Lasso Summit 2002

The Premier Lasso Event

October 14–15, 2002

Conference Manual

Featuring

Speaker Biographies

Featured Presentations

Conference Schedule

Sponsor Center

Meydenbauer Center
11100 NE 6th Street
Bellevue, Washington
USA

Lasso Summit 2002
Bellevue, Washington USA

Lasso Summit

2002 Developers' Conference

Conference Task Force

Kirk Bowman
MightyData, L.L.C.

Jim Van Heule
VH Publications, Inc.

Keynote Speaker

Bill Doerrfeld
Blue World Communications, Inc.

Guest Keynote Speaker

David Axmark
MySQL AB

Featured Conference Speakers

Kirk Bowman
MightyData L.L.C.

Duncan Cameron
Lasso Development Ltd.

Bil Corry
Mindio Corporation

Fletcher Sandbeck
Blue World Communications , Inc.

Greg Willits
Verteq, Inc.

Printed in the U.S.A.
$169.00 (US)

Lasso Summit 2002
Bellevue, Washington USA

Lasso Summit

Table of Contents

Lasso Summit

Table of Contents *(cont.)*

Lasso Summit

2002 Developers' Conference

Sponsor Center

Lasso Summit 2002 is presented in cooperation with the following sponsors.

Blue World Communications, Inc.

10900 NE 8th Street, Suite 1525
Bellevue, WA 98004
phone: 425.646.0288
fax: 425.646.0236
url: http://www.blueworld.com
email: blueworld@blueworld.com

Blue World's mission is to provide the best tools to build and serve data-driven Web sites. Blue World provides Lasso Web Data Engine, Lasso Studio and Blue World ListSearch service in fulfillment of its long-term mission to bring business to the Internet.

Point In Space

6 High Street
Great Barrington, MA 01230 U.S.A.
phone: 413.644.0289
url: http://www.pointinspace.com
email: info@pointinspace.com

Point In Space Web Hosting offers complete MySQL and FileMaker Database, Lasso 3 and Lasso Professional 6, Web, E-Mail, Listserv and FTP hosting solutions on powerful Macintosh G3 and G4 servers. Come see what sets us apart from other hosting providers - including our custom Database Manager software to allow you to remotely update your databases, in-depth logging reports for your site, as well as many hosting packages to serve your exact business needs.

Lasso Summit

Sponsor Center

digital.forest

19515 North Creek Parkway
Suite 208
Bothell, WA 98011 U.S.A.
phone: 425.483.0483
url: http://www.forest.net
email: sales@forest.net

Founded in 1994, digital.forest is the premier database- and application-hosting company serving small and medium-sized businesses and corporate workgroups around the globe.

edition.net

1220 North Street
Suite 606
Wilmington, DE 19801 U.S.A.
phone: 877.225.3821
url: http://www.edition.net
email: info@edition.net

edition.net, the Fine Arts Network, is an international innovator in Macintosh network and internet solutions providing inexpensive dedicated servers featuring FileMaker Pro and Lasso Professional 5, OSX Server including new Xserve configurations and QuickTime streaming platforms for the Arts, Humanities and Business.

Lasso Summit

Sponsor Center

Lasso Development Ltd.

United Kingdom
url: http://www.lassodevelopment.co.uk
email: djcameron@lassodevelopment.com

Lasso Development Ltd. is the company behind the popular Lasso community web sites LassoDevelopment.com, LassoScripts.com and LassoSites.com. Based out of the UK, they provide their international clients with the knowledge and skill required to build the most complex of Lasso driven applications, such as their Content Management System (LDCMS).

MightyData, L.L.C.

2025 Vista Crest Drive
Carrollton, TX 75007 U.S.A.
toll free: 800.287.0845
phone: 972.492.7523
fax: 972.394.9945
url: http://www.mightydata.com
general email: info@mightydata.com
training email: training@mightydata.com

MightyData offers information technology design, development and training services to a broad client base including Fortune 1000 companies, education institutions, small business, and non-profit groups. MightyData specializes in leading technologies including FileMaker Pro, Lasso, SQL database integration, AppleScript and Palm OS.

Lasso Summit

Sponsor Center

The Data Web Center

13424 Forest Park Drive
Grand Haven, MI 49417 U.S.A.
phone: 616.844.0066
fax: 616.844.6373
url: http://www.datawebcenter.com.com
email: sales@datawebcenter.com

The Data Web Center is the one-stop source to purchase products or services related to the Lasso industry. The bundle program provides incentives for 3rd party companies by bundling their product with the Lasso product line. The Data Web Training Center promotes any Lasso related training event. DWC Headline News provide the latest news in the Lasso industry.

Lasso Summit

Conference Schedule

Monday, October 14, 2002

8:00 AM - 9:00 AM	Registration
9:00 AM - 10:30 AM	Keynote Address by Bill Doerrfeld
10:30 AM - 11:00 AM	Morning Break
11:00 AM - 12:30 PM	MySQL by David Axmark
12:30 PM - 1:30 PM	Lunch
1:30 PM - 3:00 PM	Standardize and Automate by Greg Willits
3:00 PM - 3:30 PM	Afternoon Break
3:30 PM - 5:00 PM	LassoApps by Duncan Cameron
5:00 PM - 7:00 PM	Attendee Reception & Sponsor Showcase

Tuesday, October 15, 2002

9:00 AM - 10:30 AM	Advanced Developer Techniques by Fletcher Sandbeck
10:30 AM - 11:00 AM	Morning Break
11:00 AM - 12:30 PM	The Beginner's Guide to LP5 by Kirk Bowman
12:30 PM - 1:30 PM	Lunch
1:30 PM - 3:00 PM	Creating Custom LDML Tags: The Killer Feature You're Not Using! by Bil Corry
3:00 PM - 3:30 PM	Afternoon Break
3:30 PM - 5:00 PM	Winning the Project by Duncan Cameron

Related Events

Tuesday, October 15, 2002

6:30PM - 8:30 PM	digital.forest backyard barbeque

Wednesday, October 16, 2002

9:00 AM - 4:00 PM	LDML Pro Crash Course Training

Thursday, October 17, 2002

9:00 AM - 4:00 PM	LDML Pro Crash Course Training

Keynote Address

by Bill Doerrfeld

Blue World CEO Bill Doerrfeld will kickoff Lasso Summit 2002 with the keynote address on Monday morning. Bill will present the Lasso 6 product line and Blue World's vision for the future of Lasso products. Come hear what the father of Lasso has to say to the Lasso faithful.

This year's keynote presentation is expected to include a surprise announcement.

Biography

Bill Doerrfeld founded Blue World Communications in 1992 and has led the company to its current position as a leading vendor of tools to build and serve data-driven Web sites. Bill has been responsible for bringing the Lasso Web Data Engine and Lasso Studio products to market from concept to final release ever since the inception of the Lasso product line in 1996.

Notes

Notes

MySQL

by David Axmark

David Axmark, co-founder of MySQL AB, will discuss MySQL's commitment to the Lasso product line and how LassoMySQL is a perfect fit for dynamic database-driven Web sites.

Biography

David is one of the Founders of MySQL and has been working with MySQL since before it had a name. Before MySQL took over all time David worked as software consultant for over 15 years. The things he did included a state-of-the-art market research system (CommonLISP + CLOS+ MySQL's ISAM) and a advanced business graphics package (in 32k RAM). He has written many lines of code in 6502 and Z80 assembler, BASIC, C, CommonLisp, (Bourne)-Shell and Perl. His involvement with MySQL started with the idea to make a OpenSource SQL server to replace our internal terminal based dbtool. At MySQL David has worked with strategy, commercial aspects, installation, documentation and holding talks and tutorials. Hobbies include hiking and discgolf. David lives in Uppsala, Sweden with his family, plants and computers.

Notes

The MySQL AB Company

MySQL AB is the company that develops, supports and markets the MySQL database server globally. Our mission is to make superior data management available and affordable for all, and to contribute to building the mission-critical high-volume systems and products of tomorrow. We have made our product available at zero price under the GNU General Public License (GPL), and we also sell it under a commercial license to those who do not wish to be bound by the terms of the GPL.

The MySQL database server embodies an ingenious software architecture that maximizes speed and customisability. Extensive reuse of pieces of code within the software and an ambition to produce minimalistic but functionally rich features have resulted in a database management system unmatched in speed, compactness, stability and ease of deployment. The unique separation of the core server from the table handler makes it possible to run MySQL under strict transaction control or with ultrafast transactionless disk access, whichever is most appropriate for the situation.

Today MySQL is the most popular open source database server in the world with more than 4 million installations powering websites, datawarehouses, business applications, logging systems and more. Customers such as Yahoo! Finance, MP3.com, Motorola, NASA, Silicon Graphics, and Texas Instruments use the MySQL server in mission-critical applications.

The company was set up in Sweden by two Swedes and a Finn: David Axmark, Allan Larsson and Michael “Monty” Widenius who have worked together since the 80’s. MySQL AB is the sole owner of the MySQL server source code, the MySQL trademark and the mysql.com domain worldwide. The company is privately held and without debt, and it is financed by venture capital since July 2001.

MySQL AB employs some 55 staff around the globe, and thousands more contribute to the success of MySQL by testing the software, integrating it into other software products, and writing about it.

MySQL AB has three main sources of revenue

1. Online support and subscription services sold globally over the MySQL.com website to all users of the MySQL server.
2. Sales of commercial MySQL licenses to users and developers of software products and of products that contain software.
3. Franchise of MySQL products and services under the MySQL brand to value-added partners.

We provide our customers and partners with support services they can depend on, consulting

services, training programs, and more. In addition, to relevant advertisers we sell advertising space on our website that has more than 10,000,000 page views per month (as of January 2001).

We have described our core values as follows

We want the MySQL server to be

* The best and the most used database in the world
* Available and affordable for all
* Easy to use
* Continuously improved while remaining fast and safe
* Fun to use and improve
* Free from bugs

MySQL AB and the people of MySQL AB

* Subscribe to the Open Source philosophy
* Aim to be good citizens
* Prefer partners that share our values and mindset
* Answer email and give support
* Are a virtual company, networking with others

Record Sales, New Financing Fuel Growth at MySQL

22 Apr 2002

Dual Licensing is Winning Model for Open Source Leader

UPPSALA, Sweden - (April 22, 2002) - MySQL AB, developer of the world's most popular open source database, today announced that it is experiencing unprecedented growth, as first quarter sales of the MySQL database came in at 53 percent over projections. The company also disclosed that it has recently received an infusion of financing from its European investors, led by ABN AMRO Alfred Berg Industrifinans, a respected Scandinavian investment group and part of the ABN AMRO Bank Group.

MySQL AB is directing its growing resources to support product development and expanded worldwide commercial sales of its database. As evidence of the company's increased momentum, traffic at MySQL's web site (www.mysql.com) has grown 50 percent in the last six months. The company has also doubled in size in the past year, attracting employees from 14 countries across the globe, including a fully-staffed U.S.-based sales force.

"Our success is valid proof that open source software can be the foundation of a profitable, growing business," said Marten Mickos, MySQL AB CEO. "Our two groups of users - commercial and those that use MySQL under the free GPL license - fully support each other to create a solid business model that upholds the open source philosophy. We are thrilled about what our success means not only for MySQL, but for the open source community in general."

Dual Licensing Key to MySQL Business

The MySQL database maximizes speed and power in an extremely compact design, and it has full transactional capabilities to support business critical applications. From its inception, MySQL AB has passionately supported and promoted the open source philosophy that people should be able to freely read, redistribute, and modify the source code to promote rapid improvement and evolution of the software. MySQL now has over three million installations worldwide.

MySQL AB employs a dual-licensing structure that allows it to support its loyal user community with a sustainable business model. The MySQL database is available at no cost under the open source GPL license, and the company also provides commercial, non-GPL licenses to organizations that plan to re-sell a product containing MySQL code, or that otherwise don't want to be bound by the GPL. With the availability of the commercial license option, leading companies are now embedding the high-performance MySQL database into major software,

hardware and other electronic devices.

Blue World Communications, a company renowned for its easy-to-use and powerful software for building data-driven Web sites, has integrated the MySQL database server into its new Lasso Professional 5 product, a powerful Web application server designed for high-end Web developers. The MySQL database bundled with Lasso Professional 5 is called Lasso MySQL, and it will provide Lasso developers with a powerful SQL database server.

"Lasso customers have asked for tighter integration between Lasso and high-performance SQL databases," said Bill Doerrfeld, CEO of Blue World. "We chose the open source MySQL database primarily because of its great reputation, high-performance and ease-of-use - features that are hallmarks of the Lasso product line."

MySQL AB owns all rights to the MySQL server source code, the MySQL trademark and the mysql.com domain worldwide. The company's unique model for profitability and growth leverages its full ownership of the MySQL name and trademark, as well as its established user base and its deep expertise in the MySQL code. MySQL AB also provides its customers with a range of support, consulting and training services, as well as a MySQL partnership program.

For more information about a MySQL commercial license, please email sales@mysql.com or contact Larry Stefonic, MySQL vice president of sales, at U.S. (425) 743-5635.

About MySQL

MySQL AB develops, markets and supports the MySQL database server, the world 's most popular open source database. The MySQL database, which is free and available to everyone, has over three million users, including major corporations such as Yahoo!, Lucent Technologies, MP3.com (Vivendi Universal), Motorola, NASA, Silicon Graphics, HP, Xerox, Cisco and Agilent. It is available under the open-source GNU General Public License (GPL) or a non-GPL commercial license. MySQL is headquartered in Uppsala, Sweden, but is a true virtual company, with employees throughout the world. For more information about MySQL, please go to www.mysql.com.

# # #

Notes

Welcome.

Welcome to our corner of the world. We at digital.forest are thrilled to see the Lasso Summit come to our home, the Pacific Northwest. We've grown up here, since 1994, providing hosting and colocation services to the Lasso community worldwide. We would love to see all of you at our "Backyard BBQ" after the conclusion of the Lasso Summit Wednesday evening (rain! or shine), where you can tour our facility, and enjoy some social time, food & drink.

Please visit our website at www.forest.net.

digital.forest

Standardize and Automate: Let Your Code do the Work

by Greg Willits

We like to think rapid development is a Lasso strength. Would you like to escalate that to a strategic advantage? The LDML language allows for code with context awareness. This idea was popularized by The Corral Method, but significantly more can be done to decrease site development time. Using standardized file naming, building reusable libraries, and automating the cascading of code execution leads to faster development with less custom code. I'll compare FrameWork with FrameWork -Pro, including new features for the summit, to illustrate how to analyze your code and develop automated systems.

Biography

Greg Willits has spent too much time working at careers other than programming, and aims to fix that. Dabbling in programming since the Apple][and writing several vertical apps in various languages, he began working on Lasso driven websites in 1999. Greg has since developed several applications using Lasso. Buoyed by the genius of The Corral Method, his attention soon focused on developing sites more quickly. [FrameWork:] and [FrameWork: -Pro] represent entry and advanced level examples of much of what he has learned and developed in the way of building web sites on a foundation of standardized and automated Lasso code.

Notes

Standardize and Automate: Let Your Code do the Work

Introduction / Abstract

Proponents of Blue World Communication's Lasso Web Data Engine generally cite ease of use and rapid development as differentiating strengths of the LDML language. This paper discusses techniques that leverage LDML's capabilities into even greater rapid development performance using standardized and automated systems for much of the usual supporting code needed in every database driven web application.

The term "automated" here refers to standardized libraries which communicate with each other to automatically perform a series of individual functions without manual code patches to account for application-specific variances. To accomplish this requires the standardization of database structure, resource naming conventions, user interface components, and more. These standards and the resulting possible automation will be the focus of this paper.

Lasso Professional 5.0 and the integral Lasso MySQL database will be used for the LDML syntax and database needs. Additionally, for sample code, we'll be using [FrameWork: -Pro], an extensive application framework developed by the author. While some details the FrameWork -Pro framework will be discussed, the true purpose of this paper is to show some of the automation and standardization practices I have evolved and implemented in FrameWork -Pro, and to provide the reader with some ideas for automating his or her own code.

Dynamically Constructed Pages

Having web pages dynamically constructed as they are served can occur at many levels of flexibility and complexity, and for many purposes.

Varying degrees of complexity started with HTTP server support for server side includes, and have evolved to where 100% of a page's content is extracted from a database and assembled on the fly.

The purposes for dynamically constructing a page can be separated into two primary goals. First is to reduce the presence of redundant code and content which would reduces the efficiency of updating pages and sites. Second is to populate the page with user or environment context sensitive content.

Early methods of building dynamically constructed pages used server side includes. Portions of a page which were identical across all or numerous pages were simply segregated into separate files. A short command was written into the main HTML code to reference the segregated files. As the HTTP server loaded the page and came across the include file command, that file was loaded and inserted into the page.

This eliminated redundant information in program code that made early sites inefficient to update. With repeating information separated into a single file, changing that one file updated all pages that referenced that file. However, even SSI methods required the hard coding of file pathnames. Repetitive information could be peeled off into efficient single source locations, but there was still a number of limitations.

Today, with highly evolved middleware languages and integrated databases, the systems used to build dynamic pages can allow millions of site users to have an individually customized page. Additionally, what customized information to serve, and where it comes from can all be dynamically calculated. The programmer may not even know where the information will come from, and instead write the page construction code to adapt to a number of possible sources. Lasso Web Data Engine is such a middleware language that allows for all these possibilities.

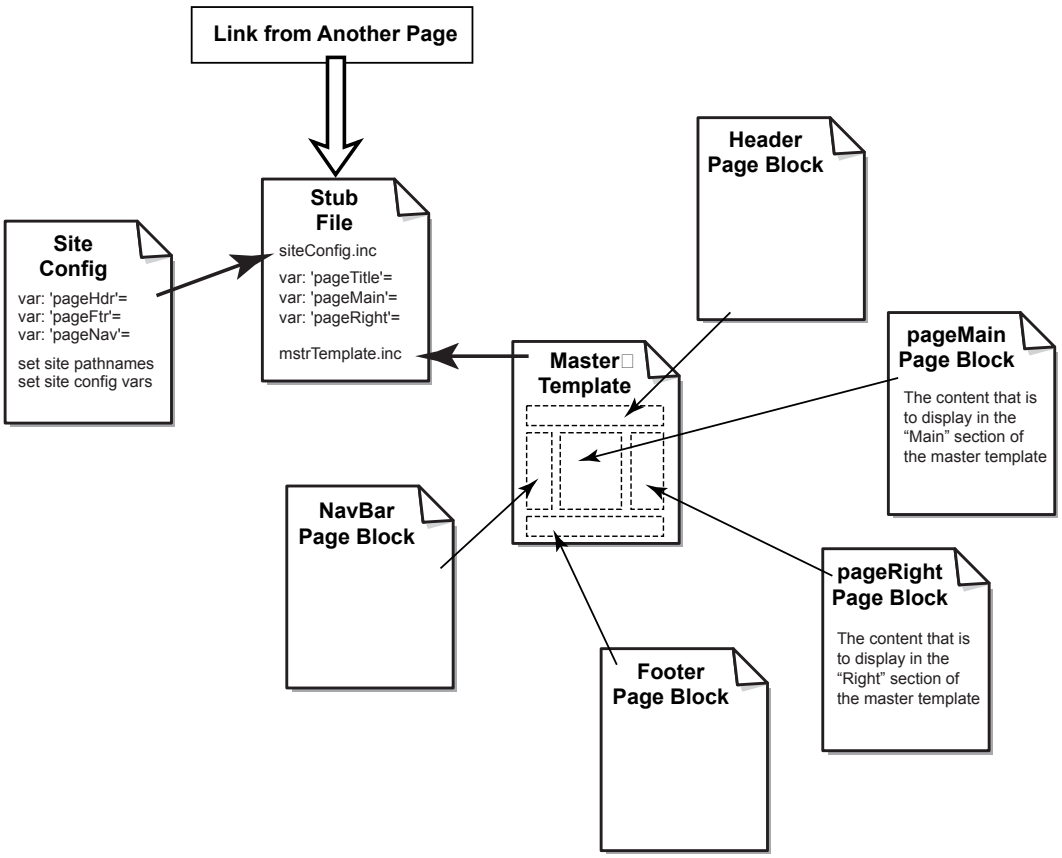
Lasso is a highly evolved, modern, middleware programming language for creating dynamically constructed web pages, web sites, and internet applications. Fundamentally, Lasso can pull content from files like server side includes and databases to insert into web pages as they are served. However, Lasso also allows complex logic systems for making decisions as to what content to serve based on any number of user or environment conditions. Not only can content be pulled from server side includes and databases, but so can program code. The code that builds the page can be as dynamic as the content.

In fact, analogous to "machines making machines," Lasso can be programmed to create Lasso code. Lasso can literally be used to solve problems where the problem isn't known until the page is served. At that point, the code itself that acquires content and constructs a page can be optimized on the fly.

Corral—Dynamic Page Building Blocks

While people using Lasso had been employing a variety of techniques to build pages dynamically for some time, Peter Bethke's March, 2001 white paper "Corral Methodology for Publishing Dynamic Web Sites using LDML" popularized some particular methods and terminology within the Lasso developer community.

As a starting point for looking at Lasso 5 code automation techniques, let's take a look at the fundamental elements of The Corral Method. Corral began life with the basic components of a stub file, siteConfig file, template file, and content file. The figure below illustrates the relationship and information flow between these components.



Let's start with the Master Template. This file contains the document description and generally defines the layout grid using place holders. By means of tables or CSS DIVs, various regions for a page's content are defined such as the header, footer, a left hand column, a center column, and a right column. Each of these elements is termed a page block. Each page block contains some LDML code which defines the location of the content file.

To fill those page blocks, we need some content files. Each page block has at least one file. Using the old server side include philosophy here, we break repetitive elements such as page headers and footers into separate files. Non-repeating page blocks such as for center and right blocks are also separated, but this is so the template can select a particular file from many possible choices.

Next, we need a way to tell the template exactly which page blocks to load. We need something to kick off the process which loads the template, which then loads the page blocks. This is the role of the stub file. Also, in a traditional static page environment, the name of the HTML file becomes the name of the page in the URL. The stub file is the file used in HTTP hyperlinks and is the file whose name shows up in the URL. The stub file is typically just a stubby little critter (hence the name?) with only a few lines of code that dictates which content files and which master template will be used to build the page.

Ultimately, the dynamic nature of the Corral Method comes down to the above three file types. However, there's one more important file that makes the Corral Method infinitely more useful. In any programming environment there are inevitably a number of information chunks that are needed to tell the whole program how to behave. Perhaps something like a default window background color, or what folder the certain files are located in. In web site programming, that means telling this information to every page. Rather than repeat the definitions in every stub file or even in every template, we can separate all that code into a single site configuration file, then simply have one line in every stub file that includes that information. This is Corral's siteConfig file, and is where environment variables are set, and where location and context specific decisions can be made. Anything code that you would find yourself repeating in preparing to build a page either gets written into or included by the siteConfig file.

Loading a Page

Basic Corral / FrameWork 1.0.2 Page Loading Method

With a basic Corral or FrameWork 1.0.2 site, page construction begins with loading the stub file. The stub file's first task is typically to include the siteConfig file. The Corral Method specifies that siteConfig calculate several useful context definition variables including the following (var names are changed to match FrameWork specs):

`c_myName` is the name of the current page (i.e. widgets_intro.lasso)
`c_myPath` is the path to the current file (i.e. /prod/widgets/)
`c_myFldr` is the immediate folder for the file (i.e. /widgets/)

We would also expect siteConfig to define some default folders where certain files stored, and default page header and footer files to be used.

```
[var:'c_hdrPath'='/site/msthd/']
[var:'c_ftrPath'='/site/msthd/']

[var:'c_myHdrFile'='hdr_genl.lasso']
[var:'c_myFtrFile'='ftr_genl.lasso']
```

After siteConfig is processed, Lasso returns to the remainder of the stub file where page specific content files are defined for the page blocks, and the master template is included. We would therefore expect all content files manually named by variables like these:

```
[var:'c_pgBlockLeft'='widgets_intro_left.lasso']
[var:'c_pgBlockMain'='widgets_intro_main.lasso']
[var:'c_pgBlockRight'='widgets_intro_right.lasso']
```

By combining the `c_my* and c_pg*` variables we can dynamically construct the location of each page block file. Where a traditional include for the left page block would look like this:

```
[include: '/prod/widgets/widgets_intro_left.lasso']
```

we can now use a simple string concatenation formula like this:

```
[include: $c_myPath + $fw_pgBlockLeft]
```

Such a formula, adapted to each page block, would be placed in the page block space holder of the master template where we wanted the corresponding files to be loaded. With this formula method, the code in the template remains fixed, and we can change the page block variables in each stub file to define each page's unique content.

While this system is a major step forward in reducing the coding effort, we can take it a step further to minimize the programmer's tasks.

SIDEBAR: In FrameWork 1.0.2 there are also section config files. These are siteConfig-like files dedicated to setting site section variables (perhaps the color of the masthead changes for each section of the site). In FrameWork: -Pro, section-specific variables are handled in some new ways. By default, the sectionConfig variables are incorporated into siteConfig with a folder name based conditional tree. Alternatively, separate files are still used, but in a different manner that is now integral to the new modular option.

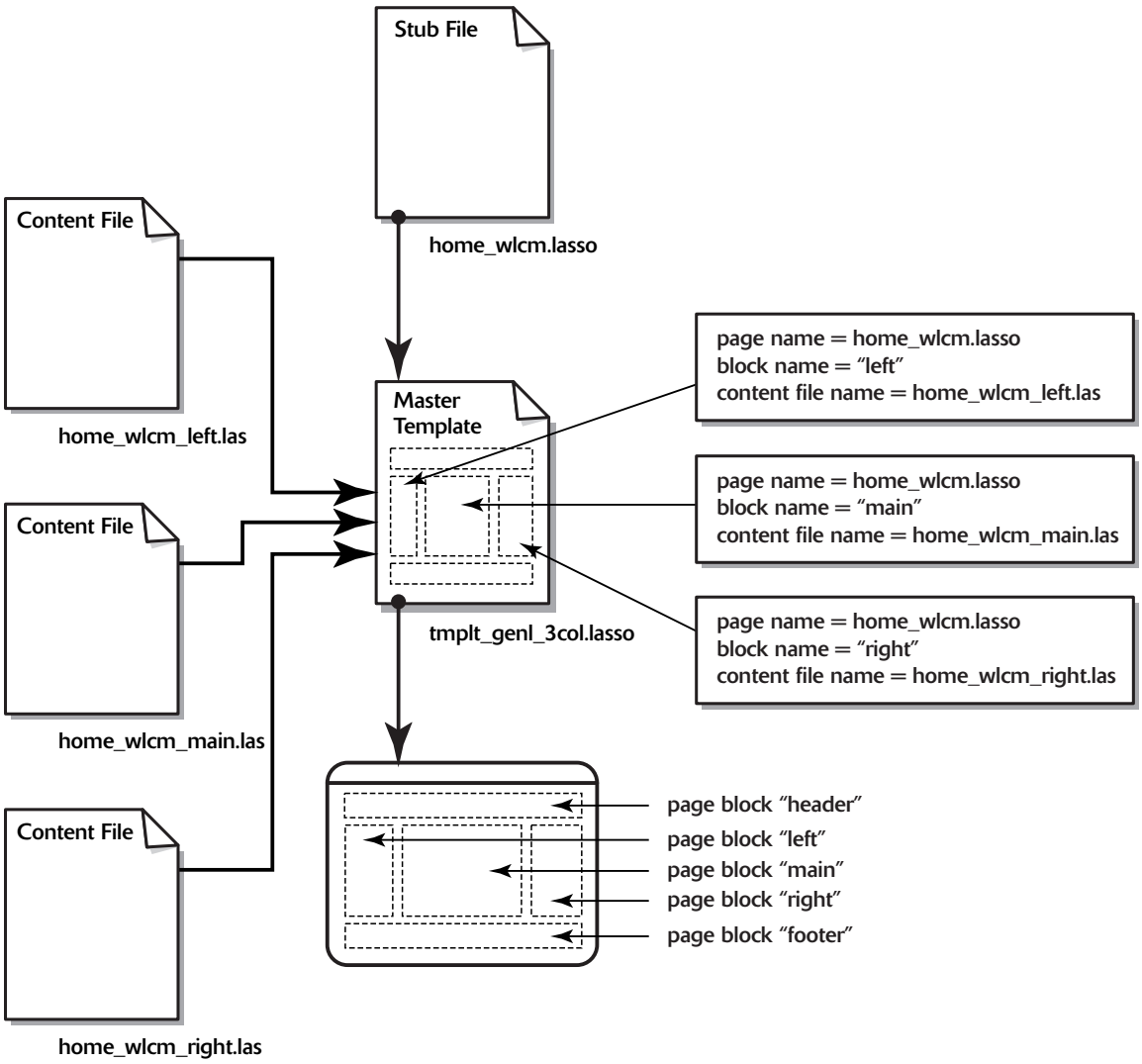
FrameWork -Pro Page Loading Method

With FrameWork -Pro, one of the goals was to eliminate all hard coding of paths, and to eliminate all manual declarations of page block filenames in the stub files. Leveraging the concept of `c_myPath and c_myName`, it stands to reason that if Lasso can determine what page is currently being loaded, and what the pathname to that file is, then there must be a method for Lasso to determine the rest of the page block content file names that need to be loaded.

To accomplish this, it was necessary to implement a structured system for the naming of all files within the site. That naming had to reflect the relationship of content sections (folders), page names (stub files), and page block names (content files). The figure below illustrates the file relationships and the resultant naming.

Before we look at how FrameWork -Pro automated the file loading process, let's look at the file name conventions employed. First, a FrameWork -Pro site's content is generally (though not strictly) organized in a flat structure where each folder is a section of common subject pages (i.e. products, support, catalog). That folder or section name becomes an integral part of all page names in that section.

File names follow a structural format of `section_pageName_blockName.lasso`. Stub files have an empty `blockName` parameter.



Therefore, a site might have the following folders, stub files, and content files:

```
/about/
    about_genl.lasso
    about_genl_left.lasso
    about_genl_main.lasso
    about_genl_right.lasso
    about_history.lasso
    about_history_left.lasso
    about_history_main.lasso
    about_history_right.lasso
```

```
/prod/
   prod_intro.lasso
   prod_intro_left.lasso
   prod_intro_main.lasso
   prod_intro_right.lasso
   /widgets/
      widgets_intro.lasso
      widgets_intro_left.lasso
      widgets_intro_main.lasso
/supp/
   supp_warranty.lasso
   supp_warranty_left.lasso
   supp_warranty_main.lasso
   supp_warranty_right.lasso
```

To reiterate our siteConfig variables:

`c_myName` is the name of the current page (i.e. widgets_intro.lasso)
`c_myPath` is the path to the current file (i.e. /prod/widgets/)
`c_myFldr` is the immediate folder for the file (i.e. /widgets/)

We've established that the folder path can be automatically represented with `c_myPath`. However, we can also automatically construct a file name for the page blocks by taking the page name, removing the `.lasso` extension, and adding the name of the page block such as `_left.lasso` which will be consistent for all files loaded into this block. Now, we can write the following into the master template:

```
[var: 'fw_pgBlock'=(string_replace:
   find='.lasso',
   replace='_left.lasso',
   (var:'c_myName'))]
[include: $c_myPath + $fw_pgBlock]
```

This would be adapted and repeated within each template page block placeholder (except the masthead items like headers and footers). With this approach, there is no need to manually define page names or block names in the stub file. The master template takes care of automatically calculating the file name for the content block.

PageLoader

One of the major goals of FrameWork -Pro's architecture is to minimize redundant code. Lasso's [include] and [library] tags make it easy and efficient to separate even small pieces of code which are repeated frequently throughout a site.

If we continue to consider the master template file, the include statement to load a page block is likely not the only code needed at that point. There is likely to be some support code surrounding that include statement. For example, in Lasso 3, if an include statement defined a file that did not exist, no error was generated, and Lasso processing continued. However, in Lasso 5 this is no longer the case. We now have to explicitly handle the possibility that there may not be a content file for a specific page block.

This can be done with with either a [file_exits] tag or [protect] tag. Within a Lasso 5 environment, the [protect] tag is probably the better choice. However, the [file_exists] tag can be used in a Lasso 3 environment. If you're still writing in Lasso 3, use [file_exists], and the code will transfer to Lasso 5.

So, our page block loading code might expand to something like this:

```
[if: (file_exists: $c_myPath + $fw_pgBlock)]
   [var: 'fw_pgBlock'=(string_replace:
      find='.lasso',
      replace='_left.lasso',
      (var:'c_myName'))]
   [include: $c_myPath + $fw_pgBlock]
[else]
   ..... run your error management routine .....
[/if]
```

Furthermore, in FrameWork there's the option to require user authentication at the block level. With that in place, the code expands yet again to this:

```
[if: ($fw_usrAppvd)=='Y']
   [if: (file_exists: $c_myPath + $fw_pgBlock)]
      [var: 'fw_pgBlock'=(string_replace:
         find='.lasso',
         replace='_left.lasso',
         (var:'c_myName'))]
      [include: $c_myPath + $fw_pgBlock]
   [else]
      ..... run error management routine .....
   [/if]
[else]
   ..... run error management .....
[/if]
```

Even without block level authentication, maybe a site has page block level user preferences for content or style. Regardless of the options, it is not uncommon for there be a variety of extra layers surrounding the basic page block include.

With each template having two, three, or more content blocks per page, we can see that even within one template, we're generating a lot of redundant code. The only difference between each one is the name of the pageBlock itself. Multiply that by several templates, and perhaps several user selectable styles or options, and there's a lot of code that's being repeated. If the programmer needs to adapt this code to another site with slightly different needs, or update this code with new features, there's a lot of repetitive maintenance work to do.

FrameWork's answer to this is two files: one called the blockManager and another called the blockLoader. Now that we're going to put all the redundant code into a separate include file, we may as well even put the routine to calculate the page block file name in that include. However, we still need to know which page block is being processed, so the code in each page block in the master template now looks something like this:

```
[var:'fw_blokName'='main']
[include: ($c_gLibsPath + 'fw_glbl_blockMngr.inc')]
```

We've reduced the page block code to the lowest possible level without redundancy. Anything that needs to change in how page blocks are processed can be written in the blockMngr include. The var declaration establishes the name of the block, and the include loads the blockManager code.

If the template has a lot of page blocks it may be worth loading the blockManager code into RAM with an include_raw. At the top of the template we'd do that with this line:

```
[var:'fw_blockLoaderCode'=
   (include_raw: ($c_gLibsPath + 'fw_glbl_blockMngr.inc'))]
```

then each page block code would look like this:

```
[var:'fw_blokName'='main']
[lasso_process:(var:'fw_blockLoaderCode')]
```

The blockManager contains all the extra layers that are typical to every page block. This includes calculating the pageblock name, which needs to be done first off. With a formula very similar to what was introduced previously, we just change it so that the block name is also a variable and end up with this:

```
#first let's calculate the name of the block we're looking for

[var: 'fw_pgBlock'=
   $c_myPath +
   (string_replace:
      $c_myName,
      -find='.lasso',
      -replace=('_' + $fw_blokName + '.lasso'))]
```

The blockManager continues with whatever layers are part of the site's features, which in this case is block level user authentication.

```
# if the block authentication shell was specified, then
# the IF clause verifies user authentication using the
# privilege defined in the stub file before loading the page block.
# Two variables are use to establish authentication. A universal
# usrAppvd variable that simply states the user is a real user,
# and a second more specific privilege. Because we don't know what
# privilege is needed for any given block, the privilege is a var
# If authentication fails (either to do timeout or unrecognized),
# then trigger FrameWork's error management system.
# The ELSE clause loads the page block without authentication

[if: (var:('fw_authBlock' + $fw_blokName))=='Y']
   [if:
      (var:(usrAppvd) == 'Y') &&
      ((var: (var:'fw_usrPrvlg')) == 'Y')]

      [include: ($c_gLibsPath + 'fw_app_blockLoader.inc')]
   [else]
      ..... error management stuff .....
   [/if]
```

```
[else]
   [include: ($c_gLibsPath + 'fw_app_blockLoader.inc')]
[/if]
[var:'fw_blokName'='']
```

You'll see that even here we don't yet load the actual content file. You'll see instead that there are two places where a blockLoader file is included. Now, the code in blockLoader isn't particularly complex, but it could be depending on the features of the site. In fact, it is expected to get more complex as more features are built in to the standard FrameWork code base. It could be argued that the code in the blockLoader should just be repeated twice in the blockManager. However, taking the redundant code theme to it fullest extent, the code has been separated into a separate file.

For most sites, the include for the blockLoader would simply be the include for the content file, and that would be it. However, FrameWork -Pro includes automated systems called actionShells to execute database actions which can be triggered at this point, and it is that code which needs to be inserted before we simply include the content file. The actionShells handle record locking, input data validation, database actions, error handling, and automated response file construction. These functions are layers, or shells, around the basic content file which would typically be the user input forms. Whether these actions are needed or not is declared in the stub file. If these actions are not needed, we simply include the content file. If the database actions are needed, then we have to include some other files instead (which in turn eventually include the content file). We'll dig into the details of those database action systems later, but for now let's take a peek at the blockLoader.

```
# if the block is the "left" block then do this:

[if: $fw_blokName == 'left']
   [protect]
      [include: (var:'fw_pgBlock', -encodeURL)]
   [/protect]

# if the block is the "main" block then do this:

[else: $fw_blokName == 'main']

   # if there's no database action, just include the content file

   [if: ((var:'fw_dbActn') == '')]
      [protect]
         [include: (var:'fw_pgBlock', -encodeURL)]
      [/protect]
```

```
        # otherwise  start by including the appropriate action shell

        [else: ((var:'fw_dbActn') == 'search')]
           [include: ((var:'c_gLibsPath') + 'fw_actn_search.inc')]
        [else: ((var:'fw_dbActn') == 'update')]
           [include: ((var:'c_gLibsPath') + 'fw_actn_updt.inc')]
        [else: ((var:'fw_dbActn') == 'updateOK')]
           [include: ((var:'c_gLibsPath') + 'fw_actn_updtOK.inc')]
        [else: ((var:'fw_dbActn') == 'delete')]
           [include: ((var:'c_gLibsPath') + 'fw_actn_dlt.inc')]
        [else: ((var:'fw_dbActn') == 'deleteOK')]
           [include: ((var:'c_gLibsPath') + 'fw_actn_dltOK.inc')]
        [else: ((var:'fw_dbActn') == 'add')]
           [include: ((var:'c_gLibsPath') + 'fw_actn_add.inc')]
        [else: ((var:'fw_dbActn') == 'addOK')]
           [include: ((var:'c_gLibsPath') + 'fw_actn_addOK.inc')]
        [else: ((var:'fw_dbActn') == 'view')]
           [include: ((var:'c_gLibsPath') + 'fw_actn_view.inc')]
        [else]
           <p>Database action defined by 'fw_dbActn' is invalid.</p>
        [/if]

    # if the block is the “right” block then do this:

    [else: $fw_blokName == 'right']
        [protect]
           [include: (var:'fw_pgBlock', -encodeURL)]
        [/protect]
    [/if]
```

As you can see, the above code could arguably just be integral to the blockManager. In the author’s opinion it is lengthy enough, and being subject to modification when new features are added, the blockLoader code warrants being is a separate file rather than maintaining it twice in the blockManager include. As shown above, the action is assumed to only be in the “main” content block, which is primarily aimed at database administration pages in a site’s Admin section. Allowing actionShells to be used in other page blocks allows database actions for purposes such as sidebar login forms, search forms, surveys, and other such database interactions. (If the actionShells needed to be available to multiple page blocks, guess what? I’d pull that if conditional tree into a separate file to eliminate repeating it redundantly.)

To summarize this section, the blockManager / blockLoader combination automates the peripheral functions that might need to surround a content file. More importantly these routines took a code chunk that was largely identical except for the page block name, and rewrote the code slightly to generalize it. This allowed to the code be separated and treated as an independent subroutine.

By taking the approach of recognizing redundant code, and restructuring the details to use variables instead of hard coded specifics, we’ve eliminated the task of repetitively writing that code each time that we alter or create a new template.

Authentication Manager

FrameWork 1.0.2 introduced a system of authenticating users based on an application specific database that enabled privileges beyond the core search, add, update, delete privileges maintained by Lasso's own user administration system.

FrameWork -Pro utilizes a virtually identical approach except that it has been streamlined to a level of greater automation. In FrameWork 1.0.2 each FileMaker database had a dedicated include to establish the user's profile and privileges for that database. In FrameWork -Pro, a universal routine has been created eliminating the need to write multiple profiling includes for each web site.

> SIDEBAR: FrameWork -Pro uses a session based authentication process now. Upon login, the user is searched for. If found, a sessionID and time stamp are stored in the user's record. From that point on, the sessionID is forwarded page to page. Each page is reauthenticated by looking for the sessionID as seen in the WHERE clause above. Then the current time is compared to the session time stamp to determine whether the session is fresh.

One of the naming conventions for FrameWork -Pro is the concept of a web site "section." A section can be many things, but we'll generalize it to be any number of files used to generate pages of a related subject.

Each section has a code. That code is the name of the top level folder that contains the primary content files for that section (or the stub files for the content pages if you're one to separate the content files from the stub files). In FrameWork -Pro that code is used as the prefix to every file in that section, and is used in the site users database as a prefix to the privileges for each table. For example, on the ldml.gregwillits.ws site, there is an articles section named "artcls." Every page in that section begins with `artcls_` as a prefix. In the table that contains user privileges, the privilege fields are named `artclsView`, `artclsAdd`, `artclsUpdate`, `artclsApprove`, etc.

By reading either the top folder of the path to a file or the prefix to any file (available by splitting the filename at the underscores), the section code can be acquired and used to automate references to any number of folders, files, or variables associated with that section. This technique was used in FrameWork -Pro to generalize the user profile routine.

The user authentication routine in FrameWork is probably like most others. The visitor supplied name and password are searched for in a database. If they're found and validated, then the user is marked as approved. Many systems might end there with a single variable that states the user's status. In FrameWork, several variables are set according to the user's privileges profile. Usually, FrameWork only loads the privileges germane to the table being accessed rather the user's entire profile, but both methods may be appropriate.

In FrameWork 1.0.2 a block of variable assignments were named and made specific to the database table that privileges were being assigned for. Such a block looked something like this:

```
[var:
    'usrAppvd'='Yes',
    'usrRcrd'=(field:'rcrdNo'),
    'usrAcct'=(field:'userAccount'),
    'usrName'=((field:'userFirstNm') + ' ' + (field:'userLastNm')),
    'usrEmail'=(field:'userEmail'),
    'usrGroup'=(field:'rcrdGroup'),
    'suprUsr'=(field:'superUser'),

    'artclsVw'=(field:'artclsView'),
    'artclsAdd'=(field:'artclsAdd'),
    'artclsUpdt'=(field:'artclsUpdate'),
    'artclsDlt'=(field:'artclsDelete'),
    'artclsAppv'=(field:'artclsApprove'),
    'artclsGrp'=(field:'artclsGroup')]
```

With this code, it is necessary to repeat the whole routine for each table that privileges need to be acquired for. Additionally, because the variables are also unique to the table, it is necessary to edit any pages or duplicate any includes that used these privilege variables. This means lots of manual and redundant coding.

In FrameWork -Pro, this block was updated to take advantage of the section code, and now looks like this:

```
[var:
    'usrAppvd'='Y',
    'usrRcrd'=(field:'rcrdNo'),
    'usrAcct'=(field:'userAccount'),
    'usrName'=((field:'userFirstNm') + ' ' + (field:'userLastNm')),
    'usrEmail'=(field:'userEmail'),
    'usrGroup'=(field:'rcrdGroup'),
    'suprUsr'=(field:'superUser'),

    'usrVw'=(field:$fw_sctnCode + 'View'),
    'usrAdd'=(field:$fw_sctnCode + 'Add'),
    'usrUpdt'=(field:$fw_sctnCode + 'Update'),
    'usrDlt'=(field:$fw_sctnCode + 'Delete'),
    'usrAppv'=(field:$fw_sctnCode + 'Approve'),
    'usrGrp'=(field:$fw_sctnCode + 'Group')]
```

Two things have been changed. First, the section code was used to build the name of the table field on the fly. This generalized the routine so it can be used for any section. Second, the privilege variables are no longer named specific to the table. Upon looking deeper at that aspect, it was simply not necessary for the majority of uses. Certainly there may be occasions where separate variables might be necessary, in which case an alternate authentication routine could be used.

Using the section code to build the field names also comes in useful for building the list of variables to be returned by the SQL statement in the search for the user's name and password:

```
-sql=('SELECT
    rcrdNo, rcrdAppvd, rcrdGroup,
    superUser, userFirstNm, userLastNm,
    userAccount, userEmail, userPass, ' +
    $fw_sctnCode + 'View, ' +
    $fw_sctnCode + 'Add, ' +
    $fw_sctnCode + 'Update, ' +
    $fw_sctnCode + 'Delete, ' +
    $fw_sctnCode + 'Approve, ' +
    $fw_sctnCode + 'Group, ' +
    'sessionNo, sessionTime ' +
    ' FROM ' + $fw_usrsDatabase + '.' + $fw_usrsTable +
    ' WHERE sessionNo' + '="' + $fw_s + '"')]
```

Thus, the entire getProfile routine has been updated to a universal include saving coding time and eliminating another group of largely redundant files.

Database Actions

FrameWork -Pro includes an intricate system of standardized routines that completely automate database actions and common peripheral tasks such as record locking, input data validation, database actions, error handling, and automated response file construction.

The focus of this discussion won't attempt to explain every detail of these files (that's a job for the FrameWork -Pro documentation), but rather along with basic explanations of the logic flow, the emphasis will be on the techniques used to make the routines automated and generalized for any database and table.

With the availability of actionShells in FrameWork -Pro, the programmer is relieved of having to write vast amounts of code to support user interaction with the database. In FrameWork -Pro, generating database administrative functionality requires little more than writing the code for the forms involved. In addition to the forms, the programmer writes a simple definition file of each database table, prepares about a half dozen variables in the stub files for the form and response page, and writes out rules in a standardized format for field validation (input error messaging is handled automatically).

There are several sets of files to manage several unique actions: add form, add response, update form, update response, delete form, delete response, and searching to generate a list of found records. In that the files are used to wrap the core action with a spectrum of standard functionality, the main files involved are called actionShells.

Each action has a slightly different logic flow, but as an example, below is the sequence of tasks performed by the update form (update) and update response (updateOK) actions.

```
Update Action Shell: locks the record to be edited, retrieves record data to display
in the update form.

   Verify a recordID value exists
      Search for the current record
      Test for errors caused by the search
         Is record currently locked? No? Continue
            Generate a lockID
            Write lockID to record to prevent another
            user from making changes to the record while
            this user prepares edits
            Test for errors caused by writing the lockID
               Search for record again and retrieve all fields
               Test for errors
                  Include the update form pageBlock
               Report Errors
            Report Errors
         Report record is locked
      Report Errors
   Report No Record Number Error

Update Response Action Shell: validates the input data, validates the lock session,
builds and executes an SQL UPDATE statement specific to the form inputs, builds
response page HTML.

   Convert action_params to vars (done in siteConfig)
   Validate input values
   Test for errors reported by validation include
      Verify a recordID exists
         Validate that the record lock is valid
            Read table definition file
            Read array of form input params
            Create corresponding array of field names
            Build SQL UPDATE statement from arrays
            Execute SQL statement
            Test for update action errors
               Search for record to acquire field data for display
               Test for errors
                  Construct standard response page HTML
                  which confirms success and offers
                  continuation options (or include a
                  custom response if specified)
               Report Errors
            Report Errors
         Report lock expiration or invalid session
      Report no record ID error
   Display Input Error Messages
```

As can be seen, the actionShells perform a significant number of tasks and offer an extensive level of error trapping. In the following sections, we'll look at the various functions these two actions shells offer, and how they were written to allow general purpose automation.

To start, each stub file for a page using one of the actionShells has several variables to define which triggers and enables the automation of the actionShell routines. The variable block looks like this:

```
[var:
    'fw_dbActn'='update',
    'fw_dbName'=(var:'fw_artclsDatabase'),
    'fw_dbTable'=(var:'fw_artclsTable'),
    'fw_dbPKey'='rcrdNo',
    'fw_dbKeyVal'=(var:'fw_r'),
    'fw_usrPrvlg'='usrUpdt',
    'fw_cnfrmFld'='artclsTitle']
```

Where:

fw_dbActn = add|addOK|delete|deleteOK|update|updateOK|search|view
fw_dbName = the name of the database
fw_dbTable = the name of the table
fw_dbFlds = a comma separated list of all fields to be returned (search action only)
fw_dbPKey = the name of the primary key field
fw_dbKeyVal = the value of the primary key for the record being acted upon
fw_usrPrvlg = the name of the var for which the user must have a Y value
fw_cnfrmFld = the name of a field to reflect in the action response

Record Locking

In any administrative environment where more than one person can manage database data, including situations where users can update their own profiles or preferences, there exists the potential that two users will attempt to edit record data at the same time.

Imagine the following scenario. Milhouse loads a record into his browser to make changes. Ingrid loads up the same record. Milhouse has to go get his dog out of the trash can. Meanwhile, Ingrid makes changes and submits the update, and heads out for her hot date. Milhouse having banished his dog, returns to make changes to the record. First off, he now no longer has accurate information. Secondly, when he saves his changes, he will unknowingly overwrite the changes made by Ingrid. Later that night, Ingrid, returning early from a miserable dating experience, searches the database and sees that her changes aren't there, convincing her to give up on computers as well as men.

While we can't help Ingrid's faith in men, we can sure do something to help her not lose faith in computers. To do that, we needed to tell Ingrid that the record she wanted to update was currently in use by Milhouse.

The first task of the update action shell is to see whether a record is locked. If it is, then FrameWork -Pro displays a message informing the user the record is busy. To avoid Microsoft-esque vagueries in our messages, we tell the user that the record may be busy for up to X minutes, and possibly less. How do we know this? A global variable defines the lock session duration much like a user authentication session. When Milhouse loaded the record, he was also informed that he had X minutes to complete the record update. If he failed to complete it in time his record update session would expire. The code also supports the option of displaying who is currently accessing the record. (Who knows we might have hooked Ingrid and Milhouse up, then they'd both have something better to do than update databases at night).

Every FrameWork -Pro database table contains four standard fields for use with record locking:

- rcrdLock = Y|empty
- rcrdLockID = a random session ID
- rcrdLockTime = a Lasso date and time
- rcrdLockOwnr = user name that owns the lock

When a record is requested for update, and found not to be already locked, a random ID value is generated and stored in a dedicated variable named `fw_k`. This value is also stored in the `rcrdLockID` field. At the same time the `rcrdLock` field is set to `Y` and the current date and time is saved to `rcrdLockTime`. The use of `rcrdLockOwnr` is optional. A value is stored, but whether it is displayed or not is up to the programmer. It is not displayed by default. The functionality of these fields is identical to typical user session management features. Below is the code. As you can see from the variables, the code is generalized to work for any record in any table.

```
# fw_k is generated by a random ID generator FrameWork library

[var:'fw_rLen'='9']
[library: ($c_gLibsPath + 'fw_glbl_makeRefcNo.inc')]
[var:'fw_k'=(var:'fw_rfcNo')]

[inline:
   -username=$fw_gClientNm,
   -password=$fw_gClientPw,
   -database=$fw_dbName,
   -sql=(
      'UPDATE ' + $fw_dbName + '.' + $fw_dbTable + ' SET ' +
      'rcrdLock="Y", ' +
      'rcrdLockTime="' + (Date_GetCurrentDate) + '", ' +
      'rcrdLockOwnr="' + $usrName + '", ' +
      'rcrdLockID="' + $fw_k + '" ' +
      'WHERE ' + $fw_dbPKey + '="' + $fw_dbKeyVal + '"')]
```

When the update form is actually submitted, the lock fields are checked to ensure the record is still locked, that the requesting update is the same session that the record was locked with, and that the lock has not expired.

The code that does all this is:

```
[inline:
   -username=$fw_gClientNm,
   -password=$fw_gClientPw,
   -database=$fw_dbName,
   -sql=(
      'SELECT rcrdNo, rcrdLock, rcrdLockID,
      rcrdLockTime, rcrdLockOwnr ' +
      ' FROM ' + $fw_dbName + '.' + $fw_dbTable +
      ' WHERE ' + $fw_dbPKey + '="' + $fw_dbKeyVal + '"')]
```

```
[If: (error_currentError)==(error_noError)]

   [var:'fw_lockDelta'=(Date_Difference:
      (Date_GetCurrentDate),
      (field:'rcrdLockTime'),
      MinutesBetween)]

   [if:
      ((field:'rcrdLock') == 'Y') &&
      ((field:'rcrdLockID') == (var:'fw_k')) &&
      ((var:'fw_lockDelta') < (var:'fw_gLockDelay'))]

   # all is good, we can proceed with the Update
```

To enable this to work, the update form must include a hidden input named `fw_k` which contains the variable `fw_k`. The value is created by the actionShell, but the programmer must be sure to include the hidden input so the value is passed to the response file, and thus the actionOK shell.

This same locking system is used for the delete action as well. This prevents a second user from trying to update a record that is likely about to be deleted. Record locking is not necessary for adds or searches.

By including some standardized fields into every table, managing record locking in this fashion as part of the actionShells automates the availability of this advanced level feature to every table in a solution with nothing more than maintaining a dedicated lock session variable in the update and delete forms.

Building INSERT and UPDATE SQL Statements

In the record locking steps, the fields being dealt with are known and fixed. That is, building a site to FrameWork - Pro conventions dictates that the lock fields exist in every table.

When adding or updating records, the main record fields are not known, and thus must be determined on the fly. Additionally, Lasso must know how to map the form input fields to specific database fields. As a security tactic, this should not be accomplished by naming the inputs with the same name as the database fields. Therefore, to provide Lasso with this information, it is necessary to create a matrix which a) lists the fields in the database table, and b) maps form inputs to those fields. Such a definition matrix is stored in a simple text file which would look like this:

```
[output_none]
apv=rcrdAppvd
ownr=rcrdOwnr
grp=rcrdGroup
rdy=artclReady
dat=artclDate
ttl=artclTitle
athr=artclAuthor
cat=artclCategory
lyt=artclLayout
syn=artclSynopsis
cpy=artclStory
[/output_none]
```

This is a simple correlation of the form input names to the database fields names. Any form which has anything to do with this database table must be consistent in its use of the form input names identified in this matrix. It is not necessary to include the lock fields as we know their names, and they have fixed functions.

The next problem is that we have to know exactly what data the forms are submitting. We cannot simply include every field in the SQL statement even if the form didn't have an input for a given field. Doing so would overwrite default values in INSERT statements designed to populate a field when no data was submitted, and of course, in an UPDATE, an empty value will be written potentially erasing existing data. So, before we can construct a list of fields for the SQL statement, we first need to acquire the list of inputs.

FrameWork -Pro has a standard library named fw_glbl_getForm.inc which is included at the end of the siteConfig file. It does a few things, but part of the code loops through the action_params list of name value pairs. Every param which does not begin with a hyphen is a form input field. (If I remember right, this routine was lifted straight from the LP5 docs). Every param without a hyphen is converted to a variable of the same name as the form input.

If the form had inputs of `apv, dat, ttl, cpy`, then this routine would create variables with those same names.

```
[loop: (action_params)->size]
   [if: !(action_params)->(get:loop_count)->
      (first)->(beginswith: '-')]
      [var:
         ((action_params)->(get:loop_count)->first) =
         ((action_params)->(get:loop_count)->second)]
   [/if]
[/loop]
```

This routine is performed automatically at the end of the siteConfig file. To ensure it is only run if the preceding page was a submitted form, the siteConfig include looks like this:

```
[if: (Client_FormMethod)=='POST']
   [include: ($c_gLibsPath + 'fw_glbl_getForm.inc')]
[/if]
```

Therefore, by the time the actionShell is loaded, all the form inputs have been converted to vars using the names we have identified in the database table definition matrix. The next step is to build an array of var names and an array of fields names which correspond in order. To do that, we read through the database table definition matrix. The code that builds these arrays looks like this:

```
# fw_fileErr stores the error message, if any, from the file access routine
# fw_filePath stores the full path to the file being accessed
# fw_tableDefn stores the full unparsed contents of the text file
# fw_paramNames stores the array of form param names
  corresponding to the fieldNames
# fw_fieldNames stores the array of field names
  corresponding to the paramNames

[var:
   'fw_fileErr'='',
   'fw_filePath'='',
   'fw_tableDefn'='',
   'fw_paramNames'=(array),
   'fw_fieldNames'=(array)]
```

```
    # read the file

    [inline: -nothing,
        -username=$fw_gClientNm,
        -password=$fw_gClientPw]

        [file_control]

            # build a path in one of two ways depending on whether the
            # site is flagged to run as FrameWork Modular or Classic
            # c_mFilesPath and c_gFilesPath are defined in siteConfig
            # the name of the defn table is dbTable_{tableName}.lasso
            # the table name was defined in the stub

            [if: $fw_gModular = 'Y']
                [$fw_filePath=(
                    $c_mFilesPath +
                    'dbTable_' + $fw_dbTable +
                    '.lasso')]
            [else]
                [$fw_filePath=(
                    $c_gFilesPath +
                    'dbTable_' + $fw_dbTable +
                    '.lasso')]
            [/if]

            [if: (file_exists: $fw_filePath)]
                [$fw_tableDefn=(file_read:($fw_filePath))->(split:'\r')]
            [/if]

            [$fw_fileErr=(file_currenterror)]
        [/file_control]
    [/inline]

    # create the array of form param names and field names
    # the file should be contained by an output_none, to prevent it
      from being readable as a plain text file in a browser,
      so we want to skip those lines
    # also, the file allows blank lines for easier reading

    [loop: ($fw_tableDefn->size)]
        [var:'fw_fieldDefn'=($fw_tableDefn->(get:loopcount))]
        [if: ($fw_fieldDefn == '') || ($fw_fieldDefn >> 'output_none')]
        [else]
            [$fw_paramNames->
                (insert:($fw_fieldDefn->(split:'=')->(get:1)))]
            [$fw_fieldNames->
                (insert:($fw_fieldDefn->(split:'=')->(get:2)))]
        [/if]
    [/loop]
```

Now we have a specific list of field names and variables that might need to be added into the SQL statement. We can start building the SQL statement with standard clauses, then we can add the unknown fields and values. To do that we'll loop through the table definition arrays. If a variable exists with the same name as an input field name, then we know that data was submitted for the corresponding field. We can grab the field name and input data and add it to the SQL statement. This translates the arrays into a fixed known list of field names affected by the submitted form and the corresponding data values.

```
# first, we start with the known fixed fields in a FrameWork
# compatible table

[var:'fw_sqlActn'='']
[$fw_sqlActn = 'UPDATE ' + $fw_dbName + '.' + $fw_dbTable]
[$fw_sqlActn += (' SET ')]
[$fw_sqlActn += ('rcrdModified="' +
   ((date_format:
      (Date_GetCurrentDate),
      DateFormat='%Y-%m-%d %T') +
   ' - ' + (var:'usrName')) + '", ')]

# we can reset the record lock fields at this point

[$fw_sqlActn += ('rcrdLock="' + '", ')]
[$fw_sqlActn += ('rcrdLockTime="' + '", ')]
[$fw_sqlActn += ('rcrdLockOwnr="' + '", ')]
[$fw_sqlActn += ('rcrdLockID="' + '", ')]

# loop through the table definition arrays and see if
# a particular field was submitted by the form
# we do that by reading the array and seeing if a var with
# name exists, if so, include it in the SQL statement

[loop: ($fw_paramNames->size)]
   [if: (var_defined:$fw_paramNames->(get:loop_count))]
      [$fw_sqlActn +=
         (($fw_fieldNames->get:loop_count) + '="' +
         (var:($fw_paramNames->get:loop_count)) + '", ')]
   [/if]
[/loop]

# we end up with a trailing comma that has to be deleted

[$fw_sqlActn->(removetrailing:', ')]

# finally add the reference to the keyfield

[$fw_sqlActn += (' WHERE ' + $fw_dbPKey + '="' + $fw_dbKeyVal + '"')]
```

All that's left to do is execute the statement:

```
[inline:
   -username=$fw_gClientNm,
   -password=$fw_gClientPw,
   -database=$fw_dbName,
   -sql=$fw_sqlActn]
```

Bingo—a 100% standardized and automated SQL execution system. The programmer's only chore is to create the simple table definition text file, and define a handful of variables in the stub file. With the actionShells, not only are the SQL statements automated, but the peripheral functions of record locking, input validation, and error management are all integrated and automated as well.

To summarize, there were three steps involved in automating the building of an SQL statement. First, was the routine to automatically snag form parameters and convert them to variables. Second was the process of reading the table definition matrix and creating corresponding arrays of form input names and field names. Third, was the code that created the SQL statement by looping through the table definition arrays and looking for matches in the defined variable space.

This process is identical for adds, updates, and searches, although there are slight variations in the third step to create the appropriate SQL statement.

Error Management

There are numerous methods in Lasso to handle error management, and with Lasso Professional 5, there are even more. At one level, the programmer can rely on Lasso's own internal error trapping. Using the automatically called error.lasso page, it's fairly easy to create a less geeky error message for your users than the default "blue box" that Lasso generates. LP5's new Protect, Fail, and Handle tags make it even easier than previous versions to build highly tuned customized error management.

Once an application has been written and debugged, the majority of non-syntax errors that need to be communicated to site visitors are likely going to be related in one way or another to data input filtering and database actions.

In FrameWork 1.0.2 error handling code is written into each format file that required a database action or input filter. The [if]...[else]...[/if] structures used to test for errors are standardized, and the error messages are separated into includes for easier maintenance. That structure looks something like this for an update response page:

```
# from FrameWork 1.0.2
[if:
   (var:'usrAppvd',encodenone)=='Yes' &&
   (var:'propUpdt',encodenone)=='Yes']

   [inline: -search,
      -clientusername=(var:'fw_gClientNm',encodenone),
      -clientpassword=(var:'fw_gClientPw',encodenone),
      -database='properties.fp4',
      -layout='allFields',
      -op='eq',
      'rcrdNo'=(var:'fwr',encodenone),
      -token.skip=(token_value:'skip',encoderaw)]
```

```
            [if: (found_count) != 0]
               [If:(error_currentError)==(error_noError)]
                  [var_set:'fwrid'=(recordID_value)]

                  [inline: -update,
                     -clientusername=(var:'fw_gClientNm',encodenone),
                     -clientpassword=(var:'fw_gClientPw',encodenone),
                     -database='properties.fp4',
                     -layout='allFields',
                     -recordID=(var:'fwrid',encodenone),
                     etc...]

                     [If:(error_currentError)==(error_noError)]

                        ..... display confirmation .....

                     [else]
                        [include: '/site/libs/fw_glbl_errorMngr.inc']
                     [/If]
                  [/inline]
               [else]
                  [include: '/site/libs/fw_glbl_errorMngr.inc']
               [/if]
            [else]
               [include:'/site/libs/fw_glbl_notFound.inc']
            [/If]
         [/inline]
      [else]
         [include:'/site/libs/fw_glbl_noPass.inc']
      [/if]
```

Though the methods were standardized, and the messages separated as includes, this still required a lot of redundant code. If we wanted to add some more features to error handling methods, it required updating a lot of pages.

In FrameWork -Pro, the database actionShells removed the redundancy of the error testing structures to a single source location. In fact, segregating that error management was a prime motivator for creating the actionShells. Another feature that was developed for FrameWork -Pro was an improved error message system. This was originally developed in Lasso 3 and has been transferred to Lasso 5 with little change except for syntax. Therefore, it doesn't make use LP5's new tags, but the concepts generally replicate the methods one would employ with the Fail tag.

Below is the error handling structure of the updateOK actionShell from FrameWork -Pro with the productive code removed:

```
[$fw_dataValid='Y']
[library: ($c_mAdmnLibsPath + 'admn_validate.inc')]
[if: $fw_dataValid == 'Y']

   [if: (var:'fw_r') != '']
      [inline:
         -username=$fw_gClientNm,
         -password=$fw_gClientPw,
         -database=$fw_dbName,
         -sql= acquire record lock status fields ]

         [If: (error_currentError)==(error_noError)]
            [if: ..... test for record lock status ..... ]

               ..... read table definition matrix .....

               ..... create SQL statement .....

               [inline:
                  -username=$fw_gClientNm,
                  -password=$fw_gClientPw,
                  -database=$fw_dbName,
                  -sql=$fw_sqlActn]

                  [if: (error_currentError)!=(error_noError)]
                     [$fw_Error += (
                     '5024' + ':::' +
                     'Record could not be added due to Lasso error ' +
                     (error_currentError: errorCode) + '\r')]
                  [/if]
                  [if: $fw_error != '']
                     [include: ($c_gLibsPath + 'fw_glbl_errorMngr.inc')]
                     [abort]
                  [/if]
               [/inline]

               [if: (var:'fw_cnfrmFld') == '']
                  [var:'fw_cnfrmFld'='rcrdNo']
               [/if]
               [inline:
                  -username=$fw_gClientNm,
                  -password=$fw_gClientPw,
                  -database=(var:'fw_dbName'),
                  -sql= acquire fields for confirmation display ]

                  [if: (error_currentError)==(error_noError)]
                     ..... build standard  or include custom
                           confirmation  page .....
```

```
                    [else]
                       [if: (error_currentError)==(error_noRecordsFound)]
                          [$fw_error += (
                          '5021' + ':::' +
                          'Record number not found when trying to
                           acquire record data' + '\r')]
                       [else]
                          [$fw_error += (
                          '5022' + ':::' +
                          'Record data could not be acquired due to
                           Lasso error ' +
                          (error_currentError: errorCode) + '\r')]
                       [/if]
                    [/if]
                 [/inline]

              [else]
                 [$fw_error += (
                    '5031' + ':::' +
                    (date_add:
                       (field:'rcrdLockTime'),
                       minute=(var:'fw_gLockDelay')) + '\r')]
              [/if]

           [else]
              [If: (error_currentError)==(error_noRecordsFound)]
                 [$fw_error += (
                    '5021' + ':::' +
                    'Record number not found when trying to
                     acquire recordID' + '\r')]
              [else]
                 [$fw_error += (
                    '5022' + ':::' +
                    'Record could not be found due to Lasso error ' +
                    (error_currentError: errorCode) + '\r')]
              [/if]
           [/if]
        [/inline]

    [else]
       [$fw_error += (
          '5022' + ':::' +
          'Record number empty' + '\r')]
    [/if]

[else]
   [output_none]<!--
      ..... the field validation include built up the fw_error variable
      ..... nothing else needs to be done here -->[/output_none]
[/if]
```

The [if] test structures are typical of LDML error testing. What is unique is what is written in the [else] clauses. FrameWork - Pro utilizes an internal variable `fw_error` to store a delimited string of error codes and messages. The data structure still carries its Lasso 3 heritage, and could now be stored as an array of pairs in LP5, but that's for a future update.

The error message building portion of the code does three things. First it prepends the current error message with any previous error messages. Thus, the system can trap multiple, sequential, non-fatal errors and report them all. Second, an error code is added to the `fw_error` variable. Third, an error string is added. This string is not necessarily the error message, but rather a string that can be used to augment the error message which is actually generated by a separate routine. For most cases in the actionShells, the string is really just a useful note to denote what the error code means. The real usefulness of the error string comes into play with the field validation system which will be discussed in the next section.

The purpose for this error code based system was to provide greater granularity in error messages that would be specific to the steps being executed by the actionShell.

If an error occurs, the `fw_error` variable now becomes non-empty. A non-empty condition is tested for at the end of the blockManager with this code:

```
[if: $fw_Error != '']
   [include: ($c_gLibsPath + 'fw_glbl_errorMngr.inc')]
[/if]
```

A second error trapping system is used to capture fatal Lasso syntax errors. For this, the file error.lasso is coded like a stub file to simply snag the error code and message provided by Lasso and include a template specific to reporting errors. The template in turn includes the fw_glbl_errorMngr.inc file.

```
[var:
   'fw_crntError'=(error_currenterror),
   'fw_crntErrCode'=(Error_CurrentError: ErrorCode)]
[include:'/site/c_siteConfig.lasso']

[var:'c_pgTtl'='Error']
[var:'c_hdrFile'='hdr_err.las']

[include:
   ((var:'c_gTmpltsPath') +
   'tmplt' + $c_myStyle + '_error.lasso')]
[abort]
```

The errorMngr library does a few things. First, it determines whether the error being reported is a Lasso fatal error or an internal FrameWork error:

```
[if: $fw_Error != '']
   [include:($c_gLibsPath + 'fw_glbl_errorFW.inc')]
[else]
   [include:($c_gLibsPath + 'fw_glbl_errorLasso.inc')]
[/if]
```

Each of the two error handling includes specializes in dealing with the respective internal FrameWork and Lasso error types.

The errorFW.inc file splits the `fw_error` string into its individual codes and messages and loops through each set using an error code based conditional tree to compile an error message for display. Skipping the string splitting code, a portion of the error code tree looks like this:

```
[else: ($fw_errCode == '5010')]

   <p><b>User Login Not Recognized</b></p>
   <p>That name and password are not recognized for access. Perhaps
      you entered them incorrectly.</p>
   [var:'fw_errReturn'='Y']

[else: ($fw_errCode == '5012')]

   <p><b>User Session Expired</b></p>
   <p>Your authorization or security session has expired. Please
      login again.</p>
   <p>For security reasons, when your browser has not connected
      from one secured page to another secured page after
      [var:'fw_gSessnTimeout'] minutes, the server considers the
      connection inactive and cancels the login you started with.
      Simply log back in, and you may continue.</p>
   [var:'fw_errLogin'='Y']

[else: ($fw_errCode == '5030')]

   <p><b>Record is Busy</b></p>
   <p>The record you have requested is currently being edited. The
      record will be available within [var:'fw_gLockDelay'] minutes,
      and may be available sooner.</p>
   [var:'fw_errReturn'='Y']

[else: ($fw_errCode == '5032')]

   <p><b>Record Lock has Failed</b></p>
   <p>The record you are editing could not be locked due to an error.
      (Error #5032 occured on page [var:'c_myName'].
      [var:'fw_errMsg']).</p>
   [var:'fw_errReturn'='Y']
```

This system allows for the definition of numerous very specific errors with the error message content controlled centrally in this one file.

The errorLasso.inc file is used to pass along information from a fatal Lasso error. Instead of displaying the Lasso blue box, the error code and message are delivered in more user friendly fashion.

At the end of each error message, we would want to present the visitor with some options. If a session expired we would want the visitor to log back in. In many cases, we simply need the visitor to return to the previous page. You'll notice that the last statement for error code 5012 sets a variable named `fw_errLogin` and 5030 sets a variable `fw_errReturn`. Once the errorFW.inc file has concluded processing, and reverts back to the errorMngr.inc file, the errorMngr checks a variety of variables like fw_errLogin and fw_errReturn. These are standardized responses applicable to any number of error messages. If fw_errLogin == Y, then a link to the login page is

displayed. If fw_errReturn == Y, then a link to return to the previous page is displayed. Placing this code in the errorMngr.inc file prevents it from being redundantly repeated with each error message in the errorFW.inc file.

Finally, another element of the error manager that is automated is the start of an email message to the site's support group. One of the setup variables for a FrameWork site is a support email address. When an error occurs that is likely due to a programming problem, a message is displayed asking the user to email the site support group. To get a jump start on providing the information needed, the following code is used to create an email subject and body:

```
[var:'fw_errMailMsg'= (
   'Error \"' +
   (var:'fw_crntError') +
   '\" (' +
   (var:'fw_crntErrCode') +
   ') occurred on page \"' +
   $fw_gHome + $c_myPath + $c_myName +
   '\"')]

... code snipped ...

# this message is displayed on the web page

<p>If the error occurs again, please send a message to our tech
   support staff at <a href="mailto:[var:'fw_gSupportEmail']?
   subject=Website Error
   &body=[var:'fw_errMailMsg']">[var:'fw_gSupportEmail']</a>
   and report that you got the following error message:</p>

... code snipped ...
```

Of course, additional information could easily be added to the body message to provide as much relevant feedback as is needed. The body message could even be customized based upon the actual error code.

Input Validation

Input validation is a specialized subset of error management. Instead of testing for errors during database actions, the tests compare input data with some rules of data conformity. If the tests fail, we need to inform the user.

With the structure of FrameWork -Pro's error management system, it was easily extended to handle field validation and the task of building a display message to inform the user of input data errors.

To automate as much of the validation process as possible, you'll recall that the first step of the updateOK action shell was to validate input form data. This is accomplished by initializing a validation status flag, including a validation rules file, then testing the status flag before continuing with the database update action. The relevant code from the beginning of an updateOK or an addOK action shell looks like this:

```
[$fw_dataValid='Y']
[library: ($c_mLibsPath + 'form_validate.inc')]
[if: $fw_dataValid == 'Y']

   ..... continue action shell .....

[else]
   [include: ($c_gLibsPath + 'fw_glbl_errorMngr.inc')]
[/if]
```

The form_validate.inc file looks like this:

```
# password input cannot be empty

[if:
   (var_defined:(var:'upw')) &&
   (var:'upw') == '']

   [$fw_Error += (
          '5101' + ':::' +
          'Password' + '\r')]
   [var:'fw_dataValid'='N'][/if]

# password input cannot contain spaces

[if:
   (var_defined: (var:'upw')) &&
   ((String_CountFields: delimiter=' ', (var:'upw')) > '1')]

   [$fw_Error += (
          '5108' + ':::' +
          'Password' + '\r')]
   [var:'fw_dataValid'='N'][/if]
```

As you can see the response to a failed test is the same concatenation of a code and message string to the `fw_error` variable as was used for the actionShell errors. In this case, the message string is the name or label of the field as shown on the user's input form. This string will be used in the message displayed to the user. Additionally, the `fw_dataValid` flag variable is set to N so that upon return to the actionShell, the action will not process, and instead the error manager is loaded.

One extra feature to note is that each actual data validation incorporates a test to see if the variable that contains the field data has been defined. This is a direct tie-in to the automated getForm library which converts form inputs into vars. This extra test allows us to build a single validation code file per database table with every input field that requires validation. When the validation code is processed, only those fields which are part of the form will actually be tested. This eliminates the need to create multiple form-specific validation files that would result in a lot of redundant code prone to inconsistency errors.

When validating form input, it is of course possible for several fields to have invalid data. We certainly don't want the user to be informed one field at a time, so rather than stop processing at the first field failure, we process the whole form and collate all error messages into a single display. It is for this purpose that the error manager and the `fw_error` variable are set up to contain multiple sequential errors.

There are a number of predefined field validation codes and messages built into the error manager. Below is an example of a few of the codes defined in the errorFW.inc file. The `fw_errValHdr` variable is used to ensure that a title is displayed above the messages, but that it is not repeated as the routine loops through each error code in the `fw_error` variable.

```
[if: (var:'fw_errValHdr') != 'Y']
   <p class=subhdblk>Form Information Requires Corrections</p>
[/if]

[if: ($fw_errCode == '5100')]
[else: ($fw_errCode == '5101')]
   <p>The field <b>[var:'fw_errMsg']</b> cannot be blank.</p>
   [var:'fw_errReturn'='Y','fw_errValHdr'='Y']

[else: ($fw_errCode == '5102')]
   <p>The field <b>[var:'fw_errMsg']</b> must only contain
      digits 0-9.</p>
   [var:'fw_errReturn'='Y','fw_errValHdr'='Y']

[else: ($fw_errCode == '5103')]
   <p>The field <b>[var:'fw_errMsg']</b> must be a valid number.</p>
   [var:'fw_errReturn'='Y','fw_errValHdr'='Y']

[else: ($fw_errCode == '5104')]
   <p>The field <b>[var:'fw_errMsg']</b> must contain only
      letters.</p>
   [var:'fw_errReturn'='Y','fw_errValHdr'='Y']
```

Field validation is largely automated in that the step of including validation is integrated into the actionShells, and the construction of an error message is automated using the error management system. The programmer still has to define the specific tests required for specific form inputs, but this is confined to a single file using a simple, repeatable method for each field which significantly minimizes the task.

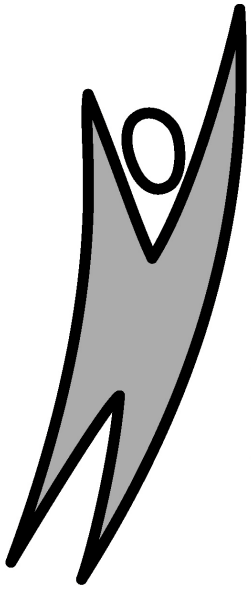
Summary

In this paper we've taken a look at several common supporting code systems used in the development of dynamically constructed web sites using Lasso Professional 5. Byusing standardized resource naming conventions, and variable driven resource referencing, we've demonstated methods in which these systems can be coded for automatic execution with only minimal trigger variables and configuration code.

We also discussed some scenarios where redundant code has been segregated into include files which can be more readily maintained for improved performance or functionality.

By combining these practices and creating a standardized library of functional routines, the developer can eliminate a significant amount of the redundant programming that is often done, and a significant amount of troubleshooting and debugging that goes with it.

For more information about [FrameWork: -Pro], visit ldml.gregwillits.ws.



Lasso
Development

The Company

www.LassoDevelopment.co.uk

The Community

www.LassoDevelopment.com

The Application

www.LassoCMS.com

LassoApps: Protecting Your Code

by Duncan Cameron

Want to develop applications and sell them as a product while protecting your source code? Want to protect your clients from accidentally deleting source files? Duncan will explain the benefits of using LassoApps with your applications opposed to distributing accessible source code. As well as guiding wanting developers through structure guidelines when planning and developing large LassoApp driven applications.

Biography

Duncan has been specializing in building database driven applications since 1986. Getting involved with Lasso in early 1997, Duncan is the founder of LassoDevelopment.com and the UK Lasso Association as well as the author of 'Lasso Professional 5 Developer's Guide' and co-author of the Lasso Professional 5 vs PHP white paper.

Notes

Protection with LassoApps

LassoApps are one of the greatest features of the Lasso product line, as they allow developers and development companies to put together solutions with the following additional features:

1. Source Protection
2. Packaged Solution

Source Protection

Commercial Perspective

There is the community that agrees that open source is the way we should provide our applications to allow education, security awareness and bug fixing.

While this is true to an extent, we still need to be aware that many of the developers and development companies who provide applications to paying clients do so to earn a living, and the source code to the applications they develop is at the heart of that living.

This is one of the reasons why LassoApps are so appealing to the commercial community, as they can convert their current open source applications into a source secure application, and ensure that their hard sweat and tears while developing the application is protected (Intellectual Property).

The benefits to this include the facility of developing an application to distribute and sell to multiple organisations with the knowledge that the source code is protected from piracy.

Client Perspective

As in the past with web based applications, they have been required to be run the majority of the time on the client's servers.

For some clients especially those who are not web application aware, the many files and directory structure of an application can be mind blowing, and bring concern that it can be maintained, supported within the client's premises.

LassoApps give the developer and development company the reassurance to inform the client that because the application is running from a single file, no files can accidentally be deleted or moved because they are all compiled into one LassoApp application file.

This is a big selling feature of the LassoApp based application outside of intellectual property protection.

Packaged Solution

File Size

Large web based applications can contain files and images that run into the thousands, which can bring a headache to the transport of the application to client servers. However, with a LassoApp based solution the application can be only one binary compiled file when presented to the client.

This ensures that the file structure is kept to a minimum and can be presented on a floppy disk, or even emailed to the client from the slowest connection speeds.

From a logistical perspective this creates a reassurance with the client, as they don't have the concerns as previously mentioned with the non-skilled employees installing the application. As it can be easily inserted into the web root folder as picking up a set of keys from a table.

Portability

The portability of the LassoApp application brings additional benefits such as, allowing a demonstration version of the application to be distributed with ease.

Types of easy distribution:

1. Email Distribution
2. File Download Distribution
3. CD Distribution
4. Floppy Disk Distribution

This opens up your application to become commercially viable, and bring the benefits to smaller development companies to develop larger solutions and have the reassurance that the application can be distributed with ease, thus more clients having the facility to get hold of the file easily.

How do LassoApps work?

LassoApps contain formatted lasso files such as text files with the suffix '.lasso' and images, compiled into a single binary file and loaded to through RAM and Cache. This means that while the Lasso enabled web server is running, you can load a LassoApp based application once and from that point onwards, the application resides in memory and is not loaded again.

This brings the advantage of speed to the application, as the server is already running your application and no longer needs to load it, and just calls the relevant element of the LassoApp to process the request sent.

To identify if you are currently using a LassoApp based application, the serving web site URL will include the following suffix

.LassoApp

What are requirements to running LassoApps?

In order to run a LassoApp based application, the web server must be running Lasso provided by Blue World Communications Inc. http://www.blueworld.com

Preparing for a LassoApp application

When developing a LassoApp driven application, there are a few things to be aware of and also to get in order, these are:

1. File type restrictions
2. Custom Tag Caution
3. File Structure
4. Linking and Including Files

What type of files can be included within a LassoApp application?

LassoApp driven applications can include lasso formatted text files that would contain the source code of the application and image files such as .gif and .jpg file types.

If you were looking to include files such as a Flash or Director file, you would compile these files independently and then include them within a folder to reside with your compiled LassoApp application.

Custom Tag Caution

One element to a LassoApp you need to be fully aware of, is the usage of Custom Tags, this is because when a Custom Tag is loaded into memory, the tag assigns its name and sits there until used.

If you create another application with a custom tag of the same name, you will receive an error, unless you place the following code into your custom tag:

```
<?lassoscript
        If: !(Lasso_TagExists: 'customTagName');

                ……..
        /if;
?>
```

This then will check to see if a custom tag already exists with the same name, and if it does it will not generate the custom tag and place it into memory.

This also has to bring caution to Custom Tags with the same name, however doing a completely different function.

Naming is important.

File Structure Rules

The file structure needs to be carefully planned to ensure that the application knows the location of each included file. When including files a LassoApp will always look for the file from the root of the LassoApp structure.

Lets progress through a small LassoApp application being prepared. The application will contain an image called logo.gif and several files containing the scripts for the application.

The files of the application will be contained within a folder called ‘Example’, which will then contain three more folders called ‘Images’, ‘Library’ and ‘Tables’.

example	–	contains all files, folders and images that work with the application.
images	–	contains all images that are used within the application.
library	–	contains all library files used within the application.
tables	–	contains all table templates used within the application.

All other files will remain in the root of the folder.

LassoApp Structure Example

```
example Folder
 |
 |--- image Folder
 |      |
 |      |--- logo.gif
 |
 |--- library Folder
 |      |
 |      |--- inc_lib_conf.lasso
 |      |--- inc_lib_datevars.lasso
```

```
|
|--- tables Folder
|      |
|      |--- table_ae_difference.lasso
|      |--- table_ae_format.lasso
|      |--- table_v_search.lasso
|
|
|--- index.lasso
|--- indx_main.lasso
|--- searchR_main.lasso
|--- template_public.lasso
```

When planning the file structure of a LassoApp application, you need to aware of some set rules to ensure your application won't hit any problems.

The first rule is:

You can NOT include a file from a file that is sitting up root from the current file location.

What I mean is:
(see the table structure to assist with the following explanation)

We CAN include the file in tables called table_v_search.lasso from the root file called indx_main.lasso because it is down root in location.

What we can NOT do is call indx_main.lasso from the file table_v_search.lasso within the tables folder because it is up root in location.

One alternative to the previous solution is to have all of the files sitting within one folder and prefix all the names with clear descriptions such as:

```
example Folder
|
|--- image Folder
|      |
|      |--- logo.gif
|
|--- index.lasso
|--- indx_main.lasso
|--- library_inc_conf.lasso
|--- library_inc_datevars.lasso
|--- searchR_main.lasso
|--- table_ae_difference.lasso
|--- table_ae_format.lasso
|--- table_v_search.lasso
|--- template_public.lasso
```

This will allow you to identify the files quickly.

Large LassoApp applications can contain large number of files so as you can see planning is important with file structures. Once you grasp the file structure you will be progressing nicely to start building your LassoApp application.

How do I link or include files within a LassoApp?

The LassoApp is developed exactly the same as if you where constructing a normal web site with linking to files, file response pages, displaying images and including inlines with a few additions:

All images and files when compiled into a LassoApp are given a reference point, that Lasso uses to call the right file/image, so in order for Lasso to log these reference points you need to surround you links and images with the LDML tag [LassoApp_Link]. Once this has been done, it will be included in the LassoApp and positioned ready for the running of the LassoApp.

An example of using the [LassoApp_Link] tag is shown below:

Anchor Tag Link

```
<A HREF="[LassoApp_Link: 'index.lasso']">Homepage</A>
```

Form Action

```
<FORM ACTION="[LassoApp_Link: 'index.lasso']" METHOD="POST">
        ...
</FORM>
```

Image

```
        <IMG SRC="[LassoApp_Link: '.images/logo.gif']" BORDER="0">
```

Also this does NOT affect the running of the application without it being a LassoApp, as Lasso just ignores the LDML tag, but it IS required for the link or image to work when used within a LassoApp to work.

If you are looking to link to one of the files within the folder but contain a parameter with a value, this is how it will look:

```
<A HREF="[LassoApp_Link: 'index.lasso']?process=1">Homepage</A>
```

As you can see it is only the filename you surround with the tag.

If you are using [Response_Filepath] as your link or a form action, you do NOT need to surround this with [LassoApp_Link] as the page is known and will call itself.

When working with the tags [Include] and [Library] these will need to look like:

```
[Include: (LassoApp_Link:'header.lasso')]
```

or

```
[Library: (LassoApp_Link:'site_config.lasso')]
```

Remote Links

LassoApps are NOT restricted to linking only to files that are contained within the LassoApp file and can link to remote files simply by using the normal link:

```
<A HREF="http://www.lassodevelopment.com">Working Link</A>
```

Variable Usage

The LDML tag [LassoApp_Link] cannot contain variables, this means the following would NOT work:

```
[if: $condition ==’test1’]
        [var:’page’ = ‘one’]
[else: $condition ==’test2’]
        [var:’page’ = ‘two’]
[/if]

[include: (LassoApp_Link: $page + ‘_main.lasso’)]
```

So what to do is the following:

```
[if: $condition ==’test1’]
        [var:’page’ = (LassoApp_Link:‘one_main.lasso’)]
[else: $condition ==’test2’]
        [var:’page’ = (LassoApp_Link:‘two_main.lasso’)]
[/if]

[include: $page]
```

We don’t need to include the LDML tag [LassoApp_Link] in the above include because we have already set it within the variable $page.

Naming the LassoApp

When naming LassoApps its important to ensure that you don't give them the same name as the ones that Lasso uses for administration purposes; Admin.LassoApp and Startup.LassoApp as these will conflict with the Lasso set-up on the machine the LassoApp is running on.

Another thing to watch out for is duplication of LassoApps, this can be done accidently by using a third parties LassoApp with the same name of one you are already using, I suggest prefixing your LassoApps with your initials e.g. DC_example.LassoApp

Also I recommend keeping a sequence going when developing large applications that will be maintained with upgrades and updates.

For example, the first file name could be:

DC_example_1a

Then while you are building in version 1, you just need to progress the a to b:

DC_example_1b

This is an easy way to control your upgrades and updates, because you don’t need to change it to version 2a until you are happy that it qualifies for a full number change.

Building the LassoApp

There are two ways to build a LassoApp, either through the Lasso Administration Panel under the menu heading BUILD and the sub menu LASSOAPP BUILDER or using the [LassoApp_Create] LDML tag.

There are advantages to both methods:

The LDML tag enables granted users to create LassoApps directly from their hosted Lasso enabled folder, as well as the facility to create an automated LassoApp generator via a public interface.

The Admin builder allows the developer to keep a backup of the last version of the application that was built. This is because the folder would not be used for serving LassoApps but to only build the application, so no temptation is there to change the code until its been tested via a serving folder.

Using the LDML tag

The [LassoApp_Create] LDML tag requires three parameters:

1. -Root where the folder that contains the files, sub folders and images to be included in the LassoApp. This needs to be a full path from the root of the machine and not the web server.

2. -Entry the file that is the first page to load when the LassoApp is opened. This path is just the relative filename from the folder set in -Root.

3. -Result where you want the file to be placed once built. This needs to be the full path from the root of the machine and not the web server.

On a windows setup the tag will look like this:

```
[LassoApp_Create:    -Root='C:\\InetPub\\wwwroot\\AppFolder\\',
                     -Entry='index.lasso',
                     -Result='C:\\InetPub\\wwwroot\\AppFolder.LassoApp']
```

Notice we need to use the escape character \ to allow Lasso to read the full path correctly as we have used the backslash \ for the Windows directory setup.

Once this tag is loaded within a page in Lasso, it will take the Folder 'AppFolder' and create the file AppFolder.LassoApp and deposit it within the wwwroot folder.

When the LassoApp is called via the browser it will call the page index.lasso that is compiled within the LassoApp

On an OS X setup the LDML tag [LassoApp_Create] may look like this:

```
[LassoApp_Create:    -Root='/Library/WebServer/Documents/AppFolder/',
                     -Entry='index.lasso',
                     -Result='/Library/WebServer/Documents/AppFolder.LassoApp']
```

Using the Lasso Administration Interface

To use the Lasso Administration builder, you need to:

1. place the folder containing the files and images for the LassoApp into the Admin/BuildLassoApps folder within the Lasso Professional 6 application folder.

As with the LDML tag method, the name of the folder will become the LassoApp file name.

2. Open up your browser and then launch the Lasso Administration panel.

Once the Lasso Administration panel is open:

3. Click on BUILD.

4. Then select LassoApp Builder.

5. If your folder is the only LassoApp ready to be built then you will immediately see the folder name appear within the drop down available, if not sift through the names until you find the folder name you called your LassoApp.

6. Then you will be asked for the entry filename.

This will be the index file of your application, so if you called your index file index.lasso then this is what you type into this box.

This is where Lasso will visit first to then work through every link and image surrounded with the [LassoApp_Link] LDML tag to start referencing them.

7. Click on Create LassoApp

Once complete your LassoApp file will be awaiting your collection within the Admin/BuildLassoApps folder where you deposited your folder originally.

Take your file and then load it into your web serving folder and type in the URL including the full LassoApp filename.

Possible Errors while building a LassoApp

One great feature of LassoApp building is the error reporting, this will happen when the following occurs:

Syntax Error
Missing Referenced File
Empty File

Syntax errors, are great because you can use the LassoApp builder as a great debugging tool, while developing your application.

If you receive an error that does not contain a reason just a number -9968, this is generally because you have provided a page that is blank and contains no content.

Navigating LassoApps

If you want to call specific files or images within a LassoApp, this can be done by using the -ResponseLassoApp tag as follows via a URL:

http://domain.com/example.LassoApp?-ResponseLassoApp=pagename.lasso

Summary

To summarise LassoApps in a short paragraph, I would have to say:

If you have not yet tried using LassoApps or have been worried about trying the function out, give it a shot. They are fun, thought provoking and straight forward.

LassoApps provide both the application developer and the development company the functionality of offering large browser based applications that offer easy distribution, protection of intellectual property and the scope of a large commercial opportunity.

“Web based applications with a twist” ☺

Duncan Cameron

edition.net the Fine Arts Network

manage your own dedicated server
dedicated platforms - single and multi-domain hosting versions

your choice of MacOS, MacOSX or Linux - FileMaker Pro hosting featuring Lasso
MacOSX Server featuring QuickTime - on-demand and live broadcast streaming solutions

Our control panels make it easy to deploy your multi-domain server

pricing for edition.net FileMaker Pro solutions

fm10012 - edition.net database solution - MacOS, single domain, 2gig storage
FMPro Unlimited, Web Companion Web Connector, Lasso — $85 monthly

fm10017 - edition.net dedicated server MacOS - multi-domain, 4gig storage
FMPro Unlimited, Web Companion Web Connector, Lasso, branded name servers
edition.net dual-server technology - database server + www server — $140 monthly

osx10022 - dedicated server solution - MacOSX Server, multi-domain,
4gig storage, 256mb ram, WebStar V + FileMaker Unlimited, Lasso Professional 5,
supports multiple data sources, 200 user webmail, branded name servers — $140 monthly

osx10021 - dedicated server solution - MacOSX Server, multi-domain,
40gig storage, 512mb ram, WebStar V + FileMaker Unlimited, Lasso Professional 5,
supports multiple data sources, 200 user webmail, branded name servers — $220 monthly

osx10023 and osx10024 dedicated server solutions featuring Xserve

Lasso Web Data Engine

all solutions feature unlimited transfer, dedicated cgi-bin, ftp access, edition.net server contol panels,
dedicated security control panel, dedicated email server@yourdomain, daily backup, uninterrupted power supply (UPS),
high speed internet connectivity, flat-rate pricing with no traffic surcharges, 24 / 7 monitoring

FileMaker Solutions Alliance

custom configurations always available
finally – the control you have been looking for!

ADC Apple Developer Connection

Apple Solution Experts

edition.net
http://www.edition.net
info@edition.net
+1 (877) 225-3821 toll free sales

EGM

Advanced Developers Techniques

by Fletcher Sandbeck

This presentation will provide details about advanced developer techniques for Lasso including syntax checking, debugging, automatic error recovery, error logging and reporting, and more.

Biography

Fletcher Sandbeck is the Lasso Product Specialist for Blue World Communications. He has worked on the Lasso Studio, Lasso Developer, and Lasso Professional product lines. His many roles at Blue World have included feature specifications, engineering of Lasso Studio and the Lasso Administration interface, beta program coordination, and documentation. Prior to joining Blue World, Fletcher was an active Lasso developer and member of the Lasso community for many years with his family owned firm, Cumuli Design. Fletcher is a graduate of the Massachusetts Institute of Technology with a bachelors degree in Mathematics with Computer Science and has worked for MIT, Caltech, and the University of Washington.

Notes

Advanced Developer Techniques

By Fletcher Sandbeck

Blue World Communications

Introduction

This paper provides a series of briefings of advanced developer techniques which are made possible by new features of Lasso Professional 6. Many of these techniques are covered in the Extending Lasso 6 Guide. Hopefully this paper will serve to provide additional details of when these techniques are useful and how best to implement them in real solutions.

The techniques presented in this paper are by no means exhaustive. Each technique merely scratches the surface of what is possible with Lasso Professional 6.

I have tried to avoid going over basic developer techniques in this paper. For Lasso users who are new to creating custom tags and data types it is best to read this paper in conjunction with the Extending Lasso 6 Guide or the other papers presented at the Lasso Summit.

Compound Expressions

Lasso Professional 6 introduces a few new syntax constructs which are primarily useful to advanced developers. These include compound expressions which will be discussed here and referenced which are discussed later.

A compound expression is one or more tag calls written as if in a LassoScript, but enclosed by curly brackets { }. They can be thought of us a type of LassoScript, or a C-style execution block, or as a custom tag without a name.

Compound expressions can be executed immediately using the [Tag->Eval] tag. This evaluates the compound expression in the environment of the current page. The following compound expression returns a variable $condition if it is not equal to null, otherwise ‘’ is returned.

```
[var: ‘test’ = { if: ($condition == null); return: ‘’; /if; return: $condition; }->Eval]
```

The expression can also be written with indentation for better readability.

```
[var: ‘test’ =
    {
        if: ($condition == null);
            return: ‘’;
        /if;
        return: $condition;
    }->Eval
]
```

This can be useful for performing multi-tag calculations inline without creating a custom tag or using extra variables, but what gets really interesting is using compound expressions as mini tags.

A compound expression is an object just like any other in Lasso. It is an object of type ‘tag’ which can be stored in variables, even serialized into sessions are restored on a later page. The following two lines are equivalent to the code above, but now the code stored in the $myTest variable can be called multiple times without rewriting it.

```
[var: ‘myTest’ = { if: ($condition == null); return: ‘’; /if; return: $condition; }]
[var: ‘test’ = $myTest->Eval]
```

A mini-tag can be created in a compound expression by calling it using the [Tag->Run] tag rather than the [Tag->Eval] tag. The [Tag->Run] tag allows parameters to be passed and even for member tags to be called. The following variation of the compound expression above fetches a parameter like a tag and returns it only if it is not equal to null.

```
[var: 'myTag' =
    {
        if: (params->(get:1) == null);
            return: '';
        /if;
|       return: params->(get:1);
    }
]
```

Now this 'tag' can be called using the [Tag->Run] tag as follows. The parameters of the tag are passed in the -Params parameter as an array or a single parameter can be passed straight.

```
[var: 'test' = $myTag->(Run: -Params=$condition)]
```

This 'tag' can now be changed into a real tag by giving it a name. This is accomplished by installing the compound expression into the [Tags] map. Then, the tag can be called like any other. The [Tags] map uses tag names as the key and tag objects as the value.

```
[Tags->(Insert: 'myTag', $myTag)]
```

Now the new tag [myTag] can be called like any other tag.

```
[var: 'test' = (myTag: $condition)]
```

One final example will serve to demonstrate how compound expressions can be very useful in cutting down the amount of code that needs to be written to accomplish complex tasks.

The example is creating a tag that performs the same operation on each element of an array. We create a tag that takes two parameters, an array and a compound expression that performs an operation on one element of the array.

```
[Define_Tag: 'exArray', -Required='myArray', -Required='myTag']
    [iterate: #myArray, (local: 'myItem')]
        [#myItem = #myTag->(Run: -Params=#myItem)]
    [/iterate]
[/Define_Tag]
```

This very simple tag can now be used to modify an array in place using a compound expression. For example, the following code squares every element in an array of integers.

```
[var: 'myArray' = (array: 1, 2, 3, 4, 5)]
[exArray: $myArray, { return: (params->(get: 1) * params->(get:1)) }]
[output: $myArray]
-> (array: 1, 4, 9, 16, 25)
```

Global Variables and References

Lasso Professional has always supported global variables, but Lasso Professional 6 includes some new tags that make them really easy to use. This section discusses a solution that uses custom container tags, global variables, and references to implement a server-wide page element caching mechanism. The goal is to have a very easy-to-use cache system that allows page elements such as a top-ten list or message of the day to be generated periodically then served from the cache until they expire.

The new container tag is called [exCache] ... [/exCache] and is defined in the LassoScript in the code file that accompanies this document. It requires a -Name parameter which names the cached content. An optional -Expires parameter specifies when new content should be generated in seconds. An optional -Condition parameter if true refreshes cached content immediately. An optional -Log parameter helps with debugging.

The code splits into three parts. The first part sets up the environment for the tag and parses incoming parameters. The next part decides whether a refresh is needed by examining items stored in the cache, the -Expired parameter, and the -Condition parameter. The final part of the tag evaluates the contents of the container tag and stores it in the cache if necessary. The part of the tag that uses global variables is actually very short and is recreated below.

First, the [Global_Defined] tag is used to check whether an existing cache storage variable has been created. If not, a new global variable is created to store the cache. This global variable will immediately be available to all other processes and format files in Lasso Service.

```
// Initialize the global cache storage
if: !(global_defined: 'ex_Cache_Storage');
    global: 'ex_Cache_Storage' = (map);
/if;
```

Next, a reference to the global cache storage is stored locally. The @ symbol stores a reference rather than a copy of the referenced object. This reduces the memory overhead of the tag and allows us to reference the global variable as if it were just a local variable.

```
local: 'storage' = @(global: 'ex_Cache_Storage');
```

The next line searches for an element in the cache storage map that has the same name as the -Name parameter. This line uses the reference to the global cache that we have stored in the local 'storage' variable. Rather than copying the element out of the global cache a reference to the named element is stored in a local variable named 'cache'.

```
local: 'cache' = @#storage->(find: #name);
```

The local variable #cache will be referenced throughout the tag, but will always refer to the element stored in the global map. For example, when the [Run_Children] tag is used to have Lasso execute the contents of the container tag, the result is stored in the global map directly by inserting it into the #cache variable.

```
local: 'contents' = (run_children);
#cache->(insert: 'date' = (string:date));
#cache->(insert: 'contents' = @#contents);
```

The following line is not strictly necessary since the elements in the global cache map have already been changed, but it is included for the case where the -Name parameter could not be found in the cache and needs to be inserted for the first time.

```
    #storage->(insert: #name = #cache);
```

The [exCache] ... [/exCache] tags are very easy to use. Simply surround a portion of the page in them and specify how long you want the output to remain cached. Any database action performed within the tags will only be executed when needed to refresh the cached version.

For example, the following would refresh a stock quote every hour using the [XML_RPC] tag. The cache ensures that the stock quote tag isn't called over and over again unnecessarily.

```
[exCache: -name='stockdisplay', -expires=3600]
Apple's Stock Price: [Stock_Quote: 'AAPL'] as of [Date].
[/exCache]
```

Or, the following would perform a database search once a day to create the 10 flavors of the day for a soup restaurant. The database search need only be performed once since the soup selections are all set in advance.

```
[exCache: -name='soupsoftheday', -expires=3600 * 24]
    [Inline: -Search, -Database='Soups', -Table='Schedule',
    'Soup_Date'=date->(format: '%Q %T')]
        [Records]
            <br>[Loop_Count]: [Field: 'Soup_Name']
        [/Records]
    [/Inline]
[/exCache]
```

This tag works best when the contents of the tag takes a long time to generate but doesn't change frequently.

Data Types

This section briefly discusses some of the new features available to Data Types creators in Lasso Professional 6. These include operator overloading and the ability to create a tag that is called whenever any unknown tag is called.

Vectors and Symbol Overloading

The first example will be a Vector data type that allows mathematical operations to be applied to all of the elements of a vector simultaneously. Vectors will support operations against other vectors or against scalar values.

Note: this is not intended to be a full mathematical package for vectors, but is motivation for a discussion of the features of data types.

We define the vector data type as a sub-class of the built-in array data type. This allows us to use the array member tags 'for free'. We do replace the [Null->onConvert] callback tag so that our type is properly output as an exVector and not as an array.

```
[Define_type: 'exVector', 'array']
    ...
    [Define_Tag: 'onConvert', -Required='type']
        [local: 'result' = (exVector)]
        [loop: self->size]
            [#result->(insert: '(' + (string: self->(get: loop_count)) + ')' )]
        [/loop]
        [return: self->type + ': ' + #result->(join: ', ')]
    [/Define_Tag]
[/Define_Type]
```

A vector is initialized just like an array. [exVector: 1, 2, 3, 4, 5] is a vector with five integer elements.

The exVector type can now implement several custom tags which allow it to override built-in LP6 symbols such as >>, +, -, *, /, %, etc. This is done by creating custom tags with the name of the symbol inside the definition of the [exVector] data type. When that symbol is called the other parameter of the symbol will be passed as a parameter to the tag.

The following tag overrides the built-in addition + operator. It checks the data type of the parameter to the tag. If it is a vector then the elements from both vectors are added to each other. If it is a scaler then the parameter is added to each element in the vector.

```
[Define_Tag: '+', -Required='item']
    [local: 'result' = (exVector)]
    [if: (#item->type == self->type)]
        [loop: self->size]
            [#result->(insert: self->(get: loop_count) +
                #item->(get: loop_count))]
        [/loop]
    [else: ((array: 'string', 'integer', 'decimal') >> #item->type)]
        [loop: self->size]
            [#result->(insert: self->(get: loop_count) + #item)]
        [/loop]
    [/if]
    [return: #result]
[/Define_Tag]
```

This symbol can now be used with [exVector] objects. [(exVector: 1, 2, 3, 4, 5) + 5] would return [exVector: 6, 7, 8, 9, 10] and [(exVector: 1, 2, 3, 4, 5) + (exVector: 1, 2, 3, 4, 5)] would return [exVector: 2, 4, 6, 8, 10].

The += operator is similar, but needs to change the current vector in-place rather than returning any results. In order to preserve memory the += operator is used to modify the elements of the 'vector' array directly.

```
[Define_Tag: '+=', -Required='item']
    [if: (#item->type == self->type)]
        [loop: self->size]
            [self->(get: loop_count) += #item->(get: loop_count)]
        [/loop]
    [else: ((array: 'string', 'integer', 'decimal') >> #item->type)]
        [loop: self->size]
            [self->(get: loop_count) += #item]
        [/loop]
    [/if]
[/Define_Tag]
```

The rest of the mathematical symbols can be modified similarly so that the [exVector] type supports the full range of mathematical, comparison, and assignment symbols.

The [exVector] tag can now be used for mathematical manipulations as follows. The following code which performs mathematical operations on two vectors produces the results shown.

```
[var: 'myVector1' = (exVector: 1, 2, 3, 4, 5)]
[var: 'myVector2' = (exVector: 5, 4, 3, 2, 1)]
<br>Vector Alpha: [output: $myVector1]
<br>Vector Beta: [output: $myVector2]
<br>Alpha + Beta: [output: $myVector1 + $myVector2]
<br>Alpha - Beta: [output: $myVector1 - $myVector2]
<br>Alpha * Beta: [output: $myVector1 * $myVector2]
```

```
Vector Alpha: exVector: (1), (2), (3), (4), (5)
Vector Beta: exVector: (5), (4), (3), (2), (1)
Alpha + Beta: exVector: (6), (6), (6), (6), (6)
Alpha - Beta: exVector: (-4), (-2), (0), (2), (4)
Alpha * Beta: exVector: (5), (8), (9), (8), (5)
```

The following code which performs mathematical operations between a vector and a scalar value produces the results shown.

```
[var: 'myVector' = (exVector: 1, 2, 3, 4, 5)]
[var: 'myScalar' = 10]
<br>Vector: [output: $myVector]
<br>Scalar: [output: $myScalar]
<br>Alpha + Beta: [output: $myVector + $myScalar]
<br>Alpha - Beta: [output: $myVector - $myScalar]
<br>Alpha * Beta: [output: $myVector * $myScalar]

Vector: exVector: (1), (2), (3), (4), (5)
Scalar: 10
Vector + Scalar: exVector: (11), (12), (13), (14), (15)
Vector - Scalar: exVector: (-9), (-8), (-7), (-6), (-5)
Vector * Scalar: exVector: (10), (20), (30), (40), (50)
```

Colors and Unknown Tags

The next example will be an [exColor] data type that represents an HTML color. It will have a number of member tags which allow the hexadecimal values for various colors to be returned, e.g. [exColor->Red], [exColor->White], etc. It would be laborious to use [Define_Tag] to create each of these member tags, but we can use a shortcut.

The [Null->_UnknownTag] tag is called whenever an unknown member tag is called for a custom type. We can define this tag and return a proper color value if possible or a warning message otherwise.

First, we create a global color map that can be shared by all the pages on the server.

```
[if: !(Global_Defined: 'exColor_Colors')]
    [Global: 'exColor_Colors' = (Map:
        'AliceBlue'='#F0F8FF', 'AntiqueWhite'='#FAEBD7', 'Aqua'='#00FFFF', ...
    )]
[/If]
```

Then we define the unknown tag callback to check this map and return an appropriate value if the tag name can be found in the map.

```
[Define_type: 'exColor']
    [Define_Tag: '_unknowntag']
        [return: (global: 'excolor_colors')->(find: tag_name)]
    [/Define_Tag]
[/Define_Type]
```

Now, the [exColor] type can be used to return different colors. For example, a table with different colors in each cell could be returned with the following code.

```
<table border=1 cellspacing=3 cellpadding=3>
    <tr>
        <td><font color="[exColor->red]">red</font></td>
        <td><font color="[exColor->blue]">blue</font></td>
        <td><font color="[exColor->green]">green</font></td>
    </tr>
</table>
```

Debugging Techniques

We'll close out this paper with a brief discussion of some useful debugging techniques in Lasso Professional 6. These include new log tags, new map tags, and the handle tags.

Log Tags

The new log tags allow for error messages to be flagged with a level and routed to one or more destinations. This can be invaluable for seeing the information that you need to see when you are debugging. Properly setting up logging levels is important to end-users as well so they can throttle back the messages produced by your solutions if necessary.

The three log levels are:

• Critical - These are issues that need to be brought to the attention of the system administrator. Critical issues may affect the proper operation of Lasso Service and may require the system administrator to make some changes. By default logged to LassoErrors.txt, to the errors database, and to the console.

• Warning - These are issues that don't affect the proper operation of Lasso Service, but may affect a particular page execution. Informative messages about the state of the server are usually warnings. By default logged only to the console.

• Detail - These are detailed messages about the proper operation of the server. These include raw SQL statements. By default logged only to the console.

For example, the Lasso Connector for Lasso MySQL logs all SQL statements which are issued by the user at the detail level, logs any errors that occur while running a SQL statement at the warning level, and logs errors connecting with the MySQL service at the critical level.

Messages can be routed to three destinations:

• File - The LassoErrors.txt file in the same directory as Lasso Service. This can be opened in a text editor or 'tail' can be used to watch as messages are added to this file. This can be useful for monitoring Lasso Service without restarting into console mode.

• Console - The console of Lasso Service is visible when Lasso Service is launched as an application on Windows or using the consoleLassoService.command script on Mac OS X.

• Database - The errors database can be viewed in the Monitor section of Lasso Administration. Logging messages to the errors database can be useful since you can search the database and see if any new errors have occurred.

The global administrator can change the routing of error messages using [Log_SetDestination] or they can be adjusted in Lasso Administration. For example, the following three tags route all messages to their defaults. The changes take effect immediately allowing you to toggle on and off messages as needed.

```
[Auth_Admin]
[Log_SetDestination: Log_Level_Critical, Log_Destination_Database,
    Log_Destination_Console, Log_Destination_File]
[Log_SetDestination: Log_Level_Warning, Log_Destination_Console]
[Log_SetDestination: Log_Level_Detail, Log_Destination_Console]
```

Messages can be sent to any of the three levels using the [Log_Critical], [Log_Warning], and [Log_Detail] tags. These tags accept a message as a parameter. They are not container tags like [Log] ... [/Log].

One very useful technique is to turn off logging of detail messages to the Console, then log all your error messages as warnings. This creates a much less chatty display and allows your messages to be seen more easily.

```
[Log_SetDestination: Log_Level_Detail]
[Log_Warning: 'My Message']
```

Another technique is to log your error messages to the database, for example by routing warnings to the database, and flag all your warning messages with a unique identifier. Then search the errors database for your messages.

```
[Log_SetDestination: Log_Level_Warning, Log_Destination_Database]
[Log_Warning: '[example] My Message']
[Inline: -database='lasso_internal', -table='_errors', 'message'='[example]', -search]
...
[/inline]
```

Map Tags

A couple new map tags are very useful for debugging. For example, to see what variables have been defined on a page you can use the [Map->Keys] tag on the [Vars] map. [Vars->Keys] returns a list of all the variables that are defined on the current page that is a lot easier to read than the [Vars] output.

Similarly, [Globals->Keys] can be used to list all globals and [Locals->Keys] can be used to list all local variables. [Tags->Keys] can even be used to list all currently defined tags.

Handle Tags

The [Handle] ... [/Handle] tags can be very useful for detecting specific error situations and responding to them. A pair of [Handle] ... [/Handle] tags execute at the end of a page or container, after all the other code on the has executed. They perform the code contained inside only if a condition is met.

For example, the following code placed anywhere on a page (outside of any container tags) will execute when the page is finished executing and will output some information to the console.

```
[Handle]
    [Log_Detail: (Date) + ' ' + (Response_FilePath) + ' ' + (Error_CurrentError)]
[/Handle]
```

When combined with the [Protect] ... [/Protect] tags, handle tags can be used to recover from an error. For example, if a page is surrounded by [Protect] ... [/Protect] tags then any errors will be caught and a [Handle] ... [/Handle] block will execute immediately. The example below returns a custom error page to the visitor and send email to the site administrator letting them know the error occurred.

```
[Protect]
    ... page contents ...
    [Handle: (Error_CurrentError != Error_NoError)]
        [Email_Send: -host='mail.example.com',
            -to='administrator@example.com',
            -from='administrator@example.com',
            -subject='Site Error: ' + (date) + ' ' + (response_filepath) + ' ' + (error_currenterror),
            -body=(variable: '__http_response__')]
    [Redirect_URL: 'error.lasso']
    [/Handle]
[/Protect]
```

The Beginner's Guide To LP5

by Kirk Bowman

This session will explore how to begin developing Web sites with Lasso 5. It will present the Lasso architecture and methodology in terms a newbie can understand. Get started with Lasso from the ground up and avoid common pitfalls and mistakes discussed in this session.

Biography

Kirk has been developing database solutions since 1991. In 1997 he began developing Web solutions with a beta of FileMaker Pro 4.0 and CDML. Shortly thereafter he created his first Web solution with Lasso 1.0--a nationwide benefits site system for a Temple-Inland.

Kirk is now president of MightyData, LLC in Dallas, Texas offering technology design, development and training services. He has written several FileMaker and Lasso training courses as well as articles for ISO FileMaker Magazine and FileMaker Advisor. His latest Lasso projects include Benefits2002.com, a benefits enrollment site for Willis of Texas and SEIRS (Special Education and Related Services), Web-based special education administration software.

NOTE: This paper will be provided during the session.

Notes

Notes

Creating and Using Custom LDML Tags

by Bil Corry

Judging from the 28 custom LDML tags available at LassoScripts.com (five of which I wrote), it's clear one of the killer features of LP5 is going unused (or you're not sharing :-). After attending my presentation, you will know and understand the following:

- How to create custom LDML tags (it's sooooo easy!)
- Using custom LDML tags to reduce your coding by reusing code (save time)
- Using custom LDML tags to abstract complicated code into more understandable modular code (easier debugging)
- Using custom LDML tags for automation (less work)
- When NOT to use custom LDML tags (avoid mistakes)
- How to share your custom LDML tags with others (win big on karma)

I will be covering the basics of creating custom LDML tags for the beginners, illustrating a general framework of LDML tag creation for the intermediate users, and will delve into automating Lasso for the advanced users. So regardless if you are new or old to custom LDML tags, this presentation is for you.

Biography

Based in San Diego, California, Bil Corry is President and founder of Mindio Corporation, an organization dedicated to providing custom Lasso solutions and technologies. Bil has been using Lasso since version 2.0 and is a member of the Lasso Partner Alliance. Bil is also a partner in LassoWare, a publisher of quality Lasso developer solutions. Bil holds a degree in Computer Science with a minor in Women's Studies and was awarded a certificate of recognition from Eastman Kodak Company in March 2002 for his creation of the Kodak Wireless Reporting System using Lasso Professional 5.

Notes

Creating and Using Custom LDML Tags

LassoSummit 2002
Seattle, Washington

Presented by:
Bil Corry
President
Mindio Corporation

Introduction

Custom Lasso Dynamic Markup Language (LDML) Tags allow you, the developer, to extend the functionality of Lasso that is indistinguishable from the built-in functionality provided "out-of-the-box." Not only are you able to build your own custom LDML tags, but you can also integrate and use custom tags created by other developers.

This paper will cover the what, why, when, and how of custom LDML tags. What are custom LDML tags? Why would you use them? When should (or shouldn't) you use them? How do you create them and use them?

This paper is meant as an introduction to custom LDML tags and while it will touch on some advance topics, it is not meant as a complete guide to custom LDML tag creation. This paper does assume some general familiarity with Lasso, such as what is a "tag" and how do you use a "tag". Familiarity with arrays and maps is helpful when learning how to retrieve parameters passed to the LDML custom tag.

What are custom LDML tags?

Custom LDML tags are simply new tags defined and created in LDML. Once created, they can be used just like any built-in Lasso tag. A basic example of a custom tag looks like this:

```
[define_tag:'today']
      [return: date_getcurrentdate]
[/define_tag]
```

This custom tag `[today]` would return the current date and time. How you define custom tags will be covered later in this paper.

Lasso actually provides three methods of extending the functionality of Lasso: LCAPI tags, LJAPI tags and LDML tags. LCAPI and LJAPI are outside the scope of this paper; we will instead be focusing on the third method, custom LDML tags. For the remainder of the paper, "custom LDML tags" and "custom tags" will be used interchangeably.

As an aside, LCAPI allows you to create Lasso tags in the C programming language and LJAPI allows you to create Lasso tags in the Java programming language. While both allow advanced tag creation, only LJAPI tags are cross-platform compatible (custom LDML tags, the focus of this paper, are cross-platform compatible as well).

Why use custom LDML tags?

Custom tags have three big benefits: code abstraction, code re-use, and code automation. Each one will be discussed further:

Code abstraction is the ability to take a complicated set of code and condense it into a simple tag or group of tags. A good example of this is `[email_send]`. That one tag contains over 1100 lines of code (including comments) to

send email. So rather than you having to type 1100 lines of code each time you want to send an email, you use a tag to do it for you.

Code abstraction has an additional benefit in that there is less debugging. By using a single tag to execute a thousand lines of code, you now only have to debug the thousand lines of code once, and have it work everywhere the tag is used. If the same one thousand lines of code were contained in multiple Lasso pages, keeping all of them up-to-date would be a nightmare (we're avoiding the topic of includes for those who know what includes are).

This leads us into the next benefit, code re-use. Code re-use is a pretty simple idea, write a thousand-line custom tag once and re-use it infinitely. No more copying and pasting complicated code between Lasso pages, just create the custom tag and use it.

The last big benefit is code automation. Code automation is the ability to have Lasso perform tasks so that you don't have to. A good example of this is my custom tag `[session_form]`. `[session_form]` allows the developer to call a single tag and have it add the session information into every form on that page (rather than the developer having to manually add it).

So instead of having to do this:

```
<form action="myPage.lasso" method="post">
<input type="hidden" name="-Session"
       value="SessionName:[session_id:-name='SessionName']">
<input type="hidden" name="-Session"
       value="MyOtherSession:[session_id:-name='MyOtherSession']">
Username: <input type="text" name="username"><br>
Password: <input type="password" name="password"><br>
</form>
```

You can instead do this:

```
<form action="myPage.lasso" method="post">
Username: <input type="text" name="username"><br>
Password: <input type="password" name="password"><br>
</form>
[session_form]
```

So if the session name should change or the number of sessions changes, the custom tag will still work correctly and requires no additional debugging. The manual method requires the developer remember the exact format for adding session fields to a form; a typo or change in the session name will cause the session propagation to fail.

When should you use custom LDML tags?

There is not an exact science for knowing when to use or not use a custom tag. Each situation is different and requires you weigh the benefits of creating a custom tag. As you create and use custom tags, you will being to develop a sense of what code makes an excellent tag and what code does not. Below are a few things to consider when deciding on creating a custom tag.

Scope of the custom tag: Custom tags should be created with the thought of breaking the code into logical pieces. You will want to avoid creating a tag that does too much or too little. For example, you can have a single custom tag that returns an entire page, such as this:

```
[myPage]
```

Most likely, a tag such as the above would be too large in scope to be useful. Remember, you want to be able to re-use it on other pages.

Likewise, you can have a custom tag with a scope too small to be useful. For example, say you create four custom tags to return H, T, M, L respectively. Then you could use them like this:

```
<[h][t][m][l]>
```

It's not helpful to have a tag that simply returns a H, T, M or L. The scope is way to small and requires more work and Lasso processing power than the benefit justifies.

Redundancy of the custom tag: Custom tags should reduce your coding, not be redundant. It doesn't make sense to have a custom tag that does the same thing as how the tag accomplishes its task. For example, the example I started the paper out with looked like this:

```
[define_tag:'today']
      [return: date_getcurrentdate]
[/define_tag]
```

Since `[today]` is simply the same thing as `[date_getcurrentdate]`, it does not make sense as a custom tag. However, if `[today]` were defined to return today's date and time in an alternative format to [date_getcurrentdate], then it would make more sense.

Portability of the custom tag: Creating a solution with custom tags means that you have to provide not only the solution, but the custom tags as well. In some shared settings, such as a hosted solution, you may not be able to define global tags. Instead, you will need to define each custom tag on every page that it is used. While this is not a problem, it does require you understand where and how your solution will be hosted.

How do you use custom LDML tags?

In looking at how you use custom tags, there are actually two considerations: where do you define them so that you can use them and how do you call them on a page once you've defined them.

Custom tags can be defined in one of two places, either they can be defined on the page that uses them or they can be defined globally (but not both since custom tags can not be defined twice on the same page). To define them on the same page that calls them, simply place the `[define_tag]` code above where you are using it. Generally, you will want the custom tag defined at the very top of the page; however, it is not a requirement and the tag may be actually defined anywhere on the page so long as it is above where it is called.

To define a custom tag globally, place the custom tag in its own .Lasso file (I use the naming convention TagName.Lasso). Then place the .Lasso file in the LassoStartup folder. The exact location of this directory differs by platform and installation choice, so consult your Lasso Setup Guide for the exact location or do a file search. Once your custom tag .Lasso file is located inside of LassoStartup, simply restart LassoService (or reboot if you're unfamiliar with how to restart LassoService) and your custom tag is now available globally, just like the built-in tags. Be sure to debug your custom tag thoroughly before adding it to LassoStartup; once placed in LassoStartup it becomes hard to debug and requires restarting LassoService for every change you make.

Using custom tags are identical to the built-in tags; simply call the custom tag by name. They are indistinguishable from built-in tags in how they are used.

How do you create custom LDML tags?

We've now covered everything except how to actually code a custom tag! Let's start by using the absolute minimum required to create a custom tag:

```
1       [define_tag:'myAlert']
2       [/define_tag]
```

On line 1, we tell Lasso we want to create a custom tag called 'myAlert'. This will be the name you use when calling it from your .Lasso page:

```
        [myAlert]
```

Line 2 is simply letting Lasso know it has reached the end of the custom tag definition. `[myAlert]` as defined above doesn't actually do anything if called, but Lasso will recognize it as a valid tag.

(A small aside, when coding in Lasso, line numbers are not used. I've provided them in my coding examples only for easy identification of particular code segments.)

Custom tags can be named anything consisting of letters, numbers and underscores. Custom tags should not begin with an underscore since Blue World reserves this notation to denote internal tags used by Lasso. When deciding how to name your custom tag, Blue World recommends that "in order to prevent confusion between custom tags created by different developers, all custom tags should start with an identifier for the author of the custom tag, followed by an underscore, then the name of the custom tag." (SOURCE: Extending Lasso Guide, Blue World ©2002)

So for our custom tag above to conform to Blue World's recommendation, it should be `[bc_myAlert]` ("bc" for Bil Corry). This will also allow other developers to easily see which tags in your code are custom tags, this is especially helpful when working with a team of developers or distributing your custom tag to the Lasso community.

As defined above, `[myAlert]` doesn't do anything. To be useful it has to do something, so let's have it email an alert each time it's called:

```
1       [define_tag:'myAlert']
2               [email_send:
3                       -host='smtp.mindio.com',
4                       -to='bil@mindio.com',
5                       -from='lassoServer@mindio.com',
6                       -subject='myALERT!',
7                       -body='myAlert was triggered!']
8       [/define_tag]
```

Again, line 1 tells Lasso that you are creating a custom tag called 'myAlert'. Lines 2 to 7 are simply just the `[email_send]` tag that sends an email alert. Line 8 tells Lasso it has reached the end of the custom tag definition. So each time I want to receive an alert from Lasso, instead of having to do this:

```
        [email_send:
               -host='smtp.mindio.com',
               -to='bil@mindio.com',
               -from='lassoServer@mindio.com',
               -subject='myALERT!',
               -body='myAlert was triggered!']
```

I now can do this:

```
[myAlert]
```

What would make this custom tag even more useful is if you could have it email what the alert actually was. To do that, we will need to pass the alert message to the custom tag. There two ways to pass information into a custom tag, using variables and using parameters. Let's look at how both are accomplished.

Passing information into a custom tag using a variable (or variables) is probably the easiest way for someone just starting out creating custom tags or someone unfamiliar with arrays and maps. The tradeoff is calling the tag requires not only the tag be called, but the variables also need to be declared and a value set. Generally, I don't recommend this method as it allows for more error when coding and will confuse other developers when reviewing your code. Should you choose this method, I'd recommend naming the variables 'tagName_varName' to make it explicit that it's used for passing information into a custom tag. So going back to our `[myAlert]` custom tag, we can pass our alert message using a variable by defining the tag like this:

```
1      [define_tag:'myAlert']
2            [email_send:
3                  -host='smtp.mindio.com',
4                  -to='bil@mindio.com',
5                  -from='lassoServer@mindio.com',
6                  -subject='myALERT!',
7                  -body='myAlert was triggered!  The alert was ' +
8                  $myAlert_alertMessage]
9      [/define_tag]
```

Then to use it, we do this:

```
[var:'myAlert_alertMessage' = 'Jim is not wearing the same shirt!']
[myAlert]
```

This will send an email with the body of the email reading "myAlert was triggered! Jim is not wearing the same shirt!" Note that if the variable 'myAlert_alertMessage' isn't defined when calling `[myAlert]`, Lasso will throw an error just like any Lasso page. I will cover checking that the correct parameters were passed a little later in the paper.

Let's create the same tag, but this time using explicitly passed parameters. Before we do, let's go over your options when passing parameters.

There are two types of parameters that can be passed, named parameters and unnamed parameters. You can pass as many of each as your custom tag requires. Named parameters are parameters that have a name and are assigned some value. They are in the form of:

```
[someTag: paramName=paramValue, paramName=paramValue, etc.]
```

Unnamed parameters are simply values passed to the custom tag and are in the form of:

```
[someTag: paramValue, paramValue, etc.]
```

When the custom tag can expect either a named parameter or a unnamed parameter value, we calls those parameters "keyword parameters." Keyword parameters are simply parameters that you know the name of ahead of time. Standard convention requires all keyword parameters to begin with a hyphen. Non-keyword parameters are parameters that you won't know the name of ahead of time, such as passing the name and value of a database field or just simply passing a value.

It's important to understand these distinctions, so here are a few examples.

Here's an example of a keyword named parameter (-victim) with a keyword unnamed parameter (-copy):

```
[brainDownload: -victim='Jessup', -copy]
```

Here's an example of a non-keyword named parameter (victim) with a non-keyword unnamed parameter (copy):

```
[brainDownload: victim='Jessup', copy]
```

Not a lot of difference, right?!?! It is mostly a semantic issue. If the tag expects a parameter called "victim", then we would put a hyphen in front of it and make it "-victim" (it's a keyword). If the tag can optionally accept a parameter called "victim", the name is still known ahead of time so again, we put a hyphen in front of it and make it "-victim" (again, a keyword).

If the tag accepts the name of a database field and its value, but we don't know which database field, then the name is not known to us ahead of time so it is not a keyword. We would not place hyphen in front of it since we don't even know what it's called!

Parameters that are passed into a custom tag are accessed via the Lasso tag `[params]`. `[params]` returns an array that contains, in order, the parameters that were passed to the custom tag. Keyword named parameters may be accessed individually via the Lasso tag `[named_param]` or can be found by looping through `[params]` or by their position in the parameters. All other parameters can only be accessed by looping through `[params]` or by their position in the parameters.

Let's look at an example of passing all types of parameters and then learn how to retrieve them inside of `[define_tag]`:

```
[brainDownload: -victim='Jessup', favorite_food='sushi', 1, -copy]
[brainDownload: -victim='Willits', favorite_activity='auto racing', 2, -copy]
```

So in the above two examples, "-victim" is a keyword named parameter, 1 and 2 are unnamed parameters, "favorite_food" and "favorite_activity" are non-keyword named parameters, and –copy is a keyword unnamed parameter. If you didn't quite get that, don't sweat it; read the rest of this paper and come back later. It'll probably make more sense.

Here's how `[brainDownload]` would retrieve the parameters:

```
1       [define_tag:'brainDownload']
2
3               <!-- first get the keyword named parameter –victim -->
4               [local:'victim' = named_param:'-victim' ]
5
6               <!-- next get the non-keyword named parameters -->
7               [local:'non_keywords'=(array)]
8               [loop:params->size]
9                       <!-- is it a pair and doesn't being with '-' -->
10                      [if: params->get:loop_count->type == 'pair' &&
11                      !(params->get:loop_count->first->beginswith:'-')]
12                              [#non_keywords->insert: (params->get:loop_count->first =
13                              params->get:loop_count->second) ]
14                      [/if]
15              [/loop]
16
```

```
17          <!-- last, get the unnamed parameters -->
18          [local:'unnamed'=(array)]
19          [local:'unnamed_keyword'=(array)]
20          [loop:params->size]
21                <!-- unnamed params will not be a pair -->
22                [if: params->get:loop_count->type != 'pair']
23                      <!-- check if keyword -->
24                      [if: params->get:loop_count->type == 'string']
25                            [if: params->get:loop_count->beginswith:'-']
26                                  [#unnamed_keyword->
27                                  insert: (params->get:loop_count)]
28                            [else]
29                                  [#unnamed->insert:(params->get:loop_count)]
30                            [/if]
31                      [else]
32                            [#unnamed->insert:(params->get:loop_count)]
33                      [/if]
34                [/if]
35          [/loop]
36
37          ... do stuff with the parameters ...
38
39    [/define_tag]
```

So #victim would contain a string of the name of the victim, #non_keywords would contain an array of all non-keyword named parameters and their values, #unnamed would contain an array of all unnamed non-keyword parameters and #unnamed_keyword would contain an array of all unnamed keyword parameters. Notice that keyword named parameters only requires a single line (line 4) while the other methods require more work? Naturally, you can loop through the `[params]` only once and pick out the parameters as they come up; I've segregated the parameter types to make it easier to see how each type of parameter is retrieved.

The above example assumes that the parameters can be passed in any order, which is the safest way to code your custom tags. However, you can trade safety for ease of coding by instead requiring that the parameters be passed in a certain order. In that case, the tag could be written as:

```
1     [define_tag:'brainDownload']
2           [local:'victim' = params->get:1 ]
3           [local: params->get:2->first = params->get:2->second ]
4           [local:'id' = params->get:3 ]
5           [local:'action'= params->get:4 ]
6           ... do stuff with the parameters ...
7     [/define_tag]
```

As you can see, it's much simpler to code, but less flexible for the developer to use.

So let's go back to our `[myAlert]` example, we wanted to pass the alert as a parameter. So the tag will be called like this:

```
      [myAlert: 'Jim is not wearing the same shirt!']
```

So we modify our custom tag to be the following:

```
1     [define_tag:'myAlert']
2           [email_send:
3                 -host='smtp.mindio.com',
4                 -to='bil@mindio.com',
5                 -from='lassoServer@mindio.com',
```

```
6                   -subject='myALERT!',
7                   -body='myAlert was triggered!  The alert was ' +
8                   (params->get:1)]
9       [/define_tag]
```

That covers passing parameters in, but what about passing information back out?

Information can be returned back to the calling page in one of four ways; note that only `[return]` and using variables is formally documented by Blue World. The other two methods, -autooutput and $__html_reply__ are undocumented and are subject to change. I cover –autooutput below, $__html_reply__ is an advanced method and isn't covered by this paper.

`[return]` is entirely straight-forward. Simply pass the information you want to return to the calling page using `[return: passThisBack]`. `[return]` is limited to returning only one item, so if you need to return more than one item, you can return an array or map that instead contains multiple items. Here are a few examples:

[return: 15]	->	returns the integer 15
[return: 'doghouse']	->	returns the string 'doghouse'
[return: (array: 15,'doghouse')]	->	returns an array containing 15 and 'doghouse'

Let's take a look at a simple example of how it's used. Say we have a custom tag called 'html' that simply returns 'html':

```
[define_tag:'html']
       [return:'html']
[/define_tag]
```

Then we could use it like this:

```
<[html]> ... web page ... </[html]>
```

```
Which when processed by Lasso returns this:
```

```
<html> ... web page ... </html>
```

But say you have an even better idea, to have a custom tag for the opening and closing <HTML> statements:

```
[define_tag:'html_open']
       [return:'<html>']
[/define_tag]

[define_tag:'html_close']
       [return:'</html>']
[/define_tag]
```

Then we could use it like this:

```
[html_open] ... web page ... [html_close]
```

But wait, Lasso returns this:

```
&#60;html&#62; ... web page ... &#60;/html&#62;
```

That's because Lasso is HTML encoding the response from the tag. You can instead do this:

```
[html_open:-encodenone] ... web page ... [html_close:-encodenone]
```

But it's a pain each time you want to return HTML elements to have to specify –encodenone. And if you forget, you'll get encoded HTML elements. That's where the undocumented "-autooutput" comes in handy. It'll return everything contained in the custom tag as if it was on the page itself. Note that you don't use `[return]` when using –autooutput, you just format the inside of the tag with how you want the data returned to the calling page.

So our HTML tags rewritten using –autooutput would look like:

```
[define_tag:'html_open',-autooutput]
      <html>
[/define_tag]

[define_tag:'html_close',-autooutput]
      </html>
[/define_tag]
```

Then we could use it like this:

```
[html_open] ... web page ... [html_close]
```

And Lasso will return this:

```
<html> ... web page ... </html>
```

Another way we could have done it is by passing back the data in a variable:

```
[define_tag:'html_open']
      [var:'html'='<html>']
[/define_tag]

[define_tag:'html_close']
      [var:'html'='</html>']
[/define_tag]
```

Then we could use it like this:

```
[html_open][$html] ... web page ... [html_close][$html]
```

And Lasso will return this:

```
<html> ... web page ... </html>
```

The nice thing about passing data back as a variable is that you can return more than one value easily. Just create a variable for each value you want to return. So if you have a function that splits a full name into a first name and last name, you could do something like this:

```
[define_tag:'split_name']
      [var:'first_name'=params->get:1->split:' '->get:1]
      [var:'last_name'=params->get:1->split:' '->get:2]
[/define_tag]
```

Then we use it like this:

```
[split_name:'Bil Corry']
Name = [$first_name] [$last_name]
```

Which returns:

```
Name = Bil Corry
```

An important note about variables inside of custom tags: if you're NOT intending to return the value in a variable to the calling page, then you should use locals instead variables inside of your custom tag. Locals are identical to variables except they are only available to the custom tag and not to the calling page.

Here's a simple example to illustrate:

```
[define_tag:'test']
       [local:'theLocal'=1]
       [var:'theVar'=2]
[/define_tag]
```

Then when calling the tag:

```
[test]
theVar=[$theVar]
theLocal=[#theLocal]
```

It returns:

theVar=2
theLocal=Error: The local variable "theLocal" has not been declared

That's it! You now have the basics to create your own powerful dynamic custom tags. Happy coding!

Summary

Custom LDML tags allow any Lasso developer to extend the functionality of Lasso using only LDML. This means any developer that can code in LDML can create powerful extensions to the Lasso language that are cross-platform compatible.

In this paper, I've outlined what custom LDML tags are, why you would use them, when you should (or shouldn't) use them, how you create them and how you use them. This covers the foundation of extending the Lasso language. I would encourage you to further pursue custom tag creation with LCAPI and LJAPI for even more advanced custom tags. For more information, Blue World has written an entire manual on extending Lasso; it is called "Extending Lasso Guide" and the electronic version is provided with each Lasso purchase. A hardcopy of this guide is available for purchase separately.

As you create custom LDML tags, I hope you will join me in providing your custom tags to the Lasso community. A directory of Lasso solution websites is available here:

```
http://www.blueworld.com/blueworld/products/lassosolutions.html
```

By contributing to the Lasso community, we all benefit. I look forward to seeing your solutions.

About the Author

Based in San Diego, California, Bil Corry is President and founder of Mindio Corporation, an organization dedicated to providing custom Lasso solutions and technologies. Bil has been using Lasso since version 2.0 and is a member of the Lasso Partner Alliance. Bil is also a partner in LassoWare, a publisher of quality Lasso developer solutions. Bil holds a degree in Computer Science with a minor in Women's Studies and was awarded a certificate of recognition from Eastman Kodak Company in March 2002 for his creation of the Kodak Wireless Reporting System using Lasso Professional 5.

Contact

Bil Corry, President
bil@mindio.com
Mindio Corporation
1523 Stone Edge Circle
El Cajon, CA 92021-2981
Ph: 760-402-1830

Copyright

(Revised September 3, 2002)

Winning The Project

by Duncan Cameron

Ever lost a project but can't understand why? You know your solution is the best for the need but the client just didn't get it? Duncan will lead you through a simple process to assist you in ensuring that your next project is a winner. Covering topics such as

1. Understanding the need.
2. Careful planning.
3. Explaining the solution.
4. Adding the personal touch.

Biography

Based in San Diego, California, Bil Corry is President and founder of Duncan has been specializing in building database driven applications since 1986. Getting involved with Lasso in early 1997, Duncan is the founder of LassoDevelopment.com and the UK Lasso Association as well as the author of 'Lasso Professional 5 Developer's Guide' and co-author of the Lasso Professional 5 vs PHP white paper.

NOTE: This paper will be provided during the session.

Lasso Summit

2002 Developer's Conference

Conference CD

consulting
FileMaker & Lasso
training

Design and development services:

- Database Design
- Web Site Design
- Project Management
- FileMaker Development
- Lasso Development
- SQL Database Development
- Palm Handheld Integration

Public and private training classes:

- Introduction to FileMaker
- Beyond the Basics
- Philosophy of Scripting I
- Philosophy of Scripting II
- Lasso 5/6 Crash Course

Hands-On Training Classes!

Check out our FileMaker/Lasso success stories!
Willis of Texas - http://www.filemaker.com/customers/450.html
Clovis Schools - http://www.filemaker.com/customers/1005.html

"This system works like a dream. I'd say it's cut our time in half. What's more, we were able to eliminate a staff position and reduce our overhead." *— Leiann Staggs, Sunbelt Power Controls*

"What was once a horror for me is now a simple matter with the FileMaker-based system that MightyData shaped for us. It has dramatically improved the quality of service and communication we can provide students, and at a reduced cost to our taxpayers." *— Debbie Marlowe, Independence Public Schools*

"Kirk is awesome. Now I'm ready to make a killer FileMaker solution!" *— Barry Evleth, Ingram Micro*

"You exceeded my expectations in both quality and content!" *— Melyssa St. Michael, UltraFit*

"These guys know their stuff, but more importantly, they know how to teach." *— Bruce Newlin, AFM-Dawn Computers*

"Kirk Bowman is gifted with the art of instructing and teaching." *— Kari Rhame, Clear Creek ISD*

MIGHTYDATA™

Database Building & Training

Call 800-287-0845

or see our web site: www.mightydata.com

info@mightydata.com • Dallas, Texas