID]['projtrack'][QID]['worker'][AUTH_EDIT] = 'dick, jane';`

`$this->m_data[APPID]['projtrack'][QID]['worker'][AUTH_DELETE] = 'dick, jane';`

**Discussion:** These settings will (also) result in the automatically hiding or showing of corresponding action icons in the GUI of your SynApp2 generated application, depending on username.

AUTH\_PROCESS

**Purpose:** Restrict use of any SynApp2 process function to a specific set of users

**Synopsis:** `$this->m_data[APPID][<appid>][AUTH_PROCESS][<function>] = <usernames CSV>;`

**Example:** `$this->m_data[APPID][payroll][AUTH_PROCESS]['payroll_end_of_month'] = 'dick, jane';`

**Discussion:** This provides an extra layer of safety, by preventing unauthorized function invocation regardless of how a process related request originates.

## DATABASE

**Purpose:** Allow an application to use a database where the database name differs from the value of the application APPID

**Synopsis:** `this->m_data[APPID][<appid>][DATABASE] = <database name>;`

**Example:** `$this->m_data[APPID]['app'][DATABASE] = 'foo';`

**Discussion:** none

## TABLE

**Purpose:** Support reference to a basis table where the table name differs from the QID value.

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][TABLE] = <table name>;`

**Example:** `$this->m_data[APPID]['app'][QID]['table1'][TABLE] = 'blah';`

**Discussion:** Mapping a QID value to a table name is most often used to establish of a new data view, upon which generated pages and reports and their associated customization can be based.

An explicit mapping of QID value to table name may be called a secondary QueryID. A primary (or natural) QueryID mapping for every table is assumed - i.e. QID value equals table name.

**This element is typically manipulated via the SynApp2 Tools - QueryID page** and stored in the synapp2.inc.php file for the generated application.

## KEYMAP

**Purpose:** Provide an alternative to the <database>.\_keymap\_ table that that normally supports the SynApp2 keymap mechanism

**Synopsis:** `$this->m_data[DATABASE][<database name>][KEYMAP][] = <value>;`

**Example:** `$this->m_data[DATABASE]['sample'][KEYMAP][] =  
array(TABLE_NAME=>'table1',  
COL_NAME=>'table2_fk',  
JOIN_TABLE=>'table2',  
JOIN_COL=>'pk_column');`

**Discussion:** This particular customization keys off of a DATABASE name rather than APPID value. The sub-identifiers TABLE\_NAME, COL\_NAME, JOIN\_TABLE and JOIN\_COL are used to name the array elements.

## ORDER

**Purpose:** Change the default sort order for a specific QID

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][ORDER] = <SQL sort order expression>;`

**Example:** `$this->m_data[APPID]['app'][QID]['table1'][ORDER] = 'table1.column1';`

**Discussion:** The default sort order is ascending by primary key (PK) value.

**This element is typically manipulated via the SynApp2 Options - Order page** and stored in the synapp2.inc.php file for the generated application.

## MACRO

**Purpose:** Override the default expansion behavior for a specific foreign key (FK) column

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][MACRO][<key column>] = <column expression>;`

**Example:** `$this->m_data[APPID]['app'][QID]['t1'][MACRO]['t1.id_t2'] = "concat(t2.lastName, ', ', t2.firstName)";`

**Discussion:** By default, the SynApp2 SQL generator will expand foreign key (FK) columns to be the value of the first non-key field/column of the referenced record. The column expression will apply wherever the FK column would naturally appear. The expression will also apply to a corresponding drop-down list unless overridden by a LIST\_MACRO. See below.

**This element is typically manipulated via the SynApp2 Options Macro page** and stored in the synapp2.inc.php file for the generated application.

## LIST\_MACRO

**Purpose:** Override the default expansion behavior for a specific foreign key (FK) column used for drop-down lists

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][LIST_MACRO][<key column>] = <column expression>;`

**Example:** `$this->m_data[APPID]['app'][QID]['t1'][LIST_MACRO]['t1.id_t2'] = "concat(t2.firstName, ' ', t2.lastName)";`

**Discussion:** By default, the SynApp2 SQL generator will expand foreign key (FK) columns to be the value of the first non-key field/column of the referenced record.

**This element is typically manipulated via the SynApp2 Options List Macro page** and stored in the synapp2.inc.php file for the generated application.

## LIST\_MACRO\_ORDER

**Purpose:** Override the default sort order of drop-down list

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][LIST_MACRO_ORDER] = <sort order expression>;`

**Example:** `$this->m_data[APPID]['app'][QID]['table1'][LIST_MACRO_ORDER] = 'table2.column2';`

**Discussion:** **This element is typically manipulated via the SynApp2 Options List Macro Order page** and stored in the synapp2.inc.php file for the generated application.

## ~~LIST\_MACRO\_FILTER~~

**Purpose:** ~~Limit the drop-down list elements~~

**Synopsis:** ~~`$this->m_data[APPID][<appid>][QID][<qid>][LIST_MACRO_FILTER] = <filter expression>;`~~

**Example:** ~~`$this->m_data[APPID]['app'][QID]['employee'][LIST_MACRO_FILTER] = "dept.name = 'Engineering' or dept.name = 'Test'";`~~

**Discussion:** ~~none~~

## EXTRA

**Purpose:** Define a supplemental column expression

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][EXTRA][<basis column>][<column alias>] = <column expression>;`

**Example:** `$this->m_data[APPID]['app'][QID]['table1'][EXTRA]['table1.id_table2']['amount'] = "table2.qty * table2.price";`

**Discussion:** The column expression may contain references to any (qualified) table.column names that can logically be joined to the QID (basis) table. The SynApp2 SQL generator will automatically produce any (left) join clauses needed to satisfy the expression.

The column expression may be a sub-query. Be sure to surround the expression with parentheses, e.g. "(select count(\*) from line\_item where line\_item.id\_invoice = id)"

**This element is typically manipulated via the SynApp2 Options - Extra page** and stored in the synapp2.inc.php file for the generated application.

## OMIT

**Purpose:** Suppress nav tab(s) or report sub-tab(s)

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][OMIT][<tab identifier>] = <tabs CSV>;`

**Example:** `$this->m_data[APPID]['app'][OMIT][NAV] = 'honor_title, order_mode, orders';`  
`$this->m_data[APPID]['app'][OMIT][RPT] = 'genre';`

**Discussion:** This identifier is **not** used to suppress form column(s), as of SynApp2 version 0.1.4 and later. Use COL\_OMIT.

## INCL

**Purpose:** Generate supplemental nav tab(s) or report sub-tab(s)

**Synopsis:** `$this->m_data[APPID][<appid>][INCL][NAV] = <elements CSV>;`  
`$this->m_data[APPID][<appid>][INCL][RPT] = <elements CSV>;`  
`$this->m_data[APPID][<appid>][INCL][NAV][] = array(A_HREF => <filename>, A_TEXT => <tab text>);`

**Example:** `$this->m_data[APPID]['app'][INCL][NAV] = 'month_end_processing';`  
`$this->m_data[APPID]['app'][INCL][RPT] = 'sales summary, accounts receivable detail';`

**Discussion:** The value of a [NAV] element implies that a page named <element>.htm will be opened via the nav tab. If the visible nav tab name is to differ from the page filename, then the value may be expressed as an array (e.g. `$this->m_data[APPID]['synapp2'][INCL][NAV][] = array(A_HREF=>'pagegen', A_TEXT=>'PageGen');`)

The value of A\_HREF implies that a page named <filename>.htm will be opened via the nav tab. The sub-identifiers A\_HREF and A\_TEXT are used to name the value array elements. Note the empty [] square brackets.

The value of a [RPT] element implies that a page named <element>.report.htm will be opened via the report sub-tab.

## TAB\_ORDER

**Purpose:** Control sequence of page navigation tab(s) or report sub-tab(s)

**Synopsis:** `$this->m_data[APPID][<appid>][TAB_ORDER][NAV] = <elements CSV>;`  
`$this->m_data[APPID][<appid>][TAB_ORDER][RPT] = <elements CSV>;`

**Example:** `$this->m_data[APPID]['app'][TAB_ORDER][NAV] =`  
`' tab_c, tab_a'; // tab_c, tab_a, tab_b, tab_d, ..., tab_N`

**Discussion:** List tab/page names, from left to right, in the sequence you want them to appear. You do not have to list all of the tabs for the application. Any unspecified tab(s) will appear **after** the explicitly named ones. Use page name values as described for the identifier INCL above.

## QUERY

**Purpose:** Override the SynApp2 query generator and/or map a complete SQL statement to a specific QID

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][QUERY] =`  
`<SQL statement>;`

**Example:** `$this->m_data[APPID]['app'][QID]['table1'][QUERY] =`  
`'select * from table1 where column1 > 0';`

**Discussion:** none

## ~~SUB\_QUERY DEPRECATED~~

**Purpose:** ~~Add a column that is the result of a sub-query~~

**Synopsis:** ~~`$this->m_data[APPID][<appid>][QID][<qid>][SUBQUERY]`~~  
~~`[<column name>] = <(SQL statement)>;`~~

**Example:** ~~`$this->m_data[APPID]['app'][QID]['customer'][SUBQUERY]`~~  
~~`['orders_placed'] =`~~  
~~`'(select count(*) from orders where orders.id_customer =`~~  
~~`customer.id)';`~~

**Discussion:** ~~Note the parentheses surrounding the <(SQL statement)>. The parentheses are required. The <column name> becomes the column alias in the resulting query expression constructed by the SynApp2 SQL generator.~~

## FETCH

**Purpose:** Retrieve a column value in response to the onChange event of an iform <select> (drop-down list) element, or registered lookup on a pending ACTION\_INSERT, and/or on (JavaScript - synapp2.js) calls to do\_lookup() or do\_stat()

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][FETCH]`  
`[<column name>] = <column expression>;`

Example: `$this->m_data[APPID]['app'][QID]['item'][FETCH]  
['subtotal'] = 'format(sum(item.qty * items.price), 2)';`

Discussion: This mechanism triggers and processes a request/response cycle that is useful for reflecting a column value that depends on a/the value(s) of a newly selected record.

#### TITLE

Purpose: Provide an alternative title for a report

Synopsis: `$this->m_data[APPID][<appid>][QID][<qid>][<form identifier>]  
[TITLE] = <report title>;`

Example: `$this->m_data[APPID]['app'][QID]['item'][RFORM]  
[TITLE] = 'Sales Detail Report';`

Discussion: This attribute is only meaningful for RFORM (Report Form, i.e. PDF report)  
**This element is typically manipulated via the SynApp2 Options - Reporting page** and stored in the synapp2.inc.php file for the generated application.

#### LEGEND

Purpose: RESERVED

Synopsis:

Example:

Discussion:

#### LIMIT\_ROWS

Purpose: Override the default number of rows in a TFORM (select) form

Synopsis: `$this->m_data[APPID][<appid>][QID][<qid>][TFORM][LIMIT_ROWS] =  
<rows>`

Example: `$this->m_data[APPID]['app'][QID]['item'][TFORM][LIMIT_ROWS] =  
'100';`

Discussion: This attribute is only meaningful for TFORM. The value is applied to all SynApp2 generated pages where a TFORM appears for the associated QID. The default value for LIMIT\_ROWS is 5.

**This attribute is typically manipulated via the SynApp2 Options Display page** and stored in the synapp2.inc.php file for the generated application.

## COL\_OMIT

**Purpose:** Suppress a form column

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][<form identifier>][<column name>][COL_OMIT] = <flag>;`

**Example:** `$this->m_data[APPID]['app'][QID]['emp'][SFORM]['pwd'][COL_OMIT] = '1'; // true`

**Discussion:** Use this attribute to reduce clutter on forms. SFORM and DFORM often benefit from having fewer columns.

**This attribute is typically manipulated via the SynApp2 Options Display page** and stored in the synapp2.inc.php file for the generated application.

## COL\_SIZE

**Purpose:** Control column width on forms and reports

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][<form identifier>][<column name>][COL_SIZE] = <width>;`

**Example:** `$this->m_data[APPID]['app'][QID]['emp'][SFORM]['pwd'][COL_SIZE] = '20';`

**Discussion:** The width units are not strictly defined and the effect can vary depending on the rendering client. The visible width roughly corresponds to 'number of characters' but you may have to adjust the value to get the result you're looking for.

When width is applied to TFORM columns, implemented with HTML table elements, the [CSS] style attribute for 'white-space' is forced to a value of 'normal'. This allows word-wrapping to occur within cells.

**This attribute is typically manipulated via the SynApp2 Options Display page** and stored in the synapp2.inc.php file for the generated application.

## COL\_ALIGN

**Purpose:** Control column alignment on forms and reports

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][<form identifier>][<column name>][COL_ALIGN] = <alignment>;`

**Example:** `$this->m_data[APPID]['app'][QID]['item'][RFORM]['amount'][COL_ALIGN] = ALIGN_R;`

**Discussion:** This attribute is only meaningful for TFORM (Select Form) and RFORM (Report Form, i.e. PDF report) columns.

The <alignment> attribute value identifiers are ALIGN\_L, ALIGN\_J,

ALIGN\_C, and ALIGN\_R.

**This attribute is typically manipulated via the SynApp2 Options Display page** and stored in the synapp2.inc.php file for the generated application.

## COL\_FORMAT

**Purpose:** Control appearance of column data on reports

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][<form identifier>][<column name>][COL_FORMAT] = <function[,format]>;`

**Example:**

```
$this->m_data[APPID]['app'][QID]['item'][RFORM]['each']  
[COL_FORMAT] =  
    'sprintf, $%.2f'; // currency dollars.cents  
$this->  
>m_data[APPID]['app'][QID]['invoice'][RFORM]['invoice_date']  
[COL_FORMAT] =  
    'col_format_date, d M Y'; // e.g. 01 Jan 2009  
$this->  
>m_data[APPID]['app'][QID]['invoice'][RFORM]['invoice_date']  
[COL_FORMAT] =  
    'col_format_date, d/m/y'; // e.g. 01/01/09
```

**Discussion:** This attribute is only meaningful for RFORM (Report Form, i.e. PDF report) columns.

Any PHP function may be invoked directly by this mechanism as long as the function signature follows either of these calling conventions: `<function>(<value>)` or `function(<format>, <value>)`.

To invoke a function with a different signature, create a 'wrapper' function to transform the arguments. Specify the name of wrapper function as the value of the COL\_FORMAT attribute.

Formatting functions should return a string.

SynApp2 provides the following pre-defined wrapper:

```
function col format date($fmt, $value)  
{  
    return date_format(date_create($value), $fmt);  
}
```

See: <http://www.php.net/manual/en/function.date.php> for information about formatting date values.

See: <http://www.php.net/manual/en/function.sprintf.php> for information about formatting other values.

**This attribute is typically manipulated via the SynApp2 Options Display page** and stored in the synapp2.inc.php file for the generated application.

## COL\_EDITOR

**Purpose:** Control the method used to enter column values

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][<form identifier>][<column name>][COL_EDITOR] = <editor>;`

**Example:** `$this->m_data[APPID]['app'][QID]['customer_orders'][IFORM]['special_instructions'][COL_EDITOR] = EDITOR_TEXTAREA;  
$this->m_data[APPID]['app'][QID]['item'][IFORM]['description'][COL_EDITOR] = EDITOR_INPUT;`

**Discussion:** This attribute is only meaningful for IFORM (Input Form) columns. EDITOR\_INPUT is the default editor for column values unless their length exceeds 40 or their data type is 'text' (i.e. text-blob).

The <editor> attribute value identifiers are EDITOR\_INPUT and EDITOR\_TEXTAREA.

**This attribute is typically manipulated via the SynApp2 Options Display page** and stored in the synapp2.inc.php file for the generated application.

## COL\_EDITOR\_ROWS

**Purpose:** Control the number of rows for EDITOR\_TEXTAREA

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][<form identifier>][<column name>][COL_EDITOR_ROWS] = <rows>;`

**Example:** `$this->m_data[APPID]['app'][QID]['customer_orders'][IFORM]['special_instructions'][COL_EDITOR_ROWS] = '10';`

**Discussion:** This attribute is only meaningful for IFORM (Input Form) columns and when EDITOR\_TEXTAREA is the COL\_EDITOR.

**This attribute is typically manipulated via the SynApp2 Options Display page** and stored in the synapp2.inc.php file for the generated application.

## COL\_ATTRS

**Purpose:** RESERVED

**Synopsis:**

**Example:**

**Discussion:**

## COL\_ORDER

**Purpose:** Control column output sequence of a query result

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][<form identifier>][COL_ORDER] = <column names CSV>;`

**Example:** `$this->m_data[APPID]['app'][QID]['item'][RFORM][COL_ORDER] =  
          'col_c, col_a'; // col_c, col_a, col_b, col_d, ..., col_N`

**Discussion:** This element is only meaningful for RFORM (Report Form, i.e. PDF report) columns. List column names, from left to right, in the sequence you want them to appear.

You do not have to list all columns from the data set. Any unspecified column(s) will be output **after** the explicitly named ones, in the same sequence as they appear in the query result.

**This element is typically manipulated via the SynApp2 Options Col Order page** and stored in the synapp2.inc.php file for the generated application.

## RPT\_PAGE\_SIZE

**Purpose:** Control page dimensions for reports

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][RPT_PAGE_SIZE] = <page size>;`

**Example:** `$this->m_data[APPID]['app'][QID]['item'][RPT_PAGE_SIZE] =  
          PAGE_SIZE_LETTER;`

**Discussion:** The <page size> attribute value identifiers are PAGE\_SIZE\_LETTER, PAGE\_SIZE\_LEGAL, PAGE\_SIZE\_A3, PAGE\_SIZE\_A4 and PAGE\_SIZE\_A5.

**This attribute is typically manipulated via the SynApp2 Options Reporting page** and stored in the synapp2.inc.php file for the generated application.

## RPT\_PAGE\_ORIENTATION

**Purpose:** Control page orientation for reports

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][RPT_PAGE_ORIENTATION] = <page orientation>;`

**Example:** `$this->m_data[APPID]['app'][QID]['item'][RPT_PAGE_ORIENTATION] =  
          PAGE_ORIENTATION_LANDSCAPE;`

**Discussion:** The <page orientation> attribute value identifiers are PAGE\_ORIENTATION\_PORTRAIT and PAGE\_ORIENTATION\_LANDSCAPE.

**This attribute is typically manipulated via the SynApp2 Options Reporting page** and stored in the synapp2.inc.php file for the generated application.

## RPT\_FONT\_SIZE

**Purpose:** Control point size of font for reports

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][RPT_FONT_SIZE] = <point size>;`

**Example:** `$this->m_data[APPID]['app'][QID]['item'][RPT_FONT_SIZE] = '12';`

**Discussion:** The default value for <point size> is 8.

**This attribute is typically manipulated via the SynApp2 Options Reporting page** and stored in the synapp2.inc.php file for the generated application.

## RPT\_FONT\_FAMILY

**Purpose:** Control typeface for reports

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][RPT_FONT_FAMILY] = <typeface>;`

**Example:** `$this->m_data[APPID]['app'][QID]['item'][RPT_FONT_FAMILY] = FONT_FAMILY_COURIER;`

**Discussion:** The <typeface> attribute value identifiers are FONT\_FAMILY\_ARIAL, FONT\_FAMILY\_COURIER, FONT\_FAMILY\_HELVETICA and FONT\_FAMILY\_TIMES.

**This attribute is typically manipulated via the SynApp2 Options Reporting page** and stored in the synapp2.inc.php file for the generated application.

## RPT\_CELL\_BORDER

**Purpose:** Control surrounding outline of data cells for reports

**Synopsis:** `$this->m_data[APPID][RPT_CELL_BORDER] = <flag>;`

**Example:** `this->m_data[APPID]['app'][RPT_CELL_BORDER] = SWITCH_ON;`

**Discussion:** This attribute affects **all** RFORM (Report Form, i.e. PDF report) output for the generated application. The default value is: SWITCH\_OFF.

**This attribute is typically manipulated via the SynApp2 Options Reporting page** and stored in the synapp2.inc.php file for the generated application.

## RPT\_ROW\_SHADE

**Purpose:** Control alternating background highlight of report rows

**Synopsis:** `$this->m_data[APPID][RPT_ROW_SHADE] = <flag>;`

**Example:** `$this->m_data[APPID]['app'][RPT_ROW_SHADE] = SWITCH_OFF;`

**Discussion:** This attribute affects **all** RFORM (Report Form, i.e. PDF report) output for the generated application. If RPT\_ROW\_SHADE and DETAIL\_SUMMARY\_COLS are both active for a report, row shading occurs only for summary rows. The default value is: SWITCH\_ON.

**This attribute is typically manipulated via the SynApp2 Options Reporting page** and stored in the synapp2.inc.php file for the generated application.

#### RPT\_ROW\_SIZE

**Purpose:** Force a uniform row height for PDF reports [with images]

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][RPT_ROW_SIZE] = <point size>;`

**Example:** `$this->m_data[APPID]['app'][QID]['item'][RPT_ROW_SIZE] = '12'; // em`

**Discussion:** The size units are not strictly defined and the effect can vary depending on report font size. The row height value roughly corresponds to 'a multiple of character height'. Adjust the value based upon evaluation of actual report output. Choose a value that produces the best compromise between number of report rows per page and image size.

This element is intended for use when a query result column has been customized with a MEDIA\_TYPE of TYPE\_IMAGE. Images are automatically scaled to fit, while maintaining the original aspect ratio. The best results are typically achieved with images that have square proportions, and have been pre-processed for minimum file size, with native dimensions reasonably close to the final output size.

#### DETAIL\_SUMMARY\_COLS

**Purpose:** Add an aggregation row for specific column of a report

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][DETAIL_SUMMARY_COLS][] = <column name>, <group_by_col_name>[, <suppress left N cols>;`

**Example:** `$this->m_data[APPID]['app'][QID]['item'][DETAIL_SUMMARY_COLS][] = 'discount, invoice_number, 3'; // invoice_number, date, cust`

`$this->m_data[APPID]['app'][QID]['item'][DETAIL_SUMMARY_COLS][] = 'amount, invoice_number';`

**Discussion:** For each record group, statistics are computed for: count, sum, ave, min and max. An output row is added, after each record group, to reflect the value of the sum statistic for the specified column name. A grand total row is added, to the end of the report, to reflect the value of the sum statistic for the specified column name. Data columns, whose values repeat for every row of a group, can be suppressed. Specify the actual number of columns to suppress. The ORDER specified for the QID of any report, that employs the DETAIL\_SUMMARY\_COLS mechanism,

should rely on an expression that depends **strongly** upon the 'group\_by\_col\_name'. Important: Note the empty square braces [] that are part of the key expression. Do not fail to include them.

**This element is typically manipulated via the SynApp2 Options Detail Summary Cols page** and stored in the synapp2.inc.php file for the generated application.

#### EXPORT\_TYPE

**Purpose:** Register the report data export formatting preference

**Synopsis:** `$this->m_data[APPID][<appid>][EXPORT_TYPE] =  
[<export type>]; // app-level`

`$this->m_data[APPID][<appid>][QID][<qid>][EXPORT_TYPE] =  
<export type>; // qid-specific`

**Example:** `$this->m_data[APPID]['app'][EXPORT_TYPE] =  
EXPORT_TYPE_ODS;`

**Discussion:** There are two forms for this customization element. The <export type> attribute value identifiers are EXPORT\_TYPE\_TXT\_TAB, EXPORT\_TYPE\_CSV and EXPORT\_TYPE\_ODS.

The default format is EXPORT\_TYPE\_TXT\_TAB - Text (Tab-delimited) and is readily imported by most electronic spreadsheet programs. Implementation of EXPORT\_TYPE\_TXT\_CSV - Text (Comma-separated values) is pending, as of SynApp2 version 1.8.0beta\_3. EXPORT\_TYPE\_ODS - OpenDocument Spreadsheet supports imbedded images, but can only be read by ODS capable programs.

#### VALIDATOR

**Purpose:** Register a column value validation function name for a specific QID and validation context key

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][VALIDATOR]  
[<context key>] = <function name>;`

**Example:** `include ('my_functions.php');`

`$this->m_data[APPID]['app'][QID]['reservation'][VALIDATOR]  
[VALIDATOR_SECONDARY] = 'validate_checkout_date';`

**Discussion:** The return value of a validation function is a string.

An empty return string indicates successful validation. Indicate validation failure with the exact message text you want to appear in the iform feedback label element bound to the col\_var.

Look at \_shared\_/validate.php to see how the validation context key applies and how the return value of your function(s) affects program flow.

The validation <context key> may be VALIDATOR\_PRIMARY or VALIDATOR\_SECONDARY.

Be aware that any function mapped with the VALIDATOR\_PRIMARY <context key> will override (i.e. replace) the automatic intrinsic primary validation of the named col\_var.

Be sure to define your functions with the correct parameters for the <context key> you use to map them.

The form of a custom **primary** validation function is:

```
function <function_name>(
    $col_name,
    $col_var,
    $display_name,
    &$col_vars)
{
    $status_msg = '';

    if ($col_var <test expression>)
    {
        $status_msg = "{$display_name} failed validation.";
    }

    return $status_msg;
}
```

The form of a custom **secondary** validation function is:

```
function <function_name>(
    $col_name,
    $col_var,
    $display_name,
    &$col_vars,
    &$feedback)
{
    $status_msg = '';

    $other_col_var_is_valid =
        !is_set($feedback[<other_col_name>]) &&
        is_set($col_vars[<other_col_name>])) ? true : false ;

    if ($other_col_var_is_valid &&
        ($col_var <test expression>
        $col_vars[<other_col_name>]['v']))
    {
        $status_msg = "{$display_name} failed validation
                        against <other_col_name>.";
    }

    return $status_msg;
}
```

Parameters passed to your function(s) are:

\$col\_name -- the name of the col\_var subject being validated

\$col\_var -- the posted value (or array of values) associated with the col\_name

\$display\_name - the computed display name for col\_name

`$col_vars` -- a reference to an array of `col_vars` posted with the action request

`$feedback` - a reference to the output buffer array for validation failure messages (supplied only for context `VALIDATOR_SECONDARY`)

Some parameters are passed by reference for the purpose of efficiency. They should be treated as constants by your functions.

## PROCESS

**Purpose:** Register a function name for a specific QID and process context key

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][PROCESS][<context key>] = <function name>;`

**Example:** `include ('my_functions.php');`

```
$this->m_data[APPID]['app'][QID]['month_end'][PROCESS]
[PROCESS_MAIN] = 'month_end_processing';
```

**Discussion:** The return value of a processing function is bool.

Look at `_shared_/action.php` to see how the process context key applies and how the return value of your function(s) affects program flow.

The form of a custom processing function is:

```
function <function_name>(&$args, &$action, &$adhoc_markup)
{
    $adhoc_markup[] = "<h1>Hello World!</h1>";

    if (isset($args['name']))
    {
        $value = $args['name'];
        // code . . .
    }

    return true; // continue action
}
```

Parameters passed to your function(s) are:

`$args` - a reference to an array of name, value pairs passed from your GUI via the action request mechanism

`$action` -- a reference to the action object instance managing the request/response cycle

`$adhoc_markup` -- a reference to an array (of lines) that will be returned to your browser and inserted directly as the innerHTML of your adhoc container element

## MEDIA\_TYPE

**Purpose:** Register a media type for a query result column

**Synopsis:** `$this->m_data[APPID][<appid>][QID][<qid>][MEDIA_TYPE][<column name>] = <media type>;`

**Example:** `$this->m_data[APPID]['app'][QID]['item'][MEDIA_TYPE]['photo'] = TYPE_IMAGE;`

**Discussion:** The <media type> attribute value identifiers are TYPE\_ANY, TYPE\_TEXT, TYPE\_VIDEO and TYPE\_IMAGE. The designation affects handling of the column for during report generation and ties the column to the media location definition, if any, for that type.

## TEMP\_LOC

**Purpose:** Register an applications relative directory for temporary files

**Synopsis:** `$this->m_data[APPID][<appid>][TEMP_LOC] = <apps relative directory>; // app-level`

`$this->m_data[APPID][<appid>][QID][<qid>][TEMP_LOC] = <apps relative directory>; // qid-specific`

**Example:** `$this->m_data[APPID]['app'][TEMP_LOC] = 'app_tmp';`

**Discussion:** There are two forms for this customization element. In the absence of customization, server-dependent default behavior applies. If the PHP function\_exists('sys\_get\_temp\_dir') and it returns a valid directory, then that value is used, otherwise the applications directory, parent of the generated application directory, is used.

As of version 1.8.0beta\_3, temporary files are only used to support EXPORT\_TYPE\_ODS.

## MEDIA\_LOC

**Purpose:** Register an applications relative directory for media files

**Synopsis:** `$this->m_data[APPID][<appid>][MEDIA_LOC][<media type>] = <apps relative directory>; // app-level`

`$this->m_data[APPID][<appid>][QID][<qid>][MEDIA_LOC][<media type>] = <apps relative directory>; // qid-specific`

**Example:** `$this->m_data[APPID]['app'][MEDIA_LOC][TYPE_IMAGE] = 'app_img';`

**Discussion:** There are two forms for this customization element. The <media type> attribute value identifiers are TYPE\_ANY, TYPE\_TEXT, TYPE\_VIDEO and TYPE\_IMAGE. This element is used when generated PDF reporting detects query result columns with MEDIA\_TYPE customization. Media location can also be incorporated into EXTRA column expressions, used to produce thumbnail images (for Select Form presentation).



## Resources

### *Reference Books*

*JavaScript: The Definitive Guide*, David Flanagan - <http://oreilly.com/catalog/9780596101992>

*Bulletproof Web Design*, Dan Cederholm - <http://simplebits.com/publications/bulletproof>

### *Links*

SynApp2 Documentation - [http://www.synapp2.org/main/?page\\_id=6](http://www.synapp2.org/main/?page_id=6)

SynApp2 Support Forums - <http://synapp2.org/forum>

PHP Manual (English) - <http://www.php.net/manual/en>

MySQL 5.1 Reference Manual (English) - <http://dev.mysql.com/doc/refman/5.1/en>

Oracle® Database SQL Reference -  
[http://download.oracle.com/docs/cd/B19306\\_01/server.102/b14200/toc.htm](http://download.oracle.com/docs/cd/B19306_01/server.102/b14200/toc.htm)

Oracle 10g Express Edition -  
<http://www.oracle.com/technetwork/database/express-edition/overview/index.html>

phpMyAdmin - <http://www.phpmyadmin.net>

Web Standards - <http://www.w3.org/standards>

Markup Validation Service - <http://validator.w3.org>

CSS Validation Service - <http://jigsaw.w3.org/css-validator>

HTML Tutorial - <http://www.w3schools.com/html>

CSS Tutorial - <http://www.w3schools.com/css>

JavaScript Tutorial - <http://www.w3schools.com/jsref>

PHP Tutorial - <http://www.w3schools.com/php>

SQL Tutorial - <http://www.w3schools.com/sql>