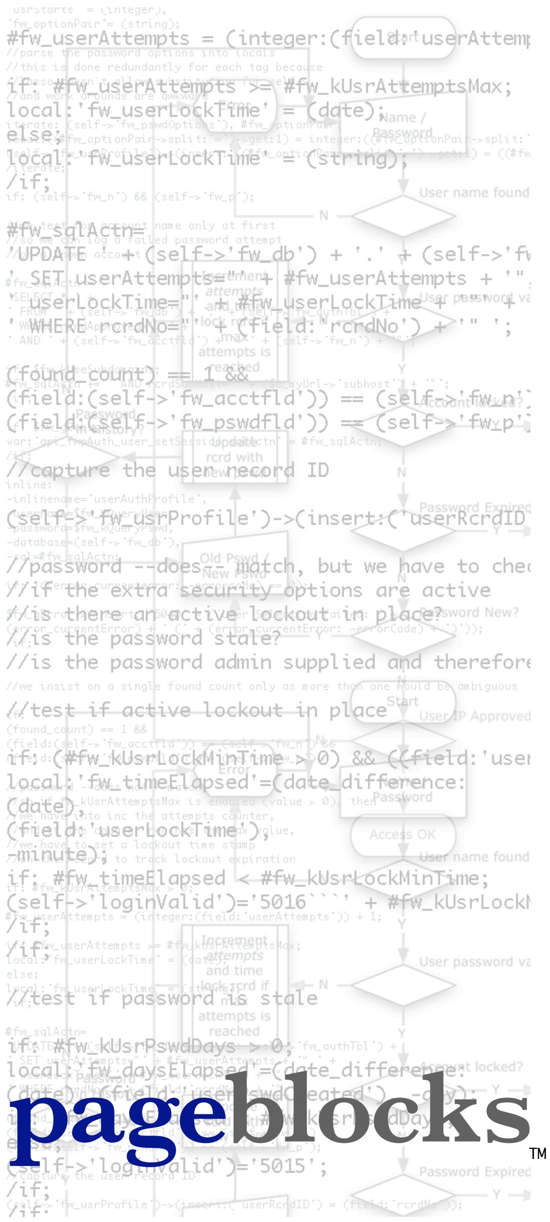
PageBlocks
Internet Application Framework
for Lasso Professional Server 8.1 / 8.5

Developer Guide rev 9

Release 5.3.0, August 17, 2007



Preface

This preliminary documentation is somewhat rough (organizationally and graphically). It's currently a mix of Guide and Reference, though I want the docs to eventually evolve into separate volumes with a how-to Guide to discuss application and solution design, and a Reference to be the in-depth technical review of the code. You'll likely have to dig into the source code and read these notes iteratively. As much as possible, I tried to push the details of the tags themselves into the reference database for now.

Eventually, I'd like to try to accomplish what probably should be two separate tasks: to introduce the reader to several prevailing technologies in software development, and the specific details of the PageBlocks web application framework. The general programming technologies information is important for two reasons: first the developer should be aware of the prevailing practices that today would be considered essential qualities of professionally developed software, and secondly most of these technologies are implemented in the PageBlocks framework to some degree, so an understanding of them will inherently lead to improved understanding as to how and why PageBlocks is organized the way it is.

It is assumed the reader has at least some experience using Lasso, or perhaps is new to Lasso but has experience with some other form of programming. This text will not attempt to explain rudimentary programming terms or practices, but will make an effort to bring a novice into the realm of intermediate status.

For now, I'm using Apple's Pages program for editing (to get taste of its abilities). It's not too bad, but has limited control for integrating illustrations the way I'd like to, so layout is a little on the boring side.

Acknowledgments

Thanks to Peter Bethke for his original white paper The Corral Methodology *and his efforts, along with the support of John May at Point In Space, to provide a central resource for discussing and exploring site structure and dynamic page assembly. Those early days got me off to a good running start.*

Thanks to Johan Solve for sharing the work and sample source code of his OneFile methods (and the countless tips and guidance to the Lasso community).

To everyone that has looked at, played with, and developed with my code and provided some feedback, a huge thanks to you. A special thanks to Nikolaj de Fine Licht for his extensive testing and feedback.

To the Lasso developer community, and particularly to those whose shoulders I stand on, I've appreciated the tips, guidance, and discussions. Hopefully you'll find something useful in how I've interpreted and packaged what you've taught me, and everything I've learned so far.

Version 5.3 Update Highlights

Version 5.3.0 Update

- the pbStart and pbJedi projects include a completely new /siteStringsMVS/ module which provides a more complete interface to getting started with content for Multi-View Strings. It taps in multiple data tables now to access individual languages, and it allows editing two languages side by side so that a reference language can be pulled up for helping to edit additional language/media/variant entries. It's by no means a perfect interface for the job, but it's a good start, and it provides a demonstration for subclassing `fwp_recordData` as well as building multi-purpose, single-page functionality. That module is a tiny version of what is going on in the EDP system. The original /siteStrings/ module is left in the projects as a simpler version that can only work with a single language. Use either it or /siteStringsMVS/ in your projects if you you're using `appStrings`, but you don't need both.
- fixed some bugs in `fwp_validator` tha could cause crashes if a input value was null
- added `->getResults` and `->resultsReset` to `fwp_validator` so that results can be copied into a new object, and results can be reset so the validator can be easily used on multiple record objects in series. `fwp_recordData` now uses `->getResults` automatically to encapsulate validation results. The validator uses `->resultsReset` on its own prior to each validation to ensure the results buffer is clear.
- changed `fwpActn_loadTableModel` (which is used by `fwp_recordData` to load tableModel files as needed) to run as `-privileged` so the use of `database_tableNames` would not need Show Table Schema rights in SiteAdmin. This simplifies setup requirements added in 5.2.
- in `fwp_recordData`, removed `fw_` prefix from `->fw_validateInputs` member tag name to open it up for direct access. This should not impact any existing code.
- several new features added to `fwp_recordData` to improve subclassing and validation
  - added `-wherePairs` option to `->getRecordUsingLock` to improve flexibility in defining which record to obtain. It's not a full implementation of the `-where` option in `->select`, but it now allows locking records using AND in the WHERE clause. I'll be looking implementing complete `-where` functionality for a future relaease.
  - added `->'validationResults' instance var to` to contain validation data results. It is a map that has three keys: `errorMsgs`, `coreCodes`, `appCodes` which are identical in content and purpose as those same instance vars of the `fwp_validator`.
  - added `->showMsgsFor` to display error messages for a given field. Its role and usage is the same as `$fw_validator->showMsgs`, however this new tag improves the encapsulation of models built with `fwp_recordData`. Accepts the name of the input as a nameless parameter.
  - added `->errorExistsFor` to test whether there is an error for a given input to a model. Accepts the name of the input as a nameless parameter, and returns true or false. Check the `validationResults` to determine what the error is if necessary.
  - changed `-asRecordsMaps` to return the maps using the inputs names rather than the field names, to get the old version for compatabillity change all existing `-withRecordsMaps` and `-asRecordsMaps` parameters to `-withRecordsMapsFields` or `-asRecordsMapsFields` (these are

synonyms). This expands the ability to write all code using the input names rather than field names.

- updated `_fwpAPI_init` to solve a problem with Apache 2.2 not adding DirectoryIndex file names to URLs with trailing slashes. However, this fix forces the use of default.lasso as an index file. You can edit that line to use your prefered file name for now. I'll investgate smarter adaptions for a future release.
- added `$fw_logoutPagePath` which gets redirected to after the logout is completed
- added `-ignoreCache` option to `fwp_rcrdsList` to bypass the cached list status for when the list needs to be reset to clear previous search and pagination values
- fixed some bugs in fwpFile_uploadManager.ctyp
- added valueLists_languages_en-us.cnfg which provides an official list of ISO languages and codes.
- several changes to CSS files for WinIE fixes. Mostly use the * prefix hack for monospaced font size overrides for IE6 and IE7, and added `overflow: hidden;` to some div specs.
- added `div#stdadminfilterbar` to pb_apiStyles.css, and added color specs to Validation Msgs
- added `fwptxtmenuhilitelink` class to `<a>` tags in fwpGui_menuHText.ctag and fwpGui_menuVText.ctag to allow for better style control over hilighted links in text menus

Migrating to 5.3.0

If you have a project started with an earlier release of 5.2, there's some things to rearrange and update for 5.2.5 that warrant some step-by-step details.

Update _initMasters

In section (H), look for this line (it may be modified in your project):

```
//  $fw_loginPagePath      = '/sitemngr/';
```

and add this line after it.

```
//  $fw_logoutPagePath     = '/sitemngr/';
```

This new variable designates the page the app should be redirected to after a Logout link is clicked. When the logout link is clicked it leaves a nasty URL in the browser with a now stale session ID. PB 5.3 will redirect to the URL in `$fw_logoutPagePath` after a logout has completed in order to clean up the URL.

Replace Obsolete Names

In version 5.2.x and earlier, `fwp_recordData` used an `-asRecordsMaps` parameter in some of the member tags like ->select to return records as an array of maps. The map keys were the data table field names. In 5.3 this has been changed so that the map keys are the input named defined in the tableModel config file. If you're using `-asRecordsMaps` in your code, this change will break your code, but there is an easy fix

Change all occurrences of `-asRecordsMaps` to `-asRecordsMapsFields` and the maps will be returned just like they were in 5.2.x.

Version 5.2 Update Highlights

Version 5.2.5 Update

- extended the Lasso `[string]` data type to include the following member tags (improved version of similar `fwpStr` tags that have existed for some time):
  - `->getLeft` : gets the leftmost (n) characters of a string
  - `->getRight` : gets the rightmost (n) characters of a string
  - `->getWords` : gets the first (n) words of a string
  - `->getSentences` : gets the first (n) sentences of a string based on splitting at `'. '`
  - `->getParagraphs` : gets the first (n) paragraphs of a string with paragraph detection including double `br` tags, `p` tags, and line feeds.
- extended the Lasso `[integer]` data type to include the following member tags which return a date string based on an integer. Use these like `$sessionExpires = 30->minutesFromNow`
  - `->secondsFromNow`
  - `->minutesFromNow`
  - `->hoursFromNow`
  - `->daysFromNow`
  - `->weeksFromNow`
  - `->monthsFromNow`
  - `->yearsFromNow`
  - `->secondsAgo`
  - `->minutesAgo`
  - `->hoursAgo`
  - `->daysAgo`
  - `->weeksAgo`
  - `->monthsAgo`
  - `->yearsAgo`
- extended the Lasso `[integer]` data type to include about 300 unit conversions for units of data size (bytes), length/distance, weight, fluid volume, and area. Units include American standard and Metric. Look at file fwpExtn_integer.lgc for details of each available conversion.
- much improved support for custom configurations of template `<head>` `meta` tags, `script` tags for external JavaScript libraries, and `link` tags for external CSS files through a new `$fw_headContent` object with the following member tags:
  - `->addMetaTag` : adds a pair to define the name and content of a meta tag
  - `->addHttpEquiv` : adds a pair to define the http-equiv and content of a meta tag
  - `->addCssFile` : adds a full path name to an external CSS file
  - `->addScriptFile` : adds a full path name to an external JavaScript file
  - `->removeMetaTag, ->removeHttpEquiv, ->removeCssFile, ->removeScriptFile` : corresponding remove methods can be used to modify a baseline of adds for a specific module or page

- eliminated the vars `fw_HTTPexpires`, `fw_HTTPauthor`, `fw_HTTPdesc`, and `fw_HTTPkeyWords` in favor of using the new `$fw_headContent` object.
- _defineCSS.lgc and _defineJavaScript.lgc continue to be options for creating customized logic, but both are disabled by default now.
- eliminated already obsolete `$fw_gUserPermsVars` from _initMasters
- eliminated already obsolete `$fw_pgRepeatLogic`, `$fw_pgRepeatAbove`, and `$fw_pgRepeatBelow` from _initMasters
- fixed typo for Spotlight connector in _fwpAPI_init
- bug fix to `fwpPage_loadTemplate` that was not including _definePageHead.lgc
- fixed a bug in fwpGUI_listRcrds.ctag that could cause lists draw without data
- updated some files in /siteUsers/ to use input names rather than field names in the queries and list config files (not a critical functional change, just a change to example code)

Migrating to 5.2.5

If you have a project started with an earlier release of 5.2, there's some things to rearrange and update for 5.2.5 that warrant some step-by-step details.

Update _initMasters

Generally speaking look for the lines with the //comments and variable declarations for the following variables and delete these lines:

- `fw_HTTPexpires`, `fw_HTTPauthor`, `fw_HTTPdesc`, and `fw_HTTPkeyWords`
- `fw_pgRepeatLogic`, `fw_pgRepeatAbove`, and `fw_pgRepeatBelow`
- `fw_gUserPermsVars`

In section (H), eliminate some of the now obsolete comment lines by modifing the top of the help comments to look like the following:

```
//  $fw_headContent    see docs
//  $fw_pageModes      see docs

//  $fw_client
//  ->decimalChar      ('.') default character for decimal in fwpNum and fwpMath tags
..... etc.....
```

Still in section (H), after the line:

```
//  all these vars can be overidden in _pageConfig
```

insert the following default lines above the `$fw_pageMode` lines:

```
//  $fw_headContent->(addHttpEquiv:   (pair:'expires'='0'));

    $fw_headContent->(addMetaTag:     (pair:'author'=''));
    $fw_headContent->(addMetaTag:     (pair:'description'=''));
    $fw_headContent->(addMetaTag:     (pair:'keywords'=''));

    $fw_headContent->(addCssFile:     ($fw_sPath->'css') + 'pb_apiStyles.css');
    $fw_headContent->(addCssFile:     ($fw_sPath->'css') + 'styles_main.css');

    $fw_headContent->(addScriptFile: ($fw_sPath->'js') + 'scripts_main.js');
```

Assuming you have definitions for those meta tags, transfer the content from the now obsolete vars to the declarations above.

Compare your modifications to the _initMaster.lgc file in the /pbStart/ project folder.

Update /_admin/ Files

If your application was using the styles_admin.css file to modify how the admin sections looked, you'll need to add the following `->addCssFile` line in _pageConfig for the affected modules such as /siteUsers/ and /sitestrings/:

```
if: ($fw_requestPage->'path') >> '_admin';
    $fw_headContent->(addCssFile: ($fw_sPath->'css') + 'styles_admin.css');
```

Eliminate Obsoleted Vars

Depending on which versions of the demo files your app may have initially been created with, there may be instances of some of the now obsoleted vars, so do a global search for these, and eliminate them.

- `fw_HTTPexpires`, `fw_HTTPauthor`, `fw_HTTPdesc`, and `fw_HTTPkeyWords`
- `fw_gUserPermsVars`

Version 5.2.4 Update

- added /siteMngr, /siteUsers and other admin features to pbStart file set.
- updated login procedure in mngr_login2_main.lgc to test for errors from `->authenticate` before trying the `->authorize` step. This will help to accurately report login lockouts and other specific login errors.
- updated error message 5016 in strings_coreErrors_en-us.cnfg
- bug fix for `fwp_recordData->delete` that required an -inputs parameter (which served no purpose)
- added `-formSpec` option to fwpErr_validator.ctyp to make it easier to use the validator on raw POST and GET data. See documentation section *Using fwp_validator on POST and GET*
- bug fixes to fwpEDP_controller.ctyp
- fixed sorting bug introduced to fwpGui_listRcrds.ctyp by a recent code update
- some minor updates to pb_apiStyles.css
- fixed a case where the `fwpLog_err` tag was bing invoked even if `$fw_gLogErr` was set to false
- added trace calls in log tags
- added a trace call to the `->insert` action of `$fw_error` so that we can see exactly when errors occur in the trace

Version 5.2.2/5.2.3 Update

Some minor fixes were release with 5.2.1, but with 5.2.2 there's a a major update to how value lists are created and used. There's a whole new system with new capabillities documented in detail in this updated Guide. The previous value list tags are deprecated but still available with the documentation for those moved to Appanedix A. There's no urgent need to rewrite existing value list code in applications. The old tags will likely remain available for some time. However, using the new value list data type will help improve the logic/display separation of your code.

Additonally, there are new fwpGIS tags for acquiring a U.S. address geocode from the Google API and from the Yahoo API. (5.2.3 added -minPrecision to these tags, and fixed a bug).

Added `fwpNum_dec5` and `fwpNum_dec6` tags.

Version 5.2.0 Overview

- New—integrated database migrations management
- New—integrated deployment mode detection
- New—Nemoy search engine (Winner of first Lasso Programming Challenge)
- Enhanced feedback for misconfiguration and developer errors
- Enhanced query abstraction system
- Enhanced API startup process for better 3rd party integration

- Integrated template head and page wrapup tasks into templateLoader to prevent framework updates from overwriting developer code
- Separated all default definitions for CSS styles embedded in framework HTML generators into a separate CSS file for easy modification or overriding
- Several minor fixes, improvements, and a few new tags

- The init process of the API files that go into /LassoStartup/ has been updated. This won't affect any application code, but allows API sets to be initialized individually, included or excluded, and allows application code to test for dependencies of API libraries. See the documentation section *API Startup and Dependencies*
- The page assembly process has been updated to simplify the developer's code, and to help isolate API updates from overwriting developer code in certain areas. In previous versions, the files in /_pbLibs/ were mixtures of API code and developer code. The intent was to allow the developer to modify these core processes. The default processes have now been fully integrated into the template loader, and files in this folder are now either additive or alternative to the built-in processes. This means framework updates no longer require the developer to scour through these files looking for update modifications.
- New multi-deployment management enables resource and behavior modification on-the-fly based on the domain name the site runs as. This allows the use of separate databases when running the site under production, test, development, etc. For example, when the site runs as www.example.com, it will use the database mydb_prod, but when run locally as www.example.dev it will automatically use mydb_test. Declare as many cases as needed to cover production, beta, test, development, etc.
- Added numerous error handlers to catch common misconfiguration and errors in setup and use of PageBlocks tags and types. Prior to these, it was common for misconfigurations to result in obscure Lasso error messages. Now, more specific information is provided with each error, and the tagTracer now displays more error information to help identify exactly when in the page processing the error happened.
- A new `$fw_apiError` object is used to capture programming errors so that `$fw_error` can be focused on providing error feedback to end users. This allows messages to show alternative versions based on whether debugging is enabled or not.
- Added escalation values to `$fw_debug`. Before this change `$fw_debug` was boolean. It is now an integer represented by the constants `fw_kDisabled`, `fw_kEnabled`, `fw_kChatty`, `fw_kVerbose`. This allows debugging messages to be invoked under various degrees of detail. In particular, this is now used to add messages to the tagTracer so that only basic tag calls are logged when debugging is enabled at the base level. Detailed inner workings are logged when `$fw_debug` is set to `fw_kChatty` or `fw_kVerbose`.
- Added new `$fw_criticalLog` to enable certain critical error events to be written to the Lasso SiteAdmin Errors log. The new error handling includes `log_critical` statements when setup, database availability, and config file access errors are encountered.
- Updated all `fwpDate` tags to account for how Lasso 8.5.2 changed the way it treats date casting with empty or non-date-string inputs.

- The fwp_recordData custom type has been updated to allow the abstracted input names from tableModel to be used in query phrase fragments in places of real field names. This now allows the abstracted input names to be used universally in application code.
- The fwp_recordData and fwp_rcrdList custom types have improved options for GROUP BY and ORDER BY.
- Added new fwp_recordLockStore type which stores muliple record locking values. Prior to this update, only one record lock ID per session was retained, making it possible to lose track of some locks under certain error conditions. This update will prevent several scenarios that would require locks to expire before re-gaining access to those records. The fwp_recordData custom type has been updated to make use of this new lockStore.
- Fixed bugs in fwpActAdaptor_mysql that could prevent some date data types from being saved, and changed quoting of some query strings.
- The fwpGUI_checkbox, fwpGUI_listBox, fwpGUI_radioBtn, and fwpGUI_popup value list tags all have a new input to specify the html id attribute. Previously it was forced to be the saem as the name attribute. This update allows them to be different.
- fwpUtil_showVars has improved output formatting for HTTP headers
- fwpStr_randomID has new options to force use of uppercase letters and for inserting hyphens at specified intervals. This enables the generation of IDs like J7KL-TY-OPI8.
- Added new tag fwpGIS_geocoderCoords.
- Added isAlphaNumericHyphen and isAlphaNumericSpace validations to $fw_validator.
- Added drawHiddenExtras and drawHomeFilterInputs methods to fwp_edpView.
- Added -makeMap option to fwpCnfg_splitPairs
- Added new fwpCnfg_splitBlocks and fwpCnfg_splitLabels tags
- Added forwardedFor and forwardedHost instance vars to fwp_client.
- Added $fw_client->ip (as a member tag and instance var). Looks for X-Forwarded-For in the page headers and returns that value if present, else returns the normal client_ip value. In a load-balanced muli-server setup, client_ip is usually going to return the IP address of the load balancing machine that forwarded the request. The X-Forwarded-For value is often added to headers to reflect the original client IP address (just like you'd normally expect from client_ip). You should use $fw_client->ip everywhere that would normally use client_ip to make your applications ready for load-balanced installations. (All internal API code has been updated this way).
- In fwpErr_validator, changed #thisMsgFirstWord to #msgFirstWord to be consistent with the documentation
- In fwpErr_validator, changed hasLabel code to be processed if it starts with a [so MVS code can be used to display multi-language strings for field names
- In fwpErr_validator, update the date|isDate validation code to allow an =euroDate option to force inpout value to be interpreted as a dd/mm/yyyy date.
- The fwp_user->testPswd method has been updated to use the standard validation error strings system.

- In `fwpGui_menuVText.ctag` and `fwpGui_menuHText.ctag` are updated to process menu name fields from the config file if the menu item string starts with a `[` so MVS code can be used to display multi-language strings for menu titles
- adjusted the _atBegin routine to allow default.lasso and index.lasso to be used as “index” files. Either one can be used in any given folder, usage does not have to be one or the other.
- added `useAppStringsDataTables` in `$fw_pageModes` to control whether the appStrings system should look up a requested string in the appstrings data tables or not. This also elminates the version 5.1.x requirement of having the appStrings_en_us table present in a PageBlocks database whether it was needed or not.
- added some improvements to the Debug display output
- cleaned up the auto generated `<label>` html for checkboxes and radio buttons, and added `loop_count` to the auto generated `id` values.

Migrating to 5.2

Prior to replacing your /LassoStartup files, make a backup copy of /LassoStartup/_fwpAPI_init.lgc as you'll need to transfer any adjustments you've made in there to the new version. You may not remember making changes from the defaults, so make a copy and verify that, or you'll be blaming changes in app behavior on incompatibilities instead of misconfiguration.

The vast majority of changes introduced in 5.2 are backwards compatible with 5.1.x code, with some exceptions. First and foremost, all the tags, types, and variables that were deprecated in 5.1 have been eliminated. This is to reduce the extra clutter these created, and to eliminate the processing time they consumed. Refer to the 5.1 Migration Details in the section below, and update all deprecated items in your application code. The majority of these can be done with search and replace.

Variable Name Changes

The following variables (left column) have been replaced by the objects in the right column. These changes have been made with immediate effect. The old variables are not available. (Note that `fw_k` names are constants not variables).

```
$fw_gUrlParamsChar              = fw_kUrlParamsChar
$fw_gServerMode                 = $fw_serverMode
$fw_helpEmail                   = $fw_accountsHelpEmail
$fw_helpPhone                   = $fw_accountsHelpPhone
```

Filename Changes

```
_pageMapper.lgc                 = _pageRouter.lgc
_fwpAPI_initVars.lgc            = _fwpAPI_init.lgc (won't impact app code)
_fwpAPI_initCache.ctyp          = _fwpAPI_initcache.ctyp (won't impact app code)

fwpPage_setCSS.lgc              = _defineCSS.lgc (now optional)
fwpPage_jsScriptsLib.lgc        = _defineJavaScript.lgc (now optional)
fwpPage_templateHead.lgc        = _definePageHead.lgc (now optional)
fwpPage_wrapup.lgc              = _definePageWrapup.lgc (now optional)
```

Updating /_pbLibs/

fwpPage_templateHead.lgc

The file fwpPage_templateHead can usually just be deleted. If you have made modifications to this file to incorporate custom meta tags, unique special JavaScript loading, or other changes, then read through the new docs section on *Using /_pbLibs/* to see how the new options can be used.

If necessary, you can rename the old file to _definePageHead.lgc, and in /LassoStartup/_fwpAPI_init, make sure that the definition for the constant `fw_kUseDefinePageHead` is set to true. (Restart the Site if needed).

fwpPage_wrapup.lgc

The file fwpPage_wrapup can usually just be deleted. If you have made modifications to this file to customize the debug output `topVars` or `clearVars`, these lists are now defined in _initMasters. If you have modified what was section (B) to add vars to the `->addVars` list, you can do that in the new _defineWrapup.lgc file (the example file in the pbStart kit includes that as default code).

fwpPage_setCSS.lgc

Rename your existing file to _defineCSS.lgc, and in /LassoStartup/_fwpAPI_init, make sure that the definition for the constant `fw_kUseDefineCSS` is set to true. (Restart the Site if needed).

fwpPage_jsScriptsLib.lgc

Rename your existing file to _defineJavaScript.lgc, and in /LassoStartup/_fwpAPI_init, make sure that the definition for the constant `fw_kUseDefineJavaScript` is set to true. (Restart the Site if needed). Also read the new docs section *Integrating JavaScript Libraries* for more details on the topic as I think the use of this file has been largely misunderstood in the past.

Updating Master Templates

The new integrated templateLoader cause a few changes to master template files. The good news is that they get simpler.

Previous version templates all had the following first line of code which can now be removed completely:

```
<?lassoscript include: ($fw_sPath->'apiLibs') + 'fwpPage_templateHead.lgc'; ?>
```

Your templates files may also have the following code:

```
[if: $fw_construction || $fw_dataOffline || $fw_maintenance]
    [include: ($fw_sPath->'msthd') + 'alerts.dsp']
[/if]
```

which should be replaced by

```
[fwpPage_loadAlerts]
```

And, lastly, each template file should have ended with a </html> tag. That should be removed as the template loader adds it for you.

So, now, your templates should start and end with the <body> </body> tags.

Updating _initMasters.lgc

New Debug Modes

In Section (E), `$fw_debug` is now used a different way. Prior to 5.2, it was a simple boolean value. This has been replaced by using one of four constants as a value. I find it handy to have all four values assigned like this:

```
    $fw_resetAllCaches     = false;
    $fw_debug              = fw_kVerbose;
    $fw_debug              = fw_kChatty;
    $fw_debug              = fw_kEnabled;
    $fw_debug              = fw_kDisabled;
    $fw_debugLog           = false;
    $fw_debugTimers        = false;
    $fw_debugIPFilter      = '127.0.0.1';
```

Note that all four values are listed. Of course, when run like this, each subsequent assignment will override the former one. So, listed like this, the net result is that `$fw_debug` is disabled. To enabled the chatty mode, I comment out the lower two options, and end up with this.

```
    $fw_resetAllCaches     = false;
    $fw_debug              = fw_kVerbose;
    $fw_debug              = fw_kChatty;
//  $fw_debug              = fw_kEnabled;
//  $fw_debug              = fw_kDisabled;
    $fw_debugLog           = false;
    $fw_debugTimers        = false;
    $fw_debugIPFilter      = '127.0.0.1';
```

The last assignment is now the new value. If this bothers you, then you can certainly use a single line assignment and change the value as needed.

Add the following new variables just below the debug vars section. This is now where you manage the output of vars in the debug display.

```
$fw_debugTopVars      = '';
$fw_debugClearVars    = 'fw_gQueryUser, fw_gQueryPswd, fw_gFilesUser, fw_gFilesPswd, ' +
                        'fw_gUploadUser, fw_gUploadPswd, fw_gPassthruUser, ' +
                        ' fw_gPassthruPswd, fw_gDatabases, fw_gTables, ' +
                        ' fw_gDbTableModels, fw_gHexMap, fw_rollovers';
```

Variable changes

In section (F), cut this declaration:

```
$fw_gServerMode = 'dev';
```

and paste it to section (H) just above `$fw_uploadMIMEs`. Change the name from `fw_gServerMode` to `fw_serverMode` so you have this in section (H) (your value assignments may be different):

```
//  $fw_client->(setVariant:'default');

    $fw_serverMode        = 'http';

    $fw_uploadMIMEs       = array;
    $fw_uploadSizeMax     = 0;
```

In section (H) remove these lines. These are now controlled by application-wide constants in /LassoStartup/_fwpAPI_init.lgc.

```
$fw_pageModes->enableSetCSS;
$fw_pageModes->enableJScripts;
```

New Deployment Management

In Section (E) add the following variables with domain values that make sense for your application. See the document section for this new feature for a detailed explanation.

```
$fw_deploymentHosts->(insert: 'pageblocks.org'   = 'production');
$fw_deploymentHosts->(insert: 'pbtest.dev'       = 'test');
$fw_deploymentHosts->(insert: 'pb.dev'           = 'dev');

$fw_deploymentMode = $fw_deploymentHosts->(find: ($fw_requestPage->'domain'));
```

In conjunction with this change, you'll now want to update the database declarations in section (K) of the _initMasters. Create a `select/case` structure similar to this:

```
select: $fw_deploymentMode;
    case:'production';
        $fw_gDatabases = (map:
            'auth'  = 'pbdemo_production',
            'logs'  = 'pbdemo_production',
            'site'  = 'pbdemo_production');
    case:'test';
        $fw_gDatabases = (map:
            'auth'  = 'pbdemo_test',
            'logs'  = 'pbdemo_test',
            'site'  = 'pbdemo_test');
    case;
        $fw_gDatabases = (map:
            'auth'  = 'pbdemo_dev',
            'logs'  = 'pbdemo_dev',
            'site'  = 'pbdemo_dev');
/select;
```

For this example, the use of production, test, and development (dev) deployments have been created. Based on the domain name the site is running under, the application will use the corresponding database.

These names of production, test, and dev are not mandatory. By declaring domains and labels in `$fw_deploymentHosts` you can name deployments any way you want. Perhaps you want one named beta or demo.

If the sub-domain name needs to be the determining factor, then change this line

```
from:   $fw_deploymentMode = $fw_deploymentHosts->(find: ($fw_requestPage->'domain'));
to:     $fw_deploymentMode = $fw_deploymentHosts->(find: ($fw_requestPage->'subdomain'));
```

and that will allow demo.example.com to run a separate database than www.example.com.

Reorganization of strings_coreErrors_en-us.cnfg

As part of the improved diferentiation between errors intended for developers vs end users, it was necessary to reorganize the internals of the strings_coreErrors_en-us.cnfg file. Error codes that were intended for end users have generally remained the same, but many of the internal errors designed to inform the developer have changed codes and text.

If you have developed, or are using, any alternative language versions of the coreErrors file, you will need to update those files to be congruent to the new organization and text.

New Tags/Types

Database Tags/Types

- fwpActn_encodeLIKE.ctag
- fwpActn_lockStore.ctyp

Config File Tags

- fwpCnfg_splitBlocks.ctag
- fwpCnfg_splitLabels.ctag

Formatting Tags

- fwpDate_isoDateTime
- fwpDate_isValid
- fwpStr_markWebLinks

Nemoy Search Tools

- fwpSrch_nemoyController.ctyp
- fwpSrch_nemoyCriteria.ctyp
- fwpSrch_nemoyQuery.ctyp
- fwpSrch_nemoyAdaptorMySQL.ctyp
- fwpSrch_nemoyAdaptorSqlite.ctyp

- fwpSrch_nreMysqlmod.ctyp
- fwpSrch_nreRobertson.ctyp
- fwpSrch_nreWordcount.ctyp

New Tag Param Names

Several tags have updated parameter names. PageBlocks 5.0 often used a boolean value as a technique to create on/off type parameters. This has been changing with each release to simply having the param present or not. Additionally, the names of many params have been changed to improve readability. In these cases the old param names continue to be supported for backwards compatibility.

One area with several changes is the `fwpCnfg` tags. Several tags supported a `-nosplit` option which has been changed to `-withoutSplit`. For example,

```
fwpCnfg_splitComma: #theseInputs, -withoutSplit;
```

has improved readability over

```
fwpCnfg_splitComma: #theseInputs, -nosplit;
```

While the meaning is certainly not obscure in either case, the former represents the style of readability that PageBlocks APIs are evolving towards.

Consult the details of each tag in the docs or reference database, but the following are some changes that affect several of the config tags:

```
-nosplit       -withoutSplit
-nocache       -withoutCache
-noTrim        -withoutTrim
-nomerge       -withoutMerging
-despace       -removeWhiteSpace or -removeExtraTabs
```

It won’t be necessary to change these params as the old names are still supported, but as you work on sections of code, it may be worth updating them to minimize the impact of future deprecation.

New Page Not Found Handler

The files /_apiLibs/fwpPage_errMngr.lgc and /_apiLibs/fwpPage_pageNotFound.lgc are both obsoleted, and no longer included as of 5.2.0. This requires some minor modification to the file template_filenotfound.dsp, and to all _pageConfig.lgc files.

Updating template_filenotfound.dsp

The demo files included a section of code like this:

```
<div id="pgblock2colwnmain">
    [include:($fw_sPath->'apiLibs') + 'fwpPage_pageNotFound.lgc']
    [include: ($fw_sPath->'apiLibs') + 'fwpErr_mngr.lgc']
    [($fw_error->'errorMsgs')]
</div>
```

The div has no importance, but the three Lasso lines inside it do. These should be changed to the following (only the first two Lasso lines are different):

```
<div id="pgblock2colwnmain">
    [fwpPage_pageNotFound]
    [$fw_error->handleAllErrors]
    [($fw_error->'errorMsgs')]
</div>
```

Updating _pageConfig.lgc

The select/case statements that identified each page will almost always end like this:

```
case;
    include: ($fw_sPath->'apiLibs') + 'fwpPage_pageNotFound.lgc';
```

And, the admin sections would have looked like this:

```
case;
    var:'fw_pgAuthRequired'    = false;
    include: ($fw_sPath->'apiLibs') + 'fwpPage_pageNotFound.lgc';
```

Both of these can be replaced with the following:

```
case:
    fwpPage_pageNotFound;
```

The `fwpPage_pageNotFound` tag takes an optional `-pgTitle` parameter to define the string that will show as the page title in the browser window title bar. It is “Page Not Found” by default.

Version 5.1 Update Highlights

Overview

- New Multi-View Strings (MVS) add multiple languages for error messages, validation messages, application interface strings, and application content including support for displays-specific string versions (desktop, PDA, etc), and client-specific string versions for hosted applications
- New tools and refinements for debugging, troubleshooting, and performance optimization including:
    - an improved structured display of variable contents,
    - a built-in multi-timer object for tracking performance of framework and arbitrary sections of application code, and
    - a new tag tracer to provide a "stack trace" of which section of code have been processed (handy for tracking down conditional paths the code is executing and finding unexpected recursions, etc)
- Improved object oriented design of error handling system
    - the previous system that used files in /_pbLibs/ has been completely replaced with a custom type and the new MVS string system
    - `$fw_error` is still the pimary error management object, and most version 5.0 code should still work as files, tags, and variables have all been supported with backwards compatibility code.
- Added the ability to use fully abstracted URLs where nothing in the URL corresponds to actual disk folders or files (required changing some variable names to be more in line with the separation of URL paths from response page paths).
- Improved object oriented design of page assembly modes, several mode variables have been reorganized as instance vars to collection objects
- Improved nomenclature consistency through variable name changes
- Improved nomenclature semantics by converting several global variables to constants
- All code files and demo databases should now be in UTF-8 character sets
- Global variable caches have been updated to be thread safe
- Several custom types have been updated to prototypes for improved performance
- `->select` queries in fwp_recordData that would fetch the found count when not needed have been optimized
- The `serverType___`, `databaseName___`, and `tableName___` elements of the tableModel_ config files are now [process]ed so they can use vars to avoid duplicating definitions
- General tag optimizations and enhancements.
- The pbComponents system of version 5.0 has been removed. It is replaced by the MVS system that enables multi-language application strings

Migration Details for 5.0 to 5.1

As PageBlocks moves to cleaner use of object oriented code, there's going to be some pain in migrating a 5.0 to a 5.1 site. Many internal changes have been made with backwards compatible preservations, but several of the bigger changes simply make backwards compatibility too complex.

Several variables and some ctags, ctypes, and includes have been deprecated in favor of new objects which consolidate the old variables, tags, and files. Most of the previous variables should all still work as they have been mapped with references into the new objects. Use of previous ctags and files should also still work, but may require replacement of existing framework files with new backwards compatible versions.

Deprecated Variables

The following page variables (left column) have all been replaced by the objects in the right column. Each of these has been mapped by reference so that the old variable should still function as it did in version 5.0, but it would be best to update all occurences of the old variables to the new objects as the deprecated features probably will be removed in the next major release.

```
$fw_errorMsg                 = $fw_error->'errorMsgs' or ->'errorMsg'
$fw_language                 = $fw_client->'language'
$fw_numDecimalChar           = $fw_client->'decimalChar'
$fw_numGroupChar             = $fw_client->'thousandsChar'

$fw_useSetCSS                = $fw_pageModes->'useSetCSS'
$fw_useJScripts              = $fw_pageModes->'useJScripts'
$fw_useJSPerPage             = $fw_pageModes->'useJSPerPage'
$fw_useBlockTemplates        = $fw_pageModes->'useBlockTemplates'
$fw_useAutoErrDisplay        = $fw_pageModes->'useAutoErrorDisplay'
$fw_useAutoRestoreSession    = $fw_pageModes->'useAutoRestoreSession'
$fw_preventCache             = $fw_pageModes->'usePreventCache'
$fw_pgRepeatLogic            = $fw_pageModes->'repeatLogicBlocks'
$fw_pgRepeatAbove            = $fw_pageModes->'repeatAboveBlocks'
$fw_pgRepeatBelow            = $fw_pageModes->'repeatBelowBlocks'

$fw_myURL                    = $fw_requestPage
```

The following page variables have all been eliminated from the framework code, and have been replaced with the new $fw_timer object. If your application code relied on any of these variables, review the updated demo code (namely the pageWrapup and footer files) and documentation for the newer object instance variables.

```
$fw_startTime                                   $fw_timerSQL
$fw_stopTime                                    $fw_timerVLists
$fw_timerConvertForm                            $fw_timerTxtMenus
$fw_timerLoadConfig                             $fw_timerStylize
$fw_timerRollover                               $fw_timerLogicBlocks
$fw_timerPageInitAll                            $fw_timerDisplayBlocks
$fw_timerActn                                   $fw_timerAuth
$fw_timerFMP                                    $fw_timerAuthSession
```

These variables have all been eliminated from the framework code. Some should have been internal use variables only, so application code may break if they were used directly. Those which were for application use are mapped to the new validator object.

```
$fw_stdInputErrorCodes            = $fw_validator->'coreCodes'
$fw_cstmInputErrorCodes           = $fw_validator->'appCodes'
$fw_inputErrorMsgs                = $fw_validator->'errorMsgs'
$fw_cstmInputErrorIdentified
$fw_customInputName
$fw_customInputCode
```

The following global variables have been deprecated in favor of using custom constants. Note that constants are not variables. They're used just like cutstom tag names.

```
$fw_gLassoVersion          = fw_kLassoVersion
$fw_gLassoPlatform         = fw_kLassoPlatform
$fw_gPageExt               = fw_kPageExt
$fw_gLogicExt              = fw_kLogicExt
$fw_gDisplayExt            = fw_kDisplayExt
$fw_gCnfgExt               = fw_kCnfgExt
$fw_gCTagExt               = fw_kCTagExt
$fw_gCTypExt               = fw_kCTypExt
```

The following variables were labeled as though they were not globals, but should have been. The New name on the right should be used.

```
$fw_errorAlertLevel        = $fw_gErrorAlertLevel
$fw_errorAlertEmail        = $fw_gErrorAlertEmail
$fw_errorAlertFrom         = $fw_gErrorAlertFrom
```

Deprecated Tags and Includes

fwp_corralPaths

This ctype has been renamed `fwp_requestPageElements` as part of the refinement of the naming system for URL request handling. This was a mostly internally used ctype, so the name change should not affect any application code.

fwpErr_validateInputs

The tag `fwpErr_validateInputs` has been deprecated in favor of the new `$fw_validator` instance of the `fwp_validator` custom type. Use of the old tag should still work, as it has been rewritten to re-direct validation to the new ctype.

/_pbLib/app_customInputValidators.lgc

This file has been deprecated in favor of the new validator ctype and a new ctype method of adding custom validation codes. This change will break existing code. See the chapter on *Input Validation* for details.

/_pbLibs/fwpErr_mngr.lgc

The entire error management system has been overhauled into the `$fw_error` instance of the `fwp_errorManager` ctype and the use of the new multi-view strings. This file has been rewritten to redirect code to the new error manager, but the files fwpErr_mngrLasso.lgc, fwpErr_mngrFWP.lgc, fwpErr_mngrAppSpecific.lgc have all been eliminated.

While a call to fwpErr_mngr.lgc will still be accepted, any ap-specific error messages and modifications to the standard messages will have to be translated into the new multi-view strings files.

New Tags and Types

fwp_requestURLElements

This new ctype stores the requested URL and provides it in string form and in array form (split at / characters). Includes the domain names as well.

fwp_requestPageElements

This is the `fwp_corralPaths` ctype renamed to better indicate it's purpose when using abstracted URLs. This ctype contains all path elements of the actual page that is handling the current requestURL.

fwpPage_routeURLToPage

Use this new tag inside the root-level _pageRouter.lgc file to redirect processing of an abstract URL to a specific PageBlocks page path for handling.

pbComponents

If your application code used pbComponents, then this is one area that will not be backwards compatible and will require some updating of your code. The new systems are similar, and recoding shouldn't be difficult.

Preparing a Site for Migration

These are things I came across having to change as I migrated a 5.0 site to the 5.1 APIs. Once I got through this, the site pretty much worked.

Updating /LassoStartup

- install new /LassoStartup files from SDK to completely replace the v500 files

Updating _pbLibs

- rename existing /_pbLibs folder to /_pbLibs500
- install new /_pbLibs folder from SDK

- if you've made any modifications to fwpPage_pageNotFound, fwpPage_setCSS, fwpPage_jsSctiptsLib, fwpPage_templateHead, or fwpPage_wrapup, then compare your changes to the new files, and update the new version 510 files

Updating error.lasso

- assuming you've made no modifications to it, replace error.lasso with the one from the new SDK

Updating _initMasters

- rename existing /_pbInit/_initMasters.lgc file to _initMasters500.lgc
- install new _initMasters.lgc file from SDK
- transfer the user info, and any other specific settings you had defined in the v500 version
- be sure to add entries to `$fw_gDbServerTypes` to correlate to the ones existing your `$fw_gDatabases`

Add MVS Strings Tools

- a new table that is required is pbstrings_en_us, even if it is empty, be sure to add this table to your application database
- copy the /site/strings/ folder from the SDK to your application
- transfer custom error messages from /_pbLibs500/fwpErr_mngrAppSpecific.lgc to /site/strings/strings_appErrors_en-us.cnfg (or whatever languages you have in use)

Update Custom Validation Codes

- copy /site/libs/validator_appSite.ctyp from the SDK into your application
- transfer custom validation code from /_pbLibs500/app_customInputValidators.lgc to /site/libs/validator_appSite.ctyp, and move the validation error strings to /site/strings/strings_appValErrors_en-us.cnfg (or whatever languages you have in use)

Update _pageConfig Files

- do a global search & replace to change `'oneFile'` to `'pageConfig'` on all your application source files. (Make sure the /LassoStartup files are not included in this search & replace).
- do a project global search & replace for `$fw_myURL` to `$fw_requestPage` (the old var should remain backwards compatible, but this is a pretty safe S&R to do and be sure you’re updated to the latest spec). The purpose for this change is more evident when you read up on the new abstract URL handler system.
- make sure each section of your _pageConfig files which defines the page element variables is surrounded by a conditional like this:

```
if: ($fw_pageMethod == 'pageConfig');
    select: $fw_requestPage->'name';
        case: bla bla bla
    /select;
/if;
```

Update CSS Files

- validation message classes have been changed to be surrounded by `<span>` instead of `<p>` so, `p.inputerrmsg` will have to be changed to `span.inputerrmsg` in your CSS file(s).

If you find other changes are needed or can be done to smooth the migration, send them to docs@pageblocks.org

Essentials

Introduction

The PageBlocks™ framework is an application development toolkit for building web applications using the Lasso language and server from OmniPilot Software.

Building on a base platform of XHTML, CSS, and Lasso, the PageBlocks web application framework takes the most popular structural methods used by Lasso developers and implements them with a cooperative collection of toolkits to automate many rote tasks in coding web sites. Libraries of custom tags, custom types, and includes are integrated to provide a robust feature set covering security, user authentication and authorization, session management, error management, input validation, user interface controls, logging, internationalization, and more.

The framework takes care of non-value-added elements that must be a part of virtually every web application so development resources can be focused on the value-added elements unique to the application.

The PageBlocks framework provides enough structure to foster rapid site development, yet is flexible enough to adapt to the style of the developer and needs of both monolithic and modular web applications. While the framework and libraries will meet the needs for many types of sites, it certainly isn't assumed that it will meet all needs. Most of the built-in tools have fairly flexible capabilities, and where they don't meet a specific need, they can be used to rapidly build working prototypes with the final code being developed to replace the default framework features as needed.

Using the framework does not force or limit the developer into the specific capabilities of the PageBlocks toolkits. The features of PageBlocks can be used or not used, and in some cases augmented. The intent is that the core components, structure, and standards of the PageBlocks framework will be sufficient for developers to create modular, reusable code to be shared among projects and even among developers. The more the PageBlocks framework standards are used, the more portable the code.

Why Use a Framework?

I view frameworks as having three key goals: minimize application design and development time, increase reliability by using well tested parts, and increase application effectiveness and efficiency through specialization. Just like we probably would not consider writing our own language from scratch to use for our projects, we can also consider not writing the framework-level tools from scratch. Using an established language spares us from the time to author, test, and document the language. It spares us the effort of developing expertise in the reliability, security, and performance issues that language programming faces. This same philosophy can be pushed a level to include the application framework.

By providing an application architecture and numerous utility routines that in theory are tested and documented, a framework frees the application programmer to focus on the unique aspects of the application while relying on existing code to perform the mundane tasks.

A framework also takes advantage of the benefits of specialization. Rarely will any one person become an expert in all areas of what a dynamic web application entails. Sure, we can become quite proficient in multiple disciplines, but let's face it, each of us has our favorite development area. Some developers would rather spend time on design than error handling. Others prefer to work on the data structures and storage systems than CSS. With the use of an independent framework, people who enjoy the low-level coding can focus on improving and expanding the framework (even within specific areas of expertise), and people who enjoy the work above that level can focus on the end user side of the application.

Another advantage to using a third-party framework is long term maintenance and customer support. We're all aware of the classic "custom program" nightmares customers experience. Bob writes an app that works fine, but for one reason or another, Betty gets called upon to see if she can add to or modify the app. Chances are Bob didn't leave behind enough documentation to really get Betty up to speed quickly, and if she's never seen Bob's code before, she's in for a learning curve that the customer probably finds hard to justify the cost of. Using a public framework to build upon gives Betty and, more importantly, your original customer a head start in picking up the application and continuing development. For independent developers, this idea of multiple developers being able to support Bob's work is important.

Feature Highlights

The underlying architecture of the PageBlocks framework was driven by two primary goals: to automate page assembly so development is focused on application-specific components, and to enable a modular structure so specific data management solutions could be reusable with as little recoding as possible.

Automated Page Assembly

Central to the PageBlocks framework is the template-driven page assembly which uses files and/or database sources for content while keeping the application logic and presentation code separated. The framework automatically adapts to both stub file and onefile style techniques of *The Corral Method*, and handles literal, extensionless, and fully abstracted URLs.

Since the days of server side includes, individual pages of web sites have been divided into components (e.g. header, sidebar, body, footer) and assembled dynamically as the page is served. This was originally done to eliminate the inefficiencies of maintaining redundant information used throughout a site such as headers, footers, and menus. However, each server-side include statement had to be explicitly coded with the file name.

Writing explicit references requires time, and in itself is inefficient in terms of maintenance and code reuse. If it became necessary to use two files for a header, every page would need updated to recode the include. The PageBlocks framework eliminates this manual file referencing. Using a structured file naming system and a page assembly engine, files which are components of pages are automatically located and included into a master XHTML template. This enables faster development, more efficient maintenance, and greater long-term reliability.

Modular Solutions Reuse

Many web sites have common information management needs such as articles, news, catalogs, membership rosters, or photo management. With much of the functionality and presentation needs, aside from style, being very similar from site to site, it stands to reason that we'd like as much of the code we develop for a solution to be as reusable as possible.

With the PageBlocks framework, it is possible for each site section to be a modular, self-contained unit. It is not self-sufficient in that it requires access to the core API, but it is self-contained in that all resources unique to that module (logic, media assets, data structures, data views, page layouts, and administration functionality) are stored within a single folder. This folder can be dropped into another PageBlocks site and expected to function with little modification while adopting the site's masthead, styles, and other site-level resources.

User Authentication and Authorization

User access control involves two distinct elements: authentication and authorization. The former involves determining that a user has valid access to protected sections of a web site. The latter involves determining what the user is permitted do within those protected sections. The PageBlocks framework has a very flexible, and easy to use built-in system to handle both. Numerous permissions for numerous users are readily managed.

User Session Management

In conjunction with the authentication system, the PageBlocks framework includes session management to control page-by-page re-authentication while automatically providing user profile data. Application-specific data may be added, removed, or updated in the sessions, and sessions may be propagated by cookie or through forms. Two session storage fields are used with one field being cleared with each login, and the other being sustained between sessions.

Error Management

Every software application will encounter input or code which is invalid. Responding to those occurrences is the job of error management. Whether errors are due to programming weaknesses, poor user input, or unexpected unavailability of resources, the application should respond gracefully by anticipating errors, trapping them, and providing the user with useful feedback to either correct or avoid the problem.

Including a robust and extensible system for error management is one of the more tedious and time-consuming tasks in programming. The PageBlocks framework provides a standardized system for defining, trapping, and responding to errors which reduces the programmer's efforts in this area.

Input Validation

Every web site should be programmed with the expectation that every user input could be malformed, invalid, or even hostile in intent. All input should be validated to assure it meets requirements for format, value range, and other applicable factors. This results not only in improved security, but also improved data integrity which raises the value of the application.

The PageBlocks framework leverages the error-management system to implement an input validation system which reduces the effort required by the programmer to test GET and POST input values and provide user feedback.

Database Action Management

The PageBlocks framework includes prewritten routines to integrate the error management, record locking, field validation, and database statement generation that accompanies common database actions. These routines are structured as shells around the core Lasso inline actions, and thus are called actionShells.

The record-locking system prevents two users from loading the same record into their browsers for editing. This prevents unsyncronized updates in multi-administrator environments. Message delivery to users during the locking process is automatically handled through the error management system.

As of version 5 of the PageBlocks framework, the API can support multiple SQL dialects through an n-tiered adaptor. A MySQL adaptor is included. Database statements for INSERT (add), UPDATE, DELETE, and SELECT (search) are all automatically written in response to form inputs using a simple configuration file for each table. Security measures are taken to eliminate risks from rewritten form hacking.

FileMaker is currently supported through the version 4 API which provides most of the same capabilities, and will be updated to the version 5 spec.

Security

Programming techniques and standards are incorporated in the PageBlocks framework to help ensure the security of the application's programming code and data. This documentation will review the principles followed which can be extended to application development as well.

Logging

The framework provide integral logging of successful and failed authenticated accesses, general application errors, and database editing actions. Logs can be turned on or off, written to file or data table, and can be configured to roll daily, weekly, monthly, or annually if written to files.

User Interface Controls

To support the rapid development goals, the PageBlocks toolkits include several custom tags to automatically create text menus and value lists for forms. A library of graphics for form control is provided, and can be shipped with applications or used for prototyping.

Data Display Formatting

Lasso has many data formatting tags, but in many cases it takes some verbose coding to get simple common formatting of dates and numbers. The PageBlocks framework has several tags to make it easier to display date and numeric data in a variety of common formats.

Image Manipulation

With the addition of ImageMagick and the ExecuChoice PassThru custom tag for Mac OS X, the developer can use PageBlocks custom types and tags to manipulate or acquire specification data from images. While Lasso's own image tags can be used for most circumstances, having a separate library of tags allows the use of updated versions of ImageMagick if needed.

Internationalization (Multi-Language Content)

As of version 5.1, the PageBlocks framework includes several tools for developing multi-language site content. The error management and input validation systems maintain language-specific strings in easy to author text files. Application strings such as page titles, form field labels, instructional paragraphs or any other element can be defined with multiple language versions in text files or in database records. Dynamic content stored in database tables can be pulled as individual records or as page-specific collections (retrieving all strings that belong to a given page with a single query).

The API for retrieving strings abstracts the source of the string (text file vs database) so that coding the use of any string is consistent. The system also caches strings retrieved from both text files and databases to improve performance.

Multi-Media and Multi-Variant Strings

In addition to building multi-language presentations, the string management tools added in version 5.1 enable an application to build media-specific presentations. Strings can be defined as being targeted to a desktop browser, a handheld device, or an aural browser so that content can be tuned to be efficient for those media.

This presentation flexibility is carried one more layer by allowing strings to be defined with arbitrary versions for a purpose meaningful to the application. As described so far, any one string can be represented in multiple languages. Within any given language, the string can have any number of media versions. A string can also have any number of variants per language or per media for any purpose. The primary purpose for including this ability is to enable hosted applications to deliver a customized interface per client by allowing strings which serve the same purpose to be flexible in their actual text. For example, there might a field label of Last Name that a client prefers to be Surname. This difference is not language driven, nor is it media driven. It is an arbitrary variant.

Site Structure

A PageBlocks web site has three fundamental components. First is the application programmer interface (API) which comprises the custom tags, custom types, and include files that provide the core of the framework and toolkit features. Most of the custom tags and custom types are located in the site /LassoStartup folder. Core libraries are found in the /_pbInit and /_pbLibs folder in the virtual host root.

Second are the application's site-wide resources. These are a collection of libraries, media files, configuration files, and other files that may be universal to the entire application. These files are consolidated in the /site/ folder.

Third are the application's module resources. Each module is an arbitrary delineation of pages and functionality which make up a portion of the web application. A module might be a catalog, a web site section for news, support, or product line information, or a photo upload manager. A module is comprised of the page content, the application logic, media files, configuration files, administration pages, and other resources needed to display and manage the content. These files are organized in /{moduleName}/_resources/ in a structure that parallels the /site/ folder structure.

If the application needs a modular architecture, the goal would be to have a module within a single folder which could be dropped into another site and have everything necessary to implement the module's content and functionality. For example, a news release section which provides display and search features might be separated into a module to be reused in multiple web site projects with only minor adjustments for presentation style.

Some applications simply would have no benefit in being designed as discrete modules. For example, a research management site might incorporate budget, planning, approval, and tracking features that don't lend themselves to being modular components. For such sites, folders can be used to segregate logical sections of the web site presentation, but all application libraries, configuration files, media assets and other files can be centralized in the site resources structure. This is what we call a monolithic structure.

XHTML and CSS

The PageBlocks framework and libraries use XHTML 1.0 Transitional using backwards compatible syntax (see http://www.w3.org/TR/xhtml1/). If you have not yet made the transition to XHTML, now is your opportunity. If you have made any effort to write valid HTML 4.01 code, then you should find the transition to XHTML to be fairly simple. Technically, you can stick to HTML 4.01 code of course, however, all PageBlocks tags that write HTML do so using XHTML 1.0T, so it won't validate to an HTML 4.01 doctype. Currently, there are no plans to maintain HTML 4.01 versions of the PageBlocks libraries, though it would be possible to allow for a global configuration var so that tags could alter their output for XHTML or HTML if someone wanted to take that job on.

CSS is used to provide all style definition within any tags that write XHTML. Even if you don't plan to use much CSS for your applications, supporting the PageBlocks tags that require CSS styling is a fairly simple task.

While I do not claim any particular degree of expertise with CSS, I am in my fourth epoch of using it, and believe I am finally on the road to using it "properly" for the most part. If you haven't used CSS, or used it only very little, the pbPadawan site might provide a good starting template for you to develop an understanding of how to use it.

The PageBlocks Framework Roadmap

As with any software, there is always room for improvement. This release of PageBlocks only touches on what is possible, and several improvements and extensions are planned and in early development. The following are the most important areas that will receive attention in coming releases. They are in no particular order at this time, and once other documentation needs have been attended to, a proper roadmap can be developed. I would encourage anyone with interests in these areas to consider getting involved in working on the code:

- continue to review existing systems for opportunities to convert older procedural code into more efficient object oriented code
- include a unit testing framework
- extend the database API to handle some relational queries automatically
- extend the database API for transaction support
- extend built-in tools for editing user permissions as templates and groups
- sessions for non-authenticated visitors
- standardized shopping cart management
- toolkits for the main developer apps

Confused Already?

If you're struggling to understand what is meant by APIs, frameworks, toolkits, architectures, platforms, etc, then take a deep breath, and let's pause for some term definitions.

Architecture

In software, the term architecture has fairly direct parallels to the use of the word as a noun defining the style and method of design and construction of a building. It can be viewed simply as the arrangement of the parts of a system in a particular order or structure.

Software architecture refers to the overall design of a software system, the organization of its components, and usually the communication between its components. Let's look at a web site for a simple example. When designing a site, we could consider the primary organization by the nature of the information presented such as products, support, and company information. This might lead us to organize the site like this:

```
/myCompany
    /products/
        /widgets/
        /sprockets/
    /support/
        /phone/
        /email/
    /company/
        /jobs/
        /investing/
```

We could also consider the functions performed on the information as the primary organization such as search, view, and modify. That might lead us to organize the site like this:

```
/myCompany
    /search/
        /products/
        /support/
        /company/
    /view/
        /products/
        /support/
        /company/
    /modify/
        /products/
        /support/
        /company/
```

The design differences between these options, and the impact that has on how the rest of the files are organized and the functions each page contains represents the architecture of the software. It is the overall design and organization of the pieces of the whole software system at the "big chunk" level.

Frameworks

If architecture is the overall organization, then a framework is the skeleton which lies between the big picture of the architecture, and the details of application-specific design.

A software framework is a collection of general purpose routines that work together to support functions for application-specific capabilities. Software frameworks are developed and optimized for particular applications. Macintosh's Carbon and Cocoa frameworks are great for developing stand-alone desktop programs, but as fancy and cool as they are, they're of no use in developing a web site. The PageBlocks framework is specific to building web sites. Whether it is well suited to your type of projects, you'll have to determine for yourself. There's no universal rule as what those tools need to be, but every dynamic web application needs error management, user authentication, and systems to pull page pieces together, and we can expect a web application framework to provide tools like that.

Toolkits and Libraries

Toolkits and libraries are collections of reusable code such as tags, functions, procedures, object classes and similar units. Each collection is usually related by the type of data it works with or functions it performs, but are general in purpose from an application perspective.

Toolkits differ from frameworks mainly in scope. Toolkits are smaller collections of code focused on a particular area, whereas a framework is comprised of multiple toolkits with code or conventions to tie them together to form cooperative capabilities.

Depending on the semantics you follow, toolkits have a higher functional role than simple libraries. Where libraries are collections of independently functioning components (such as Lasso's array or string tags), toolkits tend to have components that cooperate to form a system (such as Lasso's file tags, TCP tags, or database tags).

Application Programmer Interface (API)

If toolkits are defined and grouped by what they do, the API is the standards and methods concerned with how the toolkits are used by the programmer--the interface used by the application programmer.

In assembly language, parameters to functions are passed on a stack. The parameters must be placed on the stack in a specific order, and there is a specified number of parameters that must be placed even if the values are null. This might look something like this:

```
pha #100       // the number 100
pha $BBFA      // the memory location BBFA
_doStuff       // a macro to a library function
```

This is very different than using a lasso tag which looks like this:

```
[string_replace:
    (field:'someText'),
    -find='\r',
    -replace='<br />']
```

When developing custom routines in Lasso, there's a number of options as to how we might pass parameters. We could write a custom tag in which the method used would look like a native Lasso tag like the one above.

```
[draw_menu:
    -configFile='mymenu',
    -separator=' | ',
    -leftcap='[ ',
    -rightcap=' ]']
```

Or we could use variables and an include:

```
[var:
    'configFile'='mymenu',
    'separator'=' | ',
    'leftcap'='[ ',
    'rightcap'=' ]']
[include: '/libs/draw_menu.inc']
```

The style and methods for coding the toolkits and framework define the API. The PageBlocks API for the most part uses custom tags and custom types, a number of general application configuration variables, and a few includes.

Site Structure

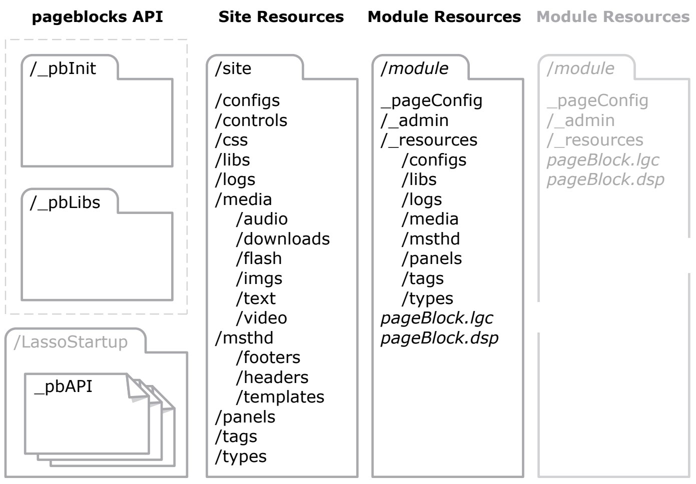
The PageBlocks framework is designed to be a foundation for information-oriented, dynamically-driven websites which manage data of several topical areas. Websites for companies and organizations and hosted applications for intranets that depend on databases and forms would be prime candidate applications. With this emphasis in mind, the native site structure, that is, the file organization, of the PageBlocks framework results from two of its primary goals: modularity and code reusability.

Data-driven web sites tend to be organized with themed groups of information into website sections such as company info, products, support, news, etc. where the feature and data sets tend to be similar from site to site. Having created and tested a given data management feature set, a developer should be able to reuse the majority of the code to reproduce that information organization and feature set on future projects. Not that the design and layout would necessarily be reused, but the data structures, logic algorithms, and human interface functions could be useful starting points to save development time and increase application reliability.

A good example is the classic news section of a website. Whether informal or following the conventions of the news industry, the information structure and presentation of news has nearly identical needs regardless of site design. A developer having created a news management section for one website should be able to drag and drop that whole collection of code and resources into another site and require little to no modification except to adapt to the site style. To accomplish that, all resources used for that site section need to be grouped together. To allow the site resources to all work together regardless of domain or directory, the program code needs a consistent method to reference local module resources and site resources. A module is a section of a site contained within a folder at the site root level. The term module was chosen to support or emphasize the modular capabilities of the PageBlocks framework.

Therefore, the PageBlocks framework is built around the concepts and use of site and module resources. Code and resources which are general purpose to the site as a whole can be developed as global, or site, resources. Code and resources which tend to belong together to present a specific class of information such as news, a product catalog, support services, a download library, etc. can be developed as modules to be reused and modified as needed across multiple projects.

For example, rather than place images from news, products and other site sections together in one /site/images/ folder, resources are grouped under module folders, then sub-divided by resource type.



The illustration above shows the overall site structure of a PageBlocks web application. This structure is more or less the same for modular and monolithic applications. The difference lies in where the developer locates the various file resources.

So, our news example would yield a path of /news/_resources/images/. Files specific to the /news/ section of the site would be organized in the /news/ folder. Whereas traditional website organization tends to require the developer to scrounge through multiple folders all over the site to pick out the pieces that make up a news section to a web site, the PageBlocks module structure allows resources specific to the site section to be quickly located, copied, and reused in multiple projects.

This structure also enables a simple API for accessing the files in these folders. All organization folders are defined as variables, but rather than having dozens of variables, there are two custom types ("ctype"), one for site resources and one for module resources with instance variable names identical to the folder name. The site ctype is `$fw_sPath` and the module ctype is `$fw_mPath`. The definition of the module ctype is modified on the fly with a prefix of the current module folder name. This means the same variable can be used for every module to reach resources local to that module. With site and module resource structures being uniform, remembering path names is easier. The final format of the variable is therefore `$fw_sPath->'libs'` and `$fw_mPath->'libs'` which is used like `[include: ($fw_sPath->'libs') + 'myFileName.lgc']` to access files.

The bulk of the custom tags and types are installed in /LassoStartup at the main or site level (usually the site level). Either the source code versions or a compiled LassoApp can be used. There are also a number of standard include libraries which are located in the virtual host root.

The PageBlocks framework website structure views all file resources as being in one of three categories: API resources, site level resources, and module level resources. The API resources are the prewritten libraries of tags, types, includes, and such. Site-level resources are a mixture of framework- and programmer-provided files that are universal to the whole site (includes, config files, templates, graphics etc.), and module resources are framework and programmer provided files that are specific to a single module of the website. Typical folder structure and files associated with these three resource areas are shown below.

Root Level Site Structure of Core PageBlocks API Features

```
    _indexRedirect.lgc - centralized redirection for URLs with no files
    _pageConfig.lgc    - root level page configuration declarations
    _pageRouter.lgc    - optional abstracted URL handler
    /_pbInit           - site & app environment initialization
    /_pbLibs           - customizable framework includes and libs
    error.lasso        - Lasso error trapping
    default.lasso      - default site file (or index.lasso)
    /site              - site wide resources
    /siteMngr          - prebuilt site-wide admin entry point
    /siteStrings       - prebuilt site-wide admin for appStrings/pageStrings content
    /siteUsers         - prebuilt site-wide admin of authenticated users
```

Site Resources: these folders are for site-wide resources.

```
/site
    /configs       - config files for PageBlocks API tags/types
    /controls      - graphics for user interface buttons/feedback
    /css           - CSS files
    /libs          - includes and libraries (logic)
    /logs          - error, auth, db access logs
    /media         - media assets (label & organize as you see fit)
        /audio
        /downloads
        /flash
        /imgs
        /text
        /video
    /msthd         - masthead graphics and includes
    /panels        - reusable HTML snippets (display)
    /tags          - custom tags (those not in LassoStartup)
    /types         - custom types (those not in LassoStartup)
```

Masthead Resources

```
/site/msthd/
    /headers       - template header includes
    /footers       - template footer includes
    /templates     - master template files
    /msthdImgs     - graphics for masthead imgs
```

Typical Module Structure

```
/example               - typical site module folder
    /_admin            - administration pages for this module
    _pageConfig.lgc    - local module page configuration declarations
    /_resources        - resources specific to this module's pages
        /configs       - config files for PageBlocks API tags/types
        /controls      - graphics for user interface buttons
        /css           - CSS files
        /libs          - includes and libraries (logic)
        /media         - media assets (label & organize as you see fit)
        /msthd         - masthead templates, graphics, and includes
        /panels        - reusable HTML snippets (display)
        /tags          - custom tags (those not in LassoStartup)
        /types         - custom types (those not in LassoStartup)
```

As the structure implies, the /_admin folder holds the "back end" administrator files for that module if any are needed. The various admin sections for the whole site are tied together by a navigation menu. The structure of the demo site shows how this can be done. Of course, this system may not be suited to every site, and certainly isn't a required methodology. Likewise the menu for admin navigation can be of any style. The text link menu of the demo app is simply an easy one to create and use for an example.

The _initX and _pageConfig Files

There is a hierarchy of configuration files to handle site, module, and page level configuration parameters. When the Lasso Site is started, a file named _fwpAPI_initVars is loaded which declares global variables used by the framework. This is done only once. Next, the ctag `fwpPage_init` is called for every page. This tag is the main page assembly process in which additional files are called to define the environment.

The first file included by `fwpPage_init` is _initMasters which provides master config vars. Next is a _initProfile which provides the ability to tailor the environment's configuration to a user's or domain's specific preference. This is followed by _initCustom where application-specific config declarations are made. Finally, the _pageConfig file is loaded to define page-specific vars like the template to be used, page title, and other such things (which could be declared further up the chain as well).

The details of each of these files is discussed further in the Page Assembly chapter.

API Resource Naming Standards

(Refer to the online reference sections at www.pageblocks.org/refc/ for additional information.)

Variables

Variables prefixed with `fw_g` are framework level globals universal to the whole site. They generally should not be changed outside of their declaration in _fwpAPI_initVars or _initMasters. They should not be changed at all based on any conditional code as changing them to suit one user will affect all users.

Variables prefixed with `fw_` but without a g designation are general-purpose, page-level variables typically for use in conjunction with the framework's custom tags, types, and includes. Most can be, and are expected to be, modified as needed by the application.

Application variables should either not be prefixed, or prefixed with something other than those used by the PageBlocks framework (which includes `fw_` and `edp_`).

Custom Tags, Includes, Libraries

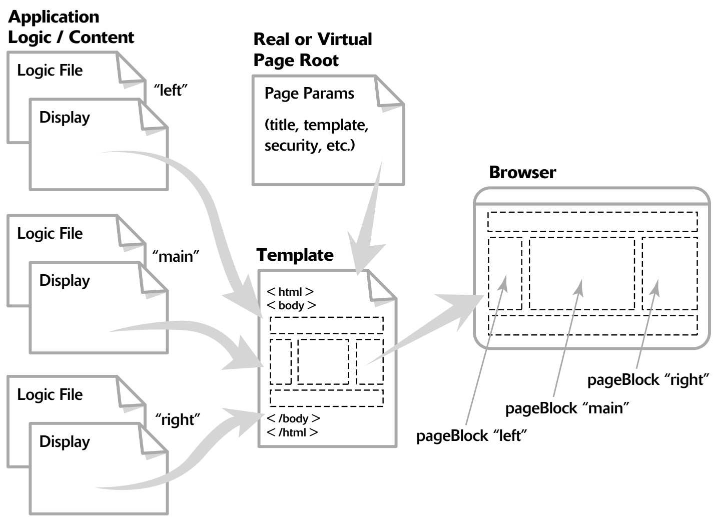
All tags, types, and libraries begin with `fwp` to denote their membership in the PageBlocks framework and API. Following the `fwp` letters will be an API toolkit identifier and underscore such as `GUI_`, `Actn_`, and `Err_` to identify which programmer's toolkit the tag or file belongs to. Some includes begin with an underscore to put them at the top of file lists and emphasize their significant role within the framework.

Custom Types

All custom types begin with `fwp_`. An identifier for a toolkit is not included primarily to help identify types from tags.

Page Assembly

The PageBlocks framework is essentially a Corral Method framework. As such, it uses HTML page layout templates and individual content blocks for template components, and it uses a nesting configuration hierarchy. The PageBlocks framework goes further than the original Corral Method description in providing a standardized structure for sites to support both monolithic and modular organization, and it supports what has been termed stub file (files with integral page config declarations) and onefile (centralized page config declarations) page configuration systems. Additionally, extensioned, extensionless and even fully abstracted URLs are easily implemented.



The page configuration method does not have to be declared. It is set up to be automatically detected, and an internal variable is set so subsequent page assembly logic decisions are automatically controlled. Any given page in any given directory can be written to use either method. The methods may be mixed within any folder.

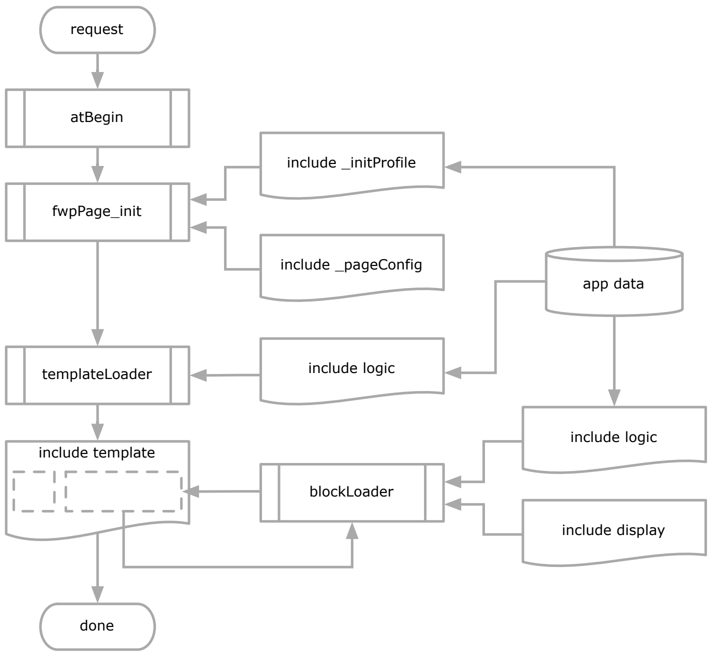
Extensionless and abstracted page URLs are achieved through Lasso's `atBegin` capabilities, so there's very little HTTP server configuration to do. A couple of custom tags also help modify URL GET form appearance and automatically reconstitute the data on the response page.

In looking at the many details and options for the page assembly process, let's first go over some general topics, then take a detailed look at the sequence of events in building a page, and finally a look at URL abstraction tehniques. We'll be using the demo fileset pbPadawan which you should probably have handy to review during the discussions below.

Configuration Nesting

The PageBlocks framework uses a nested environment configuration system to handle site, module, and page level configuration parameters. When the Lasso Site is started, a file named _fwpAPI_initVars from /LassoStartup is loaded that declares global variables used by the framework. This is done only once during the loading of the /LassoStartup files.

Next, the ctag `fwpPage_init` is called for every page. This tag is the main page assembly engine which goes through several steps to convert action_params to vars, integrate the nested configs, activate user authentication, and include the page template, and more. This custom tag is the closest relative to the Corral siteConfig include. However, the ctag only controls logic. Configuration declarations that would also be in a siteConfig file are made elsewhere. The



fwpPage_init tag can be called from a number of possible page starting points which we'll look at in detail further on.

The first config file included by fwpPage_init is _initMasters. This file has a couple of functions. First, it contains duplicate declarations of most of the vars initialized in _fwpAPI_initVars. The purpose is to provide an easy-to-access method to override the values of the vars. During development, these globals may need experimented with or altered to suit testing or debugging. Once an application's high level cnofiguration is stable, the assignments in this file can be commented out, and the values can be moved to the _fwpAPI_initVars file to eliminate the duplicate declarations, (though it isn't necessary to do so).

The second purpose of the _initMasters files is to declare the page-level, site-specific environment vars. These vars define the default values which can then be changed on a per-page basis if needed. For example, in your application, maybe most pages can be cached by proxy servers, but certain admin pages should not. The framework has a built-in handler to prevent caching which simply needs to be enabled or disabled. You would set the default condition to disable cache prevention in _initMasters, but declare it enabled for those certain admin pages. There are several of these page level variables and controls that are in _initMasters (which would not get moved to _fwpAPI_initVars as the globals can be).

The next configuration declarations are done in _initProfile which provides the ability to tailor the environment's configuration to a user's or domain's specific preference. If the master config var of $fw_gUseInitProfiles is true, then _initProfile is included. Otherwise, it does not need to exist. _initProfile is a developer defined routine. The sample file sets all include a template to show what could be done, but it is up to the developer to provide the code that acquires either the user's or domain's profile data for whatever purpose it is being used for, whether that be to set application-specific vars perhaps for unique clients of a shared application or for user personal preferences, or to determine overrides for framework vars.

The next file included by fwpPage_init, _initCustom, is purely for the declaration of application-specific setup variables. This is where you would define the custom vars needed to control your application. If you've previously had a siteConfig with var declarations, this is where they would go in a PageBlocks framework application. Whether or not this file is included is controlled by the environment var $fw_gUseInitCustom. Again, if this var is false, the file does not need to exist.

Finally, a file named _pageConfig, if present, is loaded to define page-specific vars like the template to be used, page title, whether authentication is required to access the page, and other such things. There is no control var for _pageConfig. If it is present, it is loaded. There can be a _pageConfig in the main site root, and also one in each major site module. So, if you had major site sections including /company, /products, and /support, each of those can, and normally would, have their own _pageConfig file.

The _pageConfig file used to be two separate files specific to module configuration and page configuration, but in an effort to reduce the number of framework includes per page, the two were merged.

Now, the first part of _pageConfig handles configuration details universal to a site module. This could be used to declare a header and footer to use throughout a /company module which would different from the ones used in a /products module. The second part of _pageConfig deals with specific page needs (i.e. title, template, etc).

Page and File Naming Conventions

Every page is structured by a page template. The sample pbPadawan site uses CSS to control virtually all styling, but there is no edict or restriction as to the use of XHTML/CSS or any combination of the two. However, tags which generate HTML code will currently generate code to XHTML Transitional standards. This primarily impacts the closing of tags with a /> when necessary. (It would be possible to update these tags to optionally generate HTML 4 or XHTML code if there's enough demand for it, but I'll wait to see what response is first).

Regardless of the standards used, the sectioned display regions of the page layout template are called pageblocks, and each pageblock has a programmer-defined, arbitrary reference name such as main, left, news, topten, advert, or anything else to identify it. The template is populated by using Lasso to include files into those pageblock regions. To accommodate separation of logic and display and flexibility in page assembly, each pageblock allows for a variety of possible files to be loaded. Before considering those various files, let's first look at the content loading process.

To enable the automation in page assembly that the PageBlocks framework offers, pages and their component files follow a structured naming convention. This eliminates the need to declare variables to identify content files, helps to group related page files together in file lists, and enables an easy to manage separation of logic and display.

Content in a PageBlocks site is generally (though not strictly) organized in a relatively flat structure where each folder is a section of common subject pages (i.e. company, products, support). Each of those folders is a PageBlocks application module. Each folder name (or a shortened version of it) is typically the leading component of all page names in that module. Again, this isn't a strict requirement, but rather a useful convention in having individual files identified as to where they belong within the whole site.

Next, the file name contains an identifier for the page. Thus, we end up with a file starting with module_pageName such as prod_intro or support_contacts. For classic Corral stub files, we simply end with our primary page extension such as .lasso and end up with prod_intro.lasso or support_contacts.lasso. People will tend to think this pattern is required, so I repeat that it is not. You can have pages named whatever you want, but I have found this pattern to save me a lot of grief when it comes to communicating with customers, and knowing what file I'm editing when I have many open files during development, etc.

For pageblock content and logic files (whether using stub or onefile configuration), we add a component that identifies the block name. Now we have files starting with something like prod_intro_main and support_contacts_left. To enable the separation of logic and display, we use different file extensions. Logic files are named .lgc, and display files are named .dsp. The actual extensions are configurable in _fwpAPI_initVars, but the two must have different extensions.

Most pages will have their page-specific config declarations made in _pageConfig files. For these pages, there are no actual files that would correlate directly to the URL. So if you have a URL of /products/prod_intro, there is no /products/prod_intro.lasso file. However, there will be logic (.lgc) and content (.dsp) files as needed. Therefore, a site might have files as listed below. Note that some blocks have both logic and display files, some have just display files. We'll address the issue of pages that do not use _pageConfig a little further on.

```
/about/
    _pageConfig.lgc
    about_genl_left.dsp
    about_genl_main.lgc
    about_genl_main.dsp
    about_genl_right.lgc
    about_genl_right.dsp
    about_history_left.dsp
    about_history_main.dsp
    about_history_right.dsp
/products/
    _pageConfig.lgc
    prod_intro_left.dsp
    prod_intro_main.lgc
    prod_intro_main.dsp
    prod_intro_right.lgc
    prod_intro_right.dsp
    /widgets/
        widgets_intro_left.dsp
        widgets_intro_main.dsp
/support/
    _pageConfig.lgc
    supp_warranty_left.dsp
    supp_warranty_main.dsp
    supp_warranty_right.dsp
```

Types of Page Logic and Content Files

The core of the PageBlocks framework is a page assembly process which automatically seeks for a variety of possible sources of logic and content to fill the page's master template. Sources include various format files and a general purpose content management database.

Let's look first at the format file options. As mentioned in the previous section, format files are automatically located based on their name. Assuming a page master template with left, main, and right pageBlocks, the following files are the common options for containing logic and content:

```
_pageConfig.lgc          - declares page title, template, and more

pagename.js              - page-specific javascript file
pageName.lgc             - page-wide logic file (results usable in all blocks)
pageName_left.lgc        - left pageblock logic
pageName_left.dsp        - left pageblock display
pageName_main.lgc        - main pageblock logic
pageName_main.dsp         - main pageblock display
pageName_right.lgc        - right pageblock logic
pageName_right.dsp        - right pageblock display
```

The page-wide logic file pageName.lgc is a file comprised of non-output Lasso code. It is loaded even before the template is, and can therefore be used to dynamically alter anything on the page. Normally, the pageblock logic files should contain code that affects the display of only that block's display file. If you need something that can impact the whole page, then the page-wide logic file is for that.

Repeating Content Blocks

You may be able to determine at this point that the use of pageblock display files for each page can pose a problem if you want the same content to show up on all pages. You could create a separate include, and then make pageName_left.dsp files for each page to include that separate file, but that's not very elegant.

To provide a solution for repeating content at the module level, we have the repeatXX files. These can be included on every page in the module for the specified pageblock. They can either provide content in addition to the standard pageblock files, or be used simply to provide content common to all pages.

```
repeatLogic_left.lgc      - a logic file for the left block of every page
repeatAbove_left.dsp      - a display file inserted above the standard pageblock file
repeatBelow_left.dsp      - a display file inserted below the standard pageblock file
```

Enabling Repeating Blocks

Providing this option has its own downside though. Normally, the page assembly process simply checks if a file exists and includes it if it does. This is fine for templates and pages where most of the time there's a file for each block in the template. However, in the case of these repeating content files, that can add up to a lot of extra `file_exists` tests per page when this feature is not used.

To counteract the potential for all those extra `file_exists` tests, the repeating blocks feature is normally disabled. It can be enabled using the following command either in _initMasters if most pages in the site will make use of it, or per _pageConfig file for specific modules, or even per page config declaration if only some pages in a module need it:

```
$fw_pageModes->enableRepeatBlocks;
```

Once enabled, the system also has to be instructed as to which repeating blocks will be used (the goal of minimizing the number of needless `file_exists` calls seems worth a little manual setup over automation in this case). There are three page mode commands correlating to the three repeating blocks. Each of these is assigned an array containing the block names that will use the repeating files. So, if we declare the following:

```
$fw_pageModes->(setRepeatLogic:(array:'left', 'main'));
$fw_pageModes->(setRepeatAbove:(array:'left'));
$fw_pageModes->(setRepeatBelow:(array:'left', 'right'));
```

we are stating that there are files of the following names to be used:

```
repeatLogic_left.lgc
repeatLogic_main.lgc
repeatAbove_left.dsp
repeatBelow_left.dsp
repeatBelow_right.dsp
```

If certain pages should not use the repeating content, then the `->disableRepeatBlocks` command can be used in the _pageConfig case statement for that page, or conditional code within the file can be used. This method of repeating content integration is used in the pbJedi demo Features section to include the left menu. There's more info about `$fw_pageModes` coming below.

Block Templates

Another scenario where having pageBlocks per page doesn't work well is when you have mostly dynamic content for numerous pages from a database. In this case it is more efficient to have a single set of pageblock files populated by the database where each block file acts as a mini template for just that one block.

To allow for this, the PageBlocks framework provides the blockTemplate files option. Again, assuming a classic 3 column template, we could have these files:

```
blockTemplate_main.lgc    - logic code for a display block template
blockTemplate_main.dsp    - a display block template used for multiple pages
blockTemplate_left.lgc    - logic code for a display block template
blockTemplate_left.dsp    - a display block template used for multiple pages
blockTemplate_right.lgc   - logic code for a display block template
blockTemplate_right.dsp   - a display block template used for multiple pages
```

This capability is normally disabled to prevent needless `file_exists` calls. To enable the use of the block templates for any one module or any one page, we use a pageModes command:

```
$fw_pageModes->enableBlockTemplates;
```

When block templates is enabled, the page assembly process tests for the existence of a file named blockTemplate_xxxx where xxxx is the name of the block being loaded.

This technique is used in the pbJedi demo Features section to show the main page information, and the resource links on the right side of the Features pages.

Internationalized Block Files

The framework supports several ways to display content in multiple languages. One of the methods is by using one display file as a template with each string coming from a multi-language source. Much like a display file can be used as a template for multiple news stories, a display file can also use a language value like `en-us` as a key to retrieve language-specific text.

However, there are cases where rather than pull from dynamic sources, it may be more desirable to simply load from multiple hard coded versions of a display block in the various languages needed. The page assembly process needs to know which display block to use, and language-specific display blocks need to be named uniquely.

If this method is preferred, the developer can name pageblocks with a language identification component as follows:

```
pageName_left.lgc         - left pageblock logic block
pageName_left_en.dsp      - left pageblock display block for English
pageName_left_fr.lgc      - left pageblock display block for French
```

If this language-specific identifier is to be used, the developer must enable the pageModes command:

```
$fw_pageModes->enableMultiLanguageBlocks;
```

This can be set to control whether the whole application uses this mode or whether it is page specific by using the command at a given level in the page configuration nesting (i.e. the _initX or _pageConfig files).

Self-configured Pages and Asynchronous Requests

The preferred method for page assembly would be where _pageConfig is used to centralize page configuration details so there is no need for a file (often called a stub file) used just to declare the page config details. However, there are cases when a single, self-contained file for a page is useful—stand-alone utility pages, or help systems, or other such uses where it is preferred not to entangle a page's code into the PageBlocks framework environment. Such a file would contain the entire page layout and content plus potentially a small amount of stand-alone logic. In the PageBlocks context, these are self-configured pages because all the configuration details are embedded and not located in _pageConfig. They may continue to use all the other PageBlocks features, they're just independent when it comes to configuration declarations.

Speaking of pages, in the Lasso world, because it is a web-tailored language, we tend to think of all URL requests equating to a "page," but that request does not necessarily have to have any HTML output that would result in what we'd call a page. Lasso "pages" are often coded for periodic housecleaning and triggered by the event scheduler, or to fulfill asynchronous requests from Ajax-enabled pages by processing a small dose of logic to return data, or even to fulfill a web services request. In these cases, the paradigm of a "page" doesn't really make sense, so the PageBlocks framework should allow these uses of Lasso code to also break from the page paradigm as well while still allowing the use of all the major framework features like the database abstraction ctypes, the user authentication system, logging, etc. Indeed, as with self-configuring pages, the framework allows this by allowing URL requests to bypass the _pageConfig system but still use all the other framework features.

When creating self-configured pages or asyncronous request scripts that don't need authentication, but otherwise use PageBlocks resources (vars, tags, types), call `fwpPage_init` at the top of the file. When creating self-configured pages or asyncronous request scripts that do need authentication and authorization from the `$fw_user` object, write the `fwpPage_init` tag like this:

```
fwpPage_init:
    -fw_pgAuthRequired = true,
    -fw_pgPrivilege    = 'fw_user->prvlg.invoices_update';
```

When creating self-configured pages that will invoke a PageBlocks template, call `fwpPage_init` at the top of the file and pass page declaration variables as parameters, like this:

```
$__HTML_REPLY__ = (fwpPage_init:
    -fw_pgTitle    = 'pageblocks : StubFile Basics',
    -fw_pgTemplate = '2colwn');
```

Every parameter passed to `fwpPage_init` is turned into a page variable of the same name. These vars are created at the same time that _pageConfig would otherwise be included, and therefore, the parameters, in effect, become the page config declarations.

The meanings of the parameters and variables used in the examples above will be discussed in more detail later on. Just remember to refer backto this section for examples of dealing with pages and requests that are independent of _pageConfig.

Combining Page Assembly Options

One very important thing to keep in mind is that these various options are not exclusive to each other. You are not limited to using one method or another. They can be mixed up on a page by page basis by toggling the pageModes in _pageConfig, and they can even be combined in some ways within a single page. For example, the pbJedi demo Features section uses the blockTemplates for the main and right blocks of all pages except the Intro page which uses basic static content block files. All Features pages use the repeatAbove option to include the left menu.

PageModes Control

In the sections above, the `$fw_pageModes` object was introduced to enable certain pageblocks options. The $fw_pageModes object includes several mode switches for the page assembly process. Each switch has an enable and disable command which control the boolean value of an instance variable. Below are the available options:

PageModes Instance Variables

- `->'useJSPerPage'` (boolean) : when true, a conditional in /_pbLibs/fwpPage_templateHead.lgc will include a file named pageName.js in the template `<head>` section.
- `->'usePageStrings'` (boolean) : when true, the templateLoader will call the `$fw_pageStrings->getStringsForPage` tag to retrieve records from the multi-view string data table(s) to populate the page. Refer to fwp_pageStrings.
- `->'useBlockTemplates'` (boolean) : when true, the page assembly blockLoader will load block files beginning with blockTemplate and including the name of the block such as blockTemplate_main.lgc and blockTemplate_main.dsp.
- `->'useRepeatBlocks'` (boolean) : when true, the page assembly blockLoader will load block files beginning with repeatLogic, repeatAbove, and repeatBelow and the name of the block such as repeatLogic_left.lgc and repeatAbove_main.dsp. Which blocks are to include repeating files is determined by an array of block names in the instance vars '`repeatLogicBlocks`', '`repeatAboveBlocks`', and '`repeatBelowBlocks`'.

- `->'useMultiLanguageBlocks'` (boolean) : when true, the page assembly blockLoader will load display block files which include the current value of `$fw_client->'language'` such as pageName_main_en-us.dsp. This is not applied to logic files, as they should be written to be language independent or language inclusive.
- `->'useAppStringsDataTables'` (boolean) : when true, enables the appStrings system to look up a requested string in appStrings data tables ifa string was not found from the text files.
- `->'useAutoErrorDisplay'` (boolean) : when true, the page assembly blockLoader will include the current contents of `$fw_error->'errorMsg'` at the end of the current display block. This is useful for quick prototyping, but may or may not be desirable for the final error display management of an application.
- `->'usePreventCache'` (boolean) : when true, the `fwpPage_init` process and the /_pbLibs/fwpPage_templateHead.lgc file will add headers and meta tags to prevent the caching of the page by HTTP servers.
- `->'repeatLogicBlocks'` (array) : the list of block names for which a repeatLogic_ file should be loaded if `'useRepeatBlocks'` is set to true.
- `->'repeatAboveBlocks'` (array) : the list of block names for which a repeatAbove_ file should be loaded if `'useRepeatBlocks'` is set to true.
- `->'repeatBelowBlocks'` (array) : the list of block names for which a repeatBelow_ file should be loaded if `'useRepeatBlocks'` is set to true.

PageModes Member Tags

- `->enableJScripts` = sets `'useJScripts'` to true.
- `->disableJScripts` = sets `'useJScripts'` to false.
- `->enableJSPerPage` = sets `'useJSPerPage'` to true.
- `->disableJSPerPage` = sets `'useJSPerPage'` to false.
- `->enablePageStrings` = sets `'usePageStrings'` to true.
- `->disablePageStrings` = sets `'usePageStrings'` to false.
- `->enableBlockTemplates` = sets `'useBlockTemplates'` to true.
- `->disableBlockTemplates` = sets `'useBlockTemplates'` to false.
- `->enableRepeatBlocks` = sets `'useRepeatBlocks'` to true.
- `->disableRepeatBlocks` = sets `'useRepeatBlocks'` to false.
- `->enableMultiLanguageBlocks` = sets `'useMultiLanguageBlocks'` to true.
- `->disableMultiLanguageBlocks` = sets `'useMultiLanguageBlocks'` to false.
- `->enableAutoErrorDisplay` = sets `'useAutoErrorDisplay'` to true.
- `->disableAutoErrorDisplay` = sets `'useAutoErrorDisplay'` to false.
- `->enablePreventCache` = sets `'usePreventCache'` to true.
- `->disablePreventCache` = sets `'usePreventCache'` to false.
- `->setRepeatLogic` = sets `'repeatLogicBlocks'` to the assigned array.

- `->setRepeatAbove` = sets `'repeatAboveBlocks'` to the assigned array.
- `->setRepeatBelow` = sets `'repeatBelowBlocks'` to the assigned array.

Site-Wide Options

There are some configuration adjustments that are similar in their flexibilities to pageModes, but are made on a Site-wide basis (that is, the values are common to all pages running from a given Lasso Site using the PageBlocks framework). These options tend to be for capabilities that are either used or not, and don't require a page-specific overrride.

The major capabilities of this type are defined as constants rather than variables, and can be found in /LassoStartup/_fwpAPI_init.lgc.

- `fw_kUseDefineCSS` = enables the including of /_pbLibs/_defineCSS.lgc on each page
- `fw_kUseDefineJavaScript` = enables the including of /_pbLibs/_defineJavaScript.lgc on each page
- `fw_kUseDefinePageHead` = enables the including of /_pbLibs/_definePageHead.lgc on each page
- `fw_kUseDefinePageWrapup` = enables the including of /_pbLibs/_definePageWrapup.lgc on each page
- `fw_kUsePageStringsAutoDegrade` = if a called string from pageStrings cannot be found in the exact language, media, and variant specified, enabling this switch cause pageStrings to first look for a `default` variant of the specified media, then an `all` media before reporting the string cannot be found. (Check the *Multi View Strings* chapter for more info).
- `fw_kUseAppStringsAutoDegrade` = if a called string from appStrings cannot be found from the standard text config files, enabling this switch causes appStrings to search the appStrings data table before reporting the string cannot be found. (Check the *Multi View Strings* chapter for more info).

TemplateLoader, and BlockLoader

There are two internal engines which combine to automate the page assembly process: the templateLoader and blockLoader. The templateLoader is responsible for loading components that have impact on the whole page. The blockLoader focuses on individual blocks within the page template.

Following is the exact order in which all these possible files are loaded to build a page. Normally, `atBegin` will be the first thing processed, but there could be an index file (which should include the _indexRedirect.lasso file), or a self-configured or asynchronous request. Any one of these three will end up calling the `fwpPage_init` ctag. From there, things are mostly the same:

```
- fwpPage_init
    - action_params are converted to vars
    - $fw_requestURL is created
    - $fw_requestPage is created
    - standard $fw_ environment vars and objects are created
    - _initMasters.lgc
    - _initProfile.lgc
    - $fw_sPath and $fw_mPath are created
    - files in /site/tags and /site/types are loaded
    - logged in user session vars are restored
```

```
        - _initCustom.lgc
        - _pageConfig.lgc
        - $fw_requestPage is possibly updated if redirected (which also updates
          $fw_sPath and $fw_mPath)
        - logged in user object is reconstituted if necessary
        - the templateLoader ctag is now invoked:
            - $fw_pageStrings is populated if needed
            - page-wide logic file is loaded if available
            - a master <head> is processed (or included from _definePageHead.lgc)
            - master template is loaded
            - a master page wrapup is processed
```

For each pageblock in the template, the `fwpPage_loadBlock` ctag is invoked where the loading sequence for logic and content files is:

```
    - repeatLogic .lgc file
    - standard .lgc file
    - blockTemplate .lgc file
```

Errors accumulated in `$fw_error` are checked at this point, and if necessary, the error manager is called to transfer errors into `$fw_blockLogicError`. The error manager then resets `$fw_error` for the display files. The blockLoader continues with the display files in this order:

```
    - repeatAbove .dsp file
    - standard .dsp file
    - blockTemplate .dsp file
    - repeatBelow .dsp file
```

Errors are checked again and the error manager called if needed. Errors which occurred in the logic files will be in `$fw_blockLogicError`, and errors from the display files will still be in `$fw_error`. All error messages will have been concatenated in `$fw_error->'errorMsgs'`.

Any combination of these files can be used to create a number of unique page building processes as described in previous sections. Other than knowing the above loading order, it is not necessary to understand the internal details of the loaders. While the process as a whole might be labeled "complex," it is already written, and is all hidden behind a couple of simple custom tags and file naming conventions which makes it easy to implement and work with.

Using /_pbLibs/ Options

There are four processes which the developer can customize to influence the overall template and block assembly process. These are organized differently as of version 5.2.

_definePageHead.lgc

The templateLoader automatically builds the head of each page. There's many individual variables and control switches that go into creating the head (found in the options of _initMasters and _fwpAPI_init), and these should be adequate for the majority of applications. However, should there need to be a radically different approach to the `<head>` for a particular site, then enabling `fw_kUseDefinePageHead` will cause the templateLoader to use the developer provided code in _definePageHead.lgc. This must be used for every page in the site. It's an all-or-nothing switch. The pbStart file set includes a complete example which effectively duplicates what goes on in the templateLoader. Use this as a starting point for creating your own version.

_definePageWrapup.lgc

After a template has been loaded and populated by all pageBlocks, the templateLoader performs a few wrapup tasks automatically. These tasks include logging the page access if the authLog is enabled, storing session variables, and generating the developer debug output.

Even though the master template files include a `</body>` tag, the templateLoader actually strips that tag off to provide the opportunity to add display such as the debug output, then adds the `</body>` tag back in.

If `fw_kUseDefinePageWrapup` is enabled, the _definePageWrapup.lgc file is included as the first step in the wrapup process. This allows the developer to perform actions on every page, or generate output for every page. However, the output should generally be for diagnostic or monitoring purposes.

_defineCSS.lgc

The templateLoader will automatically load two CSS files if it finds them in `($fw_sPath->'css')`: pb_apiStyles.css and style_main.css. The former contains default definitions for all styles used by PageBlocks HTML generators. You can start with the default file from pbStart, and modify it as needed. The latter file is simply a standardized name to load which contains the developer's own styles.

If the application's CSS needs are straightforward enough to be covered by these files, then the templateLoader takes care of this on its own. If the application has more complex needs, perhaps various stylesheets are needed for specific browsers, or specific skins, then there will be some conditional code to determine which sheets to include on any given page. That code belongs in _defineCSS.lgc.

If fw_kUseDefineCSS is set to true, then the template loader will assume that _defineCSS.lgc determines all CSS needs by either writing styles directly in the head, or creating the `<link>` tags needed. For example, the file might looks like this:

```
//  include the core styles

'<link rel="stylesheet" type="text/css" href="/site/css/pb_apiStyles.css" />\r';
'<link rel="stylesheet" type="text/css" href="/site/css/styles_main.css" />\r';

//  load a skin based on current site section

if: (($fw_requestPage->'path') >> '_admin') || (($fw_requestPage->'path') >> 'siteMngr');
'<link rel="stylesheet" type="text/css" href="/site/css/styles_admin.css" />\r';
/if;
```

_defineJavaScript.lgc

If fw_kUseDefineJavaScript is set to true, the templateLoader will include the _defineJavaScript.lgc file into the page `<head>`. This does not mean that all the application's JavaScript has to reside in this file, but rather this file is used to hold the logic that determines where to load the JavaScript libraries from. Whether this is one library from the /site/ or some from the /site/ and some from the module /_resources/ is up to the developer. If certain modules or pages have unique requirements, the conditional code for that can be written in this file.

Essentially, this file becomes the central processor for determines what code to include directly in the head. For example:

```
<script type="text/javascript"><!--

function viewHelp(topic) {
    aWindow=window.open(topic,"",
"toolbar=no,status=no,width=300,height=400,scrollbars=yes,resizable=yes");
}

// -->
</script>

'<script src="/javascripts/mainlib.js" type="text/javascript"></script> \r';
'<script src="/javascripts/prototype.js" type="text/javascript"></script> \r';

if: ($fw_myURL->'module') >> 'getquotes';
'<script src="/javascripts/quotecalculator.js" type="text/javascript"></script> \r';
/if;
```

From the above example, we see three things: a) that the file can contain script code directly, b) that the file can load other libraries, and c) that the file can be used to make conditional loads of libraries based on any factor.

If the apps needs are complex based on the site modules, consider using a `select` tree similar to how _pageConfig works (except using module names instead of page names).

Integrating JavaScript Libraries

PageBlocks attempts to be agnostic and flexible when it comes to JavaScript integration. Everyone has a preferred way to do this, and there are multiple needs to be accomodated. So, PageBlocks provides a few simple rules to yield the most flexibility.

First, as of version 5.2, there's a new standard location for JavaScript libraries in /site/js/ and {module}/_resources/js/. These are accessible programmatically with `($fw_sPath->'js')` and `($fw_mPath->'js')` respectively. The page assembly process makes no assumption about the presence of default libraries in these locations.

Second, the templateLoader uses the status of `$fw_pageModes->'useJSPerPage'` to determine if a page-specific library should be loaded for the given page. The file would have the name of the page and a .js extension (see *Types of Page Loic and Content Files*). This switch can be turned on/off as needed page-by-page in the _pageConfig file.

Third, if fw_kUseDefineJavaScript is set to true, the templateLoader will include the _defineJavaScript.lgc file into the page `<head>`. See the *_defineJavaScript.lgc* section above.

Index Files and Pageless URLs

In a well-crafted web application we want users to be able to use the URL as a legitimate navigation system for the web site. This requires that users be able to enter /products/ without a literal page name.

Normally, this requires an index.lasso or index.html file which then does a redirect to the file that is the default response. Authoring all of those unique index files is tedious and the redirect itself is a waste. A more universal solution to that is often handled through Apache mod-rewrite or other URL rewriting tools in the HTTP server. However, many developers either don't have the skillset to do that, or simply prefer to have application logic like that existing within the application code and not in the HTTP server configuration.

The default documented Apache setup for a PageBlocks installation will result in a request for the index file, but the framework does offer a way to avoid an HTTP redirect, and to simplify the coding to point the request to the correct page. (The demo files use default.lasso as I think that's a more representative name for the purpose, but I call them index files as that's more universally recognizable. Either index.lasso or default.lasso can be used.)

For each disk directory where you'd expect to get a /folder/ URL request, an index.lasso or default.lasso file must exist. However, the content of that file is the same regardles of its location. It is simply:

```
[include:'/_indexRedirect.lasso']
```

Which of course leads us to the _indexRedirect file. This file is a centralized location for the whole application to define what to do with a URL that ends in a slash. There's a few var declarations at the top, but essentially each URL ending in a slash is a case block with a declaration of the actual page path to be called. A case block looks like this:

```
case: '/';
    $fw_indexRedirect = '/home';

case: '/sitemngr/';
    $fw_indexRedirect = '/sitemngr/mngr_login';

case: '/demo/';
    $fw_indexRedirect = '/demo/demos';
```

The case line identifies the URL, and the `$fw_indexRedirect` var declares the page to respond with. While not as fancy as using mod_rewrite, it keeps all URL handling code inside the application source code, avoids a redirect, and is as least centralized for easier maintenance.

This technique is only for handling requests that map to real folders on the disk drive. You would not use the _indexRedirect file to handle abstracted URLs that just happen to end in a slash. Those are handled by abstract URL processing in the root level _pageConfig file. See the *Abstract URL Handling* section for more information.

Abstract URL Handling

For the most part, writing a typical literal file per URL is pretty straight forward. You can create a .lasso file with all the logic and display code intermingled just like newbie 101 code, and the framework will serve it just fine. Being a little more advanced, you can create a .lasso file which calls `fwpPage_init` and declares some config vars, and then slice up your code into logic and display components, and you're still using a literal translation of a URL into a file.

To make your app a little more modern, you can abolish the use of extensions like .lasso in your URLs, and move all your page config declarations into _pageConfig. Now we no longer have a URL that equates to a literal file. The framework handles everything to map that URL to an internal process that responds to that URL. This is the begining of abstracting URLs which allows us to generate URLs that are interpreted in order to build a response, but this URL still requires that the .dsp and .lgc pieces of the page responding to that URL exist in folders with the same names as the URL directories. So for a URL of /products/widgets/W100-24c there still has to be a /products folder with a /widgets folder in it.

To be ultra flexible in how URLs are used to request content from the web application, we sometimes want the URL to be completely independent of the source code file structure. This is a fully abstracted URL, and as of version 5.1, the PageBlocks framework has included the ability to do this more easily than before.

By definition, an abstract URL has no physical disk file association that can be assumed. So, the PageBlocks page assembly process has to first identify a place to decipher the URL to at least figure out which page to respond with. That is done in the _pageRouter file located in the web root. To make this work, the first element of the URL (each element is separated by a slash) cannot be the name of a real folder at the web root of the source code. To use an abstract URL that starts with /product/ there should *not* be a folder named /product/ in the source code at the root level. When the page assembly engine detects that there is no real folder that matches the first element of a URL (and therefore no folder-specific _pageConfig file to include), it falls back to including the root level _pageRouter file.

Inside the _pageRouter file, the developer can write whatever code is necessary to analyze the URL elements and determine which page should be processed in response. Let's say that a URL request for /catalog/2006/widgets/W100-24c/retail is received, and we want that to be handled by the page that actually exists in source code at `/products/widgetCatalog`. Our scheme is that any request where the first element is /catalog/ be sent to `/products/` and that the third element determines the product line. The other elements will just become parameters for a database query. The framework will have already created a `$fw_requestURL` object for you. It has several instance vars, the most meaningful of which is `$fw_requestURL->'elements'` which is an array of items delimited by the slash. We can read that array in our request handling code like the example code below:

```
var:'thisRequestRoot' = ($fw_requestURL-'elements')->get:1;

select: $thisRequestRoot;
    case:'catalog';

        var:'thisRequestModule' = 'products';
        var:'thisRequestProduct' = ($fw_requestURL-'elements')->get:3;
```

```
          fwpPage_routeURLToPage:
             '/' + $thisRequestModule + '/' +
             $thisRequestProduct + 'Catalog';

      case:'x'
      case:'xx'
   /select;
```

Now, everything in that code above except for the `fwpPage_routeURLToPage` tag and the `$fw_requestURL->'elements'` instance var is an arbitrary method for reading a URL and deciding what to do with it. The developer has to come up with the URL organization, and the code to decipher it (called a request handler). What the framework provides is the array in `$fw_requestURL->'elements'` and the `fwpPage_routeURLToPage` tag to forward the URL to the correct page. This forwarding does not generate an HTTP redirect, it is a simple in-framework routing.

Handling Abstract URLs that End With Slashes

URLs that end with slashes where each element matches a real folder has to be handled by the _indexRedirect file, but abstract URLs will get passed to _pageRouter to be processed by the developer's request handler. If the presence of a trailing slash is significant, it can be tested for by using

```
   if: ($fw_requestURL->endsWithSlash);
      // yes it does
      // do some special stuff
```

Deployment Modes

To help with the multiple instances that any one site may have for the purposes of development, testing, demonstrations, and production, PageBlocks 5.2 has added some simple mechanisms to enable deployment modes.

What Are Deployment Modes?

There are plenty of scenarios where running an instance of your application means it should be using data X instead of data Y, or the page display should/should not show certain things. To help make this differences ahppen automatically, we have "deployment modes." Deployment modes enable a site to use different data sources or behave in different ways depending on what domain it is being run from.

Perhaps your site is a hosted application which offers an online demo. For that demo you want a non-production-data database to be used. So, when running as www.mysite.com, you want the application to use mydb_production, and when the user jumps to the application running at www.mysitedemo.com you want the exact same instance of the application code to use mydb_demo.

Similarly, when you run a test site or a beta site, you will want those instances to use different databases. You may even want certain debug or status data, or a user feedback form to appear at the end of each page when run in these modes only.

Perhaps your site normally runs as https, but during development you use http, and you need various fully qualified redirects and links to automatically adapt. You can use the PageBlocks control variable `$fw_serverMode` as the protocol prefix to URLS like this:

```
$fw_serverMode + '://' ($fw_requestPage->'domain') + '/folder/file'
```

but something still has to change `$fw_serverMode` from http to https. The deployment modes can handle that.

Declaring Deployment Modes

A deployment mode has two attributes: a label and a URL. The label is any arbitrary name you want to use to identify a deployment mode. This might be production, test, demo, beta, development, or whatever. The URL is the full domain name or just a subdomain name corresponding to that deployment mode. Production might run at www.mysite.com while demo might run at www.mysitedemo.com, or your local development runs at www.mysite.dev. In this case we'd be using full URLs as the differentiator for deployment modes. It might be the case that the same domain name is used, but production runs at www.mysite.com, and beta runs at beta.mysite.com in which case we need to differentiate by subdomain. Let's assume we have full separate full domains.

In section (E) of the file _initMasters.lgc we insert key-value pairs into the variable `$fw_deploymentHosts` like this:

```
$fw_deploymentHosts->(insert: 'mysite.com'       = 'production');
$fw_deploymentHosts->(insert: 'mysitebeta.com'   = 'beta');
$fw_deploymentHosts->(insert: 'my.dev'           = 'dev');
```

That is followed by this line which uses PB's $fw_requestPage object to determine which domain the site is running under, looks thath up in the deployment hosts, and assigns the deplyment mode variable the corresponding value.

```
$fw_deploymentMode = $fw_deploymentHosts->(find: ($fw_requestPage->'domain'));
```

Using Deployment Mode

From this point on you can use `$fw_deploymentMode` to modify the application's behavior. The first place we'll do that is telling the application which database to use by manipulating the assigments for `$fw_gDatabases` with a structure like this:

```
select: $fw_deploymentMode;
    case:'production';
        $fw_gDatabases = (map:
            'auth'  = 'mydb_production',
            'logs'  = 'mydb_production',
            'site'  = 'mydb_production');
    case:'beta';
        $fw_gDatabases = (map:
            'auth'  = 'mydb_beta',
            'logs'  = 'mydb_beta',
            'site'  = 'mydb_beta');
    case;
        $fw_gDatabases = (map:
            'auth'  = 'mydb_dev',
            'logs'  = 'mydb_dev',
            'site'  = 'mydb_dev');
/select;
```

For the http or https server mode example above, we'd use a simple if or ternary statement like this:

```
$fw_deploymentMode == 'production'
    ? $fw_serverMode = 'https'
    | $fw_serverMode = 'http';
```

Database Migrations Management

Losing track of changes to your data schema can be problematic. If you're maintaining instances for test and production, or have several developers to keep synchronized, well, that's just plain asking for trouble.

Migrations is a methodology for keeping track of each change in a database, helping multiple instances of databases stay up to date with schema changes, and placing schema management under version control.

The dbMigrator framework in PageBlocks provides Lasso developers with a toolset to integrate migrations schema management directly within projects.

Why Use Migrations?

If you're using version control, you have the ability to save small incremental stages of development. As you introduce changes to your code you may find yourself stuck with a change that seems to have broken things unexpectedly. Whether you need to roll back to regroup, or just to compare previous code to current code, version control proves to be a very useful capability. Additionally, version control can be used to keep multiple developers coordinated.

Migrations is method for adding these kinds of capabilities to your database schema. As you evolve an application, its data schema usually evolves as well. Quite often that evolution is performed by making changes directly to the database via command line. Let's say you've made a couple dozen changes to the schema while developing the next update to your app. When it is time to apply those same changes to a test site or to the production database, are you going to remember ever change you made?

Migrations is way to maintain a functional history of schema changes which can be applied to multiple instances of databases. Once you've created a migration for your development copy, you can run that migration against the test and production database and know you'll end up with exact copies.

What Are Migrations?

Quite simply, a migration is a simple code file which makes the schema changes to your database programmatically. By scripting these updates rather than applying them directly through a database utility, the changes are captured for reuse and for a self documenting history. Each change script is kept in a file a with a sequential number. Each number represents the current version number of the database.

A utility like dbMigrator organizes these scripts, provides the tools to execute them, and most importantly provides a framework in which to author these scripts to make them easy to write and also make them largely database agnostic so writing scripts is almost identical regardless of the database engine.

Working with dbMigrator

The dbMigrator module in PageBlocks organizes, executes, and displays the results of the developer's migrations.

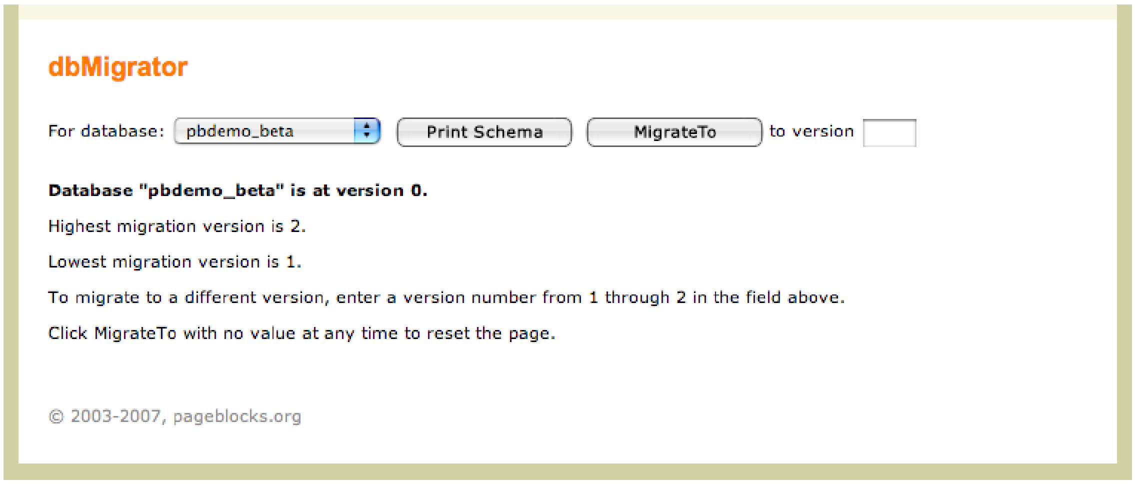
The dbMigrator utility ("dbm") runs right in the web browser on the developer's local machine or on a test server on the LAN. It can also run on a remote server, and uses the `$fw_debugIPFilter` var to determine who has access to run the dbm tools.

The application has 3 simple functions: migrateTo, printSchema, and switching the target database. MigrateTo will update or rollback the database to the version specified. PrintSchema will grab the schema for each table in the database and print an HTML table suitable for printouts. A popup menu provides a simple way to allow the actions to be targeted to the various database instances. Using the utility comes down to a simple form on a web page.

Setting up dbMigrator

Using whatever tool you prefer, create two empty databases. You can delete them later, but for now, let's use these for experimenting. Following deployment conventions, you'd want to make one of them suffixed with _dev, and one _production.

- open migrator_main.lgc an edit the var `$dbmDatabases` to be an array of names for the databases to be used project. By "databases" here, we mean the different names that the various instance of the application will use (i.e. for test, beta, production)
- use Lasso's SiteAdmin to enable all databases, and make sure your Lasso group is allowed to access each of the database and each table in the database. dbm uses the universal `$fw_gQueryUser` and `$fw_gQueryPswd` variables for access
- try the migrator URL (example: www.pb.dev/_dbm/)



Database "pbdemo_beta" is at version 0.

Highest migration version is 2.

Lowest migration version is 1.

To migrate to a different version, enter a version number from 1 through 2 in the field above.

Click MigrateTo with no value at any time to reset the page.

© 2003-2007, pageblocks.org

- you should see a screen similar to this now
- Enter 1 in the field at the far right, and click the MigrateTo button
- This should have sucessfully run the script 001 which creates the _schemaInfo table. This table must be in every database you use with dbMigrator

- Now run MigrateTo 2. This will create a set of standard empty PageBlocks tabls used for authenticated users and logging.
- Click the Print Schema button, and you should get a formatted display of table schemas for the whole database.
- To see a rollback in action, with one the databases at version 2, use MigrateTo 1 and see it modifiedback to version 1 (use Print Schema to see the details).

Writing Migration Scripts

Start by looking at the sample scripts in the /_dbm/migrations/ folder. You'll notice that each script begins with a sequential number of 001_, 002_, 003_, etc. Each script must be numbered like this. This is what allows dbMigrator know which script will upgrade the database to or roll it back to that version number. The rest of the file name is inconsequential and can be anything you need to help you know what that script does. The file must end with a .ctyp extension.

Each script is a simple ctype definition that inherits the base type of `fwp_dbmCommands`. Each script has member tags `->update` and `->rollback`. Inside the `->update` tag you write the steps to migrate the schema to the next version you need. The `->rollback` tag is used to restore the database to the state prior to the `->update` tag's steps. Whatever you do in `update`, you need to write a step to counteract it in `rollback`. The example scripts should help make this process clear.

Below are a basic set uof update and rollback scripts.

```
define_tag:'update';
    self->(addField:
        -table  = 'myNewTable',
        -name      = 'payee',
        -type      = 'string',
        -size      = '12');

    self->(modifyField:
        -table  = 'myNewTable',
        -name              = 'checkAmt',
        -type              = 'decimal',
        -precision = '8',
        -scale        = '2',
        -unsigned      = true);
/define_tag;

define_tag:'rollback';
    self->(removeField:
        -table  = 'myNewTable',
        -name      = 'payee');

    self->(modifyField:
        -table  = 'myNewTable',
        -name              = 'checkAmt',
        -type              = 'decimal',
        -precision = '6',
        -scale        = '2',
        -unsigned      = true);
/define_tag;
```

In the update tag we added a field and we modified a field. In the rollback tag you can see that reversing the addField is a simple removeField command. For the modified field, what was it before we modified it? You'll have to know what that is, and use a modifyField command to set it back to what it was.

You'll notice that each command requires a `table` parameter. That gets tedious having to repeat it, so if each command in the tag applies to the same table, you can eliminate the redundant parameters and use the following line one time before using any commands:

```
(self->'tblName')='myNewTable';
```

Schema Modification Commands

dbMigrator is written to mostly be database agnostic. Most script details you'll write will not be database engine specific. However, there will be times when you'll need to issue database-specific commands, and that can be done. the following commands are what this version of dbMigrator currently supports.

addField

```
-required = 'name'
-optional = 'table'
-optional = 'type'
-optional = 'size'
-optional = 'null'
-optional = 'default'
-optional = 'unsigned'
-optional = 'precision'
-optional = 'scale'
-optional = 'after'
```

- name is the field name
- table is the table name
- type is the column data type. See the data type reference below. If not specified, it will be set to string.
- size is the column width. It is required for all fields except the decimal type of fields which use precision and scale instead.
- null is set to True or False to indicate NULL or NOT NULL settings. If not specified, it is set to NOT NULL.
- default is used to set an explicit default value. If not specified a standard value will be used.
- unsigned is set to True or False for numeric data types only.
- precision sets the column width for numeric fields with fractional values. Precision is the 8 out of the 8,2 setting value.
- scale sets the number of fractional value columns. Scale is the 2 out of the 8,2 setting value.

- after is the fieldname of the field that the new field should be inserted after. If not specified the new field will be added based on the bahavior of the database engine (usually the end of the table)

```
    self->(addField:
        -table     = 'myNewTable',
        -name      = 'payee',
        -type      = 'string',
        -size      = '12',
        -null      = false,
        -after     = 'checkNumber');

    self->(addField:
        -table     = 'myNewTable',
        -name      = 'checkAmt',
        -type      = 'decimal',
        -precision = '8',
        -scale     = '2',
        -unsigned  = true);
```

renameField

Uses the same options as addField except that instead of name, we have

```
    -required = 'fromName'
    -required = 'toName'
```

With MySQL (and perhaps others?), a rename field requires that all definition elements be repeated, so in effect you have to declare all the same things you would if you were using addField.

```
    self->(renameField:
            -table        = 'myNewTable',
            -fromName     = 'checkAmt',
            -toName       = 'checkAmount',
            -type             = 'decimal',
            -precision = '8',
            -scale        = '2',
            -unsigned     = true);
```

removeField

```
    -required = 'name'
    -optional = 'table'

    self->(removeField:
            -table  = 'myNewTable',
            -name      = 'payee');
```

modifyField

Uses the same options as addField (except after is not used). With MySQL (and perhaps others?), a modify field requires that all definition elements be specified, so in effect you have to declare all the same things you would if you were using addField.

```
self->(modifyField:
        -table        = 'myNewTable',
        -name             = 'checkAmt',
        -type             = 'decimal',
        -precision = '6',
        -scale        = '2',
        -unsigned     = true);
```

addTable

In MySQL, the create table statement must include the definition of at least one field. So, the options for the addTable command are the same as as addField except after is not used.

```
self->(addTable:
        -table  = 'secondNewTable',
        -name       = 'rcrdNo',
        -type       = 'string',
        -size       = '16');
```

renameTable

```
-required = 'fromName'
-required = 'toName'

self->(renameTable:
        -fromName   = 'usrs',
        -toName         = 'users');
```

removeTable

```
-required = 'name'

self->(removeTable:
        -name   = 'usrs');
```

addIndex

```
-optional = 'table'
-required = 'name'
-optional = 'type'
-required = 'field'
-optional = 'length'
```

For MySQL the only required elements are the index name and the field to be indexed. The type option accepts the string values of UNIQUE, FULLTEXT, or SPATIAL.

```
self->(addIndex:
    -table     = 'myNewTable',
    -name         = 'rcrdNo',
    -field     = 'rcrdNo');
```

removeIndex

```
-optional = 'table'
-required = 'name'

self->(removeIndex:
    -table     = 'myNewTable',
    -name         = 'rcrdNo');
```

execute

The execute command will run any SQL statement provided as a single string. This allows migrations to include features not yet abstracted, or that are too cumbersome to abstract.

If you only ever work with one database engine, you might prefer to do all your migrations with execute commands. There’s nothing wrong with that. The abstracted commands are beneficial for when you work with multiple engines so you don’t have to use engine-specific syntax.

dbm Data Type Mappings for MySQL

The values on the left are valid values for the `type` parameter in fields commands like `addField`. the right column is how the map to MySQL column types. (As new adaptors are created, the options on the left will stay constant). If a value is used that is not in the left column, then that value will be passed as is in the query. This allows the migrations to pass values that the migrator doesn't yet abstract.

string	varchar
fixedString	char
radioBtns	enum
checkBoxes	set
text	text
tinytext	tinytext
smalltext	smalltext
mediumtext	mediumtext
largetext	longtext
blob	blob
tinyblob	tinyblob
smallblob	smallblob
mediumblob	mediumblob
largeblob	longblob
binary	blob
int	int
integer	int
tinyInt	tinyint
smallInt	smallint
mediumInt	mediumint
largeInt	bigint
decimal	decimal
float	float
double	double
date	date
time	time
year	year
datetime	datetime
timestamp	timestamp

Inside the dbMigrator Framework

The bulk of dbMigrator's innards can be found in the /LassoStartup/ folder in the collection of files starting with fwpDbm. To read sequentially through the code, start with the file /_dbm/migrator.lgc which creates the data needed for the database list popup menu, and creates the $fw_dbMigrator object which is the main logic controller. Follow that with /_dbm/migrator.dsp.

Application Source Code

- `fwp_dbmController`—the main controller which manages the update and rollback processes and a few other general tasks
- `fwp_dbmCommands`—the base cass for migration scripts. This class holds the abstracted fields and table modification commands.
- `fwp_dbmAdaptorNNNN`—provides the translation of the commands into database specfic queries.
- `fwp_dbmSchema`—extracts schema details for printouts

Multi-view Strings (MVS)

Whether or not you plan to use multiple languages or deliver to multiple media in your application, you need to read this chapter as the multi-view string system applies to all uses of error and validation message and strings even for a single language site delivered to desktop browsers.

MVS Concepts

The PageBlocks framework includes integrated systems for handling string data in multiple languages, multiple media, and multiple variants at the framework, application, and page content levels. Collectively, the ability to display a string in more than one language (e.g. english, klingon), and/or more than one media (e.g. desktop browser, pda), and/or more than one variant (i.e. individualized for clients of a hosted application) is called a multi-view string.

To enable this capability, the developer cannot write literal strings for display in the application code. Instead, strings must be referred to programmatically. They must be pulled from a data structure much like a specific element from a map or an instance variable from a custom type is pulled.

The framework itself has been completely updated in version 5.1 so that internal systems such as error handling and validation error messages are multi-view ready. Additionally, tools available to the developer enable the creation of an internationalized or multi-media application interface.

For applications that will use a single language and a single media, the developer must still understand some multi-view string basics as the error and validation messages are delivered using this system.

Dedicated Purpose Controllers

To provide flexibility in the multi-viewstring system, there are four distinct string categories: error message strings, validation message strings, application strings, and page content strings. Each category uses a separate controller object so the programming context of the string usage is easier to work with. Error message strings are generally delivered automatically by the error management system, but are otherwise accessed through the error manager controller (`$fw_error`). Strings are associated by error code. Similarly, the validation message strings are delivered using the standard interface to the validation controller (`$fw_validator`), and are associated by their validation codes. Application strings and page content strings have dedicated controllers (`$fw_appStrings` and `$fw_pageStrings`) used to extract the desired string data on demand. These strings are associated by arbitrary names.

Mode Control Variables

To control which strings are extracted from the pools of defined strings, there are three instance variables in the `$fw_client` object:`->language`,`->media`, and`->variant` are used to set the respective default values for the current client page.

For single-view applications (those with a single language and single media), these values are set once in the _initMasters, and that's the end of it. For multi-view applications (be it multi-language and/or multi-media), these values are to be changed by the application using whatever method the developer sees fit.

The four MVS controllers will all use these default values automatically so that they do not have to be explicitly set with each string request. However, if there is ever a need to request a string that is different than the mode defaults, then each of the values can be independently included as parameters as needed.

MVS Configuration

Strings for error and validation messages are defined in text files. Application strings can be defined in text files and/or in an SQL data table. Page strings are defined in an SQL data table (there's no value to using text files for this particular string type).

For the three string types definable with text files, there is a file dedicated to each string type: errors, validation errors, and application strings. Each string type is also separated by language so that any one file is specific to a language. Finally, following the typical overriding nature of much of PageBlocks, files can also be designed at the site or module levels.

There are a set of core files (stored in /LassoStartup/_strings/) that are an integral part of the framework code. For example, the framework error `5010` has an error message associated with it. Likewise, the validation code `an` for alphanumeric has a default message associated with it. These default messages are defined in config files that are a core component of the framework and should not be changed.

To add custom application-specific string definitions, the developer creates string config files to place in /site/strings/ or module /_resources/strings/. The core string definitions can be overriden by simply creating a definition in the app-specific files with the same string name as used in the corresponding core file.

The core files include:

- strings_coreErrors_en-us.cnfg — standard error messages
- strings_coreStrings_en-us.cnfg — standard application strings needed by the framework
- strings_coreValErrors_en-us.cnfg — standard validation error messages

To add application-specific strings, create the following files:

- strings_appErrors_en-us.cnfg — custom error messages
- strings_appStrings_en-us.cnfg — custom application strings
- strings_appValErrors_en-us.cnfg — custom validation error messages

As you can see by the file names, the language identifier is a part of the file name, and uses the conventional codes in use for identifying HTML document language[1]. In the case of the core files, it is hoped that developers with the need for a given language will translate the core files and offer them back to the PageBlocks community so they can be distributed.

With the potential for each string to have multiple versions, there was the potential for the text config file format to be quite complex. If you're developing multi-view applications, then

[1] http://www.oasis-open.org/cover/iso639a.html

some complexity is an expected part of the task, but if you're developing a single-view application, then you want simplicity. The format for these configuration files allows the developer to define exactly what needs defined and no more.

MVS Error String Configuration Blocks

The most basic format of a string definition is like this where, in the error strings config file, the `stringName` is the error code number:

```
{{stringName:
msg___      the string to display
}}
```

However, there are several options to allow for some efficiency in the string definitions and to control some optional features of the error management system on an error-specific basis. First, is the ability to define reusable string content. Let's assume we have an error message like this:

```
{{5010:
msg___      <h3 class="errorttl">User Login Not Recognized</h3>
msg___      <p>That name and password are not recognized for access.</p>
msg___      <p><a href="javascript:window.history.go(-1);">Return to prev page</a>.</p>
}}
```

The line that provides a link to return to the prev page is something that we might want to include in several error messages. It's rather redundant to have to explicitly define it over and over, and it's inefficient if we decide to alter that line and have to alter multiple copies of it. So, lines like this can be broken out and defined on their own like this:

```
{{showPrevPageLink:
msg___      <p><a href="javascript:window.history.go(-1);">Return to pre page</a>.</p>
}}
```

Now we alter the original definition to look like this:

```
{{5010:
msg___      <h3 class="errorttl">User Login Not Recognized</h3>
msg___      <p>That name and password are not recognized for access.</p>
insert___   showPrevPageLink
}}
```

The content of `showPrevPageLink` will be inserted wherever the `insert` statement is used. It is valid to add more `msg` statements to a definition after the `insert`, and it is valid to use multiple `insert` statements as needed. Using an insert statement requires that the reusable string definition be defined before it is referenced by an `insert` statement. Therefore, it is convention to define all reusable strings at the top of the configuration file.

There are additional configuration statements which can be used:

- `constant___` which defines a global constant by the name given in the `constant___` statement equal to the value of the stringName.
- `errLevel___` accepts an integer from 0 through 5 to declare the error alert level of the error (see the Error Management chapter for details)

- `logit___` accepts a Y or N string character to declare if the error is to be logged (see the Error Management chapter for details)

An error string definition block using all the available features would look like this:

```
{{5010:
constant___fw_kErrUserNotRecognized
errLevel___4
logIt___   Y
msg___     <h3 class="errorttl">User Login Not Recognized</h3>
msg___     <p>That name and password are not recognized for access.</p>
insert___  showLoginLink
}}
```

MVS Validation Error String Configuration Blocks

Defining validation error strings is virtually identical to plain error strings with the exception that the `errorLevel` and `logIt` statements are not used. The `constant` statement is valid, as are the basic `msg` and `insert` statements.

One note about creating custom error codes is that they should be defined with a leading hyphen in them. This will help prevent developers from creating validation codes that could interfere with core validation codes that may be added in the future.

MVS Application String Configuration Blocks

Defining application strings is identical to error strings with the exception that only the basic `msg` and `insert` statements have any meaningful use. Constants have no value as the string name itself should be a discernible reference name.

As with custom validation codes, it may be a smart thing to prefix application string names with either "app" or other application-specific identifier to prevent collisions with core application string names that may be added in the future. Currently, there are only two application strings defined for the framework level which are `valFirstWordThe` and `valFirstWordThis` which are literally just the words "The" and "This" which are used in the validation error strings to adapt to the sentence syntax.

It is expected the framework core strings will remain few in number, so it could be argued there is very little danger in overlap, but as a matter of design, it could still be smart for applications to use a consistent prefix.

Why not make the core strings have prefixes? Most sites are likely to be single-view apps, and it makes sense to have the framework code be the least bulky and most natural to read.

Defining MVS Language, Media, and Varient Views

So far, the above sections described how to create a single view of a string. Next, we'll cover how to create alternate languages, media, and variants.

Languages are created by using a dedicated file. Therefore, to have an american english and universal french set of application strings, you'd create the files strings_appStrings_en-us.cnfg and strings_appStrings_fr.cnfg.

Defining media and variants are done in the configuration file with nested declarations. The same method is used for all three of the string types. Let’s use an application-specific error message as an example starting point. The statements which are universal to all string versions remains the same, but statements which define the actual string are nested based on media value and variant value.

```
{{5610:
constant___    app_kNoSuchStudent
errLevel___    0
logIt___       N
variant_parent:
    msg___     <h3>Child Not Found</h3>
    msg___     <p>The child’s name you entered does not exist.</p>
    insert___  showPrevPageLink
/variant;
variant_teacher:
    msg___     <h3>No Such Student</h3>
    msg___     <p>The student name you entered does not exist.</p>
    insert___  showPrevPageLink
/variant;
}}
```

This example illustrates the scenario that an education application allows login of parents and teachers, and maintains state on the logged in user type. The user type is used by the application to create variants of strings that reflect the user’s relationship to the student. So, if we have a web form in which a student name is to be entered, and the application validates the existence of that student, it might provide an error message if the name is not found. When that error occurs, regardless of who is logged in, it is error code 5610, but we may want to differentiate the message. So we have the parent variant and the teacher variant which allows the message wording to be user type specific.

Using variants in this manner, the application code no longer has to include conditionals to determine the user type and then deliver a custom message. This tends to combine application logic in the view code, or it puts a lot of view code in the application logic. Using this method, the logic becomes a simple matter of identifying user type upon login and setting `$fw_client->’variant’` to “parent” or “teacher.” The display code is a simple matter of allowing the default error message system to call up the string for error 5610. The matter of defining possible strings is cleanly isolated to the config block above,and the decision for which string to retrieve is handled automatically by the multi-view string controller (in this case, the error manager).

Variants in text files can also be used to allow configuration of strings fr multi-client hosted applications. Each client can be configured with a unique version of a string used by the application. To allow cients to customize their own strings, you want to use the strings data table for that (discussed further on), but for strings you’d want to control, the text file is an option.

Another example is when the application is designed for compatibility with desktop browers and other browsers such as PDAs or even aural browsers. If we use the same error code example, but tune it for different browsers, we could have the following:

```
{{5610:
constant___    app_kNoSuchStudent
errLevel___    0
logIt___       N
media_all:
    msg___     <h3>Child Not Found</h3>
    msg___     <p>The child’s name you entered does not exist.</p>
    insert___  showPrevPageLink
/media;
media_aural:
    msg___     <p>Sorry, that child name does not exist.</p>
    insert___  showPrevPageLink
/media;
media_handheld:
    msg___     <p class="bold">Child name not found.</p>
    insert___  showPrevPageLink
/media;
}}
```

Ultimately, media and variant can be combined:

```
{{5610:
constant___    app_kNoSuchStudent
errLevel___    0
logIt___       N
media_all:
    variant_parent:
        msg___     <h3>Child Not Found</h3>
        msg___     <p>The child’s name you entered does not exist.</p>
        insert___  showPrevPageLink
    /variant;
    variant_teacher:
        msg___     <h3>No Such Student</h3>
        msg___     <p>The student name you entered does not exist.</p>
        insert___  showPrevPageLink
    /variant;
/media;
media_handheld:
    variant_parent:
        msg___     <p class="bold">That child name does not exist.</p>
        insert___  showPrevPageLink
    /variant;
    variant_teacher:
        msg___     <p class="bold">That student name does not exist.</p>
        insert___  showPrevPageLink
    /variant;
/media;
}}
```

The above string definition blocks indicate a few syntax rules. First, multiple views are nested by declaring a media container or a variant container inside the block. Either media or variant containers can be defined as a first nested level, but if both are used, the variant container must be inside a media container. Each container is labeled starting with either `media_` or `variant_` to identify its type, and is then followed by a unique name. That name after the underscore is the value that is used in `$fw_client->'media'` and `$fw_client->'variant'`.

MVS msg___ Definitions

As you can see form the examples above, and from looking at the core files, each `msg___` definition line is a literal HTML/Lasso code line. Each line must indeed be a single line. Text cannot wrap across lines. If you want to write short lines, and have sentences broken across several `msg___` lines, you'll have to use `nbsp;` entities at the beginnings of lines to keep spaces betweeen words. (Each line will be trimmed when loaded by the parser).

The lines are combined in the cache into a single string, and that string is `[process]`ed prior to display. So, page variables, custom tags, etc. are all legitimate content.

The main error strings in particular have three local variables that are available. They could be used like this:

```
msg___  <p>Error [#thisErrCode] (log entry [#thisErrLogID]) occurred on page:<br \>
msg___  [$fw_requestPage->'host'][$fw_requestPage->'path'][$fw_requestPage->'file']</p>
```

Where `[#thisErrCode]` and `[#thisErrData]` are passed to the error manager from the error trapping clasue that inserted the value into `$fw_error` to start with. The value for `[#thisErrLogID]` is assigned by the error manager.

MVS SQL Tables

The `$fw_appStrings` and `$fw_pageStrings` interfaces provide access to strings stored in SQL data tables. Each language has a dedicated table. So, there will be multiple tables with the exact same schema, but with content unique to a specific language. The table names can be declared in _initMasters just as with other data tables, but the default naming scheme follows the pattern seen in pbstrings_es and pbstrings_en_us. It is important to note that for hyphened language identifiers like en-us that the hyphen is changed to an underscore in the SQL data table name. This is necessary for table naming rules of some database engines. The framework code takes care of translating the characters when needed.

The data table schema is as follows:

```
CREATE TABLE `pbstrings_en_us` (
  `rcrdNo`               varchar(28)    NOT NULL default '',
  `rcrdCreated`          varchar(19)    NOT NULL default '',
  `rcrdCreatedBy`        varchar(49)    NOT NULL default '',
  `rcrdModified`         varchar(19)    NOT NULL default '',
  `rcrdModifiedBy`       varchar(49)    NOT NULL default '',
  `rcrdStatus`           char(1)        NOT NULL default 'N',
  `rcrdLock`             char(1)        NOT NULL default '',
  `rcrdLockID`           varchar(28)    NOT NULL default '',
  `rcrdLockTime`         varchar(19)    NOT NULL default '',
  `rcrdLockOwnr`         varchar(49)    NOT NULL default '',
  `stringName`           varchar(36)    NOT NULL default '',
  `stringPagePath`       varchar(255)   NOT NULL default '',
  `stringClientMedia`    varchar(12)    NOT NULL default '',
  `stringClientVariant`  varchar(24)    NOT NULL default '',
  `stringValue`          text           NOT NULL default '',
  PRIMARY KEY  (`rcrdNo`),
  KEY `pagePath` (`stringPagePath`),
  KEY `stringName` (`stringName`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8
```

This table is used to store both appStrings and pageStrings. The difference between the two is primarily semantic in that the developer specifies `stringName` for `$fw_appStrings`, and `stringPagePath` for `$fw_pageStrings`. Both, of course, use the media and variant values as well.

The `stringPagePath` values will typically be URL paths, though this is not always a requirement. When using URLs the value must match whatever the result of the expression `($fw_requestPage->'path' + $fw_requestPage->'name')` is.

When not using URLs the value can be arbitrary. For example, there may be a need to pull together several strings at one time that don't really belong to any one page (perhaps they're sidebar content, or a set of reusable section titles). In such a case the stringPagePath for each of the strings that will be collected together can have any value that is common to all strings in that set, but unique to just that set. The pageStrings controller is designed to hold only one page set at a time, so pulling more than one set of strings for the same page will require copying sets to temporary vars.

For records that are for the appStrings controller, the `stringPagePath` value can be empty.

MVS Loader and MVS Cache

The engine behind the multi-view strings system and the test file driven string type controllers is a config file loader and a caching system that simplifies the code the developer has to write, and handles opportunities to improve string retrieval performance through caching and referencing.

The MVS loader is a custom type (fwpCnfg_loadMVStrings.ctyp) used to load and parse the strings configuration files. The strings config files are loaded automatically as needed. There is no need for the developer to preload them. The `$fw_error`, `$fw_validator`, and `$fw_appStrings` controllers also use the mvsLoader to maintain a global-variable, map-based cache optimized for each controller's string data structure. The mvsLoader is used as a gateway between the controllers and the caches.

If we use the appStrings controller as the example, the sequence of events starts with `fwpPage_init` creating the `$fw_appString` object. The `$fw_appStrings onCreate` code then calls the mvsLoader to create a data cache for it. When a particular string is requested through the appStrings controller, it requests the data via the mvsLoader (which was automatically created in `fwpPage_init`, and is not something the developer ever interfaces with directly). The mvsLoader first checks the calling controller's cache, and if the string is available, it is returned directly from the cache. If the string is not available, the mvsLoader loads the appropriate language config file, and parses the text into the cache, and then returns the string to the controller. In the specific case of the appStrings controller, if the string cannot be found in the cache after the language file is loaded, then it will search the strings SQL tables. If the string is found in the appropriate language table, that data is also stored to the cache before being returned to the controller.

MVS Cache Optimization

In the MVS Configuration section above, it was mentioned that the developer needs to define only what is necessary in the config files, and no more. Consider the scenario where you have some strings that need a PDA version, but some that don't. In the config file, it could be required to define a complete PDA set of strings whether or not they were redundant to the non-PDA strings like this:

```
{{siteWelcome:
media_all:
    msg___  <p>Welcome to this Spiffy Site</p>
/media;
media_pda:
    msg___  <p>Welcome</p>
/media;
}}
{{siteNewsTitle:
media_all:
    msg___  <p>News</p>
/media;
media_pda:
    msg___  <p>News</p>
/media;
}}
```

In the `siteWelcome` string we need two definitions. In the `siteNewsTitle` string we do not. If we don't need the definition, there's no value to having the developer create it. So, the actual config code would look like this:

```
{{siteWelcome:
media_all:
    msg___  <p>Welcome to this Spiffy Site</p>
/media;
media_all:
    msg___  <p>Welcome</p>
/media;
}}
{{siteNewsTitle:
msg___      <p>News</p>
}}
```

Now, however, imagine that data converted into multi-dimensioned map (a map of maps) with the outer most map being the languages. the next layer being the string names, and the inner leyer being the variations ofthat string for media. If we use `$fw_client->'language'` and `$fw_client->'media'` to find that string in a map, and we look in the en-us language, for string name `siteNewsTitle`, and the pda version, we won't find it. That just means ther's not a pda-specific version, not that there should be no string returned. So, in the absence of a literal entry we want the system to retrieve a default value. The mvsLoader does exactly that.

When a specific media type is not found, the mvsLoader falls back to searching for a default value. Any string that does not have a media type defined is assigned the media type of "all." When the above config data is loaded into the cache the `siteNewsTitle` will have been stored just as if it were defined in a `media_all` container. So, when a `media_pda` version is not found, the loader looks for `media_all`.

It's great that the mvsLoader degrades to finding a default, but it took two searches instead of just one. To prevent that from being a recurring two searches, the mvsLoader will create a new entry for `media_pda` in the cache and use a Lasso reference to the default entry. This way the data is not duplicated, and the exact entry we want is found the next time with one search.

Retrieving MVS Strings

It was mentioned above that there are separate controllers (Lasso custom types) for each string type. The details of string retrieval for appStrings and pageStrings are discussed in the sections *fwp_appStrings Custom Type API* and *fwp_pageStrings Custom Type API* below. The details for displaying strings generated by the error manager and input validator are discussed in those respective chapters.

fwp_appStrings Custom Type API

The application strings controller `$fw_appStrings` is created automatically by the `fwpPage_init` page initialization tag. This in turn automatically creates a dedicated cache for strings defined in the string_coreStrings and strings_appStrings configuration files, and caches any strings retrieved from the strings SQL tables. The `fwp_appStrings` ctype does have some instance variables, but all are for internal use, or are taken care of during the `fwpPage_init` process.

Displaying an appString

The application strings controller `$fw_appStrings` retrieves one string per request, and uses the string name as a member tag name like `$fw_appStrings->stringName`. If `stringName` is ultimately not found, a string indicating that an unknown string was requested will be displayed.

The current values of `$fw_client->'language'`, `$fw_client->'media'`, and `$fw_client->'variant'` are used to determine which version of the string to display. However, if there is a need to display something different than what these mode variables would indicate, then one or more explicit parameters can be used such as:

```
$fw_appStrings->(stringName: -language = 'xx', -media='xy', -variant='yy');
$fw_appStrings->(stringName: -language = 'xx');
$fw_appStrings->(stringName: -media='xy', -variant='yy');
```

Clearing the appString Cache

During development, it can be necessary to clear the cache to ensure new string config definitions are used. This can be done with:

```
$fw_appStrings->cacheReset;
```

This command is already included in section (E) of _initMasters which is handy for clearing the cache with each page during development, but it may also be necessary to use programmatically to clear the cache if the app stores strings in a data table and provides an admin interface. When the value of an existing string is changed, the cache should be reset. It's not necessary to clearthe cache if new strings are added.

fwp_pageStrings Custom Type API

The page strings controller `$fw_pageStrings` is created by the `fwpPage_loadTemplate` tag when the `$fw_pageModes->enablePageStrings` tag is called. That tag should be called within the _pageConfig file. The `fwp_pageStrings` ctype does have some instance variables, but all are for internal use, or are taken care of during the instantiation step.

Retrieving and Displaying pageStrings

The page strings controller `$fw_pageStrings` retrieves string records for the current page from an MVS data table (see the section *MVS SQL Tables* above for details). Each record is translated to allow every value for `stringName` to be used as a member tag name of the object `$fw_pageStrings`. Therefore, a found set of records like this:

```
stringPagePath        stringName     stringValue
--------------------  -----------    ----------------------
/ftrs/pb_features     pgTitle        Whiz Bang Features
/ftrs/pb_features     contact        contact.me@example.com
/ftrs/pb_features     intro          PB is a spiffy...
/ftrs/pb_features     download       pb510_mac.sit
```

becomes a set of member tags like this:

```
$fw_pageStrings->pgTitle
$fw_pageStrings->contact
$fw_pageStrings->intro
$fw_pageStrings->download
```

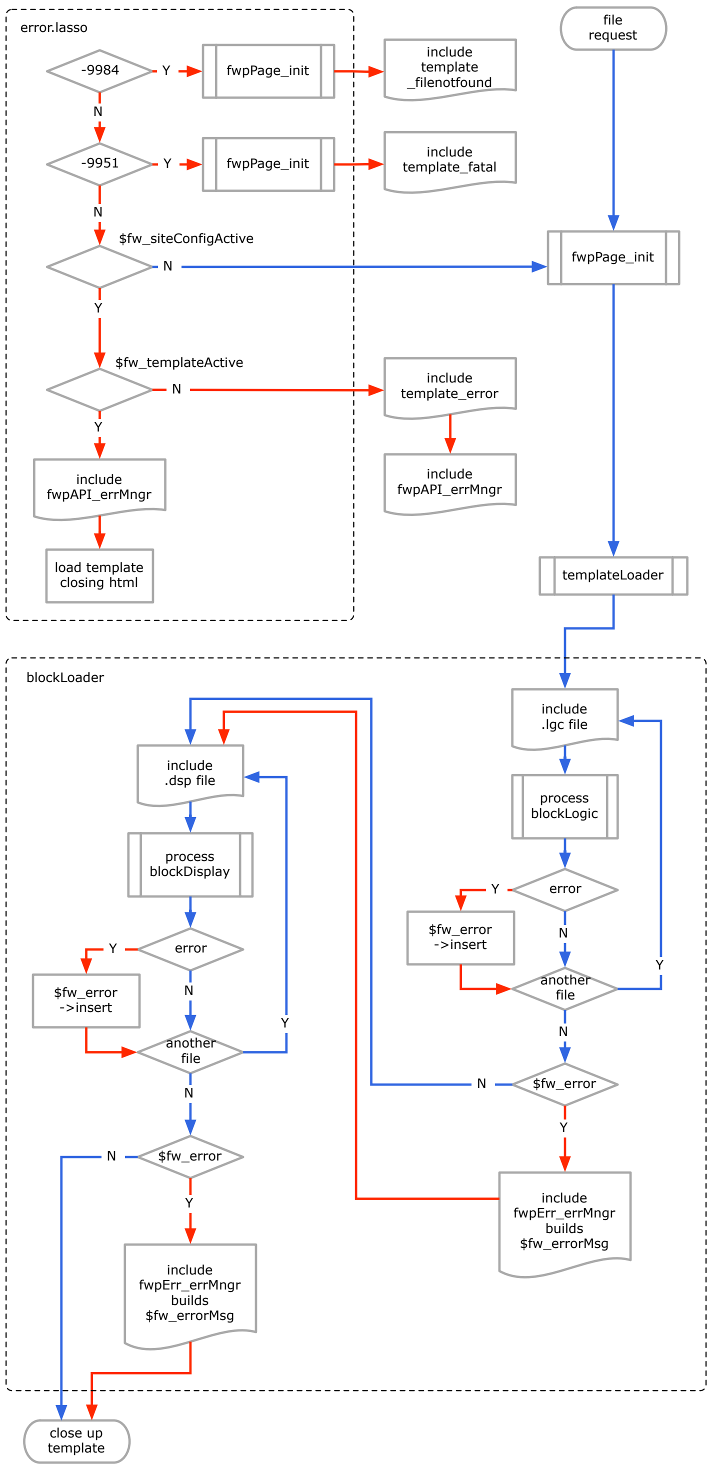
If `stringName` is ultimately not found, a string indicating that an unknown string was requested will be displayed. This custom type is created by the templateLoader so that the data is loaded before the template is loaded. This allows templates to be influenced by dynamic content.

Retrieval Modes

The retrieval process has two modes. One is designed to optimize speed, and the other to optimize data table simplicity. In the speed optimized mode, which is the default setting, the data table must have an actual record entry for every permutation of a string based on the support for media and variant versions. This allows a single query with a where clause that specifies an exact media and variant to find all strings in one step. The second mode eliminates the need for an actual record for each entry. In this mode a query is designed to retrieve all strings of all media and variant versions for that page. Lasso code is then used to determine if the exact media and variants are available, and if not, the defaults are used.

This capability is called auto degrading, and is controlled by setting the value for the definition of the constant `fw_kUsePageStringsAutoDegrade` in `_fwpAPI_initVars`. The two modes should not be used interchangeably. An application should be designed to use one mode or the other. This is why a constant is used rather than a `$fw_pageModes` setting.

Error Management



Error management is the proactive handling of both expected and unexpected errors in an application's code. Unexpected errors can include application code bugs and external resources like databases and files being unavailable. Expected errors can include poor user input (typing letters where numbers are required), database search results which find no matching records, and user logins which are invalid. In all these cases and more, a professionally developed application is expected to maintain controlled behavior when an error occurs, and provide as useful a response for the user as possible.

With Lasso, we can rely on its internal error trapping system for catching code bugs. However, if we allow Lasso's default error display to be presented to users of our web sites, that page a) looks catastrophic which doesn't instill confidence that the application is in control of itself, b) doesn't provide any information useful to end users, and c) exposes source code that may be useful to someone looking to do unfriendly things to our site. It would be better if we could trap Lasso's error gripes and create our own response to that error that would not expose source code, would look better, and would provide information that would be more useful to the end user.

At the application level we want to trap errors and interpret them in their context to provide more useful feedback to the end user. For example, if a user logs in and mistypes a password, Lasso may generate a "no records found" error. Well, sure a record wasn't found, but that could lead the user to believe his record has been deleted. It would be better to trap the error, and, leveraging the context that we're doing a search for login credentials, present the user with a message that suggests the password wasn't recognized, and perhaps it was mistyped. Lasso cannot do this for us on its own. We have to write the code that performs that interpretation (which is simply providing a message unique to the circumstances).

In its simplest form, error handling is little more than using an [if] test to check if Lasso itself is reporting an error. In many cases when performing that test we know there are some expected possibilities. With the login example, we know that a no records found error is a likely possibility. Instead of showing Lasso's error, we can write our own message as suggested above. For a login error like that, there is probably only one or two places in a whole site that needs code to handle that. However, consider testing for an expired login session. Testing for this is likely needed on several pages (possibly all pages). We do not want to be writing that code on every page. What if wanted to change what the message says to such an error? Just like we would separate a repetitively used web page footer into a dedicated file to be included into multiple pages, we want to take error handling code which is repeated throughout the application and centralize it into a general resource.

Even if we take error messages and centralize them into a single resource, we do still have to test for the errors in the places we expect them. So, if the [if] tests for errors are in the file login.lasso, how do we insert the message we want displayed from a general source?

When software errors occur, we often see error codes being used. Even in Lasso, our error testing can look at Lasso's error codes to determine what an error is. An error management system usually utilizes codes to abstract the technical definition of the error event from the message which is displayed to describe the error. For example Apache uses a 404 code for the Page Not Found message. Lasso uses -9961 to indicate a syntax error was found. This use of codes allows the message to change (perhaps for multiple human languages) without requiring the application code to change when we want to describe the error differently. Error codes can be used to efficiently identify very small differences in what caused the error. When an error is detected, the type of error can be classified and coded, and the code forwarded to a central error reporting system that standardizes the response and textual output messages.

The PageBlocks framework uses this scheme of separating codes and messages to handle application error trapping. Numerous codes have been predefined for a number of common application errors. Separate files standardize the text output that is displayed in response to those codes. The developer has open access to these files to change the messages and to create custom messages unique to the application.

Error Management in PageBlocks

Like any Lasso application, the PageBlocks code uses [if] statements to trap specific errors. However, rather than [include] specific error messages within the [else] clause of these conditional structures, error codes are defined and stacked in the error manager controller object `$fw_error`, and a single [include] is ultimately loaded by the templateLoader or blockLoader to generate the error message display and user response interface.

An error trapping routine is typically structured like this:

```
if: {conditional};
    ..... do some good stuff .....
else;
    $fw_error->(insert: 'errCode'='descriptor');
/if;
```

where the error manager insert might look like these examples:

```
$fw_error->(insert: '5100'='no records found');
```

```
    $fw_error->(insert: '5010'='access denied');
```

While this doesn't really reduce the amount of programming necessary to trap errors (it still requires writing the [if] conditionals), it does centralize the capture of error occurrences into `$fw_error`, and, as we'll see shortly, it centralizes the management of display messages and user interface which increases the flexibility and ease of defining new errors to manage.

The PageBlocks API does offer an overall reduced workload for error management because so many of the custom tags used to interface with files and databases have error management built in. Using those tags buys the error management code inside. Within the custom tags and types, error traps are still coded as [if]...[else]...[/if] constructs. On a page utilizing authentication, actionShells, and other control features, there are several application-specific errors which must be identified separately, so there can be several nested [if] containers.

When writing your own error handling conditionals, the [if] statement would be written just like you would ordinarily do. The clause between the [else]...[/if] is where the PageBlocks-specific code goes. Instead of including a file written to directly respond to that specific error, use the `$fw_error->insert` statement. Generally speaking, PageBlocks does not use Lasso's `Protect` or `Handle` tags. It's possible there could be cases where they'd be desirable to use, but I have never found a need for them, nor found a case where I thought they'd be better than the above described system.

You'll notice the insert is a pair. The first pair member is the error code that applies to the conditional which failed. The second pair member can be one of two things: either a simple descriptor which really only serves for ease of source code reading, or as a informational parameter to supplement the main message. How the second pair member is used depends on the whether the error message would benefit from clarification or not. For example, in a no records found situation, there's nothing needed to supplement a standard message. However, if a record is inaccessible because it is locked, it may be useful to report which user has the record locked. In that case the insert could look like this:

```
$fw_error->(insert:'5010'= ($fw_user->getProfile:'userName'));
```

The pair's second member can now be used to supplement the main error message string that is assembled by the error manager.

Handling Multiple Errors

While a single error might typically stop the processing of a page further, each error's [else] clause is written to insert its error code to the `$fw_error` object. The error manager is written to iterate the eror list and assemble a single message with all the appropriate information for each error encountered. Therefore, it is possible to take advantage of an ability to trap and notate multiple non-fatal errors.

Error Messages

The error manager relies on the multi-view string (MVS) system to define messages. This allows error messages, whether core to the framework or specific to the application, to be defined in multiple languages, for multiple media devices, and even in multiple variations for clients or user types. See the chapter *Multi-view Strings (MVS)* for details.

Error Logging and Automated Admin Alerts

The PageBlocks error manager allows for errors to be declared as one of five severity levels. In _initMasters, the programmer can set that all errors at or above a specified severity level generate an automated email to the site administrator.

Errors can also be logged to a text file with automated rollover management, or logged to a database table. In _initMasters the following vars can be defined:

```
//  fw_gLogErr          (true) turns on|off the logging of errors {true|false}
//  fw_gLogErrRoll      (M) sets rolling frequency of error log text files {A|M|W|D}
//  fw_gLogErrTarget    (file) sets destination of log data {file|database}

    'fw_gLogErr'            = true,
    'fw_gLogErrRoll'        = 'M',
    'fw_gLogErrTarget'      = 'database',
```

If the error log target is a text file, the file can be rolled annually, monthly, weekly, or daily. The rollover does not apply to database logging. (Use the database server/backup utilities to handled rolling of tables).

User Input Validation

A special type of error management has been developed to handle field input validation and business rules for forms. That is covered in a separate section of the documentation.

Displaying and Suppressing Error Messages

The prebuilt error message to be displayed is stored in `$fw_error->'errMsgs'`. This will contain the appropriate strings based on the current MVS language, media, and variant settings.

Specific errors can be suppressed by removing the error code from the `$fw_error` object after it has been or may have been inserted. This can be useful if the code needs to know if an error happened, but doesn't want it to be included in the error message, or a custom alternative message is preferred.

For example, the application may provide a search form. If a search yields no results, the default core message for error 5100 will be displayed. It may be preferred to display a message more tuned to the context of the search. You can't just override the default 5100 message, because there may be several searches that each need a unique message. To accomplish this, the code could first test if an error 5100 occurred, and if it had, remove that entry and insert a customized entry to take its place like this:

```
..... query right here ..…

if: $fw_error->(contains:'5100');                   // test for the error
    $fw_error->(removeCode:'5100');                 // remove the error
    $fw_error->(insert:'5605'=’custom response’);    // insert custom error
//if;
```

Error Handling during Development

All of the above discussion is aimed at errors intended to be displayed to the end user of the application. PageBlocks 5.2 has added signicant coverage of using errors to assist the developer when writing and debugging PageBlocks code.

Errors which are internal to PageBlocks tags or types that happen due to syntax problems, or insufficient condition testing prior to the use of the tag or type will now insert error codes into `$fw_apiError`. This new object separates the errors intended for the developer from those intended for end users.

When `$fw_debug` is enabled, the errors found in `$fw_apiError` will be parsed and displayed, otherwise they'll remain quiet to the end user. In some cases, where the application is likely to be failing due to this error (file not found, database not available, etc) a generic error will be shown to the end user.

Many tags and types in the PB API have been updated to provide better error information in the `$fw_tagTracer` object.

Additionally, many error are not set to also be reported using Lasso's `log_critical` if the control variable `$fw_logCritical` is set to true in _initMasters.

The combination of trapping errors that are specifically API usage errors, reporting them on screen differently than user errors, adding more details to the tag tracer, and adding many errors to the critial Lasso loghould make a significant reduction in the time it takes to understand what most errors are and where they're coming from compared to earlier version of PageBlocks.

Use $fw_apiError For Debugging

Using `$fw_apiError` is exactly the same as using `$fw_error`. Error messages are in the same MVS files, but you can't use `fwpErr_removeCode` with `$fw_apiError` as that tag is dedicated to `$fw_error` and it wouldn't make sense to remove a developer warning error anyway.

Database Interfacing

One of the constant chores in a web application is interfacing to databases and all the peripheral tasks that entails such as error handling, scrubbing form data to be sure it is formatted correctly and has no malicious code in it, prepping query statements, and handling row locking. There is a lot of reptitive code in these operations, and one of the earliest steps I ever made in creating my application framework was to group these steps in a reusable fashion. The database API has gone through a few transformations and today has some modern abstraction capabilities.

Database interfacing can be done in many ways depending on need, and the high level enterprise abstraction systems can be very complex. The goal of the PageBlocks database API isn't to provide all the modern object relational mapping capabilities that makes sense in large enterprise applications, but rather to capture the most common needs of typical small and medium web applications, and provide an API that anyone with some working knowledge of Lasso and SQL can quickly understand.

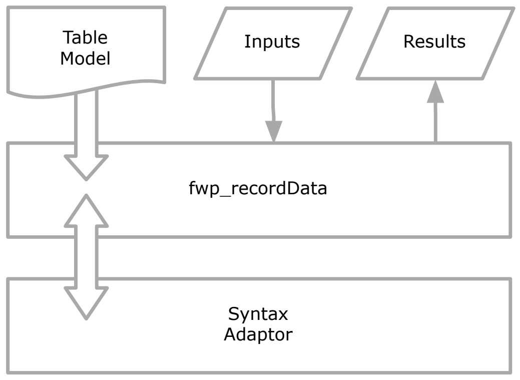
fwp_recordData Database API

(Currently fwp_recordData works for SQL databases only. For FileMaker, use the older fwp_rcrdData custom type. Eventually, the FMP interfacing will be updated with the same abilities as fwp_recordData).

The `fwp_recordData` custom type provides a general purpose interface for several data management tasks common to database driven web pages. It provides member tags for adding, deleting, updating, selecting, and duplicating records, as well as additional utility functions such as record locking and data storage.

The `fwp_recordData` API is tiered into three components. The first component is an application-specific table configuration file tableModel_{tableName}.cnfg which maps field names to form input names and lasso data types, and includes input validation rules for each field. These configuration files use on-demand loading so that their first access goes through a parsing routine to load them into a global var structure, but subsequent accesses go straight to the global var which we call a table mapper.

The second component is the `fwp_recordData` custom type. The primary purpose of this custom type is to wrap supporting routines, which are called action shells, around the core database action to integrate error management, data input validation, and record locking functionalities into a single, easy-to-use object. The `fwp_recordData` object also helps to automate user interface responses to these actions. The code in `fwp_recordData` is ignorant of query syntax, and focuses strictly on inputs, interfacing logic, and results. The `fwp_recordData` data type doesn't know exactly how to query a table. It only knows what type of query it needs at given points in its control logic. That brings us to the third layer.



The third component we call a syntax adaptor. This is the layer that contains the SQL syntax specifics. The `fwp_recordData` to adaptor interface is defined with a specific set of methods. Each method constructs a whole SQL query or a portion of a query that the `fwp_recordData` ctype will request from the adaptor. An adaptor is written for a specific syntax. Having multiple adaptors enables support for multiple database servers. With the developer's main API being the methods of `fwp_recordData`, it is possible that little or no change to much of the application code is needed to switch databases.

The `fwp_recordData` data type is not intended to be the only method a developer uses with the PageBlocks framework to interface to a database. It is a tool available to simplify the implementation of many commonly integrated functions, but it is not mandatory for all database interactions. While the `fwp_recordData` API can handle complex SQL queries, there are going to be occassions where using Lasso's inlines to communicate directly with the database will be advantageous. For example, transactions should be written out manually with Lasso's inline tags or a combination of the inline tag and `fwp_recordData` calls. The `fwp_recordData` member tags are not yet written to support transactions internally.

Aside from the logic integration capabilities, a main purpose of `fwp_recordData` is to automate the writing of `INSERT`, `DELETE`, `UPDATE`, and `SELECT` statements in response to form interaction and a few simple tag parameters. The `SELECT` actions can also be manually specified in detail to allow for complex relational SQL queries.

By consolidating these tasks and supporting routines into a single general-purpose custom type, the programmer has a stable and easy-to-use interface for many data tasks, and can focus development efforts on the more complex data and object management issues that are application specific.

Data Storage and SELECT Options

This type's primary task is not to store data, but rather to simplify the interface for managing data actions, though there are some data storage features. Data retrieved for the object are stored in Lasso's named inline containers by default. Optionally, data can be stored in an array of arrays (essentially just a reference to LP8's `records_array`), or in an array of maps which provides an easier to read interface than `records_array`. Queries which result in single record selections can also generate an individual page var for each field.

Each dataset created by an `fwp_recordData` `SELECT` contains `->'foundCount'`, `->'showFirst'`, and `->'showLast'` properties as equivalents to the built-in lasso tags. Also, each `SELECT` action returns the named inline in `->'inlinename'` and other status data such as errors, query time, and more. The full details of instance vars are explained further on.

Subclassing for Custom Models

While the main goal of `fwp_recordData` is to provide a query interface, it can be subclassed so the developer can create custom models with all the SQL interfacing built in. This in fact is the preferred way to provide custom business rules that should be applied to data, and is the means by which the developer can write more complex methods for handling relational data models. (Eventually `fwp_recordData` will be extended to recognize models built from relational data, but for now, the developer will have to handle that).

Interface Quick Glance

Create an fwp_recordData object:

```
    var:'x' = (fwp_recordData:'tableName');
```

Read-Only Instance Variables:

```
    'db'
    'tbl'
    ‘dbtbl’
    'conn'
    'keyTable'
    'foundCount'
    'firstRecord'
    'lastRecord'
    'pageCount'
    'currentPage'
    'inlineName'
    'records'
    'lock'
    'error'
    'errors'
    'validationResults'
    'inputsAreInvalid'
    'rulesAreInvalid'
    'queryParams'
    'queryString'
    'queryStringAlt'
    'queryTime'
    'tagTime'
```

Member Tags:

```
    ->onCreate
    ->cacheReset
    ->validateInputs
    ->validate
    ->getFieldForInput
    ->getInputForField
    ->sql
    ->getRecordUsingLock
    ->updateUsingLock
    ->updateUsingKeyVal
    ->update
    ->deleteUsingLock
    ->deleteUsingKeyVal
    ->delete
    ->select
    ->add
    ->duplicate
    ->insertAppError
    ->showMsgsFor
    ->errorExistsFor
```

Public Read-Only Instance Vars

- `db` (string) = the name of the actual database
- `tbl` (string) = the name of the actual table
- `dbtbl` (string) = a string in the format of `database.table` to simplify query writing
- `keyTable` (string) = the name provided when creating the object which will typically be a key for the `$fw_gTables` map (and thus not necessarily identica to the real table name in `tbl`).
- `foundCount` (integer) = the number of found records from a `SELECT` (i.e. it's the 29 of "11 through 15 of 29")
- `firstRecord` (integer) = the sequence number of the first record returned in a `SELECT` set (i.e. it's the 11 of "11 through 15 of 29")
- `lastRecord` (integer) = the sequence number of the last record returned in a `SELECT` set (i.e. it's the 15 of "11 through 15 of 29")
- `pageCount` (integer) = a number indicating the number of "pages" of records in the result if the results are divided by the amount in the LIMIT statement. So, if a query includes a LIMIT of 5 records in the results, and there are a total of 29 possible results, then `pageCount` = 6.
- `currentPage` (integer) = a number identifying which "page" of records out of the total possible results has been returned. So, if viewing records 11 through 15 of 29, `currentPage` = 3.
- `inlineName` (string) = the ID value of the named inline that a returned record set is stored under
- `records` (array) = an optional record set stored as either an array of arrays or an array of maps as dictated by the `-withRecordsArrays` or `-withRecordsMaps` options in a `SELECT`
- `lock` (string) = the ID value of record lock if acquired
- `error` (boolean) = indicates whether the member tag generated an error
- `errors` (array) = an array of pairs containing the error codes and comment strings
- `validationResults` (map) = contains the results of the `fw_validator->validate` operation performed in the `->validateInputs` tag. The map include three elements: `coreCodes`, `appCodes`, and `errorMsgs`.
- `inputsAreInvalid` (boolean) = indicates whether the submitted fields passed the basic validation requirements specified in the tableModel_ config file
- `rulesAreInvalid` (boolean) = indicates whether the submitted form passed the rules defined in the `->validate` member tag of a subclassed custom type based on `fwp_recordData`
- `queryParams` (array) = a copy of the params submitted to the tag
- `queryString` (string) = the actual SQL string executed by the member tag (sometimes more than one actual query is generated, but this will be the one that is the core of the tag's purpose)
- `queryStringAlt` (string) = if more than one actual query is generated by the tag, this will be the one that is secondary to the core of the tag's purpose
- `queryTime` (string) = the time in milliseconds it took for all SQL queries used within the member tag to execute (not just the one in `queryString`)
- `tagTime` (string) = the time in milliseconds it took for the entire member tag code to execute

Public Method –>cacheReset

Resets the global cache for all tableModel files.

Public Method –>validate

A stub method for a subclassed type to override and set the value of `'rulesAreInvalid'` to true or false based on the failure | passing of the validation code therein.

This tag is automatically triggered by `->add`, `->update`, `->updateUsingKeyfld`, and `->updateUsingLock` just after the `->validateInputs` tag has processed. The built-in tag simply returns `'rulesAreInvalid' = false`.

In order to make use of this tag and write custom validation rules outside the scope of those that could be handled through custom validation codes for `$fw_validator`, the `fwp_recordData` type needs to be subclassed. A very simple example of this would be:

```
define_type: 'myModel', 'fwp_recordData';
    define_tag:'validate';
        if: $m_expDate < date;
            (self->'rulesAreInvalid') = true;
        /if;
    /define_tag;
/define_type;
```

The above subclassed type does everything the `fwp_recordData` data type does, and it embeds custom validations that will be automatically triggered. See the section *Input Validation* for how the error message would be defined.

Private Method ->validateInputs

This tag is automatically triggered by `->add`, `->update`, `->updateUsingKeyfld`, and `->updateUsingLock` prior to performing just about any steps at all. The tag passes the info needed to use `fwp_validator`, and stores the boolean result in `->'inputsAreInvalid'`. The validation rules come straight from the tableModel_ config file.

At the end of processing, this tag then calls `->validate` to perform developer defined rules of validation.

Finally, the tag calls `fw_validator->getResults` and stores the results of validation in the instance var `->'validationResults'`. As of version 5.3, validation can now be fully encapsulated in the model object.

Public Method –>sql

Accepts a completely formed SQL query. The main purpose for using this member tag instead of a standard Lasso `inline:-sql` tag is that this one includes error trapping behavior identical to all other member tags, and will also log the query. Additionally, it can return records as a named inline, an array of arrays, or an array of maps. It also supports several of the instance vars.

All field names in the query can be the abstracted input names from the tableModel configuration file.

Supports the following instance vars:

- `foundCount`
- `inlineName`
- `records`
- `error`
- `errors`
- `queryString`
- `queryTime`
- `tagTime`

Accepts the following parameters:

- `-query` (required, string) = the complete SQL query to be executed
- `-logname` (optional, string) = arbitrary name to identify the query in the log
- `-asRecordsArrays` (optional, no value) = records to be returned in the `->'records`' instance var as an array of arrays (essentially the same thing as `records_array`). The field array will be in the same order as the fields in the `SELECT` clause of the query. (`-withRecordsArrays` is deprecated).
- `-asRecordsMaps` (optional, no value) = records to be returned in the `->'records'` instance var as an array of maps. The map keys will be the field names used in the `SELECT` clause of the query. This will be the actual field names even if abstracted tableModel input names are used. (`-withRecordsMaps` is deprecated).

Public Method –>getRecordUsingLock

Locates a record with a specified key pair, locks it by setting the lock fields, and retrieves the specified fields. The purpose is to lock a record for an update or delete form, and to retrieve the fields needed for display in the form. Specify `-withMakeVars` to convert fields to model variables.

Supports the following instance vars:

- `lock`
- `inlineName`
- `error`
- `errors`
- `queryString`
- `queryTime`
- `tagTime`

Accepts the following parameters:

- `-select` (optional, string) = a list of field names to use verbatim in the query `SELECT` clause. If not specified, then * will be used as a default.
- `-keyfld` (optional, string) = the name of the field to use as the key field. If not specified the keyfield name specified in the tableModel_ config file will be used.
- `-keyval` (optional, string) = the value to search for in the keyfield. This is required unless the parameter `-wherePairs` is used.
- `-lockval` (optional, string) = the value of a previous lock which the record may be marked by. If the record may have already been locked, pass that lock value. If the values match, the lock will be refreshed.
- `-wherePairs` (optional, array of pairs) = provides a list of inputs or field names and values that will be rendered into a `WHERE` clause with an `AND` between each pair.
- `-withMakeVars` (optional, no value) = indicates that fields retrieved will be converted to page variables named by the inputName specified in the tableModel_ config file.

Public Method –>updateUsingLock

Updates a record identified by a lock value. The record must have already been locked, and that lock ID passed to this tag. This tag along with `->getRecordUsingLock` provides full pessimistic lock control. The update sequence of events will include triggering the validation tags, updating the record, resetting the lock, logging the action (if turned on), and trapping any errors to report.

Supports the following instance vars:

- `inlineName`
- `inputsAreInvalid`
- `rulesAreInvalid`
- `error`
- `errors`
- `queryString`
- `queryTime`
- `tagTime`

Accepts the following parameters:

- `-lockval` (required, string) = the ID of a previously acquired lock
- `-inputs` (required, string) = a comma separated list of the tableModel_ config file inputNames that are allowed to contribute to the update query. This prevents field injection into queries by submitting an altered set of fields.
- `-confirmFields` (optional, string) = a list of fields to be used verbatim in a `SELECT` query to retrieve fields after the update is completed. This provides an opportunity to provide a confirmation message using data directly from the updated message.
- `-withMakeVars` (optional, no value) = indicates that fields retrieved by the `-confirmFields` step (if used) will be converted to page variables named by the inputName specified in the tableModel_ config file.
- `-withoutValidate` (optional, no value) = instructs the tag to skip the input validation processes.
- `-withoutLog` (optional, no value) = instructs the tag to not log action even if logging is on.

Public Method –>updateUsingKeyVal

Updates a record identified by any key field/value pair. It is not necessary that the record have been previously locked. If the record has been locked, and that lock is still valid, the update will be aborted. This is considered honoring an active lock. This tag offers the opportunity to do a hybrid of optimistic/pessimistic lock control. The update sequence of events will include triggering the validation tags, updating the record, resetting the lock, logging the action (if turned on), and trapping any errors to report.

Supports the following instance vars:

- `inlineName`
- `inputsAreInvalid`
- `rulesAreInvalid`
- `error`
- `errors`
- `queryString`
- `queryTime`
- `tagTime`

Accepts the following parameters:

- `-keyfld` (optional, string) = the name of the field to use as the key field. If not specified the keyfield name specified in the tableModel_ config file will be used.
- `-keyval` (required, string) = the value to search for in the keyfield.
- `-inputs` (required, string) = a comma separated list of the tableModel_ config file inputNames that are allowed to contribute to the update query. This prevents field injection into queries by submitting an altered set of fields.
- `-confirmFields` (optional, string) = a list of fields to be used verbatim in a `SELECT` query to retrieve fields after the update is completed. This provides an opportunity to provide a confirmation message using data directly from the updated message.
- `-withMakeVars` (optional, no value) = indicates that fields retrieved by the `-confirmFields` step (if used) will be converted to page variables named by the inputName specified in the tableModel_ config file.
- `-withoutValidate` (optional, no value) = instructs the tag to skip the input validation processes.
- `-withoutLog` (optional, no value) = instructs the tag to not log action even if logging is on.

Public Method –>update

Updates a record identified by any key field/value pair. It is not necessary that the record have been previously locked. This is an "update at will" process. If the record has been locked, and that lock is still valid, this tag will ignore that lock. This tag offers the opportunity to do no lock control at all or to use optimistic lock control if the developer includes his own code to maintain an update counter field. The update sequence of events will include triggering the validation tags, updating the record, resetting the lock, logging the action (if turned on), and of course trapping any errors to report.

Supports the following instance vars:

- `inlineName`
- `inputsAreInvalid`
- `rulesAreInvalid`
- `error`
- `errors`
- `queryString`
- `queryTime`
- `tagTime`

Accepts the following parameters:

- `-keyfld` (optional, string) = the name of the field to use as the key field. If not specified the keyfield name specified in the tableModel_ config file will be used.
- `-keyval` (required, string) = the value to search for in the keyfield.
- `-inputs` (required, string) = a comma separated list of the tableModel_ config file inputNames that are allowed to contribute to the update query. This prevents field injection into queries by submitting an altered set of fields.
- `-confirmFields` (optional, string) = a list of fields to be used verbatim in a `SELECT` query to retrieve fields after the update is completed. This provides an opportunity to provide a confirmation message using data directly from the updated message.
- `-withMakeVars` (optional, no value) = indicates that fields retrieved by the `-confirmFields` step (if used) will be converted to page variables named by the inputName specified in the tableModel_ config file.
- `-withoutValidate` (optional, no value) = instructs the tag to skip the input validation processes.
- `-withoutDateStamp` (optional, no value) = instructs the tag to not update the `rcrdModified` and `rcrdModifiedBy` fields as a result of the `->update` call.
- `-withoutLog` (optional, no value) = instructs the tag to not log action even if logging is on.

Public Method –>deleteUsingLock

Deletes a record identified by a lock value. The record must have already been locked, and that lock ID passed to this tag. This tag, along with `->getRecordUsingLock`, therefore provides full pessimistic lock control (preventing the updating of this record while it is staged to be deleted). The sequence of events will include retrieving the `-confirmFields` data, deleting the record, logging the action (if turned on), and trapping any errors to report.

Supports the following instance vars:

- `inlineName`
- `error`
- `errors`
- `queryString`
- `queryTime`
- `tagTime`

Accepts the following parameters:

- `-lockval` (required, string) = the ID of a previously acquired lock
- `-withFlagOnly` (optional, no value) = if this optional command is used, the record is not actually deleted, but the field `rcrdStatus` is set to an empty value which is to be interpreted by the application code as a non-existing record.
- `-confirmFields` (optional, string) = a list of fields to be used verbatim in a `SELECT` query to retrieve fields before the delete is performed. This provides an opportunity to provide a confirmation message using data directly from the deleted record.
- `-withMakeVars` (optional, no value) = indicates that fields retrieved by the `-confirmFields` step (if used) will be converted to page variables named by the inputName specified in the tableModel_ config file.
- `-withoutLog` (optional, no value) = instructs the tag to not log action even if logging is on.

Public Method –>deleteUsingKeyVal

Deletes a record identified by any key field/value pair. It is not necessary that the record have been previously locked. If the record has been locked, and that lock is still valid, the delete will be aborted. This is considered honoring an active lock. This tag offers the opportunity to do a hybrid of optimistic/pessimistic lock control. The sequence of events will include retrieving the `-confirmFields` data, deleting the record, logging the action (if turned on), and trapping any errors to report.

Supports the following instance vars:

- `inlineName`
- `error`
- `errors`
- `queryString`
- `queryTime`
- `tagTime`

Accepts the following parameters:

- `-keyfld` (optional, string) = the name of the field to use as the key field. If not specified the keyfield name specified in the tableModel_ config file will be used.
- `-keyval` (required, string) = the value to search for in the keyfield.
- `-withFlagOnly` (optional, no value) = if this optional command is used, the record is not actually deleted, but the field `rcrdStatus` is set to an empty value which is to be interpreted by the application code as a non-existing record.
- `-confirmFields` (optional, string) = a list of fields to be used verbatim in a `SELECT` query to retrieve fields before the delete is performed. This provides an opportunity to provide a confirmation message using data directly from the deleted record.
- `-withMakeVars` (optional, no value) = indicates that fields retrieved by the `-confirmFields` step (if used) will be converted to page variables named by the inputName specified in the tableModel_ config file.
- `-withoutLog` (optional, no value) = instructs the tag to not log action even if logging is on.

Public Method –>delete

Deletes a record identified by any key field/value pair. It is not necessary for the record to have been previously locked. This is a "delete at will" process. If the record has been locked, and that lock is still valid, this tag will ignore that lock. This tag offers the opportunity to do no lock control at all or to use optimistic lock control if the developer includes his own code to maintain an update counter field. The sequence of events will include retrieving the `-confirmFields` data, deleting the record, logging the action (if turned on), and trapping any errors to report.

Supports the following instance vars:

- `inlineName`
- `error`
- `errors`
- `queryString`
- `queryTime`
- `tagTime`

Accepts the following parameters:

- `-keyfld` (optional, string) = the name of the field to use as the key field. If not specified the keyfield name specified in the tableModel_ config file will be used.
- `-keyval` (required, string) = the value to search for in the keyfield.
- `-withFlagOnly` (optional, no value) = if this optional command is used, the record is not actually deleted, but the field `rcrdStatus` is set to an empty value which is to be interpreted by the application code as a non-existing record.
- `-confirmFields` (optional, string) = a list of fields to be used verbatim in a `SELECT` query to retrieve fields before the delete is performed. This provides an opportunity to provide a confirmation message using data directly from the record to be deleted.
- `-withMakeVars` (optional, no value) = indicates that fields retrieved by the `-confirmFields` step (if used) will be converted to page variables named by the inputName specified in the tableModel_ config file.
- `-withoutLog` (optional, no value) = instructs the tag to not log action even if logging is on.

Public Method –>select

The select tag has a number of capabilities to execute literal queries or to automate a variety of queries. By leveraging the information in the tableModel_ config file, the select tag can fill in the blanks for just about anything not provided through parameters. This includes automating queries for complex form-driven searches. Results can be returned in any of four formats.

Supports the following instance vars:

- `foundcount`
- `firstRecord`
- `lastRecord`
- `pageCount`
- `currentPage`
- `inlineName`
- `records`
- `error`
- `errors`
- `queryStringAlt`
- `queryString`
- `queryTime`
- `tagTime`

Accepts the following parameters:

(On their own, every parameter is optional. However, some become required based on the presence or lack of presence of other parameters. These dependencies will be pointed out).

- `-keyfld` (optional, string) = the name of the field to use as the key field. If not specified, the keyfield name specified in the tableModel_ config file will be used in the absence of any other `WHERE` qualification parameters.
- `-keyval` (optional, string) = the value to search for in the keyfield if keyfld is specified. If specified on its own without any other `WHERE` qualifications, then this value will be used in conjunction with the keyfield name specified in the tableModel_ config file.
- `-inputs` (optional, string) = a comma separated list of the tableModel_ config file inputNames that are allowed to contribute to the automation of a search form query. This prevents field injection into queries by submitting an altered set of fields. To make this parameter relevant, the `-where` parameter must be defined with the literal string value of `'form'`.
- `-select` (optional, string) = a list of field names to use verbatim in the query `SELECT` clause. If not specified, the * will be used as a default.
- `-from` (optional, string) = the verbatim clause to use in the `FROM` clause of a query. This is normally not required as a default `database.table` string is automatically used. However, this is where the developer can create joins and other complex queries.
- `-where` (optional, string) = defaults to all records if not specified, otherwise, either the `-keyval` parameter must be defined or the `-where` parameter must be defined. There are several options

for manual or automated behavior.

First, it can be the literal string `'form'` in which case the select routine will select records from a single table based upon the field and operator values submitted by a search form. The form must submit HTML inputs using the names of the tableModel_ config file inputNames. Operators can be defined with inputs named by using those same inputNames with the literal `'Op'` appended to the name (so an input named `m_firstName` would have an operator input named `m_firstNameOp`). Supported operator values include eq, bw, ew, cn, lt, lte, gt, gte, and btw (between) with a ::: separating the two values.

Second, it can be a literal string of a `WHERE` clause (without the word `WHERE`).

Third, it can be a literal string of `'lib```filename.lgc'` where filename.lgc is a file in the site or module /libs/ folder. The file will be `[process:]`ed which must result in defining a local variable named `#fw_actnWhere` that contains a complete SQL string for a `WHERE` clause.

- `-orderby` (optional, string) = a clause to be used verbatim in the query `ORDER BY` clause.
- `-groupby` (optional, string) = a clause to be used verbatim in the query `GROUP BY` clause.
- `-sort` (optional, string) = used in place of an -orderby to use a PageBlocks style sort string of `'fieldName```ASC|DESC```sort_description_to_display'`.
- `-limit` (optional, string) = can be set to a verbatim query string, or set to just a single integer to indicate the maximum returned records. The latter is the most usual case, and in fact, the parameter is usually not needed as the the environment variable `$fw_genlListMax` is used as a default value.
- `-withFoundCount` (optional, no value) = indicates that a `foundCount` valude should be determined. Making this an explicit function saves the cost of extra queries where a found count is not needed.
- `-quiet` (optional, no value) = indicates an error should *not* be reported if no records are found
- `-withMakeVars` (optional, no value) = indicates that if a single record is found, fields retrieved will be converted to page variables named by the inputName specified in the tableModel_ config file. The foundcount value must be 1 in order for this to be applied.
- `-asRecordsArrays` (optional, no value) = records to be returned in the `->'records'` instance var as an array of arrays (essentially the same thing as records_array). The field array will be in the same order as the fields in the `SELECT` clause of the query. (`-withRecordsArrays` is deprecated)
- `-asRecordsMaps` (optional, no value) = records to be returned in the `->'records'` instance var as an array of maps. The map keys will be the same as the fields in the `SELECT` clause of the query. (`-withRecordsMaps` is deprecated)

Public Method –>add

Inserts a record identified by the keyfield value. The sequence of events will include triggering the validation tags, inserting the record, logging the action (if turned on), retrieving confirmation data, and of course trapping any errors to report.

Supports the following instance vars:

- `inlineName`
- `inputsAreInvalid`
- `rulesAreInvalid`
- `error`
- `errors`
- `queryString`
- `queryTime`
- `tagTime`

Accepts the following parameters:

- `-keyval` (required, string) = the keyfield value
- `-inputs` (required, string) = a comma separated list of the tableModel_ config file inputNames that are allowed to contribute to the insert query. This prevents field injection into queries by submitting an altered set of fields.
- `-confirmFields` (optional, string) = a list of fields to be used verbatim in a `SELECT` query to retrieve fields after the add is completed. This provides an opportunity to provide a confirmation message using data directly from the added record.
- `-withMakeVars` (optional, no value) = indicates that fields retrieved by the `-confirmFields` step (if used) will be converted to page variables named by the inputName specified in the `tableModel_` config file.
- `-withoutLog` (optional, no value) = instructs the tag to not log action even if logging is on.

Public Method –>duplicate

Selects an existing record identified by the keyfield value, and inserts a copy of it with a new key value. The sequence of events will include selecting the source record, verifying the proposed key value does not exist, inserting the new record, logging the action (if turned on), retrieving confirmation data, and of course trapping any errors to report. Duplicating a record does not invoke any validation routines.

Supports the following instance vars:

- `error`
- `errors`
- `queryString`
- `queryTime`
- `tagTime`

Accepts the following parameters:

- `-keyval` (required, string) = the keyfield value of the source record to be copied
- `-newkeyval` (required, string) = the keyfield value of the new record to be created

Public Method ->showMsgsFor

`->showMsgsFor` : extracts all messages from `'errorMsgs'` of `->'validationResults'` for the specified input passed as an unnamed parameter

Public Method ->errorExistsFor

`->errorExistsFor` : returns true or false for the specific input if an error exists

Public Method ->insertAppError

`->insertAppError` : manually inserts a pair into `'errorMsgs'` of `->'validationResults'`. This would normally be used by business rule routines in `->validate`.

subclassing fwp_recordData

Developers are encouraged to use `fwp_recordData` as first generation instances or as a parent class for customized application-specific domain models.

Subclassing the `fwp_recordData` ctype allows the developer to create customized validations, and to write custom methods and instance vars to represent application-specific needs not covered by the basic built-in tags. A simple example of a subclassed ctype would be:

```
    define_type: 'myModel', 'fwp_recordData';

    define_tag:'validate';
        if: $m_expDate < date;
            (self->'rulesAreInvalid') = true;
        /if;
    /define_tag;

    define_tag:'defaultExpirationDate',
        -required = 'productType';

        select: #productType;
        case: 'fruit';
            return: date_add: date, -days = 5;
        case: 'veggie';
            return: date_add: date, -days = 10;
        case: 'frozenveggie';
            return: date_add: date, -days = 180;
        case;
            return: date_add: date, -days = 7;
        /select;
    /define_tag;
/define_type;
```

Having written this subclass, all the `fwp_recordData` vars and methods are still available. So, from here, you could do the following:

```
var:'foodItem' = (myModel:'foodItems');

$foodItem->(select: -quiet, -keyval = $thisRecord);
```

The tableModel Config File

Here's an example from the standard PageBlocks user authentication system. serverType maps to a specific database server and is used to determine which tableConnector to use to generate query syntax. The rest of the fields should be obvious.

```
#! dbTbl5
# filename = tblDefn_fwpuserauth.cnfg

{{serverType___    [$fw_gDbServerTypes->find:'site'] }}
{{databaseName___  [$fw_gDatabases->find:'site'] }}
{{tableName___     [$fw_gTables->find:'userauth'] }}
{{keyfield___      rcrdNo }}
{{lockField___     rcrdLockID }}
{{tableModel___

#inputName     fieldName         dataType   validation codes
#------------  ----------------- ---------  -----------------------------
m_rNo          rcrdNo            string
m_rNewDate     rcrdCreated       string
m_rNewBy       rcrdCreatedBy     string
m_rModDate     rcrdModified      string
m_rModBy       rcrdModifiedBy    string
m_rApv         rcrdStatus        string
m_rLok         rcrdLock          string
m_rLokid       rcrdLockID        string
m_rLoktm       rcrdLockTime      string
m_rLokown      rcrdLockOwnr      string

m_rSessTime    sessionTime       string
m_rSessVars    sessionVars       string
m_rSessKprs    sessionKeepers    string
m_rSessProf    sessionProfile    string

m_uNameF       userNameFirst     string     isRequired, hasLabel=First Name
m_uNameL       userNameLast      string     isRequired, hasLLabel=Last Name
m_uOrg         userOrg           string
m_uPhone       userPhone         string     isNumeric
m_uEmail       userEmail         string     isRequired, isEmail
m_uPw          userPswd          string
m_uHint        userHint          string     isRequired

m_uIns         userLogins        integer
m_uLastIn      userLastLogin     string
m_uTries       userAttempts      integer
m_uLokTime     userLockTime      string
m_uPwNew       userPswdNew       string
m_uPwDate      userPswdCreated   string
m_uPwHist      userPswdHistory   string
m_uHosts       userHosts         string
# these are inputs that are part of the model, but not part of the table
# these are usually interim form inputs used to collect pieces of what
# ultimately is inserted into a single field
# the purpose for including hem here is to declare validation of form inputs

i_uPhoneA      -                  -          isRequired, isNumeric, hasLength=3
i_uPhoneP      -                  -          isRequired, isNumeric, hasLength=3
i_uPhoneN      -                  -          isRequired, isNumeric, hasLength=3
i_uPhoneX      -                  -          isNumeric, hasMaxLength=5

i_up1          _                  _
i_up2          _                  _
}}
```

Examples

The below will generate `$m_uNameF`, `$m_uNameL`, and `$m_uEmail` for the record with a keyfield value of `'abcdefg'`.

```
var:'myUser' = (fwp_recordData:'pbuserauth');
$myUser->(select:
    -keyval = 'abcdefg',
    -select = 'userNameLast, userNameFirst, userEmail',
    -withMakeVars);
```

If a form has these inputs:

```
<input ..... name="m_uLastIn" ..... />
<input ..... name="m_uLastInOp" ..... />
<input ..... name="m_uNameL" ..... />
<input ..... name="m_uNameLOp" ..... />
<input ..... name="m_uIns" ..... />
<input ..... name="m_uInsOp" ..... />
```

and upon submission, we got the values of:

```
$m_uLastIn      = '2005-06-15'
$m_uLastInOp    = 'lt'
$m_uNameL       = 'M'
$m_uNameLOp     = 'lt'
$m_uLogins      = '10'
$m_uLoginsOp    = 'gte'
```

and, we used the following code:

```
var:'userList' = (fwp_recordData:'pbuserauth');

$userList->(select:
    -inputs = 'm_uNameL, m_uIns',
    -select = 'm_uNameL, m_uNameF, m_uEmail, m_uIns',
    -where  = 'form',
    -withRecordsMaps);
```

then this query would automatically be constructed:

```
SELECT userNameLast, userNameFirst, userEmail, userLogins
FROM fwpdemo.fwpuserauth
WHERE userNameLast < 'M' AND m_uLogins >= 10
```

You'll notice that the query does not include the inputs for $m_uLastIn. Why? It was not included in the `-inputs` parameter. This is the safeguard against people rewriting a form to force the hand of the automated query generator. You'll also notice that the Select clause can use the input names. Either the input names or the real field names can be used anywhere in a query.

Once our query has executed, we would have the following instance vars available:

```
$userList->'foundcount'  = 39            // or whatever the case may be
$userList->'showFirst'  = 1
$userList->'showLast'  = 39              // or whatever the case may be
$userList->'inlineName'  = 'wh7snhjw9'   // or some other ID
$userList->'records'  = an array of maps*
$userList->'error'  = false
$userList->'errors'  = (array)           // empty array
$userList->'querystring'                 // the above query
$userList->'queryTime'  = 3              // milliseconds or whatever the case may be
$userList->'tagTime'  = 10               // milliseconds or whatever the case may be

* records are in arrays, fields are map pairs
  so, we'd have a 39 element array where each array member was
  a map with four keys of userNameLast, userNameFirst, userEmail, and userLogins
```

To lock a record and retrieve data for an update form:

```
1   var:'myUser' = (fwp_recordData:'pbuserauth');
2
3   $myUser->(getRecordUsingLock:
4       -select = 'userNameLast, userNameFirst, userEmail, userLogins',
5       -keyval  = $selectedUserID,
6       -withMakeVars);
7
8   var:'myLock' = $myUser->'lock';  // will be empty if lock attempt faileded
```

Then to submit the update for that record:

```
9   var:'myUser' = (fwp_recordData:'pbuserauth');
10
11  $myUser->(updateUsingLock:
12      -lockval  = $myLock,
13      -inputs   = 'm_uNameL, m_uNameF, m_uEmail');
14
15  // we're not updating userLogins,
16  // but we fetched it above in line 4 to show in the form as static text
```

Data Type Casting

The tableModel_ configuration file includes a column for defining the Lasso data type associated with a database column. The dataType column is used to cast incoming and outgoing data passing through `fwp_recordData` into the desired Lasso data type.

Currently, the `fwp_recordData` ctype supports creating Lasso strings, decimals, integers, and dates. It will be possible to support arrays, maps, and other complex data types, and that will be worked on for a future release.

Data in Lasso variables heading into a database will be cast into the specified data type. Fields being read from a database and being converted to variables with a `-withMakeVars` option will have those variable cast as the defined Lasso data type when the variable is created.

Input Validation

The `fwp_validator` ctype provides data validation services for several built-in validation processes and for custom validation processes. It integrates validation processing and error message generation. The validator is used by `fwp_recordData` to automate validation for `INSERT` and `UPDATE` statements, but can also be used independently to validate forms that are not stored to data tables, cookies, web service requests, and any other source of data input into the application. The validator is promarily used for data format validation, but can also be used to perform business rule validations.

Let's first have a look at the API of the validator ctype, then look at how it is implemented for fwp_recordData, non-stored form data, and cookies.

fwp_validator API

We'll look at the built-in validation codes in detail further on, but for the purposes of having a reference point for this section, a validation is a short string describing the validation task such as isRequired which means the data value cannot be empty, or isAlphaNumeric which means the value can be comprised only of letters and numbers. In the API descriptions below, these strings are called validation codes.

Interface Quick Glance

An instance of fwp_validator is created automatically as `$fw_validator` during the startup of each page. Generally, you would use this instance for all validation activities.

Read-Only Instance Variables:

```
'coreCodes'
'appCodes'
‘errorMsgs’
```

Public Member Tags:

```
->validate
->insertErrorMsg
->errorExistsFor
->showMsgs
->getResults
```

Public Read-Only Instance Vars

- `->'coreCodes'` : an array of pairs which includes the input name and code of pre-defined validations that failed. Empty if all validations passed. An example pair could be `('firstName' = 'isRequired')` would mean that an input `firstName` had a validation code of `isRequired` that failed (meaning the input was empty and is not allowed to be).

- `->'appCodes'` : an array of pairs which includes the input name and code of app-specific custom validations that failed. Empty if all validations passed. An example pair could be `('zipCode' = 'usZip4')` would mean that an input `zipCode` had a custom validation code of `usZip4` that failed (presumably meaning the input did not follow the prescribed format for a US zip code in the zip+4 format).
- `->'errorMsgs'` : an array of error messages where the key is the input name, and the value is a an error message for that input. See the section on *Validation Error Messages* for details.

Public Method ->validate

Initiates validation processing of standard and custom validation codes. Returns boolean true when an error has occurred. Returns a boolean false if all validations passed.

Accepts the following parameters:

- `-formSpec` (optional/required, string) = either `-formSpec` or `-inputs` has to be provided. The param `-formSpec` (short for form specification) is generally used for validating inputs from POST, GET, cookies, or any other type of inputs that need to be validated outside of forms handled by `fwp_recordData`. It's a map where the keys are input names, and the values are a comma separated list of validation codes to process. See the section *Using fwp_validator with POST, GET, and Cookies*.
- `-inputs` (optional/required, string) = either `-inputs` or `-formSpec` has to be provided. `-inputs` is generally used in conjunction with `fwp_recordData`. It's either an array or a comma separated list of input names. These are the inputs that will processed to applicable validation rules.
- `-valCodes` (required, string) = required if `-inputs` is used, not required with `-formSpec`. This is pretty much used exclusively by `fwp_recordData` to pass the validation codes of all inputs for a given tableModel. `-inputs` provides a list of which inputs to validate out of the complete tableModel definition.
- `-usingPostForm`/`-usingGetForm`/`-usingPairs` (required, no value) = one of these three is required to declare the source of the inputs. `-usingPostForm` will search for inputs andvalues from `client_postParams`. `-usingGetForm` will search for inputs andvalues from `client_getParams`. The option `-usingPairs` allows the use of any arbitrary array of pairs as if it were `client_postParams`. This enables data from a cookie, or fields from an XML or JSON document passed to a web service request to be extracted and validated prior to acceptance by the application.

Public Method ->showMsgs

`->showMsgs` : extracts all messages from `'errorMsgs'` for the specified input passed as an unnamed parameter

Public Method ->getResults

`->getResults` : copies the `coreCodes`, `appCodes`, and `errorMsgs`, into a map with those keys and returns the map so the validation results can be copied to a new object allowing the validator to be used again on a different set of inputs.

Public Method ->errorExistsFor

`->errorExistsFor` : returns true or false for the specific input if an error exists

Public Method ->insertErrorMsg

`->insertErrorMsg` : manually inserts a pair into `'errorMsgs'` which would be used by custom validation routines

Using fwp_validator on POST, GET, Cookies, and More

For the most part, `fwp_validator` is used automatically by `fwp_recordData` to validate form data headed to a database. However, it can also be used on its own to validate forms that pass data to outside web services, or to validate POST or GET forms coming into a web service your app contains. It can even beused to validate data from cookies, XML, JSON, and other data containers that are passed into your application.

For example, let's say you had a small form to send a search request to a web service. The form includes a search string and a maximum number of results value. To validate the form prior to sending the request to the service, you would do this:

```
var:'searchValidations' = map;
var:'inputsAreInvalid' = false;

$searchValidations = (map:
    'searchString' = 'isRequired, hasMaxLength=128',
    'maxResults'   = 'isPositiveInteger');

$inputsAreInvalid = $fw_validator->(validate:
    -usingPOSTForm,
    -formSpec = $searchValidations);

if: !$inputsAreInvalid;
    include_url:.......
/if;
```

What this code fragment does is declare a map where the keys are the HTML input names, and the values are the PageBlocks validation codes (including custom codes). These are submitted to the `$fw_validator` object which returns `true` if there's a validation error. You must specify whether the incoming data will be from POST (`-usingPOSTForm`) or from GET (`-usingGETForm`).

Let's look at an example suitable to a cookie or JSON data passed in from a web service request. In each of these cases, you would first parse the cookie or JSON data into an array of pairs representing each data element name and its value. So, if we had a cookie formatted with some data like this:

```
shape-circle/color=cc9900
```

we would break that into a data structure like this:

```
$cookieData = (array:
        ('shape' = 'circle'),
        ('color' = 'cc9900'));
```

Next, we'd have a `-formSpec` of validation specifications that might look like this:

```
    $shapeCookieValidations = (map:
        'shape' = 'isRequired, isAlpha, hasMaxLength=24',
        'color' = 'isRequired, isHTMLColor');
```

And before using the cookie data we'd perform a validation like this:

```
    $cookieIsInvalid = $fw_validator->(validate:
        -usingPairs = $cookieData,
        -formSpec = $shapeCookieValidations);

    if: $cookieIsInvalid;
        ..... cookie is contaminated, so .....
        ..... revert to some default values .....
    /if;
```

The same steps could be used to verify field elements in XML or JSON data submitted to a web service, or any source of data that is being injested by the application.

The first rule for web application security is to trust nothing that comes into the application, and the validator helps us implement that rule.

Input Validation with fwp_recordData

As you've seen in the tableModel_ configuration file, the last column defines validation codes (often referred to as vcodes or valcodes in the source code) for the form input variable. When `fwp_recordData` is used to `INSERT` and `UPDATE` records, an internal `->validateInputs` tag is automatically processed. That tag in turn calls the `$fw_validator->validate` tag which verifies that each input being processed for an `INSERT` or `UPDATE` meets the validation requirements defined in the tableModel_ config file. Refer to the *Standard Validation Codes* tag section below for the details on available codes.

There are two methods to incorporate custom validations. One is to create custom validation codes to be specified in the tableModel_ config file. The other is to subclass the `fwp_recordData` ctype and include a member tag called `->validate` which will be called automatically. The former is best suited to input format validations similar to the standard codes, and the latter method is more appropriate to processing business rules.

Subclassing fwp_recordData for Business Rule Validations

In some cases, writing custom validations as described above may not be flexible enough to satisfy business rules (a.k.a. domain rules) requirements for a given data model. In such cases, it is more effective to subclass the `fwp_recordData` ctype, and override the `->validate` member tag to perform the needed business rules processing. The `>validate` member tag will be processed separately from the `>validateInputs` member tag which does the basic validation codes.

Standard Validation Codes

The following validation codes are built into the validator. The short code is the original codes used prior to 5.1.3, and the long codes are the preferred codes since 5.1.3.

•`a | isAlpha`	alpha string—value must contain only a–z and A–Z
•`n | isNumeric`	numeric string—value must contain only 0–9
•`d | isDecimal`	decimal—value must contain 0–9, period, hyphen only
•`i | isInteger`	integer—value must contain 0–9 and hyphen characters only
•`ipos | isPositiveInteger`	integer—value must contain 0–9 characters only
•`an | isAlphaNumeric`	alphanumeric string—value must contain only characters a–z, A–Z, and 0–9
•`aspc | isAlphaSpace`	alphaspace string—value must contain only a–z, A–Z, and space only
•`anu | isAlphaNumericUnderscore`	alphanumeric underscore string—value must contain only a–z, A–Z, 0–9 and underscore only
•`anh | isAlphaNumericHyphen`	alphanumeric hyphen string—value must contain only a–z, A–Z, 0–9 and hyphen only
•`ans | isAlphaNumericSymbol`	alphanumeric symbol string—value must contain only a–z, A–Z, 0–9, and characters from !@#$%^&*
•`anspc | isAlphaNumericSpace`	alphanumeric space string—value must contain only a–z, A–Z, 0–9, and space
•`req | isRequired`	required—value must contain one or more non-whitespace characters, it cannot be of length 0
•`vreq | isRequiredSelection | hasRequiredSelection | requiredSelection`	required—value list selection must contain one or more non-whitespace characters, it cannot be of length 0
•`nospc | hasNoSpace | hasNoSpaces`	no space—value must not contain any whitespace
•`eq= | isEqualTo | hasExactValue`	equal to—the value must be equal to the value of a specified form input name
•`regex= | matchesRegex`	regex—the value must match the regex pattern specified
•`min=N | hasMinValue`	minimum numeric value—must be ≥ N, where N is cast to a decimal for comparison

•max=N | hasMaxValue	maximum numeric value—cannot be greater than N, where N is cast to a decimal for comparison
•len=N | hasLength | hasExactLength	exact string length—value must be exactly N characters in length
•minlen=N | hasMinLength	minimum string length—must be ≥ to N (integer)
•maxlen=N | hasMaxLength	maximum string length—must be ≤ to N (integer)
•email | isEmail | isEmailAddress	email address—must contain an @ symbol, at least one period after the @, and one letter in each segment)
•cc | isCreditCard	valid credit card number (uses the Lasso tag Valid_CreditCard)
•link | isLink | isWebLink	href link—a safe URL (does not contain the word javascript:)
•date | isDate	date—accepts numerous date formats and converts the date to a string in the format of yyyy-mm-dd. Use the =usaDate option or =euroDate option to force the interpretation as a specific format
•pw | isPassword	password—a macro for req.an.nospc.lenmin=6
•pws | isPasswordStrong | isStrongPassword	strong password—min length is 8, requires one A–Z, one a–z, one 0–9, one symbol from !@#$%^&*
•day | isDay	numerical day of the month—a macro for i.min=1.max=31
•month | isMonth	numerical month number—a macro for i.min=1.max=12
•year | isYear	four digit year in the range of 1900-2020—a macro for i.min=1900.max=2020
•label= | hasLabel | usesLabel	friendly field name—a string to display in the error message to identify the form input. Will be processed it if starts with [
•notrim | isNotTrimmed | doNotTrim	prevents the removal of whitespace from the beginning and end of the input value (normally, all values are trimmed)
•htmlok | isHTML | hasHTML | usesHTML	prevents the conversion of angle brackets into entities (normally all angle brackets are converted to prevent HTML injection)
•scriptok | isJavaScript | hasJavaScript | usesJavaScript	prevents javascript: and <script from being converted to entities (normally the colon after javascript and the angle bracket before script are each converted to an entity, even if -htmlok is enabled, to prevent JavaScript injection)

Custom Validations

Custom validations are implemented using a fairly simple custom type to contain the logic necessary to process the validation required. These app-specific validations can be written to be site-wide or module-specific to support the modular nature of the PageBlocks framework. It is recommended that all custom validation codes be prefixed by a hyphen to prevent unintentional collision with any future extensions to the standard codes.

Each validation code is implemented as a single member tag of the custom type. The basic structure of a validation custom type looks like this:

```
 1  <?lassoScript
 2  define_type:'validator_appSite';      // or 'validator_appModule'
 3  define_tag:'-custom',
 4      -required = 'inputName',
 5      -required = 'inputValue',
 6      -optional = 'inputParam';
 7
 8      local:'inputIsValid' = false;
 9
10      if: #inputValue == 'testing';
11          #inputIsValid = true;
12      /if;
13
14      return: #inputIsValid;
15
16  /define_tag;
17  /define_type;
18  ?>
```

In the listing above, a validation code of `-custom` is implemented as a member tag starting at line 3. Each validation code member tag must have lines 4–6 defining the required and optional parameters, and line 8 which sets the default value of the local variable `inputIsValid` to `false`. After that, each validation can perform whatever transformation is needed on the input value, and whatever conditional tests are needed to meet the validation requirements. If all conditionals are satisfied, `#inputIsValid` is set to `true`. Finally, `#inputIsValid` is returned. The `$fw_validator` object will handle the integration of that response into the error messages if necessary. Below is a more complete examples of a custom validation.

```
define_type:'validator_appSite';

//  california zip code validation, sort of
define_tag:'-zipca',
    -required = 'inputName',
    -required = 'inputValue',
    -optional = 'inputParam';

    local:'inputIsValid' = false;

    if: (integer:#inputValue) > 90000
        && (integer:#inputValue) < 96999);

        #inputIsValid = true;

    /if;

    return: #inputIsValid;

/define_tag;

//  cheesy zip+4 format test
define_tag:'-zipus4',
    -required = 'inputName',
    -required = 'inputValue',
    -optional = 'inputParam';

    local:'inputIsValid' = false;

    if: #inputValue->(contains:'-');
        local:'zipA' = (#inputValue->split:'-')->get:1;
        local:'zipB' = (#inputValue->split:'-')->get:2;

        if: (integer:#zipA >= 10000)
            && (integer:#zipA <= 99999)
            && (integer:#zipB >= 0000)
            && (integer:#zipB <= 9999)
            && (string:#zipB)->size != 4;

            #inputIsValid = true;
        /if;

    return: #inputIsValid;

/define_tag;
/define_type;
```

The `inputName` will be the name attribute of the form input. The `inputValue` will be the value of the form input. The `inputParam` will be the contents of the second part of the pair that was used to declare the error. So, if `$fw_error->(insert:'5610'=$errInfo)` was used to declare the error, then the value of `$errInfo` would be the value of `#inputParam`. As with all Lasso custom tags, the values are passed by reference, so you may need to consider copying the inputs in some cases.

Custom Validation Messages

Having coded your custom validations, where do you define the error messages to display back on the form? The multi-view strings system is used for that. For site-wide validations, you'd use /site/strings/strings_appValErrors_en-us.cnfg (for whatever language), and for module-specific validations you use /{moduleName}/_resources/strings/strings_appValErrors_en-us.cnfg (again, for whatever language). Refer to the *Multi-View Strings* chapter for more details.

With each message, the local variables `#msgFirstWord`, `#inputLabel`, and `#inputParam` are all available, where `#msgFirstWord` is going to be a literal string of This or The simply as a start to the message, `#inputLabel` will be the value of a `label=Friendly Name` code from the tableModel file, and `#inputParam` is the `pair->second` value of the error declaration that was also passed to the validation member tag.

Message definitions for the example validations above might look like this:

```
{{-zipca:
constant___fw_kValNotZipCA
msg___ [#msgFirstWord] field [#inputLabel] is not a California Zip code.
}}
{{-zipus4:
constant___fw_kValNotZipUS4
msg___ [#msgFirstWord] field [#inputLabel] is not a Zip+4 code.
}}
```

If you know there will be no `label=` definition, then the `#inputLabel` can be left out, and the `#msgFirstWord` can be hard coded. However, it is probably good form to use the above structure so that if the error message is ready in case a label is defined.

Logging

The PageBlocks framework includes integral logging in three key areas: program errors, access to authentication-required pages, and execution of database queries that modify the database (adds, updates, deletes). There is also a generic log tag that can be used to create multiple custom logs. Logs can written as tab delimited disk files with rollover options, or can be written to database tables. Control of how logs are written is found in the _initMasters file.

There are four log tags: `fwpLog_auth`, `fwpLog_data`, `fwpLog_err`, and `fwp_logCustom`. Each of these is pretty much integrated to where the developer does not need to implement them except to set the setup variables in `_initMasters`. The `fwpLog_auth` tag is integrated into the `fwp_user` custom type, the `fwpLog_data` tag is integrated into the `fwpActn` tags and types, and the `fwpLog_err` tag is integrated into the error manager.

Until these docs get expanded more, you can refer to the online reference for the details about what each log stores. Each tag works the same way. In the mode that the tags write to a disk file, that writing is done inline with the page code. In the mode where they write to a database, the data is passed to the `fwpLog_asyncSQL` tag where a standard Lasso inline is triggered as an `-async` process.

Config File Tools

A few of the PageBlocks framework tools depend on configuration files. Several tags have been created to make loading and parsing of configuration data easier to work with. There's two kinds of tags in the `fwpCnfg` tag set. One type simply processes an input to a desired output. The second type serves more or less as a macro for a common set of steps, and typically involves loading the file from disk and set of process steps.

For example, the `fwpCnfg_splitLines` tag takes a line delimited string and converts it into an array. The `fwpCnfg_decomment` tag takes an array and remove lines which are empty, begin with `#` or `//`, or contain the string `output_none` in it. The `fwpCnfg_loadLines` tag loads a file (expecting it to be line delimited), splits the text into an array, and decomments it all in one tag. This way there's an assortment of steps available individually to use them where you need them, but also some all-in-one tags to make quick work of typical config file formats.

The descriptions below summarize what each tag does. More complete details are available in the online reference.

The following tags are available in the fwpCnfg tag set:

`fwpCnfg_deComment`—takes an array of items (typically retrieved from a line delimited config file) and removes items deemed non data such as empty items, items that begin with # or //, and items which contain an "output_none" string.

`fwpCnfg_loadFile`—retrieves a disk file of a specified name from the /site/configs/ folder and/or the /module/_resources/configs/ folder. If a file is found in both locations, the module file is appended to the site file unless the `-withoutMerging` option is specified.

`fwpCnfg_loadLines`—uses the `fwpCnfg_loadFile` tag to load a file, uses `fwpCnfg_splitLines` to normalize line endings and split the file at line endings, then uses `fwpCnfg_deComment` to remove non-data lines from the array. Includes an optional `-removeWhiteSpace` command to eliminate spaces and tabs which allow the files to be formatted for easy reading, but need to be removed as non-data. Also includes a `-withoutCaching` option to preventthe file from being added to the internal config file cache (which would prevent it from accessing the disk file next time).

`fwpCnfg_loadPairs`— uses the `fwpCnfg_loadFile` tag to load a file, uses `fwpCnfg_splitLines` to normalizes line endings and split the file at line endings, then uses `fwpCnfg_deComment` to remove non-data lines from the array.
Finally, uses `fwpCnfg_makePairs` to split each line at = or === to create an array of pairs. The use of === as the delimiter allows the value to contain Lasso code which could be processed. You'll see that capability used in the listTbl_ config files. Includes a `-removeWhiteSpace` command to eliminate spaces and tabs which allow the files to be formatted for easy reading, but need to be removed as non-data. Also includes a `-withoutCaching`

TIP: Something you may find useful for your own custom config file formats is the method PageBlocks uses to parse multipart files. Have a look at the tableModel_ configuration file format. You'll notice several sections delineated within {} braces. Each section begins with a `{somename___` string. In config files that have sections like this, the following regex will grab all text inside that section:

```
var:'x' = (string_findregexp:
   #fw_tableDefn,
   -find='{serverType___([^\\}]+)\\}')->last;
```

option to preventthe file from being added to the internal config file cache (which would prevent it from accessing the disk file next time).

fwpCnfg_loadProcess— uses the fwpCnfg_loadFile tag to load a file, then simply applies the process tag to the file contents. The main purpose is to allow a file to exist in either the /site/configs/ or module /_resources/configs/ level and let the tag find it.

fwpCnfg_loadVars— uses the fwpCnfg_loadFile tag to load a file, uses fwpCnfg_splitLines to normalizes line endings and split the file at line endings, then uses fwpCnfg_deComment to remove non-data lines from the array. Finally, uses fwpCnfg_makeVars to split each line at = or === to create page variables or locals. Includes a -despace option, but it only removes spaces surrounding the = or === delimiters to allow for convenient formatting. Vars are creatred as page variables by default, but can be created as locals to use inside ctags and ctypes with the option -asLocals | -intoLocals. Also includes a -withoutCaching option to preventthe file from being added to the internal config file cache (which would prevent it from accessing the disk file next time).

fwpCnfg_makeVars—accepts either an array, string, or bytes as an input and splits the string or bytes type into an array, then takes each array item in the format of x===y or x=y and creates vars or locals named x of value y. Will also take a comma separated list of variable names and create vars or locals of empty strings.

fwpCnfg_splitBlocks—accepts a {{ }} delimited block of items where each block contains a name and a data set. Returns a map of the name-data pairs (or an array is -asArray is used). The data of each block is then usually further processed into vars or locals or something.

fwpCnfg_splitComma—accepts a comma delimited string of items, eliminates the white spaces around commas, then splits it into an array. A -dontSplit option prevents the split so the tag can be used simply to normalize the comma delimiters.

fwpCnfg_splitLines—accepts a line delimited string and normalizes line endings to \r before splitting into an array. A -dontSplit option prevents the split so the tag can be used just to nomalize line endings to \r. Normally lines are trimmed unless the -dontTrim option is used.

fwpCnfg_splitPairs—accepts either an array, string, or bytes as an input and splits the string or bytes type into an array, then takes each array item in the format of x===y or x=y and creates an array of pairs (or a map if -asMap is used).

fwpCnfg_splitTabs—accepts a tab delimited string of items and splits it into an array. Includes a -removeExtraTabs option which converts sequential tabs into a single tab. This can be used on text that is tabbed to form readable columns, but where the tabs are non-data. Also includes a -dontSplit option to prevent the split so the tag can be used simply to normalize the tab delimiters. Generally, the -dontSplit would not be used unless the -removeExtraTabs option was also used.

User Management

Most modern web applications need a system for managing user access to either a portion of a site, or to pretty much the entire site. If all users have equal access and equal privileges within the site, then creating a system to manage that is not too difficult. However, if there are differences in what people can see and do, then the task can become complex. The PageBlocks user management framework provides ready-to-use tools and sample code to meet diverse application needs.

In designing a user management system, there are several distinct areas of need to address. First, we have the need to authenticate a user. That is, how do we establish who a user is, and how do we ensure the user really is who he says he is. Second, we have the need to authorize a user. This is the task of granting a user specific permissions to see specific things and perform specific tasks. Third, we need some security measures to minimize the prospect for unintended access. Fourth, we need tools to help us administrate all these tasks. The PageBlocks framework provides resources for all four of these areas.

The foundation of the PageBlocks user management system is the `fwp_user` custom type. This ctype is joined by a pre-defined data model for user profile and permissions, a configuration file for declaring application-specific authorization attributes, and an admin module starter kit to provide a customizable turn-key solution to user admin. This system is quite flexible and extensible, and should be suitable for many types of applications. The tools can be adapted to either stay out of the way for simple systems, or to step up and handle rather complex access and security needs.

An advantage to using this system is the built-in integration with the page assembly process which does most of the work for reauthenticating users between pages, and handling page-wide authorization access. In the pageConfig declarations, a page is simply flagged as requiring authenticated access, and the specific permission required for overall page access is defined. From there, the page initialization code and templateLoader automatically handle acquiring those permissions and controlling the granting or denial of access.

User Management Capabilities

Let's look at the overall capabilities first before we dive into some implementation details. We'll do that by looking at the four areas of authentication, authorization, security, and administrative management.

Authentication

Authentication is performed using two pieces of information. An account name and a password. When the user ctype is instantiated, one of the parameters is to declare which field contains the account name. The account is not hard coded to be an email address, or anything else in particular, however, if the account name parameter is not defined, it will be defaulted to the email address field. The authentication process is described in more detail in the security section, as most of the details are security oriented.

Authorization

Each user is defined with three distinct groups of information. First are profile attributes which provide information about the user. Second are permissions attributes which dictate what the user is allowed to do. Third are data access filters which provide a flexible means for the application to dynamically alter data queries based on the user to limit the scope of records the user will see. While he permissions attributes will be the primary means of authorization, in theory, all three of these information sets can be used by the application as needed.

User Profile Attributes

Profile attributes are information that tell us who the user is, and various things about the user outside of the scope of the application. At a minimum, this would include authentication info, and could additionally include data like contact info, personal preferences, etc. The PageBlocks system allows multiple profile data tables to cover user profiles which may be very different. For example, a school system might have separate tables for students and teachers as the profile data needed for each could be quite different. The `fwp_user` ctype adapts to multiple configurations as needed by allowing separate config file definitions for each user type and a declaration of which profile table (and even permissions and filters tables) to use when the object is instantiated.

User Permission Attributes

The permissions attributes is a data structure of discrete Y/N data points that indicate what the user is allowed to do within the application. The Y/N strings were used instead of 1/0 to avoid holes caused by data type mismatches, and to make the code a little easier to read.

As to what each of these data points represents is defined by the developer. To make that as painless as possible, a simple configuration file is used to declare a matrix of permissions. These permissions can be defined in arbitrary sets. Each set can have arbitrary permission points. For, example, a web site that has a news module may want to provide an online method for multiple users to write and post news articles. We already have two potential permission sets. One is that we need a set that says who is allowed to define users in the first place. Second, we need a set that says whether a user can post news stories.

Within the news set we might want to distinguish between the ability to create a new article and the ability to approve that article to show up online. How about updating or editing other users' articles? Deleting articles? Each of these abilities should be defined separately to give us the maximum flexibility.

Permission sets do not have to be aligned with modules. It's typically logical and convenient, but it is not an imposed requirement. Individual permissions can be set to govern something like the ability to see a person's phone number, or any other discrete data field. they can also be used to control whether a certain button or menu is presented to a specific user. Each permission can provide as broad or as narrow a scope as you want it to. You define each permission, and you decide with your own application code how that permission is interpreted.

You may be wondering about groups. This system intentionally shuns the concept of groups. In my experience, groups tend to cause limitations which create difficult to maintain workarounds in the application code. What I have seen trying to use groups is that there is always an exception to what individuals in a group can do. While the secretaries group may be assigned X, and Y, but not Z, inevitably there is one member in that group that should be assigned Z. So either the app code get hacked to allow that, or another group is created to cover the special circumstance. Do you remove that one person from secretaries and create a new group for just this person? Do you add that person to two groups? Both of these scenarios tend to generate orphaned rules which can be very difficult to clean up and which can lead to security holes. What I have found is that it is faster and more secure to maintain users where each users exact specific permissions are easily seen all at once, and if the application can define specific discrete permissions for greater control and security. I will admit that the downside to this system is that it is harder to make changes for large groups users all at once. This is counteracted by providing advanced tools which can be complex to write, but in my opinion that task is the lesser evil than the combined downsides of using a group based system.

User Data Access Filter Attributes

Third is the data access filters. Permissions are best used to define what a person can do. Data filters are used to dynamically limit the scope of what database records a person can view. Through the use of data access filters, database queries are written generically with the data filters filling in the blanks. A filter has four components: a table name, a field name, a matching operator, and the match value.

User File Access Attributes

One system not yet built-in, but is on the list of things to do is a system like the data access filters to control file path access. For systems which deal with stored documents, it could be advantageous to have user-specific rules for file access.

Security

There are several aspects to bolstering security within the user management system. Outside of following secure coding techniques in general, this section covers the areas of defending against login attacks, providing password management, and session management.

Now, the effectiveness of any security practice can be debated, and I'm not going to lay any particular claim that the PageBlocks framework is or is not secure. Nothing is 100% secure. Among the many things that have evolved to become a part of the framework, I have tried to pay attention to security issues, but even teams of experts leave holes, so security should be one area that people challenge the code base and remain diligent in looking for holes.

The goal of employing security-oriented practices is to make an attack difficult. The degree of difficulty should be increased with the increased value of the object being protected. However, if guaranteed security was cheap and easy, we'd all use it for even the least valuable things, yes? One of the goals of the PageBlocks framework is to provide tools that people may not take the time to create for themselves, but they'd use if available. That's where some of the security features fall. They're already built in, so you may as well employ them.

The Login Process

One of the primary concerns of controlling access is of course preventing people from simply guessing login credentials and gaining access. Part of this responsibility falls upon the user to use good passwords and properly protecting them, but the application should employ generally accepted defensive practices as well. These practices are largely seen in the series of steps that the login process goes through to authenticate a user. These are described below (not necessarily in the exact order they occur).

Defending Against Brute Force Attacks

The authentication process first looks for a match in the account name. If the account is found, the password is then verified. If the password is incorrect, the system increments a login attempt counter. When that login attempt counter reaches a specified value (defined in a config file), the account is locked for N minutes (N is defined in that same config file).

This technique helps defend against brute force login attacks where a machine rapid fires endless guesses of account name and password combinations. If a login system never locks out a user, the machine can fire away at that login page forever until a correct guess is made. Sounds improbable, but today's high end personal computers are reportedly able to perform thousands of attempts per minute with dedicated scripts that work directly with the HTTP protocol.

Note that the account is locked for N minutes. It is not locked until an admin user clears it. That is an option, but it should be used in rare cases. The reason is that an attacker can essentially create a denial of service attack on a site by attacking several accounts and locking them all up. Relying on an admin to clear all these (repeatedly?) poses a support problem. This scenario is not all that implausible when you consider a particular company that has an online app that uses an email address as the account name. It'd be pretty easy to construct an attack that tried fred@company.com and mike@company.com and so on. If the accounts clear themselves, the attack severity is lessened, but having a 10 or 15 minute lockout after 3 or 5 bad guesses severely hampers the attackers attempt to get into any one account via brute force.

Handling Disabled Accounts

Even if an account and password match are found, the login code checks several more things to ensure the account is enabled.

First, the record search includes the requirement that the field `rcrdStatus` contains a `Y`. If the record search succeeds, then it is also verified that only one record was found. User account and password combinations should of course be unique, but just in case, this check is used to prevent a security leak due to unexpected ambiguity in user records.

The next test checks whether the maximum number of failed login attempts has been reached and whether the lockout time has expired as was discussed above.

After that, password life cycle is checked to see if a password itself has an expiration date that has been exceeded. This is discussed more in the Password Management sections below.

Finally, we test if a user must be logging in from a known IP address and if indeed there is match. This is also discussed in detail in the section below.

Password Management

There are a number of configurable password management features. Each of these controlled from the authPswd_default.cnfg configuration file. The file itself is well commented to explain each setting. The following sections explain the overall features.

Quality Criteria

The user data type and configuration file allow for minimum password criteria to be defined which is then verified when creating new accounts, and when (if) users update their own passwords. The rules which can be enforced include minimum password length and individual inclusion of characters from one or more of the A-Z, a-z, 0-9, and !@#$%^&* sets.

Additionally, passwords and passphrases can be used based on user preference (not application requirement). A minimum passphrase length can be defined. This is the length at which the validation rules switch. For example, a setting of 20 would mean that once a string is at least 20 characters long, spaces become and legal the basic password character rules no longer apply. This allows your users to use which ever method they prefer, and there's no difference as far as the application is concerned.

Locking Users to IP Addresses

Another optional setting is to require that users log in from known IP addresses. If this feature is enabled, each user must have at least one defined IP address in their authentication record. More than one address is allowed. This is a universal setting in that it affects all users, and is not a use-specific setting.

Rotation and Reuse

Password rotation is the requirement that passwords be used for only short periods of time. Passwords are date stamped as to when they were created, and are allowed to be used for N days at which time the password must be changed. Another setting defines how many days in advance of the expiration that the user is informed the password needs changed. The framework sample code sets show how to use this feature to provide a change password page after the user logs in. The page can be cancelled by the user to postpone the new password.

An advanced extension to this idea is requiring that password not be reused. When this capability is enabled, each password is stored to a password history. As the user creates new passwords, each new one is compared to that user's password history to make sure it has not already been used. The configuration setting for this feature declares how many passwords to keep. Technically, this feature does have a loop hole for systems which allow users to redefine their password at any time. By simply changing the password N times in a row, the user could cycle around to reusing an old password right away. If the application allows changing of passwords, it may need some additional code to detect this attempt.

Another feature within this realm is a setting that requires a user to redefine their password upon the first use. When this feature is enabled, a user record which has been created manually in the admin system is flagged thatthe password entered by the administrator can be used only once. When the user logs in, a new password must be defined. The intent of this is to have the administrators blind as to what the user passwords are.

User Session Management

When a login succeeds, the user record is updated to include session criteria including a sessionID and a date/time stamp of when the last session refresh occurred (which starts with the login time). The user management system does not use Lasso's built-in sessions. The session control for a logged in user is built into the user custom type and the data table.

To propogate the session through multiple pages, the developer must include the sessionID in forms and links. This involves a little more attention to detail than using Lasso's session system, but it is intentional. Why? I'm not fond of Lasso's sessionID values which lack randomness. This system doesn't have the speed hit from the post processing which has to happen for Lasso to rewrite the page HTML to add sessionIDs to it. I think the developer should be in control of exactly when/where sessionIDs are propogated. I think session expiration should be part of the user's actual record not dictated by a cookie which may be compromised.

This systems provides the option to change the sessionID with every page for higher security applications.

The convention for storing and passing around the sessionID value is to use a form parameter and page variable `$fw_s`. This variable is created whether there is a sessionor not, so it is always OK to use the $ syntax. When included in forms or links, the value will be automatically restored, and the page assembly process will automatically re-authenticate the session, checking it first for expiration. So, use something like this to propogate the session:

```
<input type="hidden" name="fw_s" value="[$fw_s]" />

<a href="/mypath/mypage?fw_s=[$fw_s]>Some Link</a>
```

When the session is first created from a successful login, the user object is created with its profile, permissions, and data access information loaded. That object is then serialized and stored to the `userProfile` field of the `userauth` table. As the session is re-validated with each page, the user object is restored by unserializing the `userProfile` field. This is faster than going through the rebuilding of the user object from scratch as was done initially.

User Session Data

The user record can also be used for storing session data. There are actually two storage fields. One is for storing data that should be cleared with each new login. The other is for storing data which should survive across multiple logins and is to be cleared only when the application dictates it. These are called `sessionVars` and `sessionKeepers`. The session data storage works only for logged in users. Currently there is no system for non-logged in users, and until there is one, it is presumed that developers will use Lasso's built-in sessions for that.

Session data is manipulated by first declaring page variable names that are to be added to the session. This can be done at any point in the page. At the end of the page (or at any stage in the page that is deemed critical), a store command is used to write the data. This store command is already in the fwpPage_wrapup.lgc file to take care of storing sesion vars when there is an active session. Typical syntax would look like the following:

```
$fw_user->(addVars:'varName1, varName2'); // adds var names to a list to be stored
$fw_user->(storeVars); // writes vars to data table
```

There are several other session commands. Refer to the online reference for detailed information about those and the keepers commands.

(Note: This system has proven quite effective for my application needs to date, however, some recent projects have exposed one flaw. Currently, there is only one session. All data is stuffed into that one session, and everything is automatically restored on every page. If there's large data sets to store, they accumulate and the session can be slowed down by needlessly restoring data not even needed on a given page. I'm still driven by the original decision to provide an alternative to Lasso's built-in system for the same reasons, so I will soon be working on extending the system to allow for multiple sessions, and also sessions for non-logged in users. For now, if you have large session data sets that apply to only certain pages, then it's probably worth using a Lasso session for data, but keep using the PageBlocks user session for user authentication propogation.)

User Management Admin Starter Kit

The user management module in the pbJedi sample application includes enough code to add, edit, and remove users, and define all their associated permissions. This module uses the Event Driven Pages (EDP) toolkit for the main structure.

There are still some things to add to this module to provide advanced user management examples. The two main things to add is creating user permission templates, and providing an example system for selecting multiple users to be editing en masse.

fwp_user Authorized User API

The `fwp_user` custom type provides both the data structure of a user's profile and permissions, as well as the internal routines to manage the user's session state and other tasks. The pbJedi sample application provides an example in the /sitemngr/mngr_login files of how to establish an authorized user.

The mngr_login2.lgc file in particular shows the process of instantiating a user object and handling the login result. This involves declaring the object, calling the authentication tag, the authorization tag, storing the user state, and finally handling page redirection based on the success or failure of the login including a redirection to have the user update his password if needed. The steps before the redirection will be pretty much the same for most applications. the redirection will have to be tuned for each application. Once a login has been accepted, the framework internally handles the re-validation of the user session for each page and also re-constitutes the user object.

Currently, the user object with all its profile and permissions data is serialized upon authentication and stored. On each page, if the session passes verification, the user object is unserialized. This is faster than starting from scratch each page and rebuilding the user object, and it lends itself to working as a central user/session management system for multiple application servers.

Interface Quick Glance

Read-Only Public Instance Variables:

```
'loginAccount'
'loginPswd'
‘loginValid’
'loginGetNewPswd'
'loginDaysPswdExpires'
'sessionID'
'testpwError'
```

Public Member Tags:

```
->authenticate
->storeUser
->restoreUser
->authorize
->killSession
```

```
->getProfile
->getPrivilege
->getFilter
->restoreVarsKeepers
->getVar
->addVars
->removeVars
->storeVars
->clearVars
->getKeeper
->addKeeper
->removeKeeper
->storeKeepers
->clearKeepers
->testPswd
->updtPswdHistory
```

Special Member Tags (uses _unknownTag to handle):

```
->prof.{fieldName}
->prvlg.{permissionName} or priv.{permissionName}
->fltr.{filterName}
```

Public Read-Only Instance Vars

The bulk of the user's profile and permissions information is accessed via member tags. The following items, however, are available as simple instance vars.

- `loginAccount` (string) = the string entered as the user's account (i.e. email)
- `loginPswd` (string) = the string entered as the user's password
- `loginValid` (string) = a Y or N character indicating the login validity
- `loginGetNewPswd` (boolean) = whether the new password must be defined
- `loginDaysPswdExpires` (integer) = number of days until the password expires
- `sessionID` (string) = the string of the sessionID
- `testpwError` (boolean) = result of using `->testPswd` member tag

Public Method –>authenticate

This tag establishes whether login credentials are valid and whether the record is active and allowed to access the application. It follows several steps described in the Security section above to help prevent brute force attacks. The instance var loginValid is set, the sessionID is created, the login stats of the user record are updated, and the event whether successful or not is logged if logging is enabled.

Public Method –>authorize

This tag builds the internal data structures of the user's profile, permissions, and filters by calling internal tags. The data itself must be accessed using the `->getProfile`, `->getPrivilege`, `->getFilter` tags.

Public Method –>storeUser

After a user object has been authenticated and authorized, it is stored as serialized data back to a field within the user record. This tag writes that serialized data.

Public Method –>restoreUser

This tag pulls a serialized user out to reconstitute the user object. There is a bit of chicken and egg situation to resolve when doing this, and the framework handles that in the `fwpPage_init` code. This tag also verifies the session is still valid, and updates the session time stamp in the user record.

Public Method –>killSession

Call this tag as part of a logout process. It resets several fields in the user record to clear session state and session variables.

Public Method –>getProfile

Use this tag to acquire a profile attribute. When the internal profile data structure is created, the userauth table, and userprofile table is defined, are scanned for all field names. Each field name (except for the framework fields which begin with ‘rcrd’) is stored as an attribute name. So, if there are fields such as userNameFirst, userEmail, or profColorFave, then these profile values are retrieved using syntax like this:

```
$fw_user->(getProfile:'userEmail')
$fw_user->(getProfile:'profColorFave')
```

Public Method –>getPrivilege

This tag acquires a specific privilege value. The user’s entire permissions matrix is stored in the user object. If we assume some permissions were stored for a news module with the individual privileges of add, update, and approve, the following syntax would be used to acquire the values of those privileges:

```
if: ($fw_user->(getPrivilege:'news_add')) == 'Y';
if: ($fw_user->(getPrivilege:'news_approve')) == 'Y';
```

Public Method –>getFilter

This tag acquires either an entire filter definition or a specific filter component value. Each filter has four components: table, field, match, op (operator). Each filter is also named. An entire filter is retrieved by name which returns a map of the four components. An individual filter component is retrieved by using name_component.

```
$fw_user->(getFilter:'regionCode')
$fw_user->(getFilter:'regionCode_match')
$fw_user->(getFilter:'regionCode_op')
```

Public Method –>restoreVarsKeepers

This tag is called automatically in `fwpPage_init`, and should not need to be called by the developer. It retrieves session variables and restores them as individual page variables. If this behavior is not wanted, this step in `fwpPage_init` can be turned off by setting `$fw_pageModes->'useAutoRestoreSession'` to false. The programmer will then have to individually recall session vars using something like this:

```
$fw_user->(getVar:'name')
$fw_user->(getKeepers:'name')
```

Public Method –>getVar

Retrieves a single session variable by name like this:

```
$fw_user->(getVar:'name')
```

Public Method –>addVars

Adds a single or multiple session variable names and values to an internal list of session variables. The variables are not actually stored until the `->storeVars` tag is called. The `->addVars` tag can be anywhere within the application code, and is usually best to use right where the value that is wanted has been established. The value of the variable at that moment is what is stored. If the value changes further in the page, the original value will persist in the session unless another `->addVars` call is used (which will replace the old value).

Add vars to the session:

```
var:'color' = 'red';
var:'shape' = 'circle';
$fw_user->(addVars:'color, shape');

var:'color' = 'blue';
// at this point the session variable is still 'red'
// and that's what will be restored on the following page
// unless another ->addVars is used to save the current value
```

Public Method –>removeVars

Removes a single or multiple session variable names and values from the internal list of session. The session data on disk is not updated until `->storeVars` is called, but the data is removed immediately from the user's data structure in memory.

```
$fw_user->(removeVars:'color, shape');
```

Public Method –>storeVars

Writes the session data to the user’s record immediately. This tag is automatically called in fwpPage_wrapup.lgc, but it can be called more often within the application code if deemed necessary by the programer. There are no parameters for this tag.

Public Method –>clearVars

In one step, this tag clears the in-memory session data, and clears the data stored in the user's record. This can be used at any time, and is automatically called by the user logout action to sanitize the user's record. Remember that sessionVars data is also automatically cleared during a login (sessionKeepers data is not cleared at login).

Public Method –>getKeeper

Retrieves a single session variable from the sessionKeepers data by name like this:

```
$fw_user->(getKeeper:'name')
```

Public Method –>addKeepers

Adds a single or multiple session variable names and values to an internal list of sessionKeeper variables. The variables are not stored until the `->storeKeepers` tag is called. The `->addKeepers` tag can be anywhere within the application code, and is usually best to use right where the value that is wanted has been established. The value of the variable at that moment is what is stored. If the value changes further in the page, the original value will persist in the session unless another `->addKeepers` call is used (which will replace the old value).

Add vars to the session:

```
var:'color' = 'red';
var:'shape' = 'circle';
$fw_user->(addKeepers:'color, shape');

var:'color' = 'blue';
// at this point the session variable is still 'red'
// and that's what will be restored on the following page
// unless another ->addKeepers is used to save the current value
```

Public Method –>removeKeepers

Removes a single or multiple sessionKeeper variable names and values from the internal list of session. The session data on disk is not updated until `->storeKeepers` is called, but the data is removed immediately from the user's data structure in memory.

```
$fw_user->(removeKeepers:'color, shape');
```

Public Method –>storeKeepers

Writes the session data to the user's record immediately. This tag is automatically called in fwpPage_wrapup.lgc, but it can be called more often within the application code if deemed necessary by the programer. There are no parameters for this tag.

Public Method –>clearKeepers

In one step, this tag clears the in-memory session data, and clears the data stored in the user's record. This can be used at any time. It is not automatically called by any process in the framework.

Public Method –>testPswd

Validates that a supplied string meets the password criteria defined in the authPswd config file, and optionally tests that the string is equal to a specified variable. Returns true or false, and automatically adds error messages to `$fw_inputErrorMsgs`.

```
$fw_user->(testPswd: $newpass, -compare=$newPassAgain);
```

Public Method –>updtPswdHistory

Adds a supplied unencrypted string to the user's password history and automatically removes any values older that the specified maximum number of passwords to remember as defined in the authPswd config file.

Nemoy Search Module for Lasso

Inspired by the first Lasso Programming Challenge[2], this little tool is an easy to implement, yet flexible system for adding single-input/multi-field searches to your web site. A single input field accepts keywords with modifiers to search multiple fields of a single data table at the same time, then perform a relevance ranking algorithm of the found records.

Relevance ranking can be mathematically intensive, so a dynamic language like Lasso isn't going to provide the greatest performance when using advanced algorithms. However, by using some "good enough" adjustments to reduce the computational demand, we can create some relevance ranking with enough adjustability to provide useful results for a variety of applications.

Feature Summary

Nemoy has the following features:

- finda records which contain one or more of multiple keywords/phrases
- allows required keywords prefixed by +
- allows excluded keywords prefixed with –
- allows "begins with" keywords using * after the word fragment
- allows "ends with" keywords using * before the word fragment
- allows phrases in quotes, and allows + and – modifiers for the whole phrase
- finds records which contain any combination of the above terms
- finds records based on searching multiple fields per record
- allows weighting of fields for relevance
- uses lemmatization*, and weighting of lemmatized words vs entered words
- abstracts the query interface to enable adaptors for multiple database engines
- abstracts query interface to enable multiple data table sources
- calculates relevance for found records
- displays records sorted by relevance
- abstracts relevance interface for multiple ranking formulas
- highlights search terms in results

** current lemma dictionary does not distinguish parts of speech, and word list being used is derived from the contents of the entire ldml8 reference table and the list at http://www.lexically.net/downloads/e_lemma.zip*

[2] http://www.lassosoft.com/Community/Challenge/index.lasso?9269

Source Code Organization

The organization of code for the PageBlocks framework is different than the standalone version code released for the Challenge.

The Nemoy search tool is a combination of core classes, query adaptors, ranking engines, linguisitic references, and search configurations. The former three are added to /LassoStartup/ with the namespace of fwpSrch. An fwpSrch_init file initializes standard vars for the nemoy libraries. Linguistic references and search confs are stored in /site/libs/ or a module's /_resources/libs/ folder by convention, but are free to be located anywhere.

Processing begins in a typical PageBlocks .lgc file to declare setup variables which define available search configs, instantiate the Nemoy objects, and do any housekeeping needed to prep for the display of the results.

- /adaptors/ — each file is a custom type designed to accept query parameters and write an SQL statement for a specific database server. The only task for an adaptor is to construct the SQL statement itself, not perform the query. The adaptor is called by the ctype `fwpSrch_nemoyQuery` found in /core/.
- /core/ — these are core logic files which remain constant to all nemoy deployments (unless the PageBlocks framework is being used, in which case, the core files are already loaded with the PageBlocks API libraries).
- /linguisitcs/ — language related resource files for advanced searches. Example: lemmata includes words which are related by a root (lemma) that culd be considered when searching.
- /rankEngines/ — each file here is an implementation of a ranking algorithm within a single custom type. The Nemoy Search Module accepts multiple algorithms so it can apply uniquely tuned ranking formulas to unique data sets and purposes.

Source Code Design

When a search is submitted, a criteria object `$nemoyCriteria` is created which contains the search string, a map of parsed search terms, and a couple states of the terms as a collection. That criteria is passed to a query object `$nemoyQuery` which performs the task of searching, ranking the results, and containing the final result dataset. A `$nemoyController` object is used to house the main logic that connects the pieces together.

The query object uses composition to integrate a search configuration object, a database adaptor object, and a rank engine object. Each of these are written to a standardized API.

A search configuration object defines the database, table, fields to search and return, and other characteristics of a search. Each configuration is named. This search configuration name becomes part of the search criteria which is passed to the query object.

Likewise, the search ranking algorithm to be used for a search is a part of the search criteria object. It is independent of the search configuration.

The query object uses Lasso's `lasso_datasourceModuleName` tag to determine which database engine is being used to serve the database declared in a search configuration. The query object uses this to instantiate the appropriate database adaptor as a helper to create an SQL string to execute. Additionally, the query object instantiates a rank engine object to subsequently calculate a relevance value for each record in the found data set. The query object then sorts the results according to the relevance number.

The application developer will need to deal with results display. The display code creates the HTML for the resulting record data. Like the search form HTML, this piece is not integral to the nemoy engine, and part of the application-specific code (although the demo code provides a generic, reusable routine for this functionality).

Search Configuration with srchConfig_xxxx

A search configuration custom type declares settings for a search. This custom type is just a collection of values, there are no member tags. It is usually put in the /libs/ folder. The example below identifies the required instance variables.

```
define_type:'srchConfig_lassoTagRefc';

    local:
        'qryUser'           = $fw_gQueryUser,
        'qryPswd'           = $fw_gQueryPswd,
        'db'                = 'ldml8_reference',
        'tbl'               = 'tags',
        'keyfield'          = 'id',
        'selectFields'      = 'id, tag_name, tag_category, tag_type, tag_description',
        'requiredPhrase'    = "tag_basic='Y' AND tag_display='Y'",
        'contentFields'     = 'tag_name, tag_category, tag_description',

        'contentWeights' = (map:
                            'tag_name'         = 5,
                            'tag_category'     = 2,
                            'tag_description'  = 1),

        'lemmaWeight' = 0.6,

        'displayColumns' = (array:
                            'tag_name'         = 'Tag',
                            'tag_category'     = 'Category',
                            'tag_type'         = 'Type',
                            'tag_description'  = 'Description'),

        'highlightColumns' = (array: true, true, false, true);

/define_type;
```

Each instance variable is described below:

- `qryUser` — the `-username` value to use for the query inline
- `qryPswd` — the `-password` value to use for the query inline
- `db` — the database name
- `tbl` — the table name
- `keyfield` — the name of the primary key field
- `selectFields` — a comma separated list of fields to be returned for display or other application logic purposes
- `contentFields` — a comma separated list of fields to be searched for content

- `requiredPhrase` — an SQL WHERE clause fragment that is to be included to qualify which records can be searched. For example, if only records where `rcrdApproved='Y'` can be searched, define that phrase here. The instance var must be defined, but a value is not necessary.
- `contentWeights` — for each field name in `contentFields`, that field must be listed here along with a value which identifies the relative value/importance of finding text in that field. The size of he numbers is irrelevant. The relative difference is what matters. If finding text in the field `tag_name` is 5x more relavant/important that finding that same text in the field `tag_description`, use an integer for `tag_name` that is 5x bigger than for `tag_description`.
- `lemmaWeight` — defines the relative weighting of a lemmatized term compared to a user entered term. Should be a decimal between 0 and 1.
- `contentColumns` — for each field from `selectFields` to be displayed on screen, list that field here along with a display title for that column. the data is an array, not a map, and the fields listed must be in the same order as listed in `selectFields` (though fields can be skipped).
- `highlightColumns` — for each field from `selectFields` to be displayed on screen declare a value of true or false to indicate whether text in that display should include highlighting of the search terms.

Nemoy setup in .lgc

This file has three sections to it. The first section declares some path variables which are used throughout the other source code files, and the available rank engines and search configurations.

The second section declares the main objects: `nemoyCriteria`, `nemoyQuery`, and `nemoyController`. These vars *cannot* be renamed. The default values for `nemoyCritera` will be application dependent.

The final section is a typical handler for search submissions. Searches can be submitted by clicking a form or through a GET directly from a URL. The coordination of all components is handled by the controller `->search` method.

```
<?lassoscript

//  ---------------------------------------------
//  declare nemoy variables

$nemoyLinguisticsPath  = ($fw_mPath->'libs');
$nemoyConfigsPath      = ($fw_mPath->'libs');
$nemoySearchConfigs    = (array:
                            'pbRefc'       = 'srchConfig_pbRefc',
                            'lassoTagRefc' = 'srchConfig_lassoTagRefc');
$nemoyRankEngines      = (array:
                            'NRE_Roberston'    = 'rankEngine_robertson',
                            'NRE_MySQLmod' = 'rankEngine_mysqlmod',
                            'NRE_Simple'   = 'rankEngine_wordcount');

//  ---------------------------------------------
//  instantiate nemoy

$nemoyCriteria = (fwp_nemoyCriteria:
    -defaultSrchConfig     = ($nemoySearchConfigs->get:1)->first,
    -defaultRankEngine     = ($nemoyRankEngines->get:1)->first);
```

```
$nemoyQuery = (fwp_nemoyQuery:
    -searchConfigs     = $nemoySearchConfigs,
    -rankEnginePlugins     = $nemoyRankEngines);

$nemoyController = fwp_nemoyController;

$nemoyController->(search:
    -criteria   = $nemoyCriteria,
    -query      = $nemoyQuery);

//  ----------------------------------------
//  app-specific code for results output

library: ($fw_mPath->'libs') + 'nemoyResultsTable.ctyp';
var:'resultsTable' = nemoy_resultsTable;

?>
```

Nemoy resultsTable

This is to be application-specific code to draw the table for the demo files. The demo files have an exampe for creating a table display object. Replace it however you prefer.

Rank Engines

A rank engine custom type has a simple structure: there must be one method (member tag) called `->calcRelevanceOf`. Implementing the algorithm will be specific to the formula, but there's some things which are common to all algorithms.

The data available to the calculator include the following:

- `#records` — all records returned by the search. The data structure is an array of pairs where the first value of the pair is the rank of the record. This starts out as zero, and is then updated by the rank engine. The second value of the pair, is the data for the record which is a map of field names and contents.

```
array:
    pair: rank = map:
                        fieldName = fieldContent
                        ...
                        fieldName = fieldContent
    ...
    pair: rank = map:
                        fieldName = fieldContent
                        ...
                        fieldName = fieldContent
```

- `#allCount` — the total number of records in the data table
- `#searchWords` — an array of individual words in the search string (they'll be stripped of term modifiers like quotes and + and such, but will be in the same order that they were originally typed)
- `#contentFields` — a comma separated list of fields that were searched (copy this value or weird things will happen if you change it).
- `#contentWeights` — the weighting factors of each of the content fields

It is possible that as more complex ranking formulas are considered that the interface between the criteria, query, and rank engines will have to evolve. The existing code should probably be updated to pass all these values as a map so that implementation is more abstracted.

The file rankEngine_wordCount.ctyp is an example of a minimal formula that simply counts the number of terms. It shows the typical minimum processing that a rank engine is likely to do in iterating through records, comparing field data to search terms, and updating the rank value of the record. The other rank engines can be reviewed for more complex formulas.

Database Adaptors

The database adaptors generate SQL query strings. While it's possible to build an array of search terms for Lasso's inlines, direct SQL is preferred for control, and it's possible that rankEngine-specific adaptors could be used to implement more complex (and faster) rank formulas.

Each adaptor must implement two methods `->buildQuery` and `->buildAllCountQuery`. The former would build and return the search query, while the latter would build and return a simpler query to determine an "all records" count (accounting for the srchConfig `requiredPhrase`).

Generally speaking, the code in each adaptor should be nearly identical except to replace the text that would generate the query string itself. We're pretty much constructing simple queries with some moderate boolean logic in the WHERE clause. This logic is tightly integrated into the supported search term modifiers (+, -, quotes etc). The patterns of the boolean logic should not be changed when creating adaptors for datasource that don't have one yet.

fwpSrch_nemoyCriteria

The fwpSrch_nemoyCriteria file acquires and parses search parameters. Inputs include the `searchString` typed by the user, a `rankEngine` value, and a `srchConfig` value. The latter two allow for a UI which gives the user a choice of multiple options, or they can be hard coded.

The `->init` method loads values from `client_getParams`. (Better OO code would pass `client_getParams` to the init method, but `client_getParams` is such a common Lasso idiom it makes sense to relieve the programmer of that task.)

The `->onCreate` method can optionally be used to declare default values for `rankEngine` and `srchConfig` using `-defaultSrchConfig` and `-defaultRankEngine` if none are passed via `client_getParams`.

The internal `->parseTerms` method will parse the `searchString` into components. It generates two data structures: one for search terms (search string words including their modifiers), and one for search words (search string words stripped of modfiers).

`searchTerms` is primarily for use by the query object for constructing the SQL query string. Whereas, `searchWords` is primarily for use by the rank engine to compare search words to the field contents, and for the display code to highlight search words in the displayed content.

fwpSrch_nemoyQuery

The fwpSrch_nemoyQuery.ctyp file is the heart of the logic for the search process. It uses composition to generate helper objects for the srchConfig data, the database adaptor, and the rank engine. It does this by looking up values from srch_querySetup.ctyp to identify source code files for these objects. They are loaded as raw source code then invoked into custom types dynamically.

The `->searchFor` method invokes the database adaptor to create the SQL query strings, initiates the searches, uses the internal `->makeRowMaps` method to convert the standard Lasso recordsarray data into the data structure required for the rank engine, and finally calls the rank engine to calculate the relevance value for each row in the dataset.

GUI Menu Tools

Menu Generation Tags

Introduction

The PageBlocks fwpGUI toolkit includes menu automation systems for creating text menus in horizontal and vertical layouts, and for creating rollover menu systems. Each of these systems uses a simple text file or Lasso array for menu configuration, and provides flexible highlighting of "current" menu items.

Horizontal Text Menus (fwpGui_hTextMenu)

The tag fwpGui_hTextMenu creates a horizontal text menu from either a structured config file of the name mnuText_{menuName}.cnfg, or an array which is structured identical to the config file format. The tag allows the definition of menu end cap and separator strings for decoration. Additionally, each menu item can be defined to be highlighted if the current page matches the menu link page, or the current page is within the menu link folder. To use the highlighting, a CSS class named fwptxtmenuhilite must be defined. A span will surround the menu text when the menu item is highlighted.

In order to pass session IDs or other page-to-page values, the links of menu items can be defined to include name/value pairs. These pairs can be added in one of three formats using standard GET syntax or one of two URL modification tags (fwpPage_urlParamsFile and fwpPage_urlParamsFldrs) which embeds the pairs into the URL path.

Tag Parameters

- -menu = required : accepts either the menuName portion of a configuration file name which will be in the site or module configs folder, or the complete root absolute file pathname to a file anywhere Lasso's file tags are configured to access. The menuName portion of the configuration file would be main from the file mnuText_main.cnfg which must be located in a /configs folder.
- -tween = required : a string of characters to be displayed between each menu item
- -left = optional : a string of characters to be displayed at the left of the entire menu
- -right = optional : a string of characters to be displayed at the right of the entire menu
- -paramsGET = optional : a string of comma separated names of variables to add to the link URL as standard GET form parameters. (Note: the URL is properly defined using &amp; for the name/value pair delimiters).
- -paramsFile = optional : a string of comma separated names of variables to add to the link URL in a modified format as created by the fwpPage_urlParamsFile tag
- -paramsFldrs = optional : a string of comma separated names of variables to add to the link URL in a modified format as created by the fwpPage_urlParamsFldrs tag

Vertical Text Menus (fwpGui_vTextMenu)

The tag fwpGui_vTextMenu creates a vertical text menu from either a structured config file of the name mnuText_{menuName}.cnfg, or an array which is structured identical to the config file format. The tag allows the definition of left bullet strings and indent fill strings for decoration. Additionally, each menu item can be defined to be highlighted if the current page matches the menu link page, or the current page is within the menu link folder. To use the highlighting, a CSS class named fwptxtmenuhilite must be defined. A span will surround the menu text when the menu item is highlighted.

In order to pass session IDs or other page-to-page values, the links of menu items can be defined to include name/value pairs. These pairs can be added in one of three formats using standard URL ? syntax or one of two URL modification tags (fwpPage_urlParamsFile and fwpPage_urlParamsFldrs) which embeds the pairs into the URL.

Tag Parameters

- -menu = required : accepts either the menuName portion of a configuration file name which will be in the site or module configs folder, or the complete root absolute file pathname to a file anywhere Lasso's file tags are configured to access. The menuName portion of the configuration file would be main from the file mnuText_main.cnfg which must be located in a /configs folder.
- -left = optional : a string of characters to be displayed at the left of each menu item after any applicable indenting
- -fill = optional : a string of characters to use for each indent position. The default is a single non-breaking space per position to yield the appearance of indenting. If a fill character is specified, the indent is filled with this character pattern per position.
- -paramsGET = optional : a string of comma separated names of variables to add to the link URL as standard GET form parameters. (Note: the URL is properly defined using &amp; for the name/value pair delimiters).
- -paramsFile = optional : a string of comma separated names of variables to add to the link URL in a modified format as created by the fwpPage_urlParamsFile tag
- -paramsFldrs = optional : a string of comma separated names of variables to add to the link URL in a modified format as created by the fwpPage_urlParamsFldrs tag

Text Menu Configuration Files

The mnuText configuration file defines the text, links, and other attributes of horizontal and vertical text menus created by fwpGui_hTextMenu and fwpGui_vTextMenu. These files go in either the /site/configs/ folder or the /{module}/_resources/configs/ folder. The file extension, typically .cnfg, is programmer defined using $fw_gCnfgExt.

Each menu item is on a separate line. For both vertical and horizontal menus, there are four items:

```
menuLink```menuName```menuIndent```menuHighlight
```

where:

- menuLink - the pathname to a page
- menuName - the menu item title
- menuIndent - the number of `&nbsp;` spaces or fill characters defined by the tag to insert for indenting (ignored by horizontal menus)
- menuHighlight - the highlight method {none|page|path}. Highlighting of the current menu item is handled with a CSS span class name of `fwptxtmenuhilite`.

Vertical menus require indent values, even if set to zero. Horizontal menus will ignore indent values. The same configuration file is intentionally used so it is possible the same menu can be presented in horizontal or vertical formats. For files known to be dedicated to horizontal use, indent values can be eliminated. Files can have blank lines and lines that start with # or // for commenting.

A configuration file for vertical menus with indented categories:

```
# sample menu as seen on www.pageblocks.org
/demo/menus```Simple Shapes```0```none
/demo/menus```Square```4```none
/demo/menus```Circle```4```none
/demo/menus```Rectangle```4```none

/demo/menus```Complex Shapes```0```none
/demo/menus```Two Dimensional```4```none
/demo/menus```Trapezoid```8```none
/demo/menus```Ellipsis```8```none
/demo/menus```Three Dimensional```4```none
/demo/menus```Sphere```8```none
/demo/menus```Cylinder```8```none
/demo/menus```Pyramid```8```none
```

A typical horizontal menu:

```
# a horizontal menu example
/index```Home```page
/ftrs/ftrs_intro```Features```path
/refc/refc_list```Reference```path
/demo/demos```Demos```path
/sitemngr/```SiteManager```page
```

Text Menu Tag Examples

A typical menu of links at the bottom of a web page.

```
<div id="pgbtmtextmenu">
[fwpGui_htextmenu:
  -menu='colors',
  -tween='&nbsp;|&nbsp;',
  -left='[&nbsp;',
  -right='&nbsp;]']
</div>
```

[Red | Yellow | Blue]

To include parameters in the URL to pass to the link page:

```
[var:'p1'='fakedata']
[var:'p2'='fakedata']

<p>[fwpGui_htextmenu:
  -menu='/someplace/notinconfigs/mnuText_colors.cnfg',
  -tween='&nbsp;&bull;&nbsp;',
  -left='&bull;&nbsp;',
  -right='&nbsp;&bull;',
  -paramsStd='p1,p2']</p>
```

• Red • Yellow • Blue •

Where the link on each menu is going to include `?p1=fakedata&amp;p2=fakedata`.

Some other examples:

```
<p>[fwpGui_vtextmenu:
  -menu='shapes',
  -left=' ',
  -fill='::']</p>
```

Two Dimensional
:: Square
:: Circle
Three Dimensional
:: Cube
:: Sphere

```
<p>[fwpGui_vtextmenu:
  -menu='shapes',
  -left='&bull;&nbsp;',
  -fill='&nbsp;']</p>
```

Two Dimensional
 • Square
 • Circle
Three Dimensional
 • Cube
 • Sphere

Menus can be defined on the fly by submitting an array rather than the configuration file name:

```
<p>[fwpGui_htextmenu:
  -menu=(array:
    '/demo/menus```Menu```0```none\r',
    '/demo/menus```From```0```none\r',
    '/demo/menus```An Array```0```none\r'),
  -tween='&nbsp;|&nbsp;',
  -left='|&nbsp;',
  -right='&nbsp;|']</p>
```

Rollover Menus

The AutoRollover (`fwpGui_autoRollover`) menu generator takes a relatively simple configuration file and creates a classic image driven rollover menu object with JavaScript and HTML components. The JavaScript component would be inserted in a page template's <head> section, and the HTML component can be drawn anywhere on the page as needed. Multiple independent rollover groups can exist on a page. This system was written many moons ago, and while it's probably better to write rollover menu systems with CSS these days, the PageBlocks framework doesn't have an automated tool for that yet, and this system is still hanging around, so we may as well document it.

In addition to writing the initial code for the rollover menu, the AutoRollover object will automatically keep track of highlighting needs to display context aware changes to each menu item. User interface features include support for mouseout, mouseover, and onclick states.

A rollover menu group is defined by rows of graphic items. A row may have one or more graphic items. Each item may be:

- a static graphic with no rollover behavior and no link,
- a static graphic with no rollover behavior but with a link,
- a rollover graphic with three possible states, context aware highlighting, and a link.

With this capability, AutoRollover can be used to create an entire masthead of static and interactive graphics. On any given page, a masthead or simple menu comprised of multiple rows can have any number of rows substituted with unique context aware set of graphics. For example a two row masthead might have a consistent `mainmenu` row at the top for every page. Then, the second row might be a `submenu` that is unique for each section of the website (products, support, company). Menus can take on horizontal layouts or vertical layouts (with one item per row). In order to automate the code writing and the assembly of the menu on the page, certain file naming conventions are used for the graphc images and the configuration files.

Rollover Menu Image Files

A menu item graphic's name is divided into four parts

```
menu_{menuName}_{menuItemName}_{menuItemStatus}.{gif|jpg}
```

where:

- use the word "menu_" to denote the image is a menu item
- the name of the menu such as "main" or "prod" or "supp"
- the name of the menu item such as "intro" or "widget" or "sprocket"
- the initial state of the graphic image which is "on" or "off" or "ovr"

The final name of the graphic images would look like the following:

- menu_main_company_off.gif
- menu_main_company_on.gif
- menu_main_company_ovr.gif
- menu_prod_sprockets_off.jpg
- menu_prod_sprockets_on.jpg
- menu_prod_sprockets_ovr.jpg

Three images are required for each active rollover menu item to correspond to the item's on state, off state, and click state. The AutoRollover code uses the `off` state image for non-highlighted and `mouseout` display, the `ovr` state image for `mouseover` display, and the `on` state image for the `onclick` and highlighted display.

A non-rollover menu item graphic's name (with or without links) has three components

```
{refcString}_{menuName}_{imageItemName}.{gif|jpg}
```

where:

- any leading string such as "msthd" or "img" or "grphc"
- the name of the menu such as "main" or "prod" or "supp"
- the name of the graphic item such as "left" or "filler" or "dots"

The final name of the graphic images would look like the following:

- msthd_main_filler.gif
- msthd_main_leftcap.gif
- msthd_main_rightcap.gif

Rollover Menu Configuration Files

The format of the configuration file is relatively straight forward. Knowing how to match configuration files to the desired menu layout takes a little effort to understand given the variety of layouts and options.

Sidebar: This system was developed before its author understood how to use regex, so the config file format is a little on the unintuitive side compared to the newer config file formats for other PageBlocks tags.

Configuration file names are constructed in the format of:

```
mnuDefn_{menuGroup}_{menuName}.ext
```

- menuRover_ is a required element
- {menuGroup} is the name of the menu object such as "msthd" or "sidebar" or "nav"
- {menuName} is the name of the menu such as "main" or "prod" or "supp"
- .ext is the file extension being used for config files such as .cnfg

The group name identifies a set of one or more menus which are typically drawn together as a group such as in a masthead or a sidebar. Multiple menu objects on a page are identified using the group name.

Configuration file lines can be empty or start with a # to allow for commenting. The configuration files are structured like this example:

```
# Main Menu

*main```-```-```comp```prod```apps```supp```refc```cntc```-```-

msthd_top_logo```.gif```160```22```/default.html```N```N```N```none
msthd_top_spcrlf```.gif```40```22``````N```N```N```none
menu_main_comp```.gif```67```22```/comp/comp_tour```Y```Y```Y```path
```

```
    menu_main_prod```.gif```68```22```/prod/prod_ovr```Y```Y```Y```path
    menu_main_apps```.gif```85```22```/apps/apps_ovr```Y```Y```Y```path
    menu_main_supp```.gif```64```22```/supp/supp_phone```Y```Y```Y```path
    menu_main_refc```.gif```77```22```/refc/refc_artcls```Y```Y```Y```path
    menu_main_cntc```.gif```72```22```/cntc/cntc_genl```Y```Y```Y```path
    msthd_top_spcrrt```.gif```72```22``````N```N```N```none
    msthd_top_cap```.gif```35```22``````N```N```N```none
```

Anywhere, but preferably at the top of the file, a line like the following which must begin with an asterisk (*) must exist:

```
    *main```-```-```comp```prod```apps```supp```refc```cntc```-```-
```

Each element delimited by ``` corresponds to an image that makes up the menu. As you've likely guess by now, "main" is the name of the menu, and each of the other text elements correspond to the menu item names. The hyphens are placeholders to indicate that the position is filled with a non-rollover image. The number of delimited elements in this line must equal the number of lines used to define the images of the menu, and the elements must be in a corresponding order. The first element to the first line, etc.

This line is used to buid the JavaScript objects with basic names of `maincomp`, `mainprod`, `mainapps`, etc. These will later be turned into `maincompOn`, `maincompOff`, `maincompOvr` for the rollover states. The hyphens are used mostly for visual clarity when editing.

Following the * line, there must be a line for every graphic element in the row following this pattern:

```
filename```filetype```width```height```href```onmouseover```onmouseout```onclick```highlight
```

where:

- `filename` - the basic filename of the rollover (do not include the _on, _off, or _ovr portion)
- `filetype` -{.gif | .jpg} to define the extension of the graphic file
- `width` - image width in pixels
- `height` - image height in pixels
- `href` - the root relative link URL
- `onmouseover` -{Y | N} to include an onmouseover statement
- `onmouseout` -{Y | N} to include an onmouseout statement
- `onclick` -{Y | N} to include an onclick statement
- `highlight` -{path | page | none} identify whether the button has a location-dependent highlight status based on the path name, file name, or none

Examples:

```
    msthd_top_logo```.gif```160```22```/default.html```N```N```N```plain
    menu_main_comp```.gif```67```22```/comp/comp_tour```Y```Y```Y```path
```

The above discussion is oriented towards horizontal menus. Vertical menus are also supported. Vertical menus uses lines that begin with ! to insert a <br> between the items. The HTML is elementary, and does not use CSS at this point.

Thus, a menu of single rollover items with no extra statc images might look like this:

```
# Product Sub Menu
*prodsub:::intro:::-:::tech:::-:::perf:::-:::apps:::-:::line
menu_prodsub_intro:::.gif:::130:::20:::/prod/gf/gf_intro:::Y:::Y:::Y:::page
!
menu_prodsub_tech:::.gif:::130:::20:::/prod/gf/gf_tech:::Y:::Y:::Y:::page
!
menu_prodsub_perf:::.gif:::130:::20:::/prod/gf/gf_perf:::Y:::Y:::Y:::page
!
menu_prodsub_apps:::.gif:::130:::20:::/prod/gf/gf_apps:::Y:::Y:::Y:::page
!
menu_prodsub_line:::.gif:::130:::19:::/prod/gf/gf_line:::Y:::Y:::Y:::page
```

Creating and Drawing Rollover Menu Objects

To create a menu group create a variable of the type `fwp_autoRollover`:

```
var:'fw_rollovers'=(fwp_autorollover: -menus={array}, -imgsPath={folderPath});
```

where the array is defined as folllows

```
(array:'{menuGroupName}```{menuName}```{menuName}...')
```

which might look like the following for a menu group named `msthd` with a persistent `main` menu and a site location dependent submenu named `prod`, or the following for a simple single configuration file menu:

```
(array:'msthd```main```prod')
(array:'msthd```main')
```

To create multiple menu groups on one page, use the following format:

```
(array:'msthd```main```prod','sidebar```hotlinks')
```

That would create two menu groups named msthd and sidebar.

The PageBlocks framework uses a variable named `$fw_mnuDefns` for the menu array. If there are no rollover menus to draw on a page this variable should be set to a string. Otherwise it can be defined with a default in _initCustom such as `$fw_mnuDefns=(array:'msthd```main```prod')`, and overridden as needed in _pageConfig.

In `fwpPage_init`, the following routine tests the variable and creates a rollover menu object if an array has been defined.

```
if: $fw_mnuDefns->type == 'array';
    if: var:'fw_s';
        var:'fw_rollovers'=(fwp_autorollover: -menus=$fw_mnuDefns, -paramsFile='fw_s');
    else;
        var:'fw_rollovers'=(fwp_autorollover: -menus=$fw_mnuDefns);
    /if;
/if;
```

To attach name/value pairs to the URL in one of three formats, use the optional params `-paramsFile`, `-paramsFldr`, or `-paramsGET` and specify a comma separated list of variable names.

The code for all the JavaScript and HTML is prewritten and stored in instance variables of the menu object. To insert the JavaScript code for all menus defined for the page in the HTML `<head>`, use the following instance variable of the object:

```
$fw_rollovers->'jsCode';
```

To insert the HTML code of a particular menu into the page <body>, use the instance variable of the object of the menu name:

```
$fw_rollovers->'msthd';
```

If multiple menus were defined, the following would draw two menus:

```
$fw_rollovers->'msthd';
$fw_rollovers->'sidebar';
```

Updating _pbLibs for Using autoRollover

Add these functions to your JavaScript code in fwpPage_jsScriptsLib.lgc or whereever youi're storing scripts.

```
function imgOn(imgName) {
    document[imgName].src=eval(imgName + "On.src");
}
function imgOff(imgName) {
    document[imgName].src=eval(imgName + "Off.src");
}
```

Add this lines to the end of fwpPage_templateHead.lgc

```
((var:'fw_mnuDefns')->type == 'array') ? $fw_rollovers->'jsCode';
```

fwp_valueList Data Type

In release 5.2.2, the previous value list tags have been replaced by a single unified value list type. The new type uses a new config file format consistent with other PageBlocks config files, can also pull lists from a data table, is multi-language compatible, and has greater flexibility in applying custom list attributes. Some of the flexibilities available with the old tags which were rarely used have been eliminated to gain simplicity and performance in the new value list type.

The `fwp_valueList` type creates independent objects which can be rendered as an HTML value list in the form of a popup menu, checkboxes, radio buttons, or list box. Each value list has one or more options and behaviors unique to its list type, but for the most part, all share a majority of common features and programming requirements.

Each value list style is able to display any combination of default selection(s) which represents an initial state or the state of an application variable, database field, or other data container. The draw code of the data type will generate all necessary HTML to display and control the value list including customized HTML attributes.

The list of selection options can be supplied to a value list object from an array of pairs, from a specially formatted text file, or from a specially structured data table. Generally speaking, the text files are the most common source as they're convenient for grouping multiple lists in one place and, like many PageBlocks features, allow for site and module-specific versions. File contents are cached so performance is not hindered by repeated reading of the text file. The database table source is more effective for situations where several applications maintained as separate PageBlocks projects need to share a common value list source to avoid having to maintain redundant copies of config files.

General Syntax and Usage

The `fwp_valueList` type follows typical PageBlocks custom type conventions, and has several optional parameters. Usage has two steps: creating the value list object (typically done in a block .lgc file or app-specific custom type), and drawing the object (done in the block .dsp or app-specific view object). Most parameters are defined when the list object is created. A few can be defined as the object is drawn. Just to create a visual to work from, below is an example of what usage might look like:

```
//  creating a list from a config file named valueList_genl_en-us.cnfg

var:'m_weekDays' = (fwp_valueList:
        -file          = 'genl',
        -list          = 'weekdaysLong',
        - attributes   = (map:
            'name'     = 'm_weekDays'));

//  drawing the list

$m_weekDays->(drawAsPopup:
    -currentValue  = field:'eventDay',
    -tabIndex      = 3);
```

Value List Common Features

Parameters

Following practices which strive to separate application logic from user interface, most parameters are defined when the value list object ("vlist") is created, but a few can also be defined with the drawing member tag to allow for changes at the view layer. This separation is a significant change from the previous value list tags where all coding took place in the display code, but brings another PageBlocks tool in line with modern coding practices.

Another major change from the previous tags is how HTML attributes are handled. Attributes are defined in a map passed as a single parameter when the vlist is created. Each of these attibutes are added to the HTML exactly as submitted. This gives the developer greater control over the vlist code. To also provide simplicity, several attributes are added with defaults if none are specified at object creation which can reduce the amount of code to write.

Parameters Common to All Value List Types

- `-file`—defines which config file to use. Config file names follow the naming convention of valueList_{name}_ {language}.cnfg. The `-file` parameter declares the {name} part. The internal code will automatically determine language if is not passed by the `-language` parameter. So, to use the vaue list file valueLists_products_en-us.cnfg, you'd declare `-file='products'`. Refer to the section *Preparing Value List Text Files* for detailed information about file format.
- `-table`—the reference name of the data table to use from the `$fw_gTables` map. Pass just the reference name, not the `$fw_gTables` value. So, if `$fw_gTables` has entries like this
```
'pbrefc'              = 'pbReference',
'vlists_en-us'        = 'valueLists_en_us',
'vlists_it'           = 'valueLists_it',
'appstrings_en-us'    = 'appStrings_en_us',
'appstrings_it'       = 'appStrings_it',
```
  define the table like one of these methods
```
-table = 'vlists_en-us'
-table = 'vlists_' + ($fw_client:'language')
```
- `-scope`—applicable only if the `-table` parameter is used, this option defines whether a list defined as having site scope or module scope should be used. In the database table, lists which are applicable to the whole site are defined in the `listScope` field with "site" as the scope. Lists which are applicable to a specifc module are defined with `listScope` as the name of that module (using the module folder name).
- `-list`—defines the exact list to use. This can either be an array or an array of pairs to hard code a list or pass a dynamic list. It will most typically be the list name of a list definition from a config file or data table.
- `-titleOption`—the behavior of this one changes depending on list style. For popup and listbox lists, this defines a extra selection option to appear at the top of the list. Typically this would be used to add a "Please select..." or a blank entry at the top of the list. This allows the same list to be used for multiple displays. For example, in a data entry form, the list can have "Please Select..." at the top while in a search form could have "All" at the top. For checkbox and radio buttons, this parameter will add a `<fieldset>` around the tag with the `<legend>` containing the `titleOption` value.

- `-currentValue`—declares a value that should be selected/checked when the list is drawn. For popups and radio buttons, this must be a single value. For checkboxes and list boxes, this can be a single value or multiple values declared as a single string delimited with \r. This format was chosen to make it easier to pass field contents which retain the \r formatting of the original selection.
- `-withoutCaching`—this parameter has no value. It declares that the resulting HTML generated from this list should not be cached. usually lists should be cached for better performance, but there are occasions where the list should not be cached because it is subject to dynamic content changes.
- `-attributes`—a map of HTML attribute name-value pairs. For popups and list boxes, this declares the attributes of the `<select>` tag. For radios and checkboxes, this declares the attributes of each `<input>` tag. The minimum requirement is for the attribute `name` to be declared. All others are optional. See the *Defining Attributes* section for more details.

Parameters Specific to Radio Buttons and List Box Lists

- `-asVertical`—declares the the list should be displayed vertically. Each list item is separated with a `<br />`.
- `-asHorizontal`—declares the the list should be displayed horizontally. Each list item will have three trailing `&nbsp;`

Defining Attributes

For popups and list boxes, attributes are applied to the `<select>` tag. For radios and checkboxes, attributes apply to each `<input>` tag. The minimum requirement is for the attribute `name` to be declared. All others are optional, some are added with default values (explained below).

The atributes values are supplied as a map when the object is created. Each attribute in the map is added exactly as submitted (exceptions are explained in the *Default Attributes* section below). This allows the developer to customize attributes as needed, and even allows for non-standard or future standards to be accommodated.

Default Attributes

- `name` will always be added, even if empty.
- `id` will always be added. It will be the same as `name` if an alternate value is not specified. For radios and checkboxes, `id` is automatically incremented with a numer appended to the name supplied (e.g. `myIDName1`, `myIDName2`, `myIDName3`...).
- `class` will always be added. Default values depend on the list type and will be `fwppopup`, `fwplistbox`, `fwpradiobtn`, or `fwpcheckbox` if an alternate is not specified.
- `size` is hard coded as 1 for popups. It defaults to 4 for list boxes if an alternate is not specified.
- `tabindex` is automatically incremented for radios and checkboxes if an initial value is provided. The `tabindex` value is not cached, so the same list can be used from cache in different cases with unique `tabindex` values for each use.

- `disabled` will disable the entire list for popups and listboxes, but for checkboxes and radio buttons, if specific options are defined, only they will be disabled (use a `\r` or comma delimited list of options).

Drawing

There are member tags for drawing each list type:

- `->drawAsPopup`
- `->drawAsRadioButtons`
- `->drawAsCheckBoxes`
- `->drawAsListBox`

Parameters for Drawing

If all parameters are defined at the vlist creation, the drawing tags need no parameters, However, while all parameters can be defined at object creation, there maybe occassion to specify some details at the time the HTML is drawn. The draw tags allow for the these parameters to be defined:

- `-disabled`
- `-currentValue`
- `-tabIndex`
- `-id`
- `-class`

CSS Compatibility

The HTML code generated for each list includes a CSS class definition of: `fwppopup`, `fwpcheckbox`, `fwpradiobtns`, or `fwplistbox`. If a `css` attribute is not specified, the preceding class names will be automatically used. If you include these values in your style sheets, that style will be applied to the output of the tag.

RadioButton and CheckBox Styles

For a better user interface over the default HTML/browser behavior, the code for each radio button and check box includes an HTML `<label>` tag to create clickable labels (for browers that support that). A `class` parameter is added with the same value as the class for the `<input>` tag. Additionally, a `<style>` attribute is added to keep the mouse pointer icon an arrow when over the `<label>` text. Finally, each selection item is wrapped with <span class="nobr"> so that the `nobr` class can define a no-break to keep each item's text from word-wrapping in the display.

If the `-titleOption` parameter was defined, the `<fieldset>` and `<legend>` tags will also have classes with the same values as the `<input>` tags.

```
<span class="nobr"><label class="fwpcheckbox" style="cursor:arrow"><input type="checkbox"
value="Mon" name="m_weekDay" tabindex="15" id="m_weekDay2"  class="fwpcheckbox" />Mon</
label></span><br />
```

Even though the label, input, fieldset, and legend tags are using the same class name, you can create different rules through selectors.

Examples

Example 1:

```
var:'jumpToMonth' = (fwp_valueList
    -file          = 'genl',
    -list          = 'monthsLong',
    -titleOption   = 'Pick a calendar to view...',
    -currentValue  = (date->month:-long),
    -attributes    = (map:
        'name'     = 'jumpToMonth',
        'tabIndex' = 9,
        'onchange' = 'jumpToPage(this.value)'));

$jumpToMonth->drawAsPopup;
```

This code creates a popup menu where the list options have been pulled from the file valueLists_general_en-us.cnfg, the current month will be preselected, and where upon selecting a value, the `onchange` attribute will run the JavaScript function `jumpToPage`.

Example 2:

```
var:'m_availableDays' = (fwp_valueList:
    -file          = 'genl',
    -list          = 'daysShort',
    -attributes    = (map:
        'name'     = 'm_availableDays'));

$m_availableDays->(drawAsCheckBoxes:
    -disable = 'Sun\rSat');
```

This list will display the days of the week as horizontal checkboxes, and disable the Sun and Sat boxes.

Preparing Value List Selection Options

Value list selection options can be defined via arrays or arrays of pairs to submit directly through the `-list` parameter, via config files with sets of pre-defined options through the `-file` parameter, or via data tables with sets of pre-defined options through the `-table` parameter.

Defining Lists with Arrays

If a simple array is passed to the -list parameter, it will create a list where the display and the values are the same.

```
-list = (array: 'Small', 'Medium','Large');
```

as a popup will create the following

```
<option value="Small">Small</option>
<option value="Medium">Medium</option>
<option value="Large">Large</option>
```

If an array of pairs is passed to the -list parameter, it will create a list where the display is the first part of the pair, and the values are the second part.

```
-list = (array: 'Small'='S', 'Medium'='M,'Large'='L');
```

as a popup will create the following

```
<option value="S">Small</option>
<option value="M">Medium</option>
<option value="L">Large</option>
```

Defining Lists with Config Files

Value list config files contain one or more named list definitions. Config file names follow the naming convention of valueList_{name}_ {language}.cnfg. The `-file` parameter of creating a vlist object declares the {name} part. So, to use the value list file valueLists_products_en-us.cnfg, you'd declare `-file='products'`. Create a file unique to each language used by the application.

Each list in the config file is in the following format:

```
{{listName:
listOption
listOption
}}
```

where listOption has one of two allowed formats:

```
sharedDisplayValueName
// or
display___  value
```

In the case of the former, a single string is used as both the displayed string of the selection and as the value. In the latter the display string and value string are distinct.

Below is an example file content that defines three value lists. Lines are significant, but whitespace between display and value strings is not significant.

```
{{daysLong:
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
}}

{{daysShort:
Sun
Mon
Tue
Wed
Thu
Fri
Sat
}}

{{daysLongShort:
Sunday___   Sun
Monday___   Mon
Tuesday___ Tue
Wednesday___ Wed
Thursday___ Thu
Friday___   Fri
Saturday___ Sat
}}
```

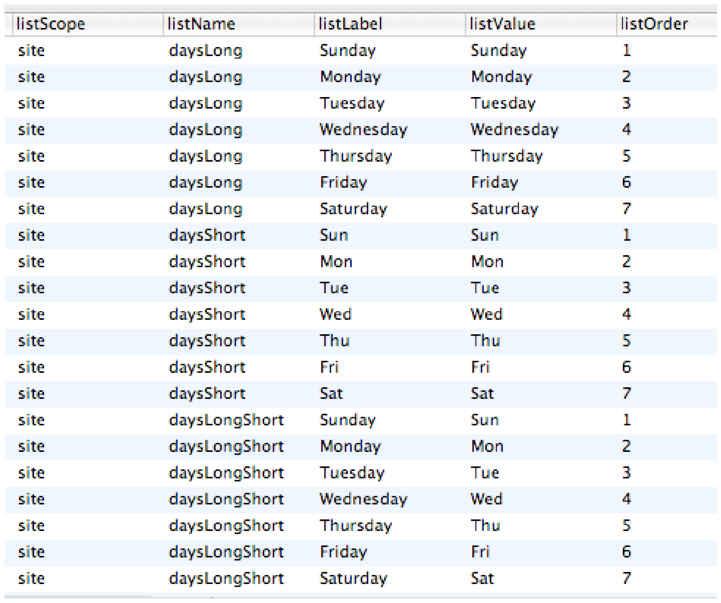
Defining Lists with Data Tables

Data tables can be used to store list selections. A prescribed schema is required:

```
CREATE TABLE `valueLists_en_us` (
  `rcrdNo`         varchar(24)    NOT NULL default '',
  `rcrdCreated`    datetime       NOT NULL default '0000-00-00 00:00:00',
  `rcrdCreatedBy`  varchar(49)    NOT NULL default '',
  `rcrdModified`   datetime       NOT NULL default '0000-00-00 00:00:00',
  `rcrdModifiedBy` varchar(49)    NOT NULL default '',
  `rcrdStatus`     char(1)        NOT NULL default 'N',
  `rcrdLock`       char(1)        NOT NULL default '',
  `rcrdLockID`     varchar(12)    NOT NULL default '',
  `rcrdLockTime`   datetime       NOT NULL default '0000-00-00 00:00:00',
  `rcrdLockOwnr`   varchar(49)    NOT NULL default '',
  `listScope`      varchar(16)    NOT NULL default '',
  `listName`       varchar(48)    NOT NULL default '',
  `listLabel`      varchar(128)   NOT NULL default '',
  `listValue`      varchar(128)   NOT NULL default '',
  `listOrder`      int(4)         NOT NULL default '0',
  PRIMARY KEY  (`rcrdNo`),
  KEY `rcrdNo` (`rcrdNo`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COMMENT='Name-Value sets for GUI value lists'
/* of course table options will vary by application */
/* listXXX fields sizes can be adjusted as you see fit */
```

The typical PageBlocks `rcrdXXX` fields are only necessary if you plan to build a PageBlocks module to edit the table. If your data values will be generated some other way, it is OK to have a table with only the `listXXX` fields. Create a table specific to each language to be used in the application (it does not use the appStrings format). Also, you'll need a tableModel_ config file for each table.

- `listScope` will contain the literal value of "site" or the folder name of the module the list is specific to.
- `listName` is name of the selection list
- `listLabel` is string to be displayed to the user for that specific selection option
- `listValue` is string value of the selection option
- `listOrder` is the order the option is to be displayed within the named group



listScope	listName	listLabel	listValue	listOrder
site	daysLong	Sunday	Sunday	1
site	daysLong	Monday	Monday	2
site	daysLong	Tuesday	Tuesday	3
site	daysLong	Wednesday	Wednesday	4
site	daysLong	Thursday	Thursday	5
site	daysLong	Friday	Friday	6
site	daysLong	Saturday	Saturday	7
site	daysShort	Sun	Sun	1
site	daysShort	Mon	Mon	2
site	daysShort	Tue	Tue	3
site	daysShort	Wed	Wed	4
site	daysShort	Thu	Thu	5
site	daysShort	Fri	Fri	6
site	daysShort	Sat	Sat	7
site	daysLongShort	Sunday	Sun	1
site	daysLongShort	Monday	Mon	2
site	daysLongShort	Tuesday	Tue	3
site	daysLongShort	Wednesday	Wed	4
site	daysLongShort	Thursday	Thu	5
site	daysLongShort	Friday	Fri	6
site	daysLongShort	Saturday	Sat	7

Loading Lists as Data Structures

Sometimes you'll need lists to be loaded as a standard Lasso array, array of pairs, or a map. For example, if you save a short form of a selection to a data table, but then want to show that selection as plain text without a value list, you need to convert it back to the long form. For that, it is handy to have the list available as a map where the short forms are the keys. To load the list into a map, we can use `fwpCnfg` file tags.

```
var:'monthsShortLong' = fwpCnfg_loadBlocks:'valueLists_time_en-us.cnfg';
$monthsShortLong = $monthsShortLong->find:'monthsShortLong';
$monthsShortLong = (fwpCnfg_splitBlockLines:$monthsShortLong, -intoMap);
```

The first line loads the whole config file into `$monthsShortLong` as a map where the keys are the names of the lists, and the values are the entire list as a string. The second line selects just the list we're interested in and sets `$monthsShortLong` as a string of the list options. The third line converts that string into a map of option keys and values that we can use like this:

```
$monthsShortLong->(find:$m_month); // displays long version of the list option
```

Formatting Tags

Lasso has many data formatting tags, but in many cases it takes some verbose coding to get simple common formatting of dates and numbers. The PageBlocks framework has several tags to make it easier to display data in a variety of common formats. The tags sets aren't exhaustive, part of their usefulness is that there are only a few of them to remember. Check the online reference for the tags listed in the sections below.

Date/Time

fwpDate_12hrTime - shows 12 hour time component of a date string

fwpDate_24hrTime - shows 24 hour time component of a date string

fwpDate_age - calculates age in years between two dates

fwpDate_euroMMLong4 - generates 15 January 2005 from a date string

fwpDate_euroMMShort4 - generates 15 Jan 2005 from a date string

fwpDate_euroSlash2 - generates 15/01/05 from a date string

fwpDate_euroSlash4 - generates 15/01/2005 from a date string

fwpDate_isLeapYear - yields true if year is a leap year

fwpDate_isoDate - generates 2005-01-15 from a date string

fwpDate_isoDateTime - generates 2005-01-15 09:22:45 from a date string

fwpDate_isValid - accepts inputs in multple numeric and textual formats like `15/04/2006` or `Apr 15 2006` and determines if the date is valid. Returns null if not valid, a Lasso date string `mm/dd/yyyy` if valid, or `yyyy-mm-dd` if `-returnAsQ` is specified

fwpDate_mmLong - generates January 15 from a date string

fwpDate_mmLong4 - generates January 15 2005 from a date string

fwpDate_mmShort - generates Jan 15 from a date string

fwpDate_mmShort4 - generates Jan 15 2005 from a date string

fwpDate_mmyySlash - generates 01/05 from a date string

fwpDate_relative - generates arbitrary text like "Today" based on relative date differences

fwpDate_usSlash2 - generates 01/15/05 from a date string

fwpDate_usSlash4 - generates 01/15/2005 from a date string

Numerics

fwpNum_dec1 - generates a decimal formatted with separators to the 10ths order of precision
fwpNum_dec2 - generates a decimal formatted with separators to the 100ths order of precision
fwpNum_dec3 - generates a decimal formatted with separators to the 1000ths order of precision
fwpNum_dec4 - generates a decimal formatted with separators to the 10000ths order of precision
fwpNum_int - generates a integer formatted with separators
fwpNum_intArray - generates an array of integers

Strings

fwpStr_getLeft - gets n left characters from a string with space sensitivity
fwpStr_getParagraphs - gets n paragraphs from a string
fwpStr_getRight - gets n right characters from a string with space sensitivity
fwpStr_getSentences - gets n sentences from a string
fwpStr_getWords - gets n words from a string
fwpStr_highlight - adds CSS spans around search words in a string
fwpStr_markWebLinks - replaces words/phrases with assigned <a> tag links
fwpStr_maskRight - combines strings like REQ-000000 and 1228 to form REQ-001228
fwpStr_phoneGetPHA - gets 999 from packed USA phone string of 9995551212
fwpStr_phoneGetPHP - gets 555 from packed USA phone string of 9995551212
fwpStr_phoneGetPHN - gets 1212 from packed USA phone string of 9995551212
fwpStr_phoneGetPHX - gets extension numbers from packed USA phone string of 99955512120012
fwpStr_phonePack - takes three field phone number inputs and combines them for db storage
fwpStr_phoneUnpack - splits a packed string into three components for form display
fwpStr_randomID - generates a random string of n characters in length
fwpStr_randomPswd - generates a random string of n characters in length leaving out the visually confusable characters of zero, letter O, one, letter I, etc.

FWP_IntArray
Integer Array Generator

Description

This tag generates an array of integers with both absolute and relative start and end parameters. It was originally developed to provide a dynamically generated list of years relative to the current year as an input for value lists, but has been expanded to allow for general purpose use. Based on the start and end values, the tag will automatically determine an ascending or descending order to the values.

Parameters

- `-first`—any integer value defining the start of the integer array. If not specified, the value is assumed to be zero. If the `-origin` option is specified then this value will be added to the value of `-origin` to determine the starting value of the array. To make the starting value less than the value of `-origin`, define `-first` as negative.
- `-last`—any integer value defining the end of the integer array. If not specified, the value is assumed to be zero. If the `-origin` option is specified then this value will be added to the value of `-origin` to determine the final value of the array.
- `-origin`—an integer value for the start of the integer array, or use the literal year or years to have the origin be the 4 digit value of the current year.

Examples

```
[fwp_INTarray: -first=5, -last=-5]
Result: 5 4 3 2 1 0 -1 -2 -3 -4 -5

[fwp_INTarray: -origin=10, -first=3, -last=-3]
Result: 13 12 11 10 9 8 7

[fwp_INTarray: -origin='year', -first=-3, -last=3]
Result: 2000 2001 2002 2003 2004 2005 2006

[fwp_INTarray: -first=6]
Result: 6 5 4 3 2 1 0
```

Event Driven Pages (EDP)

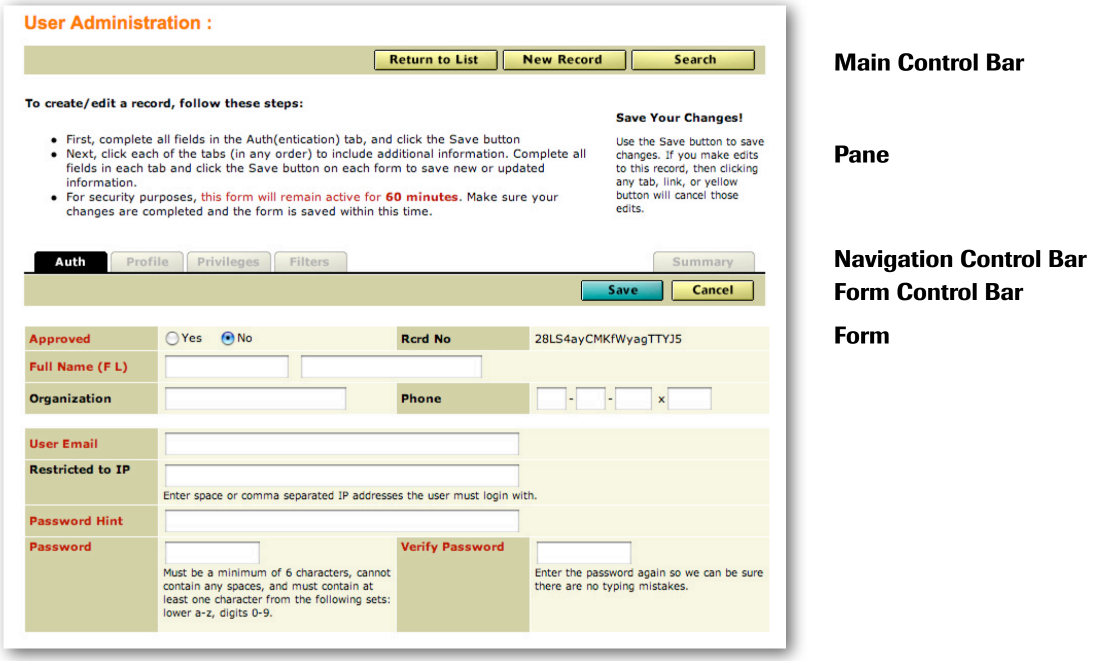
Event Driven Pages (EDP) in version 5.0 of the PageBlocks framework is a collection of custom types and code standards which automate user interface generation and response handling for data entry pages. EDP is built upon existing services in the PageBlocks framework, so it works as an extension to the existing page assembly engine not a replacement for it.

EDP is intended for use with multi-form workflow processes—a sequence of forms which are sequentially or randomly completed with each form a component of a larger data set that must be completed as a whole. For example, a registration process which might use separate forms to collect contact information, billing information, membership profile, and application preferences. Rather than presenting this to the user a single lengthy form, it can be divided into topical areas without a commitment to complete the entire data set at one time. Another example could be product configuration which requires that a sequence of forms be completed in a specific order with the second form being dependent on the first form, etc.

EDP supports flexible processes, but typically there is a begin form and a summary form with zero or more intermediate forms. The begin form is a mandatory first form to be completed. The summary form would be used to provide a non-editable summary or formatted preview of all form data. The collection of forms and supporting files to manage a given data set is called an EDP Module.

The purpose of EDP is to standardize operations where possible and focus the developer's efforts on form design/content and business rules. Navigation response and control, and many data source handling tasks are automated. Using Lasso custom type ("ctype") APIs, the majority of EDP code is invisible to the developer, yet every standard task can be easily overloaded when that task needs modified for special cases.

EDP Figure 1: A typical "Begin" form page



To create/edit a record, follow these steps:

- First, complete all fields in the Auth(entication) tab, and click the Save button
- Next, click each of the tabs (in any order) to include additional information. Complete all fields in each tab and click the Save button on each form to save new or updated information.
- For security purposes, this form will remain active for **60 minutes**. Make sure your changes are completed and the form is saved within this time.

Save Your Changes!

Use the Save button to save changes. If you make edits to this record, then clicking any tab, link, or yellow button will cancel those edits.

Approved	Yes [✓ No]	Rcrd No	28LS4ayCMKfWyagTTYJ5
Full Name (F L)			
Organization		Phone	- - x
User Email			
Restricted to IP	Enter space or comma separated IP addresses the user must login with.		
Password Hint			
Password	Must be a minimum of 6 characters, cannot contain any spaces, and must contain at least one character from the following sets: lower a-z, digits 0-9.	Verify Password	Enter the password again so we can be sure there are no typing mistakes.

EDP implements a standard MVC pattern though it is not a generic MVC system for all pages of a site (though there is potential to extend it to that purpose). EDP may be suitable for a variety of page handling needs, but the current version is focused on multi-form editing processes most typically applied to web site admin back ends and data-entry-centric intranet applications.

User Interface

User Interface Elements

To provide a visual reference of a typical EDP application, let's dive right into an example with a typical user interface. Each EDP module will usually have a home page which presents a list of records to edit, and a series of form pages starting with the begin form.

The home page, if used, would normally have a main control bar and the list of records. It could include a list filter (a simple form to search based on the fields shown in the list), and list navigation to scroll through multiple "pages" of records. Don't get hung up on the style of the sample images, layout is flexible.

The form pages include standard display elements called the main control bar, navigation control bar, form control bar, form, and panes. Figure 1 shows a typical form page (from the standard sample PageBlocks user administration module).

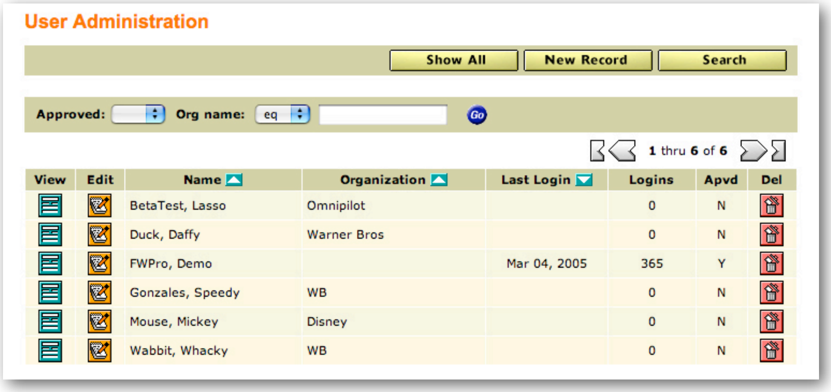
The main control bar provides functions universal to the entire EDP module—buttons to create a new record, define module preferences (not shown in Fig 1), search module records, and return to the module's main record list. These are standard functions. Custom functions can also be added.

The navigation bar provides navigation between the various forms. The built-in controller currently expects graphics as click targets and automatically chooses dim versions to make all but the begin form tab inactive for new records. ("Tabs" here just means a graphic image for a button, it can be horizontal, vertical, any design, etc). A text menu version is being worked on for a future update.

The form control bar provides form submission functions. Standard functions include Save, Delete, and Cancel, though other custom functions can be created as well. Delete is typically only shown on the summary form.

The form is what it seems—a data entry HTML form (except that the form tags themselves are not part of the form file, but we'll go over that later). The same form is used for new records and update records (although mode is detected and can be used to gerenate differences if needed). All fields are populated using Lasso variable tags (using

EDP Figure 2: A typical "Home" form page with a list filter


User Administration

View	Edit	Name	Organization	Last Login	Logins	Apvd	Del
		BetaTest, Lasso	Omnipilot		0	N	
		Duck, Daffy	Warner Bros		0	N	
		FWPro, Demo		Mar 04, 2005	365	Y	
		Gonzales, Speedy	WB		0	N	
		Mouse, Mickey	Disney		0	N	
		Wabbit, Whacky	WB		0	N	

$) not data table field tags. The EDP system automatically takes care of creating empty vars to enable the $ syntax. This provides complete separation of the form from the data source.

Panes, such as the instructional pane in Figure 1, are non-data-entry screen regions inserted where needed on the page for whatever purpose they may be needed (descriptive help, reference info, etc).

For an admin app, every page would likely have the main and navigation control bars, but the main control bar isn't mandatory. Most pages will have a form control bar and form. There also may be any number of custom control bars or areas—custom functionality needed for specific pages. Search filters for record lists would be custom controls. A custom control area is nothing more than an HTML/LDML snippet inserted at some point on the page within a HTML form container. For other other apps such as a signup series of forms, there probably would not be a home form, as there'd be no need for a list, and there would likely not be a main control bar either. The point being that EDP was originallt developed for a somewhat narrow purpose, but there is flexibility in design and what features are used.

Typical Workflow

Each EDP module can have a home form. That form would typically be a list of the records managed by that EDP module. The list would typically provide links to at least edit or delete the record, and perhaps allow the record to be duplicated or include other functions. That list may also have common search needs in a simple search filter (a small form on the list page). Advanced searches, if needed, can be performed on a separate Search page.

The first form (which must be named begin) is the mandatory form that must be completed and saved before any other form can be completed. When new records are created, it is the begin form that is always started and saved first. All subsequent forms by default can be selected at random, but the business logic can dictate that forms be completed in a specific sequence. The last form is usually a summary or preview. If used, it must be named summary, but can be visually labeled Preview or any other name that makes sense to the application (same is true for begin). Whenever a form is saved or cancelled, the landing point for that page action can be the summary form, or the next form in sequence, or any form based on business rules. The form is saved before the next page to display is even determined, so the result of the form save can be used to influence which form is displayed next.

MVC Pattern and EDP

The EDP system is constructed as a Model View Controller (MVC) pattern which works inside the standard PageBlocks page assembly engine. This means the standard PageBlocks lgc/dsp page system still governs basic page assembly, and inside those files we invoke the EDP system. Behind the scenes, there are five Lasso ctypes which provide all the workflow control, data source CRUD (create, read, update, delete) interfacing, and standard user interface functionality of EDP. The view ctype is used directly as a library of drawing routines, but the model and controller ctypes are abstract classes to be subclassed in the application.

Each form (home, begin, summary and others) within the EDP module will have a developer-created controller ctype and model ctype. These ctypes are sub-classed from master classes provided by the framework. They're easy to create and require only a handful of lines of app-specific code each to engage the full functionality of the EDP system. Typically, there can be much more code to allow for app-specific validation and business rules, but in a straight-forward module where all validation can be handled through the table config file and there are no business rules, there is very little programming required.

The controller ctype concerns itself with responses to button clicks from all GUI control bars. There are several pre-defined buttons which are handled automatically. The standard response may be overridden for customized responses, and custom buttons are also accommodated. A button's response may be as simple as jumping to another form in the module, or it may invoke a database select or update (for which the model ctype is called upon to execute).

The model ctype concerns itself with interfacing to data sources to manage the selecting, adding, updating, and deleting of records. The model's abstract ctype is built upon the PageBlocks `fwpActn_recordData` data type. Many database actions are handled automatically requiring no application-specific query code at all in the EDP model ctypes, especially if the data being handled is from a single table. If more complex data needs to be handled, the developer can readily integrate app-specific queries where needed.

In addition to the one controller and one model ctype for each tab in the EDP module, the developer also creates a main controller. The main controller is a factory pattern which is used to instantiate the controller/model ctypes as needed.

As for views, typically, an EDP module is just one PageBlocks "page." That is, there is only one page name, and one page's worth of lgc and dsp files. Indeed, most EDP content will be implemented within one page block of that page. The content of that page block is changed based on the user's button clicks, but it all happens within one page. So, drawing page displays is handled by a combination of the page block dsp file and methods of the EDP view ctype which provides EDP-specific routines to dynamically alter various elements of the user interface. For example, a single form control bar file is used with an integral configuration data block. The EDP view ctype uses this one file to dynamically render which buttons will be drawn for any given form's form control bar. Each form and each control bar is a separate file. The assembly of the components into the page layout is determined by the developer in the coding of the page block's dsp file. (The current view system will need to evolve into some better classes in future versions to allow more flexibility in the configuration capabilities. After using this for two major apps, I've found some limitations.)

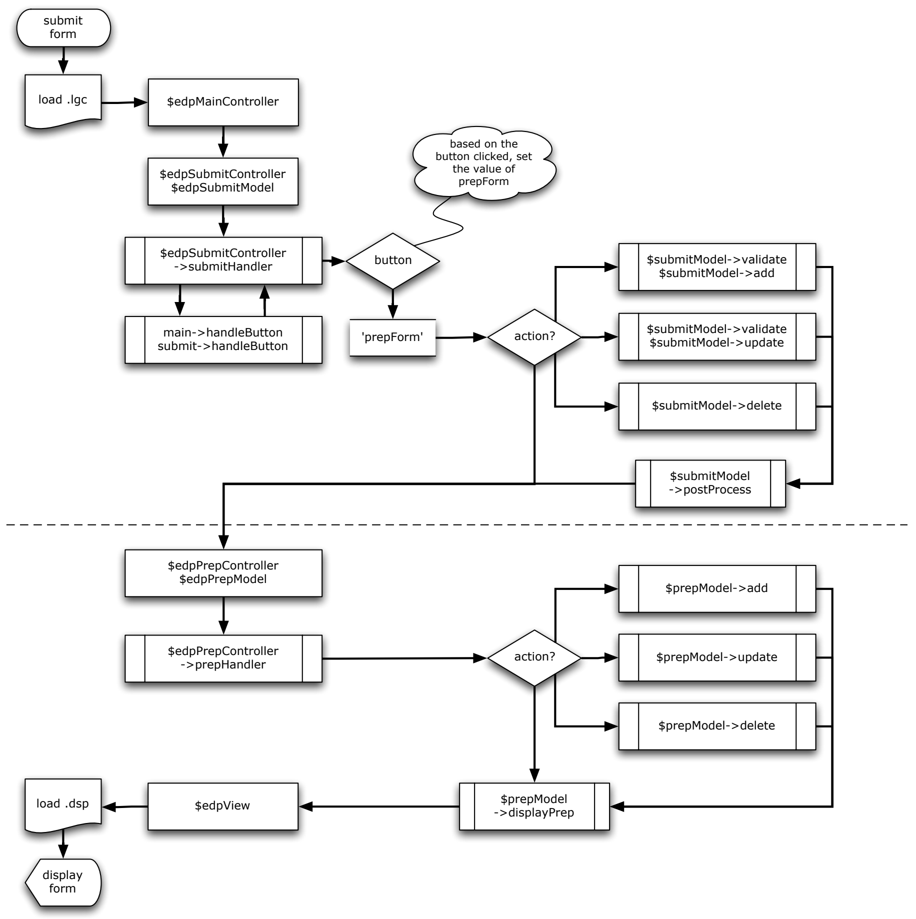
Each of the ctypes the developer has to create are fairly simple, follow consistent patterns, and can be built from provided sample files.

Code Process Flow

Have a look at Figure 3 which shows the overall logic flow of the EDP system from the time a form is submitted to when the next form is displayed. Notice first the dotted line across the illustration. Processing a page with EDP involves two distinct phases. In the first phase, represented above the dotted line, EDP concerns itself with the form which has been submitted. In the second phase, represented below the dotted line, EDP concerns itself with the form to be displayed. The two phases are important to keep in mind. Every member tag of the form controller and model ctypes has a role in either the submit or prep phase of processing the page. If you have a clear visualization of the distinction, it will help keep your thoughts organized as you write your code.

Following the logic diagram from the submit page starting point, the first thing to occur is that the .lgc file is loaded and the ctype `fwp_edpModule` is instantiated. Key events in `fwp_edpModule` begin with the instantiation of the `$edpMainController`. Based on the value (or lack) of `$fw_edpSubmittedForm`, passed by hidden input from the form, `$edpSubmitController` and `$edpSubmitModel` ctypes are created for the form which was just submitted. The `->handleSubmit` member tag is then called to determine which button was clicked to submit the page. That tag will call the `->handleButton` tags of the main controller and the submitted form controller to test for custom buttons. The end of that process will generate a value for the `prepForm` instance variable.

EDP Figure 3: Form Submission Flow Chart


submit form
load .lgc
$edpMainController
$edpSubmitController $edpSubmitModel
based on the button clicked, set the value of prepForm
$edpSubmitController ->submitHandler
button
main->handleButton submit->handleButton
'prepForm'
action?
$submitModel->validate $submitModel->add
$submitModel->validate $submitModel->update
$submitModel->delete
$submitModel ->postProcess
$edpPrepController $edpPrepModel
$edpPrepController ->prepHandler
action?
$prepModel->add
$prepModel->update
$prepModel->delete
load .dsp
$edpView
$prepModel ->displayPrep
display form

Next, based on which button was clicked, `->handleSubmit` will determine whether a database action is necessary. If an insert or update action is needed, `->handleSubmit` will invoke `->validate` of the submit form's model (coded by the developer), then either the internal `->add` or `->update` tag will be called as appropriate. After an add, update, or delete action, `->handleSubmit` will call the model's `->postAdd` or `->postUpdate` tag as approproate, and finally the `->postProcess` tag. This will complete the submit phase of the page processing.

The code now begins the prep phase. Based on the value of the submit controller's `prepForm` property, the appropriate controller and model for the page to be displayed will be created. Next, the prep form controller's `->prepHandler` is invoked. This process is simpler than the submit form's, but again, the button clicked will be used to determine if the form to be displayed needs to be prepared for a new record, a record update, or a delete record. Regardless of what actions needs to be taken by the parent ctype code, the `->displayPrep` tag is invoked (which is coded by the developer).

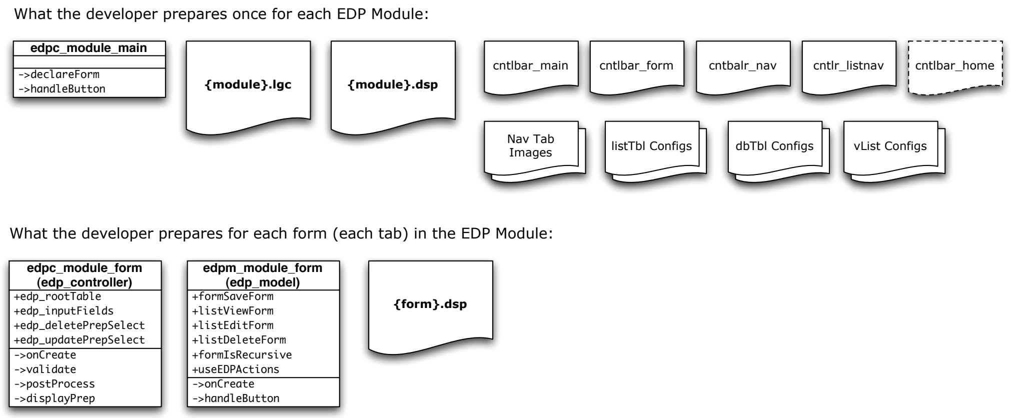
Finally, the `fwp_edpView` ctype is instantiated, and the pageblock's .dsp file is loaded.

File Organization

The EDP system uses the typical PageBlocks framework modular approach to file structure. To give EDP file snippets a structured place to exist, the following folders are in both the /site/ folder and in the module's /_resources/ folder:

- /edp_classes/—for developer authored form controller and model ctypes
- /edp_cntlbars/—for the various standard and app-specific control bars
- /edp_form/—for developer authored form bodies
- /edp_navImgs/—for images used to build the navigation tabs (note: regular button images all go in the usual PageBlocks /controls/ folder(s))
- /edp_panes/—for developer authored non-form display panes

EDP Figure 4: Files Involved with the EDP System



Creating EDP Models

There must be a model ctype file for every form in the EDP module. Each form must have a unique name which is used throughout the coding of the EDP module. There are three standard names: home, begin, and summary. The begin form is the only that is technically required. It would possible to build a single form solution that simply cycles back to itself.

Model file names will follow the pattern of edpm_ {moduleName}{tabName}.ctyp where epdm stands for EDP Model, moduleName is an arbitrary app-specific name the developer provides, and tabName is the programmatic name of the navigation menu button (a.k.a "tab") such as home, begin, and summary. The actual ctype class name must be identical to the file name (less the file extension). So, if we had a registration system programmed with EDP, and that system had tabs home, begin (which contains basic user contact info), profile (descriptions of the user), prefs (user application configuration preferences), and summary; then we'd have models with the following file and class names:

- file = edpm_registerHome.ctyp, class name = `edpm_registerHome`
- file = edpm_registerBegin.ctyp, class name = `edpm_registerBegin`
- file = edpm_registerProfile.ctyp, class name = `edpm_registerProfile`
- file = edpm_registerPrefs.ctyp, class name = `edpm_registerPrefs`
- file = edpm_registerSummary.ctyp, class name = `edpm_registerSummary`

Each of these files would be a sub-class of the built-in ctype named `fwp_edpModel` and would be a ctype defined with the typical member tags as shown below. The `onCreate` tag is required, and the others are optional, though `validate` and `displayPrep` are likely to always be used.

```
define_type: 'edpm_registerBegin', 'fwp_edpModel', -namespace='_global_';

    define_tag: 'onCreate';
    /define_tag

    define_tag:'validate';
    /define_tag;

    define_tag:'postProcess';
    /define_tag;

    define_tag:'displayPrep';
    /define_tag;

/define_type;
```

It is also possible to override built-in tags of the parent data types which handle data source interfacing for record inserts, updates and deletes. The developer can also add custom member tags to share methods with other models. The built-in tags are covered in detail in the reference, and the use of custom tags will be discussed later in this and other sections.

Model ->onCreate Tag

Of the four standard tags, only the `onCreate` tag must be defined. The `onCreate` tag declares the data source and the field details for the form. In our registration example, it was mentioned that the begin form would contain user contact data. An example `onCreate` tag might look like this:

```
define_tag:'onCreate';

    (self->'edp_rootTable') = @($fw_gTables->find:'registration');

    (self->'edp_inputFields') =
        'm_rApv, m_rgNmf, m_rgNml, m_rgCity, m_rgState, m_rgPhone, m_rgEmail';

    (self->'edp_updatePrepSelect') =
        'm_rApv, m_rgNmf, m_rgNml, m_rgCity, m_rgState, m_rgPhone, m_rgEmail';

    (self->'edp_deletePrepSelect') =
        'm_rApv, m_rgNmf, m_rgNml, m_rgCity, m_rgState, m_rgPhone, m_rgEmail';

    (self->init);

/define_tag;
```

Where the vars are defined:

- `edp_rootTable`—this is the variable of the fwpActn table map defined in `_initMasters.lgc` for the table managed by the EDP module. If more than one table is involved, pick the table you'd consider the root or main one. The variable can (should) be passed as a reference. (As of this writing, this needs to be the version 4 compatible table definition, but will change to the `$fw_gKeyTables` reference once the EDP ctypes have been updated internally to use the fwp_recordData API).
- `edp_inputFields`—this is a list of all HTML form input names (as defined the tableModel_ config file) that are allowed to be submitted when a record is created or updated with this form. This explicit list prevents the injection of non-specified form inputs into the database action that is performed.
- `edp_updatePrepSelect`— this is a list of all input names (as defined the tableModel_ config file) that are to be returned by the query SELECT (or FMP search) statement which is executed to retrieve data for displaying on an update form (which includes the begin form and all other editable forms)
- `edp_deletePrepSelect`— this is a list of all input names (as defined the tableModel_ config file) that are to be returned by the query SELECT (or FMP search) statement which is executed to retrieve data for displaying on a delete form. This may typically be the same as the summary form, but if they're different, this can be unique. If they are the same, it can be defined as a reference to `edp_deletePrepSelect` to save the duplicate typing.

Model ->validate Tag

The `validate` tag is auto-invoked by the standard response to the Save button of a form control bar (named `btnFormSave`) prior to a new record being created or an existing record being updated.

The code inside the tag is entirely up to the developer. While the name is validate, technically this could be a considered a pre-process method to perform any data manipulation or processing prior to the record being saved/updated.

If validations in the begin model are different based on a record being new or updated, the developer can use `if:((var:'fw_edpNewRcrdFlag') == 'Y')`.

The parent ctype will have already called the `fwpErr_validateInputs` ctag and the standard validation variables of `$fw_formIsValid` and `$fw_formIsNotValid` will have already been set before the `->validate` member tag is called. It will be up to the programmer to alter the values of those variables during the custom validation procerssing.

Model –>postAdd, –>postUpdate, –>postDelete, –>postProcess Tags

The `->postXXX` tags are auto-invoked by the standard response to the Save button of a form control bar (named `btnFormSave`) after a new record has been created or an existing record has been updated. The `->postProcess` tag is also invoked after a control bar Delete button (named `btnFormDelete`) is clicked. The code inside these tags is entirely up to the developer. Each is technically overriding an internal tag, so there is no need to define them in the model if they're not needed.

The processing order of these tags is `->validate`, `->postUpdate` (or `->postAdd` or `->postDelete`), and then `->postProcess`.

Model –>displayPrep Tag

The `->displayPrep` tag is auto-invoked prior to a form being displayed. For forms needing data from only a single a table, the EDP system will automatically load the data needed to populate the form, so the developer does not have to provide code for that. However, if the data loaded by EDP from the data table needs to be manipulated in any way (i.e. SSNs or phone numbers need unpacked, encrypted fields need decrypted), then that code needs to occur within the `displayPrep` tag. Additionally, if the form presents value lists or field data derived from other data tables, then those queries must be hand coded here. If anything other than a simple retrieval of the field data from the data table must be done to supply the form with everything it needs, that extra code goes in the `displayPrep` tag.

Just to be clear, the `displayPrep` tag does not create graphics for display, it prepares data required by the display. The data itself may be displayed, or be needed to control display conditionals.

Overloading fwpEDP_model Parent Tags

The built-in `fwpEDP_model` ctype provides several standard tags which are invoked by the EDP system. The standard actions provided by these tags may need to be customized for some forms. To do that, the developer creates a member tag in that form's model with the name of the built-in member tag to overload the default behavior. The developer must be aware whether the standard member tag actually needs to be customized or whether there simply needs to be special tasks performed after the standard actions in which case the `postProcess` member tag should be used.

When a built-in member tag is overloaded, the developer is responsible for performing not only the processing the application requires, but also the processing which the EDP system requires that is provided in the parent type. Each built-in member tag has unique needs. Use the EDP Parent CTypes Reference for information on exactly what an overload tag must do.

Creating EDP Form Controllers

There must be a controller ctype file for every form in the EDP module. Each form must have a unique name which is used throughout the coding of the EDP module. There are three standard names: home, begin, and summary. The begin form is the only that is technically required. It would possible to build a single form solution that simply cycles back to itself.

Controller file names will follow the pattern of edpc_{moduleName}{tabName}.ctyp where epdc stands for EDP Controller, moduleName is an arbitrary app-specific name the developer provides, and tabName is the programmatic name of the tab (i.e. home, begin, summary). The actual ctype class name must be identical to the file name (less the file extension). So, if we had a registration system programmed with EDP, and that system had tabs home, begin (which contains basic user contact info), profile (descriptions of the user), prefs (user application configuration preferences), and summary; then we'd have controllers with the following file names:

- file = edpc_registerHome.ctyp, class name = `edpc_registerHome`
- file = edpc_registerBegin.ctyp, class name = `edpc_registerBegin`
- file = edpc_registerProfile.ctyp, class name = `edpc_registerProfile`
- file = edpc_registerPrefs.ctyp, class name = `edpc_registerPrefs`
- file = edpc_registerSummary.ctyp, class name = `edpc_registerSummary`

Each of these files would be a subclass of the built-in ctype named `fwp_edpController` and would be defined with the typical member tags as shown below. The `onCreate` tag is required, but `handleButton` is optional being necessary only if there are non-standard buttons in use.

```
define_type: 'edpc_registerBegin', 'fwp_edpController', -namespace='_global_';

    define_tag: 'onCreate';
    /define_tag

    define_tag:'handleButton';
    /define_tag;

/define_type;
```

It is also possible to override built-in tags of the parent data type which handles the responses to all the standard buttons and the navigation links. The developer can also add custom member tags though it is likely that all extensions to the controller are better served through the `handleButton` tag. The built-in tags are covered in detail in the reference.

Controller ->onCreate Tag

Each controller's `onCreate` tag defines the default response pages to a few of the standard buttons of the EDP user interface. There are several properties of the controller to define as shown in the example below:

```
define_tag:'onCreate';
    (self->'formSaveForm')    = 'summary';
    (self->'formCancelForm')  = 'summary';
    (self->'listViewForm')    = 'summary';
    (self->'listEditForm')    = 'begin';
    (self->'listDeleteForm')  = 'summary';

    (self->'formIsRecursive') = true;
    (self->'useEDPActions')   = true;

/define_tag;
```

Where the vars are defined:

- `formSaveForm`—this instance var determines which form is the response page to the `btnFormSave` button click (the Save button on a form control bar). The value is the form name.
- `formCancelForm`—this instance var determines which form is the response page to the `btnFormCancel` button click (the Cancel button on a form control bar). The value is the form name. This is only defined if the button has a Cancel button.
- `listDeleteForm`—this instance var determines which form is the response page to the `btnListDelete` button click (the Delete button on a form control bar). The value is the form name. This is only defined if the form has a Delete button.
- `listViewForm`—this instance var determines which form is the response page to the `btnListView` button click. The value is the form name. This is only defined if the form uses a list that has a button to jump to a view form such as the typical `summary` type form.
- `listEditForm`—this instance var determines which form is the response page to the `btnListEdit` button click. The value is the form name. This is only defined if the form uses a list that has a button to jump to an edit form such as the `begin` form.
- `formIsRecursive`—this instance var declares a special case for some form pages. A particular form page is identified as being recursive if the page is designed to display related records of the originally selected record from home, and the page has both an add/update form and a list of sub-records on the same page. On such a page, the user would add or select for update a record from a list, and the form to add or modify a record is on that same page. When a record is added or updated, the page is refreshed and the list updated to show the record. In such a case, the vars `formSaveForm`, `listViewForm`, `listEditForm`, and `btnListDelete` are all set to the current form name, and `formIsRecursive` is set to true. The var need not be declared for forms which are not recursive.
- `useEDPActions`—this instance var determines whether the standard built-in database table actions should be invoked for this tab's page. On a typical form page like begin with a regular HTML data entry form, the EDP system will automatically attempt to locate and load a database record to populate the form. It will also invoke standard processes in response to the Save and Delete buttons of the form control bar. If for any reason those standard data actions should not be invoked for this tab, then set the `useEDPActions` instance variable to true. For example, a home page with a list of records does not need to invoke the default actions. Likewise, for a form where the developer has written custom query actions, the default actions should be disabled as well.

The values can be defined conditionally to allow for business rules that govern the flow of form presentation. For example, if you have a signup process, you would probably want the forms to be presented in order, meaning each form when saved will show the next form. Once the signup process is done, perhaps you'd want the summary form to be presented after each Save action. This can be done something like this:

```
define_tag:'onCreate';

    if: $signupCompleted;
        (self->'formSaveForm')    = 'summary';
    else;
        (self->'formSaveForm')    = 'profile';  // the form after begin
    /if;

    (self->'formCancelForm')      = 'summary';

    (self->'useEDPActions')   = true;

/define_tag;
```

Controller ->handleButton Tag

Each controller's `handleButton` is auto-invoked prior to processing the standard buttons. When a button is clicked and the page is submitted, the sequence of events is to first test whether the tab navigation buttons were clicked, then to test for custom buttons in the `main` controller's `handleButton` tag, then to test for custom buttons in the form controller's `handleButton` tag.

So, if a custom button is displayed on the begin form, and the begin form's `formSaveForm` variable is set to summary, then the `handleButton` tag of the summary controller must have code to respond to that custom button click. A typical structure for this would be like this:

```
define_tag:'handleButton';

//  ---------------------------------------------------------------------------
    if: var:'btnNameA.x';


//  ---------------------------------------------------------------------------
    else: var:'btnNameB.x';

    /if;
/define_tag;
```

It was noted that the `handleButton` tag is called in advance of handling all other standard buttons except for the navigation tabs. The sequence of events allows the opportunity to do some pre-processing for the submitted form (even before the model's `validate` member tag is called). If the developer includes a standard button name in the `handleButton` tag that code will be executed before the standard response. Note that the standard response will still be executed.

Overloading fwpEDP_controller Parent Tags

To overload the actions taken by one of the standard EDP buttons, do not use `handleButton` to catch the button click. Instead, find the parent ctype member tag that is invoked in response to the button click, and write a custom member tag to overload the parent tag.

There are two member tags of the `fwpEDP_controller` that should not be overloaded: namely the `submitHandler` and the `prepHandler` tags. These tags do the first layer of response of determining which specific member tag to invoke in response to a button click. There's simply too much going on in these tags to make it advantageous to overload them. Instead, overload the specific button response member tags.

The built-in `fwpEDP_controller` ctype provides several standard tags which are invoked by the EDP system. The standard actions provided by these tags may need to be customized for some forms. To do that, the developer creates a member tag in that form's controller with the name of the built-in member tag to overload the default behavior. When a built-in member tag is overloaded, the developer is responsible for performing not only the processing the application requires, but also some processing which the EDP system requires. Each built-in member tag has unique needs. Use the EDP Parent CTypes Reference for information on exactly what an overload tag must do.

Creating an EDP Main Controller

Each EDP module has a single main controller which serves two purposes. First, as the user navigates from form to form, the main controller instantiates the form controllers and form models of the submitted and response (a.k.a. prep) forms. In this role, the main controller serves as a factory pattern. Secondly, the main controller houses the `handleButton` member tag which provides a place to code responses to custom buttons in the main control bar.

There is only one main controller ctype file for the EDP module. It is subclassed from `fwpEDP_main`, and has a name following the pattern of edpc_{moduleName}Main.ctyp where epdc stands for EDP Controller, moduleName is an arbitrary app-specific name the developer provides, and Main is the required programmatic name. The main controller has four common but optional member tags., but has two required instance vars that require declaration. The overall structure looks like this:

```
define_type:'edpc_registerMain', 'fwpEDP_main';

    local:
        'formModels' = (map),
        'formControllers' = (map);

    define_tag:'init';
    /define_tag;

    define_tag:'handleButton';
    /define_tag;

    define_tag:'storeSelection';
    /define_tag;

    define_tag:'clearSelection';
    /define_tag;

/define_type;
```

The `->init` member tag is called automatically by the parent type's `->onCreate` tag which should not be overloaded. It can be used to initialize anything needed by the application.

The `->handleButton` tag is only necessary if custom buttons are used in the main control bar.

The `->storeSelection` tag is a conventional place to capture information about a record selected from the home list for easy reuse in subsequent forms.

The `->clearSelection` tag is a conventional place to reset the stored information about a record selected from the home list.

The form models and controllers are loaded into the `_global_` namespace for performance reasons (once loaded they remain in Lasso's tag list just as if they were loaded from LassoStartup). This requires that there be a way to make them easy to work with during development so code changes can be implemented without restarting the Lasso Site. There is a variable named `$fw_edpForceClassLoad` that can be set to `true` or `false`. Set to `true`, the ctype code is forced to be reloaded with each page making it very easy to develop with. When the code for all ctypes is stable, set the var to `false`, or just eliminate the use of it. I find it handy to declare it in the _pageConfig file like this:

```
$fw_edpForceClassLoad = true; // set to false after development is done
```

Main Controller #formModels and #formControllers Maps

The main controller's `formModels` and `formControllers` instance variables are used to associate the forms by name with their model and controller custom types so that an internal tag can make sure the model and controller types are loaded when they're needed. Populating the vars is pretty straight forward. You end up with something like this:

```
local:
    'formModels' = (map:
        'home'     = 'edpm_usrsHome',
        'begin'    = 'edpm_usrsBegin',
        'whatever' = 'edpm_usrsWhatever',
        'summary'  = 'edpm_usrsSummary'),
    'formControllers' = (map:
        'home'     = 'edpc_usrsHome',
        'begin'    = 'edpc_usrsBegin',
        'whatever' = 'edpc_usrsWhatever',
        'summary'  = 'edpc_usrsSummary');
```

Main Controller –>handleButton Tag

The main controller's `handleButton` tag is automatically invoked prior to processing the standard buttons. When a button is clicked and the page is submitted, the sequence of events is to first test whether the tab navigation buttons were clicked, then to test for custom buttons in the main controller's `handleButton` tag, then to test for custom buttons in the response page form's `handleButton` tag.

So, if a custom button is included in the main control bar, and the action taken by that custom button is identical regardless of the current form page being displayed, then the `handleButton` tag of the main controller must have code to respond to that custom button click. A typical structure for this would be like this:

```
define_tag:'handleButton';

//  ----------------------------------------------------------------------------
    if: var:'btnNameA.x';


//  ----------------------------------------------------------------------------
    else: var:'btnNameB.x';


    /if;
/define_tag;
```

It was noted that the handleButton tag is called in advance of handling all other standard buttons except for the navigation tabs. The sequence of events allows the opportunity to do some pre-processing for the submitted form. If the developer includes a standard main button name in the handleButton tag that code will be executed before the standard response to allow for some customization of the response. Note that the standard response will still be executed.

To overload the standard response altogether works contrary to what you think might be intuitive. The parent member tags which respond to main control bar button clicks are not in the parent fwpEDP_main ctype. Rather the default behavior code is in the fwpEDP_controller ctype. The reason for this is that it is expected that if the code has to be overloaded, it is likely needed to customize the response for each form. Therefore, it is easier to do that if the parent tag is already located in the form controller. Also, somewhat less importantly, it simplifies the code in the pageBlock's .lgc file to have to call only one method to test for all button clicks. If a custom behavior of a standard button is needed and the action is common to all forms, it is easier to simply create a custom button in the main control bar.

Main Controller vs. Form Controller –>handleButton Tags

For clarity, let's discuss the differences of these two member tags, and when one vs. the other should be used. Graphically speaking, the main control bar of the typical EDP user interface contains buttons that, no matter which form is being displayed, will generate an identical visual response. No matter which form is displayed, clicking the Search button displays the advanced search form, clicking the New Record button begins a new record, etc.

Therefore, graphically, buttons added to the main control bar should be for things of that nature. The code to respond to such custom buttons should be generic enough that it would be identical regardless of the current form, so the code should be located in the main controller handleButton member tag. If for some reason, the button response code has to be different for each form, then consideration should first be made as to whether the button really belongs in the main controller bar, but if it does, then the response code to the button should be in each form controller's handleButton member tag.

PageBlock .lgc File Setup

The EDP system is invoked within a template's pageblock. While the vast majority of app-specific EDP code is written as custom types, a standard pageblock logic file is used to initialize the EDP module on a given page.

```
var:'usrsEDPModule' = (fwp_edpModule:
    -mainController = 'edpc_usrsMain',
    -rcrdsListName = 'fwpUserAdmin');
```

Where `mainController` (required) is the custom type name of the module's main controller, and `rcrdsListName` (optional) is the name of the list object used on the home page if a list is used. There are no tags to call, or vars to reference in the `fwp_edpModule` data type. All the work is done with the type's `onCreate` tag. For details about what is going on inside the type, see the reference for `fwp_edpModule`.

PageBlock .dsp File Setup and EDP Views

The pageblock .dsp file is where the final assembly, so to speak, of the page display occurs. The EDP view custom type includes member tags for drawing specific elements of the typical EDP user interface, but the sequence of the drawing and some conditional coding for what to draw is determined in the page's .dsp file. While there are typical patterns for this file, it is also an area where customized assembly of control bars, forms, and panes occurs, and will require some app-specific code by the developer. Let's have a look at a typical case, and discuss each part.

The PageBlock .dsp File

First up is nothing special—just typical HTML is needed to form the page title.

```
<?LassoScript //------------------------------------------------------------ Page Heading
?>

    <h2>Module Title</h2>
```

As most forms are likely to require controlled access we use a typical PageBlocks authorization test and check for errors befor displaying the form page content, or else we display an error message. This can be handled differently based on the specific techniques employed in the application.

```
[if: ($fw_user->(getProfile:'loginValid') == 'Y') && !($fw_gErrorMsg)]
```

Assuming there's no problems with showing the form page, we first draw the main control bar. The contents of the main control bar are determined by the file /edp_cntlbars/cntlbar_main.dsp. Refer to the *Controller Display Files* section for details.

```
<?LassoScript //------------------------------------------------------------ Main Control
Bar
?>

    [$edpView->drawMainCntlr]
```

Next, the drawing of the home page (as well as the Search page if used and the Prefs page if used) vs. the rest of the "tabbed" forms is quite different, so a simple conditional is tested to see if we're drawing any of the non-tabbed-form pages. Note that to pass on the form or page currently being displayed we use the `$edpPrepController` object's `prepForm` property. The `drawForm` member tag does little more than include the file, but it abstracts the process so that we can pass a variable containing the filename.

```
<?LassoScript //------------------------------------------------------------ Home Body
?>

    [if: (($edpPrepController->'prepForm') == 'home')]

        [$edpView->drawForm:($edpPrepController->'prepForm')]
```

The srch (and prefs) forms require a little more work than the home form. You'll see that this is where the actual HTML form tags are coded. Indeed all HTML form tags used throughout the EDP module are coded in this .dsp file. In some cases (srch and prefs), it would be just as effective to code the HTML form tags in the files with the form bodies. However, putting all of the form tags in this file centralizes all those tags, reduces redundancy, and makes it easier to view all the form action declarations in one place. The srch and/or prefs code can be eliminated if they're not used, but there's also no harm aside from one more conditional test in leaving it in.

```
    [else: (($edpPrepController->'prepForm') == 'srch') ||
           (($edpPrepController->'prepForm') == 'prefs')]

        <form id="edpForm" name="edpForm" action="[$c_myURL->'self']" method="post">
           <input type="hidden" name="fw_s" value="[var:'fw_s']" />
           [$edpView->drawForm:($edpPrepController->'prepForm')]
        </form>

    [else]
```

Finally, we get to the block which draws the tabbed form pages. This examples shows how some page elements such as help panes can be user profile driven. For EDP systems where users frequently use the same module, help can be configured to be shown or hidden. New users can have them showing, while experienced users can turn the help off. (Of course this can be done with popups and CSS for on-demand veiwing, but this is just an example).

After any conditional content to display below the main control bar, the first drawn item is usually the navigation control bar. The contents are determined by the configuration data declared in the /edp_cntlbars/cntlbar_nav.dsp file.

Next is the form itself. You will have noticed that each form must propogate the PageBlocks user session. Inside the HTML form tags we draw the form control bar (/edp_cntlbars/cntlbar_form.dsp) and the form body (/edp_forms/form_{name}.dsp, where name is the form's programmatic name).

If a page is made up of more than one form, or a form with a list, or even several lists with no form, then use conditional code to determine any other files which should be drawn or included based on the current form name. The example below shows that if the current form name is vndrs that there is a file named vndrsList (actually /edp_forms/form_vndrsList.dsp) to be included below the standard form body. The developer has complete freedom to layout the page as needed here.

```
<?LassoScript //------------------------------------------------------------ Form Body
?>

        [if: $fw_user->(getProfile:'userShowHelp') == 'Y']
           [$edpView->drawPane:'help']
        [/if]

        [$edpView->drawNavCntlr]

        <form id="edpForm" name="edpForm" action="[$c_myURL->'self']" method="post">
           <input type="hidden" name="fw_s" value="[var:'fw_s']" />
           [$edpView->drawFormCntlr]
           [$edpView->drawForm:($edpPrepController->'prepForm')]
        </form>
        [if: ($edpPrepController->'prepForm') == 'vndrs']
            [$edpView->drawForm:'vndrsList']
        [/if]
    [/if]
```

And, to wrap things up we have the call to display an error message if there was a problem with eth user's authorization or any problems in the .lgc file.

```
[else]
    [$fw_gErrorMsg]
[/if]
```

If you put all of the above code pieces together, you have the skeleton for a typical EDP module's .dsp file along with some options.

The EDP View CType and View Files

EDP pages are drawn by assembling several view snippets. These files are located in standardized folders. Files unique to a specific EDP module are stored in the PageBlocks module's /_resources/ folder in /edp_forms/, /edp_cntlbars/, and /edp_panes/. Navigation tab images are in the /edp_navImgs/ folder. For files which might be common to more than one EDP module, these same folder names are located in the PageBlocks /site/ folder. The `fwpEDP_views` ctype, which is responsible for rendering these files to the display, will automatically find the snippet files in either location.

Some of the snippet files, such as the data entry forms, are straight forward HTML/LDML snippets just like you'd expect them to be. Other files, namely some of the control bars, are a combination of HTML/LDML code along with embedded EDP-specific configuration definitions. The cntlbar_nav.dsp file is a good example. This file includes a configuration data block which identifies which graphics are to be included when drawing the navigation tabs. The actual HTML/LDML code needed for the tabs is somewhat complex and tedious to build by hand (and in fact two versions are needed). So, the view ctype uses the config data to generate the required code on the fly. This is also true of the main control bar and the form control bar. If performance is critical, or the standard config and snippet setup just doesn't meet the module's needs, the developer can always hand code the complete snippet and simply `include` it instead of using the view ctype.

There are three types of snippet files managed by the `fwpEDP_views` ctype: forms, control bars, and panes which are located in the /edp_forms/, /edp_cntlbars/, and /edp_panes/ folders respectively. Form file names follow the pattern form_{name}.dsp where name is the form name (i.e. home, begin, summary). Control bar names follow the pattern cntlbar_{name}.dsp where name is either main, form, or nav corresponding to the standard control bars, or an arbitrary name if custom control bars are used (such as list filters). Pane file names follow the pattern pane_{name}.dsp where name is arbitrary.

The `fwpEDP_views` ctype generates code for typical EDP screen components and provides abstraction for including app-specific view snippets. The view ctype includes the following member tags for drawing snippets:

- `drawMainCntlr`—this is a tag dedicated to locating and processing the integral config data of the cntlbar_main.dsp file from the /edp_cntlbars/ folder. Requires no inputs.

- `drawNavCntlr`—this is a tag dedicated to locating and processing the intgeral config data of the cntlbar_nav.dsp file from the /edp_cntlbars/ folder. Requires no inputs.
- `drawFormCntlr`—this is a tag dedicated to locating and processing the intgeral config data of the cntlbar_form.dsp file from the /edp_cntlbars/ folder. Requires no inputs.
- `drawCntlr`—locates and includes an arbitrary cntlbar_{name}.dsp file from the /edp_cntlbars/ folder. Requires the name portion of the file name as an input.
- `drawForm`—locates and includes a form_{name}.dsp file from the /edp_forms/ folder. Requires the name portion of the file name as an input.
- `drawPane`—locates and includes a pane_{name}.dsp file from the /edp_panes/ folder. Requires the name portion of the file name as an input.

There are additional member tags, but they are for internal use only. There's pretty much no need to overload the member tags of the view data type. As mentioned above, if the special file formats of the main, form, and nav control bars don't allow for some unique need, it's just as easy to hand code the custom control bar and use a regular Lasso include (or even embed it in the PageBlock .dsp file).

The cntlbar_main Snippet File

The cntlbar_main file is a hybrid config and HTML/LDML file. The first part of the file includes a configuration block which defines the buttons to be included in the main control bar. The second part of the file is the display code which can include HTML and LDML code. Below is a typical example of the file. The file is processed by `fwpEDP_view->drawMainCntlbar` by first splitting the document into an array of individual lines, removing each line which is either empty or a comment, then rejoining the array into a string. Next, the config data is extracted, processed, and the resulting HTML code is inserted into the display code to replace the `{edp_mainCntlbar_buttons}` place holder. That result is then run through the Lasso `process` tag so that the display code may contain both HTML and LDML app-specific modifications.

The config data section is enclosed by the `{cntlbar_buttonList=}` container. Note that in the sample code below, the lines for `btnMainViewList`, `btnMainShowAll`, etc. are shown on two lines for this documentation but that the real code must be on single lines.

```
# ---------------------------------------------
# Config Data

# this next line is a syntax reminder
# btnName```btnImageLocation```formAction```hiddenInputsCommaList
# hiddenInputs are ```name===value, name===value, etc...
# value can be LDML code as it will be processed

{cntlbar_buttonList===

btnMainViewList```[($c_gPath->'controls') + 'btn_rtn2list_ylw.gif']```
    [$c_myURL->'self']```fw_s===[var:'fw_s']

btnMainShowAll```[($c_gPath->'controls') + 'btn_resetlist_ylw.gif']```
    [$c_myURL->'self']```fw_s===[var:'fw_s']

btnMainNewRcrd```[($c_gPath->'controls') + 'btn_newrcrd_ylw.gif']```
    [$c_myURL->'self']```fw_s===[var:'fw_s'],fw_pnl===main
```

```
#btnMainPreferences```[($c_gPath->'controls') + 'btn_preferences_ylw.gif']```
    [$c_myURL->'self']```fw_s===[var:'fw_s'],fw_pnl===pref

btnMainSearch```[($c_gPath->'controls') + 'btn_search_ylw.gif']```
    [$c_myURL->'self']```fw_s===[var:'fw_s'],fw_pnl===srch
}
```

In the above example, the line for `btnMainPreferences` is commented out. Config lines for buttons that aren't being used can be removed or simply commented out if you want to keep the code intact as a template for creating new EDP modules.

The next section of the file is for the actual display code. For the most part, the default code should work. It generates a single row table with a separate cell for each button so that each button can be its own form. This code can be added to if necessary to add other features. If a non-table design is needed, the developer will have to use a custom cntlbar file.

```
# ---------------------------------------------
# Display Code

<div id="edpcntlbarmain">
<table cellspacing="0">
    <tr>
        <td>
        </td>
        {edp_mainCntlbar_buttons}
    </tr>
</table>
</div>
```

The cntlbar_form Snippet File

The cntlbar_form file is a hybrid config and HTML/LDML file. The first part of the file includes a configuration block which defines the buttons to be included in the control bar. The second part of the file is the display code which can include HTML and LDML code. Below is a typical example of the file. The file is processed by `fwpEDP_view->drawFormCntlbar` by first splitting the document into an array of individual lines, removing each line which is either empty or a comment, then rejoining the array into a string. Next, the config data is extracted, processed, and the resulting HTML code is inserted into the display code to replace the `{edp_formCntlbar_inputs}` and `{edp_formCntlbar_buttons}` place holders. That result is then run through the Lasso `process` tag so that the display code may contain both HTML and LDML app-specific modifications. The config data section is enclosed by the `{cntlbar_buttonList=}` container.

```
# ---------------------------------------------
# Config Data

# button values are simply root rel file names for the images

{cntlbar_buttonList===
btnFormSave===[($c_gPath->'controls') + 'btn80_save_grn.gif']
btnFormCancel===[($c_gPath->'controls') + 'btn80_cancel_ylw.gif']
btnFormDelete===[($c_gPath->'controls') + 'btn80_delete_red.gif']
}
```

```
# css classes
{cntlbar_buttonClasses===
btnFormSave===formbtnsave
btnFormCancel===formbtncancel
btnFormDelete===formbtndelete
}

# tool tip strings
{cntlbar_buttonTitles===
btnFormSave===Saves the form data as shown, and jumps to the Summary page.
btnFormCancel===Ignores changes made to the form, and jumps to the Summary page.
btnFormDelete===Deletes this record.
}

# hiddenInputs are ```name===value, name===value, etc...
# value can be LDML code as it will be processed

{cntlbar_newRcrdInputs===}
{cntlbar_updateRcrdInputs===}
{cntlbar_deleteRcrdInputs===}

{cntlbar_useDefaultInputs===true}
```

If the `cntlbar_useDefaultInputs` setting is true the EDP view ctype will insert standard hidden inputs needed by EDP as well as those specified in the inputs config list. This value should normally be true. If there is some need to customize the default behavior, the value can be set to false, and the developer is responsible for seeing the application code meets EDP needs and includes all other inputs required and any condition code for them. The following list details the standard inputs created when `useDefaultsInputs` is set to true:

- New Record
  - `fw_edpNewRcrdFlag` = literal value of “Y” (no quotes)
  - `edp_keyval` = the record primary key value
- Update Record
  - `edp_keyval` = the PageBlocks rcrdLockID field value
- Delete Record
  - `edp_keyval` = the PageBlocks rcrdLockID field value

The next section of the file is for the actual display code. For the most part, the default code should work. It generates a single row table. This code can be added to if necessary to add non-standard features. A typical feature of the default EDP UI is to show a large red bar under the control bar if the form is submitted with input validation errors. This may be something the developer wants to modify. If a non-table design is needed, the developer will have to use a custom cntlbar file.

```
# ---------------------------------------------
# Display Code

[if: ($fw_gError->size) == 0]

<div id="edpcntlbarform">
    <table cellspacing="0">
```

```
        <tr>
            <td class="right">
                {edp_formCntlbar_inputs}{edp_formCntlbar_buttons}
            </td>
        </tr>
    </table>

# validation error alert bar

[if: !$fw_gFormIsNotValid]
    [include: ($c_gPath->'panels') + 'invalidEntryBar.dsp']
[/if]

</div>
[/if]
```

The cntlbar_nav Snippet File

The cntlbar_nav file is a hybrid config and HTML/LDML file. The first part of the file includes a configuration block which defines the graphics to be included in the control bar. The second part of the file is the display code which can include HTML and LDML code. Below is a typical example of the file. The file is processed by `fwpEDP_view->drawNavCntlbar` by first splitting the document into an array of individual lines, removing each line which is either empty or a comment, then rejoining the array into a string. Next, the config data is extracted, processed, and the resulting HTML code is inserted into the display code to replace the `{edp_navCntlbar_buttons}` place holder. That result is then run through the Lasso `process` tag so that the display code may contain both HTML and LDML app-specific modifications. The config data section is enclosed by the `{nav_imageList=}` container.

The `drawNavCntlbar` member tag will automatically render the tab graphics as on, off, or dimmed as needed based on context.

```
# ---------------------------------------------
# Config Data

# each imageList line is formatted as: btnName```imgLocation```width
# imgLoc does not include _on|_off|_dim file name elements -- that is added by the tag
# First button MUST be named btnNavBegin.
# The button name (less the `btnNav` string) must equal the form name
# so that `btnNavBegin` will open the form `form_begin.dsp`
# and `btnNavFilters` will open the form `form_filters.dsp`
# the button image file name can be anything

# file extension of all images must be the same
{nav_imageType===.gif}

# all image heights must be the same.
{nav_imageHeight===22}

{nav_inputList===
    fw_s===[var:'fw_s']
}
```

```
{nav_imageList===

imgFill```[($fw_sPath->'edp_navImgs') + 'tab_fill']```10

btnNavBegin```[($fw_sPath->'edp_navImgs') + 'tab_auth']```71
imgFill```[($fw_sPath->'edp_navImgs') + 'tab_fill']```3

btnNavProfile```[($fw_sPath->'edp_navImgs') + 'tab_profile']```71
imgFill```[($fw_sPath->'edp_navImgs') + 'tab_fill']```3

btnNavPrivileges```[($fw_sPath->'edp_navImgs') + 'tab_privileges']```82
imgFill```[($fw_sPath->'edp_navImgs') + 'tab_fill']```3

btnNavFilters```[($fw_sPath->'edp_navImgs') + 'tab_filters']```71
imgFill```[($fw_sPath->'edp_navImgs') + 'tab_fill']```3

imgFill```[($fw_sPath->'edp_navImgs') + 'tab_fill']```274

btnNavSummary```[($fw_sPath->'edp_navImgs') + 'tab_summary']```99
imgFill```[($fw_sPath->'edp_navImgs') + 'tab_fill']```10
}
```

The next section of the file is for the actual display code. For the most part, the default code should work. It generates a single div with a form container with the butted graphic images. This code can be added to if necessary to add non-standard features. If a some other design is needed, the developer will have to use a custom cntlbar file.

```
# --------------------------------------------
# Display Code

<div id="edpcntlbarnav">
{edp_navCntlbar_buttons}
</div>
```

How-to FAQs

This section highlights a number of typical questions about the details of how to program for EDP once you have your files all setup.

How does EDP know which form to display?

Every form has to include a hidden input named `fw_edpSubmittedForm` which contains as a value the name of the form. That form name is used throughout the EDP code.

How does EDP know which record was originally selected?

Assuming the home form is using a `fwp_listRcrds` type for the list of records, the forms for the main buttons of that list type `btnListView`, `btnListEdit`, `btnListDelete` should pass the record key value in a hidden input named `fw_r` which the PageBlocks framework will convert to `$fw_r`, which the submit form controller will trap and store to a page variable `$edp_selectedRcrdID`, and which finally is stored to the logged-in user's session variables.

How can I store info about the selected record to show on each form page?

In your main controller code, create two methods that follow this general pattern. These two methods will be called by the form controllers at applicable points in the form pages workflow.

```
//=============================================================================
//  ->storeSelection
//  trap some info about the selected (or newly created) user
//  and store to session for recall on other pages

define_tag:'storeSelection';

    if: (var:'btnListView.x') ||
        (var:'btnListEdit.x') ||
        (var:'btnListDelete.x') ||
        ((var:'fw_edpNewRcrdFlag') == 'Y');

//  at this point the record data will have been loaded and inputName vars created
//  so, use the fwp_collection type to store anything you need on other pages
//  where these vars may not be loaded
//  use any var name you prefer in place of $selectedRecord

        var:'selectedRecord' = (fwp_collection:
            -nameFirst = var:'m_uNameF',
            -nameLast  = var:'m_uNameL');

    else: (var:'btnMainNewRcrd.x');

//  for new records, create an empty object so as not to cause syntax complaints
//  by other pages, do not save to the session
```

```
        var:'selectedRecord' = (fwp_collection:
            -nameFirst = '',
            -nameLast  = '');
    /if;

    (var:$fw_gUserVarName)->(addVars:'selectedRecord');

/define_tag;

//============================================================================
//  ->clearSelection
//  clears the var of a stored selection

define_tag:'clearSelection';

    var:'selectedRecord' = fwp_collection;
    (var:$fw_gUserVarName)->(addVars:'selectedRecord');

/define_tag;
```

EDP Parent CTypes Reference

This section provides a tag by tag reference to the built-in member tags of the core parent ctypes of the EDP system. Should the developer need to overload any of these tags, this section will provide the information needed to ensure that default EDP behavior is still performed by the app-specific overload code.

DOCUMENTATION NOTE: until this section is completed, refer to the source code of the parent ctypes. For the most part, use that code as a starting template for the overload tag code, and be sure to keep any internal instance vars populated as performed by the built-in code.

fwp_edpModule Reference

Instance Vars / Page Vars

```
local:
    'fw_mainController'    = string,
    'fw_defaultPage'   = string,
    'fw_rcrdsListName' = string;
```

Member Tags

```
->onCreate     // parses inputs, invokes a series of steps to initiate EDP
->init         // performs some cleanup tasks for each page
->setDefault   // sets the default form if submittedForm is not defined
->makeMain     // creates $edpMainController
->submitForm   // loads submit ctypes,
               // calls $edpSubmitController->submitHandler
->prepForm     // loads prep ctypes,
               // calls $edpPrepController->prepHandler
->makeView     // creates $edpView
```

The initialization process started in `->onCreate` has five distinct steps: a) initialize the current form variable; b) instantiate the main controller; c) instantiate the submitted form ctypes and process the button response; d) instantiate the ctypes for the form to be prepped for display; and e) instantiate the view ctype to be ready for the page block .dsp file.

In step (a), `$fw_edpSubmittedForm` is generated by including an HTML hidden input with every tab's form to identify which form was just submitted. If the var has not been declared, it means no form has been submitted and the display should default to the home form.

In step (b) the main controller ctype is instantiated.

In step (c) the incoming submitted form is handled. The main controller's `->loadFormClasses` tag is used to create the form model and controller objects. Then, the built-in `->submitHandler` tag of the form's controller is invoked and pass along the variable names of the submit model object and the main controller object.

In step (d) the new page display is prepared. Again, the main controller's `->loadFormClasses` member tag is used to create the model and controller objects, except this time based on the `prepForm` result of the `submitHandler` tag. Then the `prepHandler` tag of the prep form's controller is called.

In step (e), the built-in view object is instantiated passing along the form name and model name of the form to be displayed.

The page variables `$edpMainController`, `$edpSubmitController`, `$edpPrepController`, `$edpSubmitModel`, and `$edpPrepModel` are created during this process.

fwp_edpController Reference

Instance Vars

```
//  instantiation inputs
local:
'prepForm'         = string,
'prepModel'        = string,
'submitForm'       = string,
'submitModel'      = string,
'mainController'   = string,
'homeListName'     = string,
'submitKeyVal'     = string,
'prepRcrdID'       = string;

//  these are populated by various tags as needed
//  and may need passed to the model
local:
'useEDPActions'           = boolean,
'listViewForm'            = string,
'listEditForm'            = string,
'listDeleteForm'          = string,
'formSaveForm'            = string,
'formCancelForm'          = string,
'formIsRecursive'         = boolean,
'formCreatesNewRecord'    = boolean;
```

Member Tags

```
->submitHandler       // first responder to a form submission

// responders to button clicks in Main button bar

->btnMainViewHome     // standard response to the btnMainViewHome.x button event
->btnMainViewList     // standard response to the btnMainViewList.x button event
->btnMainShowAll      // standard response to the btnMainShowAll.x button event
->btnMainNewRcrd      // standard response to the btnMainNewRcrd.x button event
->btnMainSearch       // standard response to the btnMainSearch.x button event
->btnMainPreferences  // standard response to the btnMainPreferences.x button event

// responders to standard buttons of the home records list

->btnListView         // shows the form dictated by (self->'listViewForm')
->btnListEdit         // shows the form dictated by (self->'listEditForm')
->btnListDelete       // shows the form dictated by (self->'listDeleteForm')

//  responders to the standard buttons of a search form page and preferences form page

->btnSrchCancel
->btnSrchSearch
->btnPrefsCancel
->btnPrefsSave
//  responders to the standard form controller buttons

->btnFormCancel
->btnFormSave
->btnFormDelete
```

```
//  misc tags

->rcrdUnlock       // frees any record whose lock ID is stored in a cookie

----------------------------------

->prepHandler      // first responder to a prep form event

->btnMainShowAllPrep   // clears any prior search criteria so all records can be shown
->btnSrchSearchPrep    // clears any prior search criteria so search form is “reset”

->addPrep          // generates an ID for the new record,
                   // generates vars so we can use $ syntax in forms
->deletePrep       // calls ->actnDeletePrep of the model object
->updatePrep       // calls ->actnUpdatePrep of the model object
->postUpdatePrep   // calls ->postUpdatePrep of the model object

->clearListInfo           // internal tag with shared functionality
->clearFormVars           // internal tag with shared functionality
->fw_getBtnListRcrdID     // internal tag with shared functionality
```

fwp_edpModel Reference

Instance Vars

```
//  these are populated by various tags as needed

local:
    'submitRcrdID'        = string,
    'rcrdLockID'          = string;

//  these are populated by the subclass onCreate tag

local:
    'edp_rootTable'           = string,
    'edp_inputFields'         = string,
    'edp_deletePrepSelect'    = string,
    'edp_updatePrepSelect'    = string,
    'edp_recordData'          = null,
    'edp_tblFieldNames'       = array,
    'edp_tblFieldNameMap'     = map,
    'edp_tblInputNames'       = array,
    'edp_tblValCodes'         = array;
```

Member Tags

```
//  these tags are expected to be overloaded

->onCreate
->validate
->postProcess
->postUpdatePrep
->postAdd
->postUpdate
->displayPrep

//  these tags are the interface to the abstract CRUD model
//  and can be overloaded if necessary to create hand coded queries

->actnAddPrep
->actnAdd
->actnUpdatePrep
->actnUpdate
->actnDeletePrep
->actnDelete
```

fwp_edpMain Reference

Instance Vars

none

Member Tags

```
->onCreate            //
->loadFormClasses     // loads ctypes if not yet loaded

//  these tags are expected to be overloaded

->init            //
->handleButton    //
->storeSlection   //
->clearSelection  //
```

fwp_edpView Reference

Instance Vars

Member Tags

```
->loadfile        // internal use
->drawbutton      // internal use to write button HTML code
->drawinput       // internal use to write input HTML code
->drawPane        // use to insert pane file
->drawForm        // use to insert form file
->drawCntlbar     // use to insert arbitrary control bar file
->drawMainCntlbar // use to insert main control bar code
->drawNavCntlbar  // use to insert nav control bar code
->drawFormCntlbar // use to insert form control bar code
```

Debugging Tools

The PageBlocks framework includes a handful of tools to help with troubleshooting, debugging, and optimizing application code whether or not you're using an IDE that offers debugging aids.

Debugging Control Variables

There are a few standard variables which control debugging modes across the entire framework. These are all found in section (I) of _initMasters.lgc.

- `$fw_debug` (integer) —set to > 0 to turn on debugging aids. Throughout the framework code there are variables which capture the state of data inside various tags & types that can be handy to view to see what's going on inside. Setting `$fw_debug` to > 0 will assign these variables and make them available to the variables output (discussed below). There are four constants used to determine the level of debug detail that is created.
  - `fw_kDisabled` (`0`) turns debugging off
  - `fw_kEnabled` (1) turns on basic debugging (shows major messages only)
  - `fw_kChatty` (3) turns on moderate debugging (shows major messages and internal details of major tags and types like loading of config files)
  - `fw_kVerbose` (5) turns on extensive debugging details (shows detailed use of almost every tag)
- `$fw_criticalLog` (boolean) — this enables writing of error messages of critical operations to write to Lasso's critical error log. This will typcally be errors related to setup errors including major configuration features, database availability, and file access errors.
- `$fw_debugIPFilter` (string) — this is a comma separated list of IP addresses of machines that the debugging display can be output to. In effect this allows the developer to turn on debugging on a production server, but limit the display to only his terminal while the app continues to be used.
- `$fw_debugTimers` (boolean) — set to true to activate and display various timers in the code which keep track of how long each piece of the application is taking to process. There are a number of timers internal to the framework code already, and the developer can add more in the application code.

In addition to the above mode control variables, the following standard objects (created in fwpPage_init) are available to aid in debugging and optimizing application code:

- `$fw_tagTracer` (fwp_tagTracer) — an instance of the `fwp_tagTracer` custom type which is used to record the sequenec in which major framework tags and custom type member tags get processed. The developer can also add trace points to the application code.
- `$fw_timer` (fwp_timer) — an instance of the `fwp_timer` custom type which stores multiple stop and start points based on named timers. It also provides totals for groups of related timers.
- `$fw_debugOutput` (fwp_showVars)— an instance of the `fwp_showVars` custom type which provides a structured nested output of all page and global variable contents including complex variables and custom types.

fwpUtil_tagTracer

This tool emulates the "stack trace" features of a compiled language debugger. It can be used to build a sequence of exactly what code points have been processed, and what variable states are at those points. Many of the framework tags and custom type member tags have trace steps already coded in them. Not every tag does, because some are just too small and benign to be concerned with. In addition to the built-in trace steps, the developer can add trace steps in the application code.

Interface Quick Glance

Create an fwp_tagTrace object (this is already done in fwpPage_init):

```
var:'fw_tagTracer' = (fwp_tagTracer);
```

Read-Only Instance Variables:

```
'traceData'
```

Member Tags:

```
->add
->tracePoint
->intervals
```

Public Read-Only Instance Vars

- `traceData` (complex) = contains individual trace points with millisecond start time stamp and optionally with variable status name-value pairs.

Public Method –>add

Inserts a trace point into `traceData`. A trace point has a name, and an optional arbitrary array of name-value pairs of variables captured at the point of the trace point add. A start time (derived from using `_date_msec`) is automatically stamped.

Tag Parameters

- `(unnamed) label` = required : a string which identifies the trace point. Inside a custom tag or custom type this would be the name of the tag or member tag.
- `(unnamed) variable` = optional : use a parameter name equal to the name of the variable, and set the parameter value to the value of the variable. Add as many as desired in this format.

Examples:

A simple insertion to identify that a particular code location has been processed:

```
$fw_tagTrace->(add:'nameOfTracePoint');
```

Inside a tag, it is best to use the tag name as the trace point label. Inside a ctype, it is best to identify both the data type and the member tag:

```
$fw_tagTrace->(add:'fwp_user->onCreate');
```

To capture the states of variables at a given trace point, include parameters in the tag equivalent to the variable names like this:

```
$fw_tagTrace->(add:'searchInline',
    -theSearchMode = $theSearchMode,
    -dbSearchUser  = global:'dbSearchUser');
```

Use a debugging switch variable to turn the tage tracing on or off:

```
$fw_debug ? $fw_tagTrace->(add:'nameOfTracePoint');
```

Public Method –>tracepoint

Retrieves a trace point from `->traceData`. This provides a raw data structure of the requested trace point. It is possible that the same trace point is encountered multiple times in the application code, so each occurrence is stored. The outer data structure is an array. Each member of the outer array is an array of pairs. The first pair will always be labeled `'traceBeganAt'` and will have a `_date_msec` integer as the value. Pairs after that will be variable names and values. The variable names will have been prefixed with $$ for global vars, $ for page vars, or # for local vars. Variable scope is determined automatically by the `fwp_tagTrace` code internally.

The only tag parameter is the trace point label.

Normally, there is no need to use this tag as it returns the data and does not format the display in a manner that will be easy to read. However, there may be times when you want to programmatically look at a tracepoint and verify that the data is what you expected to find. This would be the case if you were building Unit Tests. To print out to screen the results of the trace in an easy to read format, use the `$fwp_showVars` ctype.

Examples:

To retrieve a trace point (perhaps for automated unit testing verification):

```
var:'checkState' = $fw_tagTrace->(tracePoint:'searchInline');
```

If the searchInline tracepoint were to have been processed twice, this would return something that looks like the following:

```
array:
    pair: searchInline = array:
                pair: 'traceBeganAt' = 10707701
                pair: '$theSearchMode' = 'books'
                pair: '$$dbSerachUser' = 'estoreSearch'
    pair: searchInline = array:
                pair: 'traceBeganAt' = 10709854
                pair: '$theSearchMode' = 'movies'
                pair: '$$dbSerachUser' = 'estoreSearch'
```

Public Method –>intervals

Calculates the time interval between each trace point, and returns an array of pairs with each trace point label and the interval between them. The first trace point will have an interval of 0, and each point after that will be an integer in milliseconds.

It is important to keep in mind that the interval is a measurement of how long it took to get from trace point A to trace point B. If your trace points are inside tags, these interval values are *not* an indicator of how long a tag itself took to process. (Use `$fwp_timer` for that.)

Examples:

To retrieve the intervals:

```
var:'intervals' = $fw_tagTrace->intervals;
```

This would return something that looks like the following:

```
array:
    pair: 'tracePointA' = 0
    pair: 'searchInline' = 123
    pair: 'tracePointC' = 12
    pair: 'searchInline' = 678
```

That data would indicate that it took 123 milliseconds to get from tracePointA to searchInline, and 12 milliseconds to get from the first searchInline to tracePointC.

The best way to view the data extracted from `->intervals` is to create a var like `$intervals` and display that using `$fwp_showVars`.

fwpUtil_timer

This tool provides an easy and clean method to have multiple “stop watches” inside your code. No doubt you’ve tried something like this:

```
var:'startSomething' = _date_msec;
    // do a bunch of stuff here
var:'somethingTimer' = _date_msec - $startSomething;
```

I've used those too, and they get downright ugly after you stack up a bunch of them. Way too many variables to keep track of. This custom type provides a cleaner approach by storing all data inside the ctype under a single variable name.

Many of the framework tags and custom type member tags have timer steps already coded in them so that various parts of the page assembly process are timed (display vs logic blocks, queries, etc.). In addition to the built-in timers, the developer can add timers in the application code to measure specific routines for optimization.

Interface Quick Glance

Create an fwp_timer object (this is already done in fwpPage_init):

```
var:'fw_timer' = (fwp_timer);
```

Read-Only Instance Variables:

```
'timeData'
```

Member Tags:

```
->start
->stop
->eventTimes
->eventTotal
->groupTotal
->durations
```

Public Read-Only Instance Vars

- `timeData` (complex) = contains individual events with millisecond start time and stop time values. An event has a label, a start time, and a stop time.

Public Method –>start

Inserts an event into `timeData`.

Tag Parameters

• `(unnamed) label` = required : a string which identifies the event being timed

Examples:

To create a new event and log the start time:

```
$fw_debug ? $fw_timer->(start:'queryForCartContents');
```

Public Method –>stop

Closes an event already started in `timeData`.

Tag Parameters

• `(unnamed) label` = required : a string which identifies the event being timed

Examples:

To close an event and log the stop time:

```
$fw_debug ? $fw_timer->(stop:'queryForCartContents');
```

Public Method –>eventTimes

Returns the raw data for the event label specified. If the event was processed multiple times, the data will be an array of multiple start and stop time pairs. For a single event, the outer data wrapper is still an array, and there will be just one pair.

Tag Parameters

• `(unnamed) label` = required : a string which identifies the event to retrieve.

Examples:

To acquire the event times for a named event:

```
$fw_timer->(eventTimes:'queryForCartContents');
```

This would return something that looks like the following:

```
array:
    pair: 13220884 = 13221484
    pair: 13221497 = 13221597
```

Public Method –>eventTotal

Returns a calculated total of all instances of the event label specified. Note that it is not meaningful to return a total of all events as events may be nested and therefore times would be duplicated. Use a specific event to time the page process as a whole.

Tag Parameters

• `(unnamed) label` = required : a string which identifies the event to retrieve.

Examples:

To acquire an integer sum of event durations for a named event:

```
$fw_timer->(eventTotal:'queryForCartContents');
```

Public Method –>groupTotal

Returns a calculated total of all instances of event labels which begin with the string specified. It can be handy to group together timers to know how much total time was spent processing SQL queries. To do that, you might label every sql query event start with sqlQry. Then, you can extract a group total for all these events.

Tag Parameters

• `(unnamed) label` = required : a string which identifies the events to retrieve

Examples:

To acquire the event durations for multiple named events which all begin with sqlQry:

```
$fw_timer->(groupTotal:'sqlQry');
```

If the event list include these events:

```
userAuthenticate
sqlQryGetCart
sqlQryUpdateCart
userStoreSession
```

the `->groupTotal` would return the sum of the events `sqlQryGetCart` and `sqlQryUpdateCart`.

Public Method –>durations

Returns an array of pairs with the event name and calculated time for each instance of the event label specified. If the event was processed multiple times, the data will be an array of integers. For a single event, the outer data wrapper is still an array, and there will be just one integer. If an event name is not provided, the times for all events are returned. If an event name is provided, times for only that event are returned.

Tag Parameters

- `(unnamed) label` = optional : a string which identifies the event to retrieve.

Examples:

To acquire the event durations for all events:

```
$fw_timer->durations;
```

To acquire the event durations for a named event:

```
$fw_timer->(durations:'queryForCartContents');
```

This would return something that looks like the following:

```
array:
    'queryForCartContents' = 605
    'queryForCartContents' = 611
```

fwpUtil_showVars

This tool provides an easy to read display of variables. It can output all variables in Lasso's [vars] and [globals] maps, or output only vars you specify. It can also be configured to exclude certain variables. This custom type would typically be used at the very end of a page for detailed debug output.

Each variable is displayed with its name, value, and data type. The display of complex variables such as arrays, maps, and others that may have nested data structures are recursively navigated to format each level until a simple data type is reached (null, boolean, string, integer, decimal, date, or duration).

Interface Quick Glance

Create an fwp_showVars object (this is already done in fwpPage_wrapup.lgc):

```
var:'fw_debugOutput' = (fwp_showVars:
    -clearVars={},
    -topvars={});
```

Read-Only Instance Variables:

There are no instance variables to use.

Parameters:

`-clearvars` = a comma separated list of variable names not to include in the displayed output.

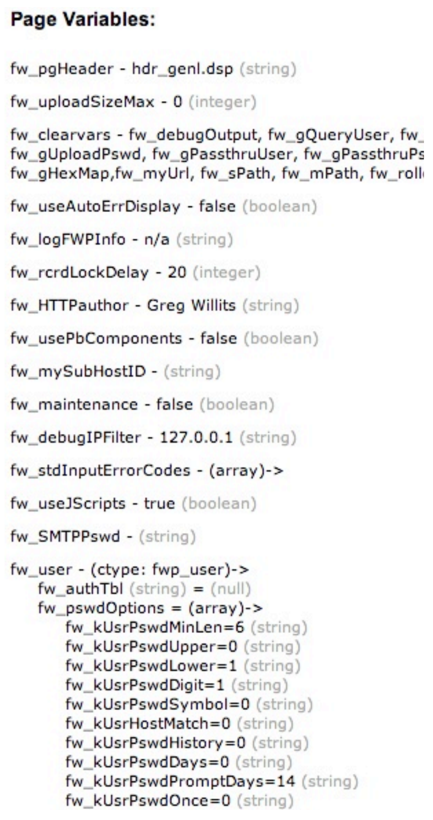
`-topvars` = a comma separated list of variable names to display in the Notable Vars section at the top of the display.

Member Tags:

```
->showAll
->showVar
```

Public Method –>showAll

Outputs a display of all page variables and all globals except for those listed in `-clearvars`. Three sections are generated. A section called Notable Vars so that vars of particular interest will be listed at the top, a section for Page Variables, and a section for Global Variables. Variables come directly from iterating Lasso's internal maps, so the display is not sorted.



```
Page Variables:

fw_pgHeader - hdr_genl.dsp (string)

fw_uploadSizeMax - 0 (integer)

fw_clearvars - fw_debugOutput, fw_gQueryUser, fw_
fw_gUploadPswd, fw_gPassthruUser, fw_gPassthruPs
fw_gHexMap,fw_myUrl, fw_sPath, fw_mPath, fw_roll

fw_useAutoErrDisplay - false (boolean)

fw_logFWPInfo - n/a (string)

fw_rcrdLockDelay - 20 (integer)

fw_HTTPauthor - Greg Willits (string)

fw_usePbComponents - false (boolean)

fw_mySubHostID - (string)

fw_maintenance - false (boolean)

fw_debugIPFilter - 127.0.0.1 (string)

fw_stdInputErrorCodes - (array)->

fw_useJScripts - true (boolean)

fw_SMTPPswd - (string)

fw_user - (ctype: fwp_user)->
    fw_authTbl (string) = (null)
    fw_pswdOptions = (array)->
        fw_kUsrPswdMinLen=6 (string)
        fw_kUsrPswdUpper=0 (string)
        fw_kUsrPswdLower=1 (string)
        fw_kUsrPswdDigit=1 (string)
        fw_kUsrPswdSymbol=0 (string)
        fw_kUsrHostMatch=0 (string)
        fw_kUsrPswdHistory=0 (string)
        fw_kUsrPswdDays=0 (string)
        fw_kUsrPswdPromptDays=14 (string)
        fw_kUsrPswdOnce=0 (string)
```

Tag Parameters

- none

Examples:

To create an output of all the vars, create an object and specify if you want certain vars at the top, and certain vars to be excluded (i.e. those that have passwords, or are just plain huge and you don't need them displayed):

```
// create the showVars object
$fw_debug ? var:'fw_debugOutput' = (fwp_showVars:
    -clearvars = 'dbConnection, gg_configFileCache',
    -topvars = 'eShopCart, userObject, sessionID');

// output the vars list
$fw_debug ? $fw_debugOutput->(showAll);
```

To create an output of only the vars you want to see:

```
// create the showVars object
$fw_debug ? var:'fw_debugOutput' = (fwp_showVars);

// output the vars list
$fw_debug ? $fw_debugOutput->(showVar:'eShopCart');
```

To create an output of timers:

```
// retrieve timer results
var:'timerResults' = $fw_timer->(durations);

// create the showVars object
$fw_debug ? var:'fw_debugOutput' = (fwp_showVars);

// output the vars list
$fw_debug ? $fw_debugOutput->(showVar:'timerResults');
```

Appendix A: Documentation for Deprecated Features

Appendix A: 5.0-style GUI Value Lists

(This documentation is from the original independent release of the fwpGUI tags. It probably needs a little updating, but will be pretty close. Double check with the online reference if needed).

Value List Tags Common Features

The four value list tags, fwpGui_Popup, fwpGui_CheckBox, fwpGui_RadioBtns, and fwpGui_ListBox, each create an HTML form input value list of the type its name suggests. Each tag has some options and behaviors unique to its list type, but for the most part, all share a majority of common features and programming requirements.

All tags are able to display any combination of default selection(s) which would be required by the application for an initial state, or to reflect an existing state of a variable, database field, or other data container. The tag will write all necessary HTML code to display and control the value list, and includes a CSS class parameter to allow display customizing.

Lists of values can be supplied to the tag as a literal value, any normal LDML substitution (variable, field, action_param, etc.), or retrieved from a specially formatted text file which contains several lists. There are several recognized list formats. Delimited string formats defined for version 1 of these tags are still valid, and there are new formats including custom delimiting and arrays which provide added capabilities.

General Syntax and Usage

The tags follow conventional LDML tag usage, and have several optional parameters. Below are some samples of what the tags might look like in use:

```
[fwpGui_popup:
    -Name='smpl1',
    -list='Red```Blue```Yellow```Green```Orange']

[fwpGui_popup:
    -Name='smpl2',
    -File=($c_filesPath + var:'myListsFile'),
    -Indx='departments',
    -Dflt=(field:'myDept'),
    -Var='companyDepts',
    -CSS='adminPopups',
    -AutoValidate]

[fwpGui_radiobtns:
    -Name='smpl3',
    -Dflt=(field:'someField'),
    -File='/site/files/cnfg_valueLists.lasso',
    -Indx=1,
    -Dflt='Blue',
    -Layout='horiz']
```

Tag Parameters

The following are parameters common to all value list tags. Each parameter has two names, a short name which is the preferred name for version 2 tags, and the original version 1 name shown in parenthesis which remains valid to allow for backwards compatibility:

Parameters Common to All Tags

- `-name`—string containing the value for the HTML name property of the field object (this also becomes the `action_param` name for Lasso).
- `-id`—string containing the value for the HTML id property of the field object. If not defined, it is set to the same value as `-name`.
- `-list`—a delimited string, array of delimited items, or an array of pairs containing the list of items to display in the value list field. The list can be constructed so that the displayed text and the selection value are the same or different. Therefore, a list of Small, Medium, Large can yield selected values of Small, Medium, Large or corresponding values such as sm, md, and lg.

  String delimiters include the version 1 defaults of ::: and +++ between display and value items for backwards compatibility. The version 2 delimiters are ``` and ___ which are more readable in text files with several lists. Either set yields the same performance. It is also possible to define custom delimiters (see the delimiter parameters later in this section).

  Arrays can define display items or display/value pairs. The latter can be delimited with ___ as a default or with a user specified string, or as an array of pairs.

  All of these options are automatically detected, so there is no need to specify which of the formats is being used (except for custom delimiters).

  Version 1 Delimiters:

  ```
  Small:::Medium:::Large
  Small+++sm:::Medium+++md:::Large+++lg
  ```

  Version 2 Delimiters:

  ````
  Small```Medium```Large
  Small___sm```Medium___md```Large___lg
  ````

  Custom Delimiters:

  ```
  Small--Medium--Large
  Small/sm--Medium/md--Large/lg
  ```

  Arrays:

  ```
  (array:'Small', 'Medium', 'Large')
  (array:'Small___sm', 'Medium___md', 'Large___'lg')
  (array:'Small'='sm', 'Medium'='md', 'Large'='lg')
  ```

- `-dflt`—a string equal to the initial menu item value that is selected when the menu is displayed. For checkbox and scroll list tags, multiple defaults can be passed as an array, as a string delimited by the `-subDelimiter`, or as a string delimited by \r as would be typical from a database field.
- `-file`—a valid Lasso file tag pathname to a file containing one or more delimited lists of items to display. There are two file formats. The version 1 file format is still supported which requires a numerical line number index into the file to specify the list to be used.

The version 2 file format is written as an LDML map, and requires the use of a map name to specify which list will be used. Refer to the section *Preparing Value List Text Files* for more detailed information about value list file formats.

- `-indx`—either an integer or string indicating which list within `-file` contains the exact list to display. An integer is required for version 1 type value list files, and a name is required for version 2 type files.
- `-var`—a string containing the name of a page variable into which the contents of the source list will be copied for the programmer's use elsewhere on the page. The variable will contain the list in the same format that was supplied to the tag. If the source was an array, the variable defined by Var will also be an array.
- `-delimiter`—a string identifying the delimiter to use between displayed items (example: `-delimiter ='**'` for One**Two**Three).
- `-subDelimiter`—a string identifying the delimiter to use between display and value items (example: `-subdelimiter ='--'` for One--1**Two--2**Three--3).
- `-lineDelimiter`—a string identifying the delimiter to use between lists in version 1 non-map text files. The default is \r.
- `-css`—a string equal to the name of the CSS class to be applied to the form field item. Each tag has its own assumed default value corresponding to the list type: fwppopup, fwpcheckbox, fwpradiobtns, fwplistbox. If CSS is not specified, the preceding class names will automatically be used.

Note that there are two possible methods to provide a series of items for the list: through `-list`, or through `-file` and `-indx`. One or the other is required, but not both. If both are specified then `source` will be used.

Parameters Specific to fwpGui_popup and fwpGui_listBox Tags

- `-blank`—"Y" or "N" to indicate whether the list should start with an empty option (a non-declaration = "Y").

Parameters Specific to fwpGui_listBox

- `-size`—an integer defining how many lines to display (a non-declaration = 4).

Parameters Specific to fwpGui_radioBtns and fwpGui_checkBox Tags

- `-layout`—"h|horiz|horizontal" or "v|vert|vertical" to indicate how the list should display (a non-declaration = vertical).

CSS Compatibility

The HTML code generated for each list includes a CSS class definition of: `fwppopup`, `fwpcheckbox`, `fwpradiobtns`, or `fwplistbox`. If `-css` is not specified, the preceding class names will automatically be used. If you include these values in your style sheets, that style will be applied to the output of the tag.

For a better user interface over the default HTML/browser behavior, the code for each radio button and check box includes an HTML label tag to create clickable labels, and a style parameter to keep the mouse pointer icon an arrow when over the label text.

File Tag Permissions

To use text files as sources of standard lists, Lasso must be configured with a user and group that allows the use of file tags and also access to the directory you will store those files in. The value list tags only require the ability to use `file_exists`, `file_readline`, and `include_raw`.

To enable the tags to access list files, the tags should be within an inline container that passes a valid username and password to Lasso. Multiple tags can be within one inline, so it is unnecessary to wrap each tag in its own inline. Instead, put an inline around the entire page area that contains the value list tags.

Examples

```
[fwpGUI_popup:
    -name='smpl',
    -list='Red```Blue```Yellow```Green```Orange']

[var:'colors'='Red```Blue```Yellow```Green```Orange']
[fwpGUI_checkbox:
    -name='smpl',
    -list=$colors]
```

These would be the simplest forms for any of the tags. For a popup, the five colors will be in the menu with a blank item at the top of the list. The blank item will be the initial selected option. For a checkbox list, the items will be displayed vertically, and none would be selected. In both cases, the values of the options will be the same as the displayed items in the list.

```
[fwpGui_popup:
    -name='smpl',
    -list='Small___S```Medium___M```Large___L',
    -dflt='M',
    -blank='N']
```

This example would display Small, Medium, Large as the available options, but the actual values of the selection would be either S, M, or L. This menu would have no empty option, thereby forcing a value to be selected. With `-dflt='M'` the initial selection displayed would be Medium. Without the `-Ddlt` option, the initial selection would be the first item in the list. Note that default selections are based on the value of the option, not the displayed text.

```
[fwpGui_popup:
    -name='smpl',
    -dflt=(field:'someField'),
    -file='/site/files/cnfg_valueLists.lasso',
    -indx=3]
```

The above sample, written in version 1 format, would select the third line of the specified file to display as options. The default value comes from a database field. Place this code in a database action inline (or retrieved named inline) to display values stored to a database.

```
[fwpGui_checkbox:
    -name='smpl',
    -list='Red```Blue___blu```Yellow___ylw',
    -dflt='Red___ylw',
    -layout='horiz']

[fwpGui_checkbox:
    -name='smpl',
    -list='Red```Blue___blu```Yellow___ylw',
    -dflt='Red\rylw',
    -layout='horiz']

[fwpGui_checkbox:
    -name='smpl',
    -list=(array:'Red'='Red','Blue'='blu','Yellow'='ylw'),
    -dflt=(array:'Red','ylw'),
    -layout='horiz']
```

These all yield the same results. The list will show the three colors in a horizontal layout. The items Red and Yellow will be checked.

```
[fwpGui_popup:
    -Name='smpl',
    -File=($c_filesPath + var:'myListsFile'),
    -Indx='departments',
    -Dflt=(field:'myDept'),
    -Var='companyDepts',
    -CSS='adminPopups']
```

Something like this would be typical usage to display the options for a field and pre-select a default value for an input form in an -update page. It also shows that the named value list read from a map type disk file will be duplicated into a page variable named companyDepts.

Preparing Value List Files

Value lists for all four of the tags can be supplied as literals, LDML substitutions, or references to a text file. There are two distinct styles of preparing text files. To allow backwards compatibility, the original plain text, one line per list file structure is still valid. However, to allow for more options and better performance, the new map file is the preferred method.

Regardless of style, multiple lists may exist in one file, and multiple files of lists are accommodated. I recommend using one value list text file for each database, or table, or set of pages, or some other unit that logically corresponds to the application. This way all the lists that pertain to a specific set of code files or data table can be easily transported to other solutions. Certainly, all value lists for an entire solution can be stored in one file if that seems an appropriate option. This is more effectively achieved with the new map file format.

Maps of Delimited Lists or Arrays for Named -Indx Reference

For new applications using version 2 tags, the new map file is likely the better method for storing lists. It requires a little more work to prepare than the original file format, but offers the significant advantage that the sequence and position of a list within the file is not fixed. Code transported among applications will not break due to the addition or removal of lists to the file. Lists are located by name, rather than by numerical line position in the file.

Additionally, the map file contents are cached. That is, when loaded, the map file is stored in a Lasso map variable. Subsequent accesses from the same web page to that same list file will recognize the map already exists, and will not re-load or re-parse the list file.

Using the map file structure, lists can still be defined as delimited strings, arrays, or arrays of pairs. Delimited strings are probably easier to read and write, but if the application code would prefer an array for use elsewhere in the page via the -var option, then an array is probably a better list structure choice. The goal in allowing multiple formats was to better accommodate the native data types an applications may already be using.

The map file format is written as a standard declaration of an LDML map variable with literal values. This file is loaded and processed by the value list tag. Below is an example of a map formatted list file written out in a way to make the lists as easy to read as possible.

The first list is a simple delimited string. The second is a simple array. In each of these cases, the list display and values will be the same for each list option. The third list is a string with two-levels of delimiting to make the value different from the display of each list option. The fourth list is an array of pairs to also have the option values differ from the display text. The fifth list shows that it is valid to have a mixture of subdelimited and non-subdelimited items.

```
[output_none]

//  Use output_none to cloak the contents of the file
//  The name of the var below must be identical to the name of the file
//  The file must be a .lasso extension
//  The name of this sample file would be ‘sampleLists.lasso'

[var:'sampleLists'=(map:

'colors'='Red```Blue```Yellow```Green',
'flavors'=(array: ‘Sweet','Bitter','Salty'),
'days'='Monday___Mon```Wednesday___Wed```Friday___Fri',
'cars'=(array: ‘Ferrari'='Modena 360','Porsche'='911 Turbo','Ford'='GT40'),
'units'='Inches___in```Pounds___lbs```Miles'

)][/output_none]
```

Delimited Strings for Numerical -Indx Reference

This file format is still allowed for backwards compatibility. The text file containing the value list definitions is prepared with each list on a separate line within the file. Each line is termined by a \r. Any name for this file is acceptable, and it can be located anywhere that your solution is able to reach. Note that the `-file` parameter requires a complete path name, not just the file name.

Each line in the text file follows this format (note the descriptions are based on the original delimiters, but it is actually possible to use any delimiters you wish so long as you also use the corresponding command options in the tags):

```
description:::item1Nm+++item1val:::item2Nm:::item3Nm
```

Where:

description—any text you want to describe the value list for easy reference. This string is not used by the value list code in any way. Any string before the first ::: is acceptable.

item1Nm—any text exactly as it is to appear on the web page for the first value item. Subsequent items are likewise included. They will appear in the list top to bottom or left to right in the left to right order that they appear in the list. Each item is separated with three colons. End with the last text item, do not include three colons at the end of the list. Any number of items is valid. Do not include a blank item for popups or listboxes, the tag does that for you as a parameter. Be sure to end the last line with a carriage return (encoded as a \r).

item1Val—(optional) any text exactly as it is to be used for the value of the selected menu item and subsequently stored in the database, or otherwise used in the application. As demonstrated above with item2Nm and item3Nm, this is optional. If the +++ delimiter is present, the tag will use the second parameter as the value for the value list option. If there is no +++ delimiter, the value is set to the same text as the option name. It is possible to mix and match this line by line.

Examples of value lists contained within a single file:

```
1   colors:::Red:::Blue:::Green:::Purple
2   yesno:::Yes+++Y:::No+++N
3   groups:::Engineering:::Sales:::Marketing:::Manufacturing:::Finance
4   our pricing options:::Annual=$220+++220.00:::Monthly=$21.95+++21.95
```

Each value list can be used for multiple fields. Therefore, several fields may use the same list of values, the actual field the value list is displayed for is defined as a tag parameter.

A value list file can contain lists for any combination of the value list custom tags. Line 1 can be for a popup list, while line 2 can be for a radio button list. In fact, different value list tags may even use the same list. For example, the colors list shown above could be used by any of the four tags. The yesno list could be used by either the popup or radio buttons tag.

One other note: a value list file can be any text file where the value list's line number can be identified. It is possible to include comments in the value list file so long as you can keep track of the line number of the needed list(s).