White Paper

# Corral Methodology for Publishing Dynamic Web Sites using LDML.

ver. 1

Peter D Bethke
March, 2001

# Table of Contents

## Introduction

The purpose of this White Paper is to introduce to the Lasso Community a methodology for structuring Lasso-driven web sites called (for now) “Corral”. The author has used this methodology for two years now to build many LDML-based applications and it has proven itself over time as an extremely useful, flexible and modular approach.

It is the intent of the author to offer this document to fellow Lasso coders for feedback, criticism, and collaboration. If these methods stand up over time, it is the author’s hope that they may be expanded on in an open-source collaborative effort to create certain standards and methods, and a common lexicon (set of shared terms).

Standards and Methods in a large part exist to provide a common framework, and lexicon, for reference. For example, the base elements of home construction are bricks, wood, cement, wires and the like. Although all homebuilders use the same raw materials, their methods can be very different. This is where Construction Standards come in to play – they allow contractors to all “be on the same page”, and “speak the same language”.

Similarly, lasso programmers all use the same language, LDML, but their methods of building sites might vary greatly, thus making collaboration more difficult, particularly in a distributed environment. Lasso is growing in popularity, and Blueworld is clearly committed to making it the best middleware choice for small and medium-sized dynamic web sites. It seems logical that we, the developers, should do what we can to bring our unique experiences together into a “critical mass” -- to push the language forward beyond just the “basics”, for the benefit of all. This happens all the time on the Lasso Talk List, and this document is offered in that spirit.

The Corral Methodology seeks to facilitate collaboration and communication by offering a structural model and lexicon for building lasso-powered web sites. The methods discussed in this paper are certainly not the only methods, or necessarily the best for each and every lasso project, but they do offer a ground-level set of beams, frames and cables, which can be expanded by any who wish to participate.

## Assumptions

The methods described in this document were developed using versions of the WDE from 2.5 to 3.6.6.2 running on the MacOS using WebStar (4.0 through 4.3) and WebTen (2.1 through 3.0). They have not been tested using a Lasso/NT environment, though the author can't see a reason why they would not work as well, given Lasso's cross-platform uniformity.

In addition, the Author makes assumptions that those reading the document are familiar with certain basic concepts in Lasso, such as Includes, Variables, Inline actions, Tokens, and File Tags. Familiarity with two of these topics, Includes and Variables, is crucial to understanding the methods presented here.

## Overview - General

The Corral Methodology uses a structured series of cascading “templates” and “includes” to build lasso format files, which are then served to the client via the Web server and the Lasso WDE (Web Database Engine). At the heart of the system is the “stub” file, a term borrowed from multimedia application development (and possibly other sources). “Stub” files in Macromedia Director terminology refer to small files that read initialization data (and other variables) and call other files to do the bulk of the work. Essentially operating as the literal “Director”, the “stub” file manages traffic and makes sure the right pieces are in place to handle the task at hand. Stub files also provide a handy place to insert global variables that influence overall application performance.

The stub file in the Corral Methodology serves the same purpose. It is a (usually) short “.lasso” format file that does not include any HTML code – rather, it declares a series of variables and processes a series of includes in a specific order to build pages in a “cascading” style, each “level” building on the previous. Starting with a “master template” and building its contents level by level (possibly incorporating “Section Templates” or even “Subsection Templates”) this structure provides tremendous flexibility over page contents and behavior when combined with a simple set of shared (essentially global) variables contained in the generic site configuration file.

Centralizing site configuration information makes it easy to port existing sites to different locations, and to create new sites from pre-defined template sets.

In addition, since the methodology is highly modular, it allows for effective component interchange and error trapping. Since most elements that build a page are called from the Stub file, “rebuilding” a page, or swapping out one element for another is as simple as changing a path reference in the stub file. In addition, if one “module” throws an error, it can be trapped, isolated and dealt with without interfering with overall site behavior.

Finally, the modular structure allows for distributed coding of application components when is more than one programmer working on the project. Given a set of global variables and a site outline, a collaborative team could easily add new components and swap out old ones without compromising site design or navigation.

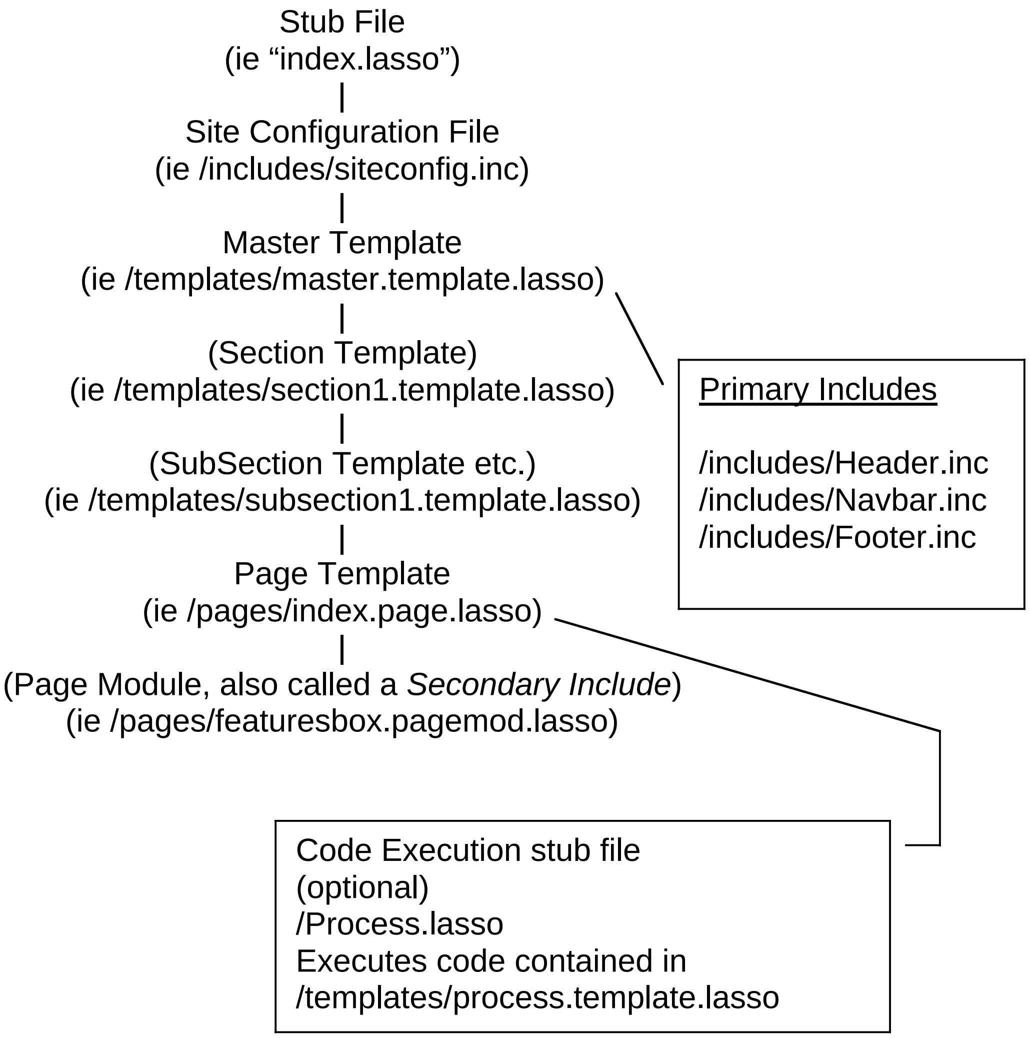
Developers use Lasso for different purposes. Some use the api as a database middleware, using search/result form pages that support an otherwise static “html”-based site. Others make use of its extensive api for non-database tasks, using file tags, string manipulation, and the like to build dynamic pages (similar to the use of php, and other popular scripting languages) that do not reference database content. Others still (including the author on occasion) combine the two methods with "log" and "file" tags to create "static/dynamic" content that is "published" to static html from dynamic lasso format pages. The Corral Methodology works with all of these systems, though its power is really apparent when applied site-wide using methods #2 and #3.

Specific discussion of the techniques for creating static publishing systems and complex error trapping routines is out of the scope of this document. What is presented here is the “skeleton” – the true version 1. It is the hope of the author that other developers might offer to contribute their ideas to the “mix” for these and any other topic that they are interested in, for only through collaboration will these concepts become broadly applicable.

So, enough of the broad picture – on to the “specifics”…

## Overview – Site Structure

The following diagram shows how a site built using the Corral Methodology might be built. Note: All vertical lines indicate files “included” based on references in the stub file. Files in parenthesis are optional.



Stub File
(ie “index.lasso”)
|
Site Configuration File
(ie /includes/siteconfig.inc)
|
Master Template
(ie /templates/master.template.lasso)
|
(Section Template)
(ie /templates/section1.template.lasso)
|
(SubSection Template etc.)
(ie /templates/subsection1.template.lasso)
|
Page Template
(ie /pages/index.page.lasso)
|
(Page Module, also called a *Secondary Include*)
(ie /pages/featuresbox.pagemod.lasso)

<u>Primary Includes</u>

/includes/Header.inc
/includes/Navbar.inc
/includes/Footer.inc

Code Execution stub file
(optional)
/Process.lasso
Executes code contained in
/templates/process.template.lasso

Final output to client: /index.lasso, incorporating fully “built” page.

## Site Structure – “File by file”.

### *The Stub File*

As described in the section “Overview – General”, the stub file (ie “index.lasso”). is a short format file whose job it is to declare the variables and trigger the actions that build the dynamic pages. Here are the contents of the stub file for the above example:

```
[include: '/includes/siteconfig.inc']

[lasso_comment]

        [var_set: 'page_title' = 'Home Page for My Site’, encodenone]
        [var_set: ‘section_template' = '/templates/section1.template.lasso',
        encodeURL]
        [var_set: 'page_template' = '/pages/index.page.lasso', encodeURL]
        [var_set: 'header’ = '/includes/header/mainheader.lasso', encodeURL]
        [var_set: 'footer' = '/includes/footer/mainfooter.lasso', encodeURL]
        [var_set: 'navbar’ = '/includes/nav/mainnav.lasso', encodeURL]

[/lasso_comment]

[include: '/templates/master.template.lasso']
```

The file begins by including the site configuration file (see below), which provides a set of useful and generic subroutines and any number of global variables that can be shared with site pages. The [lasso_comment] container is used to eliminate the “white space” generated by “invisible” commands like [var_set] and is optional, though it works well also to offset the variable declaration block.

The variables themselves are setting the path references that will be used later on the “cascading include” sequence to build the page component by component. The purpose of each declaration should be relatively clear. These variables don’t have to be declared in the body of the stub file – they can also be declared in the siteconfig.inc for global use, and overridden in the stub file for local exceptions. It makes for a very flexible system.

The heart of the system relies on the fact that variables declared in lasso format files are not scoped to the physical file itself – if that file is included in another file, its variables become part of the “parent” file, superceding previous declarations at the point of insertion. This lack of “page scoping” can be confusing at times, but proper use of it can be tremendously powerful.

The final line calls the first “template” file in the cascading sequence, usually a master template that has the most generic html formatting applied site-wide, but there are exceptions, such as the (optional) process.lasso code file – discussed later).

### *The Site Configuration File*

The site configuration file (ie “/includes/siteconfig.inc”). contains global variable information applicable to the entire site (which can be overridden at the stub file level or even lower in the hierarchy, such as page level or even page module level).

*Note: The suffix “.inc” is processed by default by lasso. The site configuration file should always have a lasso-processed suffix to prevent users from gaining access to clear-text information. Same for templates and the like.*

The site configuration file can include important variables (not required but strongly recommended) that can be used by format files. Here are 3 variables, declared in the site config file, that the author commonly uses site-wide:

```
[lasso_comment]

        [List_FromString:'URLList',response_filepath, ListDelimiter='/']

        [var_set: 'myfilename'=(List_GetItem:'URLList',ListIndex=(List_ItemCount:
        'URLList'))]

        [var_set: 'myfoldername'=(List_GetItem:'URLList',ListIndex=(Math_Sub:
        (List_ItemCount: 'URLList'), 1))]

        [var_set: 'pathtome'=(String_RemoveTrailing: Pattern=(String_GetField:
        FieldNumber=(String_CountFields: delimiter='/',(Response_FilePath)),
        delimiter='/', (Response_FilePath)), response_filepath)]

[/lasso_comment]
```

The names of the variables are fairly explanatory and lend themselves to almost English-like syntax when used in conditionals in the page body or any of the includes, such as the navigation bar.

```
        [if: (‘var: ‘myfilename’, encodenone) == ‘contactinfo.lasso’]
        <a href=”/morecontactinfo.lasso”>More Contact Info appropriate to this
        page</a>
```

```
	[/if]
```

Or, to apply this conditionally to all the pages contained in a specific directory, a developer might try this:

```
	[if: (‘var: ‘myfoldername’, encodenone) == ‘contacts’]
	<a href=”/morecontactinfo.lasso”>More Contact Info appropriate to all the
	pages in this folder (directory)</a>
	[/if]
```

Many different, useful variables that can be declared in the site config file. For example, if a developer had several pages that used lasso email inlines, this information could be shared from a central location, such as:

```
	[var_set: ‘webmasteremail’ = ‘webmaster@foo.com’]
	[var_set: ‘emailhost’ = ‘mail.foo.com’]
```

Or if that same developer had file tag routines in a page body that needed to grant (for example) read file permissions to a generic user, he could imbed a username and password in the siteconfig file that was granted read file permissions in the lasso configuration setup.

```
	[var_set: ‘readfile_uname’ = ‘randolph_mantooth’]
	[var_set: ‘readfile_pw’ = ‘emergency!’]
```

and call it later in the body of the page template:

```
[inline: -clientusername=(var: ‘readfile_uname’, encodenone), -
clientpassword=(var: ‘readfile_pw’, encodenone), -nothing]

	[file_read: ‘/somefiletoread.html’]

[/inline]
```

*(note in the above example this gets tricky if the “read” file is built using the Corral Methodology ie [file_read: ‘/somestubfile.lasso’], since the output of this action would be the unprocessed text of the stub file, with possible exposure of security information. It is also possible to use this technique to read the contents of the site config file itself. As always, file tags must be used with caution).*

Formatting information is another logical set of variables to put into the site config file. For example,

```
	[var_set: ‘backgroundcolor’ = ‘red]
```

is a variable that could be called by the master template (see below) in the <body> tag:

```
<body bgcolor=”[var: ‘backgroundcolor’, encodenone]”>
```

Or conditionally referenced like so in the site config file, below the declaration of the variable ‘myfoldername’:

```
[if: (var: ‘myfoldername’, encodenone) == ‘contacts’]
	[var_set: ‘backgroundcolor’ = ‘red]
[else: if: (var: ‘myfoldername’, encodenone) == ‘products’]
	[var_set: ‘backgroundcolor’ = ‘blue’]
[else]
	[var_set: ‘backgroundcolor’ = ‘yellow’]
[/if]
```

And called in the same manner in the master template.

### *The Master Template*

The master template does all the “work” in assembling, or at least initiating the process of assembling, all the various components referenced in the stub file.

Here is a base-level master template thrown together to create a classic “four-pane” table based site (4 rows, with header/logo, navigation bar, main page body, footer). This master template calls several variables that are declared in the site configuration file. Lasso variables are highlighted in blue. Includes are highlighted in red.

---

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"

      "http://www.w3.org/TR/1999/REC-html401-19991224/loose.dtd">
<html>

<head>
<title>[var: 'page_title', encodenone]</title>
<meta http-equiv="content-type" content="text/html; charset=iso-8859-1">
<meta name="keywords" content="[var: ‘site_keywords’, encodenone]">
</head>

<body bgcolor="[var: ‘backgroundcolor’, encodenone]" link="#666699"
vlink="#333366" alink="#666699" text="#000000" background="[var:
‘backgroundgraphic’, encodenone]">
```

```
<table width="100%" border="0" cellpadding="0" cellspacing="0">
        <tr>
                <td align="center">[include: (var: ‘header’, encodeURL)]</td>
        </tr>
        <tr>
                <td align="center" bgcolor="#CCCCCC">[include: (var: ‘navbar’,
                encodeURL)]</td>
        </tr>
        <tr>
                <td align="left">[include: (var: ‘page_template’, encodeURL)]</td>
        </tr>
        <tr>
                <td align="center" bgcolor="#CCCCCC">[include: (var: ‘footer’,
                encodeURL)]</td>
        </tr>
</table>
</body>
```

---

In this example you can start to see the “cascading” effect in action; the master template calls the page template, which itself can call page modules.

Alternatively, the developer could replace the page_template placeholder with a reference to a section_template, which itself would call the page template (or even a subsection template, etc etc,). This would be useful if one was developing a shopping cart for example, which used a set layout with common controls (view cart, check out, etc.).

(Obviously, this process can get a little carried away if over-done. As with all web developing, the key to success is the classic 5 p’s – Proper Planning Prevents Poor Performance – especially when developing a modular application).

Again, the key is to declare the paths of these files in the stub file; this gives a centralized control location for all the elements on a page, or if declared in the site config file, multiple pages. Swapping out components becomes as easy as changing a path reference in the stub file or the site config file – this process of separating code from format (html) can speed the process of developing immensely.

### *Primary Includes*

Although it is possible to hard-code common elements like headers, footers and navigation bars into the master template, often a developer may wish to modularize them and call them from the stub file via the master template.

The term “Primary Includes” comes from the fact that these elements are common to most layouts. (Secondary Includes - also called Page Modules - are elements that may be included on specific pages depending on context, like a “What’s Hot” feature box on the index page of a news site).

In the previous example of a master template, the Primary Includes are relatively straight-forward. The “header” include might simply look like this:

```
<table width=”100%”><tr><td><img src=”/images/logo.gif”><P>Welcome to The
Super Duper Site!</td></tr></table>
```

Which would yield this result in the final “rendered code” output:

```
<table width="100%" border="0" cellpadding="0" cellspacing="0">
        <tr>
                <td align="center">

                <table width=”100%” cellpadding=”5” cellspacing=”5”><tr><td><img
                src=”/images/logo.gif”><P>Welcome to the Super Duper
                Site!</td></tr></table>

                <td>
        </tr>
…….

</table>
```

Obviously, in this example the use of imbedded tables is a developer preference.

Say the developer wanted to keep a high level of flexibility in the code, because he anticipated changes in the future. The header include might look like this:

```
<table width=”100%”><tr><td><img src=”[String_Concatenate: (var: 'imagesbin’,
encodeURL), (var: 'sitelogo', encodeURL)]”><P>Welcome to [var: ‘Site_Name’,
encodenone]!</td></tr></table>
```

Thus, with a variable declaration in the site config file for the images bin (“/images/”) and site logo file (“logo.gif”) and Site Name (“Super Duper Site”), the developer can easily change all references to this information from a central location (or override it at the stub level).

Static advertising banners can also be called from the stub file, though a developer may want to include a lasso-based rotating system. Although these solutions exist, it is beyond the scope of this document to describe them. However, it should be apparent that such a system could be modularized to fit into the Corral Methodology. In fact, adapting such systems to be “Method-compliant” with a system like Corral is very useful.

Footers (which typically contain copyright information, disclaimers, and possibly navigation links) function in exactly the same way as headers.

Navigation Bars (a generic term; this could refer to any navigational construct) can really benefit from global variables like ‘myfilename’ and ‘myfoldername’ (discussed under the Site Configuration File section, above), providing location-relative results.

Expanding on that previous example, say a developer had a database of car models (ie “Ford F10 Pickup”) with a “category” value list that corresponded with specific car makers (ie “Ford, Chevy, Toyota”).

Placing all the format files within a directory called “/carsearch”, the developer creates a navigation bar that displays hyperlink for all the categories based on the values in the value list. He then places this code in the navbar include with a conditional that is location-sensitive.

```
[if: (‘var: ‘myfoldername’, encodenone) == ‘carsearch’]

        Search by Maker:
        <P>
        [inline: -database=’cars.fp3’, -layout=’bycategory’, show]

                [value_list: ‘maker’]

                <a href=”searchresult.lasso?-
                token.make=[valuelistitem]”>[valuelistitem]</a><br>

                [/value_list]

        [/inline]
[/if]
```

Again, for flexibility, the developer could declare information used in this search in the site config or stub file, thus yielding:

```
        [inline: -database=(var: ‘searchdb’, encodenone), -layout=(var:
‘searchlayout’, encodenone), show]
```

```
                [value_list: ‘maker’]

                <a href=”searchresult.lasso?-
                token.make=[valuelistitem]”>[valuelistitem]</a><br>

                [/value_list]

[/inline]
```

### *Page Templates*

Page templates contain the “meat” of the final “rendered” page that is delivered to the client.

In “classic” site building, the content that is typically subject to the most editing is the “page body”. In the Corral Methodology, this content is contained in page templates. Page templates can be stored in a common directory (for example, “/pages”), and edited directly without danger of effecting the overall site design. Page content can be quickly swapped by first creating a new page template, and then simply changing the path reference to the page template in the stub file at “publish” time.

Page templates also can contain significant lasso code, and are the most logical place to put lasso code such as inlines, searches, and other constructs.

For example, continuing the above example, the page template for the stub file “searchresult.lasso” might contain an inline search that looks something like this:

```
<table width=”100%”>
        [inline: -database=(var: ‘searchdb’, encodenone), -layout=(var:
        ‘searchlayout’, encodenone), ‘carmaker’=(token_value: ‘make’,
        encodenone), maxrecords=30, -search]
                [records]

                …. Search results formatting goes here

                [records]
        [/inline]
</table>
```

Note that the variables for searchdb and searchlayout are scoped to the result page as well and don’t need to be hard coded.

Another interesting variant on using page templates is putting both the search links and the result links on the same page template and to imbed a conditional in the code, keyed off of a token a token or form parameter passed in a url string, to trigger the resulting search.

Thus, suppose we had a search page with the stub file “searchcars.lasso”. In the stub file, we declare the page_template to be “/pages/searchcars.page.lasso”. The following code block imbedded in this single page template would service both the search and results page:

```
[if: (token_value: ‘make’, encodenone) == ‘’]

    Click a link below to find all models for a specific maker:

    [inline: -database=(var: ‘searchdb’, encodenone), -layout=(var:
‘searchlayout’, encodenone), show]

        [value_list: ‘maker’]

        <a href=”[var: ‘myfilename’, encodenone]?-
        token.make=[valuelistitem]”>[valuelistitem]</a><br>

        [/value_list]

    [/inline]

[else]
    <table width=”100%”>
        [inline: -database=(var: ‘searchdb’, encodenone), -layout=(var:
        ‘searchlayout’, encodenone), ‘carmaker’=(token_value: ‘make’,
        encodenone), maxrecords=30, -search]
            [records]

            …. Search results formatting goes here

            [records]
        [/inline]
    </table>
[/if]
```

Note how the variable “myfilename” is used in the search url to direct the result page back to itself without having to hard-code it. This makes for flexible code if the name of the stub file is changed, for example from “searchcars.lasso” to “serarchallcars.lasso”, the search hyperlinks would change automatically to <a href=”searchallcars.lasso?-token.make= …. (etc.)

### *Code Execution Stub File (optional)*

This concept is fairly advanced, and only applicable in certain situations, but it bears describing since the Corral Methodology is based on the use of includes. In a recent release of the WDE, the restrictions on the use of recursive includes were tightened (ie one file calls another as an include, which calls the first file in the same manner, etc.).

On the surface, this makes perfect sense, since one would not want to introduce a recursive include that would eat away the server's memory and eventually bomb the system. However, it also effects rather benign processes such as static file generation using the [file_write] command, if the page that is executing the [file_write] uses the same master template as the page that is being generated. Even though the "generated" page never technically appears in the body of the "generating" page, it is still treated as a recursive reference and will generate a lasso error.

Example:

The administration page '/admin/makepage.lasso', in a protected Web* realm uses the same master template as the file '/admin/filetobewritten.lasso". The admin wishes to use the nifty lasso file tags to generate a static, rendered .html file to the public "bin" for viewing (and caching by the server, which is one of the limitations of serving dynamic pages, they are not cached per se by the web server).

He also discovers that the combination of the [lasso_process] and [include] tags will duplicate the functionality of the [log] tag in regards to writing a static file, so he sets about writing the code.

But the command set:

```
[file_create: '/filetobewritten.html', fileoverwrite]
[file_write: '/filetobewritten.html', (lasso_process: (include:
'/admin/filetobewritten.lasso')), fileoverwrite]
```

…will error out, since the file makepage.lasso and the file filetobewritten.lasso use the same master template.

The solution is to use a "code execution stub". This is a stub file who is set up the exact same way as other stub files, but instead of calling the master template, it simply calls a special page template that contains necessary code and a [redirect_url] tag that passes it back to a results page. The execution stub file template does not contain any formatting, nor does it reference the master template in any way, so there is not a danger of throwing a recursive include error.

The code execution stub file, which the author calls “process.lasso”, would look something like this (assuming the resourceful developer has also declared the full url of the site in the site config file using javascript or hard-coded):

---

```
[include: '/includes/siteconfig.inc']

[lasso_comment]

        [var_set: 'page_template' = '/admin/templates/process.template.lasso’,
        encodeURL]
        [var_set: ‘formatfile’ = ‘/admin/filetobewritten.lasso’]
        [var_set: ‘staticfile’ = ‘/filetobewritten.html’]
        [var_set: 'responsefile' = (String_Concatenate: (var: ‘base_server_url’,
        encodeURL), (var: ‘pathtome’, encodeURL), ‘result.lasso’]

[/lasso_comment]

[include: (var: ‘page_template’, encodeURL)]

[redirect_url: (var: ‘responsefile’, encodeURL)]
```

---

And the process.template.lasso file might look like:

---

```
[lasso_comment]

        [file_create: (var: ‘staticfile’, encodenone), fileoverwrite]
        [file_write: (var: ‘staticfile’, encodenone), (lasso_process: (include: (var:
        ‘formatfile’, encodenone))), fileoverwrite]

[/lasso_comment]
```

---

Thus, executing sequentially, the stub file reads the site config file, executes the code contained in the code template, and then redirects to the result page (note that the redirect_url tag requires a full, not relative, path reference). The redirect string is built using variables declared in the site config file.

## Conclusion

Above all, the Corral Methodology is presented as a work-in-progress, much like the WC3 html specs.

Obviously, there are many advanced applications of the Corral Methodology that can be described, including:

1. Error Trapping
2. Advanced Static File Staging and Generation
3. Security Models
4. Style Sheets
5. Lasso Macros

Some of these areas have been developed by the author, some not. What is offered here is a roadmap, and set of suggestions for a standard methodology. It is the hope of the author that with the participation of other members of the lasso developer community, these ideas will be assessed, expanded on, and (most likely) subject to the highest levels of scrutiny.

This document is presented to the lasso community as a kind of open-source initiative; hopefully the ideas in it will spark serious debate effective lasso site modeling, modularization, portability, and the role of work-flow in the creation of large-scale Lasso applications.

## About the Author

Peter D Bethke (pdbethke@lassosmart.com) is an in independent developer of multimedia and internet applications. He has used Lasso and Filemaker Pro in conjunction with WebStar and WebTen for the past four years to develop many different applications, including an online news generator (used at Accelerate Your Mac! (www.xlr8yourmac.com), a shopping cart system with full back-end administration, a sports contest managing application with extensive static html generation capabilities, and the site albemarlekids.com, a resource for families and parents in the Charlottesville/Albemarle (Viginia) area. AlbemarleKids Online is a weekly news magazine for both print and the Internet, which uses Lasso as a backend administration and publishing system for both mediums.

All the systems described here were built using the Corral Methodology. Mr. Bethke is currently developing shareware versions of several of his products, including the news generator and sports contest manager.