Introduction

Congratulations on your purchase of Goodwin ePages, an exciting new Domino development product developed by Domino developers, for Domino developers. Goodwin ePages has been designed to give you the same level of control as the most popular web development tools on the market today with simplicity, flexibility and above all else, reusability.

You will be able to:

- Read entire web forms directly into LotusScript from the browser in a single command
- Read complex URL variable structures directly into LotusScript in a single command,
- Process Form & URL variable data without parsing for HTML delimiters and modifiers.
- Deal with field values from forms in a single for/next loop/Select case statement.
- Launch MS Word or MS Excel on a client machine where it's installed without files or attachments of any kind and populate them directly.
- Broker internet access to sensitive data where the database grants no access to internet users.
- Bypass the need for Form, View, Outline, Page or other web specific design elements as you see fit.
- Give ASP, JSP, XML & J2EE a run for it's money.

Goodwin ePages will help you:

- Improve site performance dramatically.
- Build more complex transactional applications without the need for other tools and without the need for exotic workarounds.
- Build applications and get content on the browser faster.
- Write code once and reuse it for hundreds of applications.
- Write broker apps for legacy databases without using Domino data stores.
- Build far more robust, interactive and dynamic systems.

Is Goodwin ePages for you?

- If you're interested but not sure what all this stuff is, we're going to explain it.
- If you're interested and already knowledgeable in these areas, read on to get the new techniques needed to be successful and fill in any gaps.
- If these things don't interest you then Goodwin ePages isn't for you.

What kinds of Web applications have been built purely with Goodwin ePages?

Survey Toolkit

This started out as a tool to assist the CIO of a large corporation survey his IT staff. Developed with requirements from KPMG, this tool was so successful it was expanded to allow other department heads to create and administer their own surveys from their web browsers. Some of the browser based features included:

- Administration/creation & management of multiple surveys simultaneously
- User designed forms. (NO design access or developer assistance required)
- Scatter saved responses. 30 question questionnaires needed to be saved as 30 documents to protect the anonymity of respondents.
- Mass emailing with custom "Ticket" links to stop users from entering multiple survey responses which would alter the survey metrics. No log-in required.

Custom Reporting System (10 days from concept to production)

Users in a large financial company were in constant need of new views to gather metrics for reports. The database was extremely large and with 120+ views currently in place, performance would suffer. The Custom Reporting System resolved this problem entirely. It had the following features:

- Could be pointed at any Domino database on any server.
- Administrator could make any field available for search selection.
- Administrator could give any field any other name for user interface selection list.
- User friendly design with only 3 screens in total.
- Users could select any field or fields to report on and decide on their order of importance.
- On the next screen, users would enter search criteria for none or all of their selected fields. Only the fields they selected would be displayed for search criteria.
- The last screen was their custom generated report; sorted, categorized, totaled and subtotaled. Data displayed on the report was restricted to what the user selected.

Briefing Notes Workflow system (15 days from concept to production)

A government agency needed a means of streamlining approvals for high value requests. The system had a number of interesting features:

- User configurable Workflow routing.
- User configurable collaboration access rights.
- Automated mail routing with custom intranet links
- Imbedded signature images on approval
- Would launch the document directly into MS Word as desired (No attachments or files)
- Real time status screen to track approval state of all documents or specific owners documents.
- Full security features.

Installation

The Goodwin ePages system consists of a single database and the documentation.
Goodwin ePages must be installed on a Domino server (version 4.5 or higher) and will require an active HTTP task running on the server to operate.

To install the database do the following:

1. Copy the Goodwin ePages database to your local Notes Data Directory.
2. Set Access control levels as appropriate.
    a. Developers will need Designer access.
    b. Administrators will want Manager access.
    c. The host Domino server will require designer access.
    d. We suggest a minimum of reader access for Anonymous users
3. Copy the database to the desired host server.

Registering your copy

Before Goodwin ePages will work, you will need to register your copy of Goodwin ePages and acquire a key specifically generated for your copy.

1. Make note of your server name where Goodwin ePages is installed:
    a. You'll need the fully abbreviated name (i.e. server/org/org/domain).
2. You will need the database replica ID. The replica ID may be found on the database properties in the information tab or in the database synopsis. If you generate a database only synopsis, you'll be able to copy the replica ID from the report generated by the synopsis.
3. Connect to the Goodwin ePages website and click the "Register" button. You'll be presented with 3 screens. The license agreement, the information collection screen and the final key generation screen which provides you with your key. For your convenience, the key generation screen has been designed for easy copying of the key data.
    a. You must agree to the license agreement in order to receive a key
    b. You must complete the fields provided to receive a key
4. You will receive a Key and an activation level. Go to the Activation view of the database and enter both pieces.
    a. Enter both key elements exactly as shown on the site or the key may fail.

You're ready to start using Goodwin ePages.

Concepts

Firstly, it's important to understand some of the concepts behind Goodwin ePages, so lets start there. Conventional Domino design can be broken down into two methodologies. The form and view approach which entails creating forms and views and intermixing them with other Domino design elements to create a web site. One of the big advantages in this approach is that Domino will generate a considerable amount of the HTML and JavaScript needed to provide the finished result. This approach is ideal for new developers working on fairly simple web applications but holds significant downsides for seasoned professionals trying to build complex systems. The more a system does for the developer, the more that system must assume on the developers behalf. The eventual outcome of these system assumptions is that fine control of even the simplest things are lost or made difficult with work-arounds and exotic system configurations. Aside from the initial impact to development time, effort and compromise, long term maintainability and scalability of the tool is affected. Finally, reusability is impacted by the task/presentation oriented nature of the designs.

The second common approach to Domino development is with the use of coded elements through LotusScript. Although commonly used for processing web interactions in conjunction with forms or occasionally for web output, the complete web interaction picture has seemed impossible. Let's use an example here. I have a user who wants to publish a questionnaire weekly. The number and type of questions could vary from 5 to some unknown number and they're not interested in meeting with developer types every week for adjustments. We know we can satisfy the number of questions by storing each question as a single document, wrapping them in HTML using LotusScript and then pumping those questions out to the browser. That approach would allow hundreds of questions. The problem is, how do we read the answers back? This problem crops up again and again. The minute you depart from a static form design, you depart from your ability to retrieve visitor responses.

For web developers who use LotusScript, this can be especially frustrating and affects site performance dramatically. If I've presented the visitor with a page where a field will carry information that I need for the next page, I have to open a view, locate their response and retrieve their response before I can respond back.

The other scenario that offers some pain is access levels locally or across multiple databases. DomReg is a fine example of this. You want to register the user but you don't want to give them access to the NAB. This means they need to register elsewhere and a back end agent which will run on an unpredictable pattern will be needed to add them to the NAB and groups. Add to this the idea that we now have to maintain a NAB and a DomReg database so the user can come back and update their information as needed. All this because we have no means of brokering the transaction. Wouldn't it be easier if you could just read the data directly from the browser and act on it without a form? It might surprise you to learn that the default ACL setting for all HTML files on the Goodwin ePages web site is NONE.

This is where Goodwin ePages fits in. Goodwin ePages, at its core, is a single class focused entirely on simplifying web interactions both inbound and outbound. On the inbound side, Goodwin ePages provides a single point of entry to read both URLs and HTML form data directly into the LotusScript environment for processing. All house keeping chores such as the conversion of hexadecimal values, interpretation and organization of variable/value pairings for easy interrogation and processing, are all taken care of by the Goodwin ePages web class.

For outbound web interactions, Goodwin ePages not only helps to prepare raw data for implementation in HTML, but it also addresses things like the 32K limit on string variables (which is supposed to have been 2 gigs), effectively eliminating them from the web development picture. The Goodwin ePages approach offers so much control that the Goodwin ePages class has implemented the ability to launch MS office products on the visitor's workstation without files or attachments but rather, directly from generated content.

Generally, the implementation of Goodwin ePages not only provides clean, easy to read edit areas for your HTML, JavaScript, XML and CSS data but makes it far easier to share all of these elements, including HTML pages, across multiple web pages with single design elements or even multiple sites with a single code element. Copy/pasting these elements extends component reusability even further. (ie. One HTML document and agent used for multiple pages in a site and/or across multiple sites.)
The approach itself coupled with the simplicity of use is really where reusability, maintainability and scalability benefit the most. Goodwin ePages frees you from the relatively static nature of forms and views, thus opening a whole new world of fully dynamic web development.

The Database design

Goodwin ePages is a Domino web development tool. It is intended that you will do your web development directly in a copy of the Goodwin ePages database. As such, the database design has been kept simple and to the absolute minimum required to support it's functionality and the examples provided.

Must have design elements:

- In the script libraries you'll find the "NewWebClass" library. All of the Goodwin ePages capabilities reside here. This library must be present. The source code for this library is not available. Any attempt to edit this library or copy it to another database will result in complete loss of the library.

Good to have design elements:

- The Activation view is merely there to hold the activation button. This is the button you use to access the database profile containing the activation key. In the event that a problem should occur with your key, you may need access to this button to make corrections or adjustments. Knowing that you'll be making

extensive use of agents, we opted to avoid make the activation code an agent so as not to clutter your list of agents. Navigators would also be somewhat of an annoyance if you begin adding views to the database for your needs so we avoided those as well. The view is simply a container for the activation button and the code, basic database profile code, is easily accessible.

- The Activation form is a basic form used to display your key information.
- The Activation Button. A standard "Edit database Profile" button.

Extras you might find useful:
Technically speaking, the remaining elements are only necessary to the examples. You don't need them for Goodwin ePages to work; however, we've found these simple design elements to be very useful at keeping Page code (HTML, JavaScript, CSS, etc) out of our LotusScript code, thus improving code readability. They also provide a place where we can paste in page code generated by other web development tools and edit that code. Finally, they make reusable HTML, JavaScript, CSS, etc. possible.

- HTML view and associated form. The view is a basic look up view for LotusScript code to locate HTML documents. The HTML form has a "Title" field for the lookup name of the HTML document and a text field for the HTML content.
- Script view and associated form. Script is a generic name that covers any kind of page element that would be placed in the <HEAD> of the web page. The view is a basic lookup view for LotusScript code to locate Script documents. The script form has a "Title" field for the lookup name of the script document and a text field for the script content.
- Images view and associated form. The Images view is a basic lookup view for the images documents. It also displays image descriptions so you can more easily locate the image you need. The image form is meant to be used as a place to attach web page image files. The file can then be retrieved by the HTML on your web page.

A word about Domino agents

Goodwin ePages is a LotusScript development tool and approach. As such, much of the code work you will do will be in domino agents. To create a Domino agent suitable for Goodwin ePages interactions, ensure the following:

1. Ensure that the agent is set to shared.
2. Ensure the agent is set to run from Action menu or Agent list. Either should be fine.
3. Ensure that the agent is set to "Run Once" as opposed to acting on any search criteria. You can use search criteria if you need to but for basic page input/output, "Run Once" is what you need.
4. Ensure that you have 'Use "NewWebClass"' in agent Declarations event. Without this your code won't be able to access any of the NewWebClass features.
5. Ensure your agent is signed by someone with the minimum access required for the task.

By default, Domino agent security is only restricted by the access level of the user who has signed the agent. If the administrator has signed the agent, then the agent has access to anything the administrator does. If the server's ID is used to sign the agent, then the agent has server level access to resources. In instances where multi database operations or ODBC/ADO operations are being performed by the agent, this level of authority may be desirable, especially considering that data access security can be implemented within the agent itself. However, there are instances where having the agent run under the logged in user identity would be better or simply easier to implement. For this, the "Run agent as web user" check box can be used to force the agent to run with the access rights of the web user who is accessing the agent. It can be found in the agent properties. Be sure they have all the rights that the agent will need.

Broker Agents

Agent access rights over and above user access rights can be used to great effect. Let's say you have data you'd like to make available to web visitors but the data exists in a database that you don't want web visitors to have direct access to.
Web Agents can be signed with the access level of someone who has total access to the data. The agent can then be written to access the data on the web visitor's behalf in a "Broker" capacity. The secured database could maintain its "No Access" settings, thus keeping it very secure, while the agent could "Jump the security wall" to collect and report the data as needed. The agent could be coded to be highly specific with regard to what it will retrieve and report and can still check visitor log in information and if needed, compare that with NAB groups or ACL entries for it's own security screening.

Inbound browser communications

For the purpose of this documentation, inbound browser communication is considered to be any attempt by the browser to send information to the Goodwin ePages application. This includes URL data (Clicking a link with encoded data in it) or HTML form data (Someone completing a form). Goodwin ePages operates independently for all inbound communications it is asked to do. This means that any URL or HTML form may be submitted directly at your agent without any Domino forms, files or other resources to help interpret the incoming information. In fact, Goodwin ePages applications can be, and have been designed without any web forms, views or any other Domino design element of any kind (Custom reporting system). Inbound URL information and HTML form information can be read directly into the script code for analysis and processing as the developer sees fit. There are multiple training examples of this within the Goodwin ePages database but we'll cover off on the basics here.

Specialized URLs for the uninitiated

Before we get too deep into things, lets cover off on some URL communication techniques you may not be familiar with.

Standard URL: http://www.jgoodwin.com

URL with variables: http://www.jgoodwin.com&id=649E46&name=Joe+blow

We've all seen both of the URL examples listed above. The standard URL really requires no explanation, but the specialized URL may offer a bit of mystery so we want to make sure that mystery is cleared up.
In the second URL example shown above (URL with variables) is an URL with variables encoded in it. This is often done to communicate information from the browser back to the server code that is processing your browser interactions. These specialized URLs could exist in links, buttons or even in form action tags. What we want to clear up here is how the variables work.
The variable portion of the URL begins with an ampersand character ("&"). The first word after the ampersand is the variable name, in this case, "ID". This can be any word that you might choose to imagine. It's arbitrary and its name is completely up to you. The server won't analyze it at all nor does it even care. Identification and processing of this variable name is completely up to your code. LotusScript won't see it as a code variable though. It's just a name we use to locate the value.

The variable is followed by an equal sign. This is the way you separate the variable name from its associated value.

Finally, we have the value that belongs to the variable. In our example, "649E46"

This completes a single variable/value pair. If we have more pairs to communicate, we repeat the process beginning with another ampersand character.

URLs have limits and those limits can be tight. The browsers often have a limit on the length of the URL they will allow and in testing what we have done has shown limits in the length of the URL that Domino will accept into its CGI variables, despite a higher limit on the browser.
If you have extensive data to return from a web page that cannot be returned as a form, we suggest HTTP "POST" buttons. This is a technique for coding a PAGE with a button that has an entire FORM attached to the button itself.

Reading URLs & Forms

To code for incoming Browser communications, simply set up an agent as described above and code the following;

```
Dim WebPage As New Web
Dim Parsed As Variant

Parsed=WebPage.checkCGI("")
```

"Dim Webpage as New Web", creates an instance of the Web class. Our LotusScript agent named the class instance, "Webpage".

“Dim Parsed as Variant”, creates a variable called Parsed whose data type has not been defined yet.
Finally, “Parsed = WebPage.checkCGI(“”)”, reads the inbound browser data, parses it and prepares it for use, then organizes it into Variable/Value pairs and delivers it to the Parsed Variant which it now defines as a 2 dimension array of string elements.

If the URL data sent to the agent looked something like this:

http://www.jgoodwin.com&name=John+Goodwin&tool=Lotus%20Domino

The checkCGI method would populate Parsed with this array:

Element Access Code:	**Parsed(1,1)**	**Parsed(1,2)**
Variable name & value:	Name	John Goodwin
Variable name & value:	Tool	Lotus Domino

At this point I can write a simple for-next loop to cycle through the entire array. A select case statement inside the loop would identify variable names I’m looking for and actions to take on the values returned, as in this example:

```
For index = LBound(Parsed) to UBound(Parsed)
        Select Case UCase(Parsed(Index,1))
                Case “NAME”
                        Username = Parsed(Index,2)
                Case “TOOL”
                        Software = Parsed(Index,2)
        End Case
Next
```

The Goodwin ePages database contains fully tested and working agents that utilize this technique. The OpenPage agent is one such example that accepts a PAGE parameter to determine which page it should send from the database to the browser and an URL parameter which it uses to open a web page in a different application, server or from the web.

Or the data can be written directly to the database as a new document such as in this example;

```
Set Doc = New NotesDocument(db)
Doc.form = “MyForm”

For index = LBound(Parsed) to UBound(Parsed)
        Call Doc.AppendItemValue(Parsed(Index,1), Parsed(Index,2))
Next

Call Doc.Save(True, True)
```

As we've done above, a Select Case statement could be included in the for/next loop to interpret and process the incoming data before it's written to the document.

Important Note: Goodwin ePages is designed to deal in variable/value pairs as "&Variable=value". If you send single elements as in "&Element&another+element" the system can respond with unpredictable results including anything from distorted data to errors occurring within the web class itself.

Form data works in exactly the same manner. In fact, the code for collecting web browser data and saving received data to a new document could be used, "as is", for both HTML Form returns as well as URL return scenarios without any regard for which is actually occurring. The code for examining entries within the returned data could be used but would only have an impact if the returned data contained the identified variables in the URL or identically named fields in an HTML form. Such Select Case statements are best custom written for your particular need but it's worth noting that a single Select Case statement can be written to handle any number of situations without error, so you could write a single piece of code to deal with dozens or even hundreds of URLs and forms.

There is one specific area within the CheckCGI() method where care must be taken. This is in the use of quotes within return data. CheckCGI() has been specifically coded to recognize data in quotes primarily for use in identifying referral URLs within URL parameters. The use of single quote pairs poses no issue, but embedding quote pairs within quote pairs will result in distorted data returns since the system can no longer discern Referral URL information from normal URL information. Quote pairs are possible within quote pairs as long as the most outer quotes are double quotes and inner quote pairs are single quotes. Even with this, extra care must be taken in the encoding of the URL.

There is one area of Goodwin ePages still under development. This area is downloads. CheckCGI() can see the multipart data but at this point in time, cannot handle it. Thus, you'll want to perform downloads though the normal Domino forms design approach. This is the only known limit to Goodwin ePages web interactions.

Finally, CheckCGI() returns results in a 2 dimension array. Arrays have some special problems associated with checking the validity of an array. If any array commands, even those intended to check the array, are issued against an empty array variable, errors can result. In order to help you maintain consistent code that can handle multiple situations, we've implemented a simple mechanism to examine the returned array even if the URL or form returned had no data. If the URL or form sent from the browser had no data for CheckCGI() to prepare for you, CheckCGI() will return a 2 dimension array with a single element entry where the entry reads, "No Data".
To check for this, just check to see if element (1,1) contains "No Data". If so, skip all array processing on the returned array.

We'll go into detail about all of the methods & properties and their features later, but for the majority of your browser interactions, this is all you'll need.

Outbound Browser Communications

Outbound communications is considered to be any time your application sends data to the visitors web browser for display. Goodwin ePages has extensive coding to help in this area, making it easier to produce the results you're looking for.

Goodwin ePages breaks a web page down into two basic components, the <HEAD> area and the <BODY> area. The <HEAD> portion of the page is populated via the Head property and is limited to 32K of text, by the environment. The <BODY> portion of the page is populated by the StoreHTML() method. This method is employed to resolve size limitations of string variables in LotusScript, effectively putting the body text size limit well outside of most reasonable needs.

Previous to Goodwin ePages, code to generate HTML became extremely complex very quickly. Not only did you have to manage variables to hold the HTML for the web page, but you also had to manage code to collect data, analyze it and manage additional variables for processing collected data into the required HTML. The StoreHTML() property along with the Goodwin ePages removes all that. The web page class provided with Goodwin ePages represents a persistent web page under construction. No matter what code you're processing, your web page under construction in memory remains completely unaffected and managed, leaving you and your code to focus on the task at hand. StoreHTML() is your "Store and forget" method, allowing you to send the next piece of HTML data to the page and move on.

Store HTML() performs one other subtle task; the insertion of carriage returns for every line of HTML you store to the class. You can't really appreciate the value of this little thing until you've had to debug some HTML tables in the page source with the source as a single huge lump of text. With these carriage returns, you won't see a single huge mass of text but rather nicely terminated lines of HTML as with any normal page. This immensely improves readability, which speaks to maintainability and debugging.

PrintHTML() is the method used to send the collected HTML page data to the browser. Once you're satisfied that you've completed the encoding of your page, use the PrintHTML() method to send the page to the visitor. Optionally, PrintHTML() will include <HEAD> and <BODY> tags in the appropriate places in your page. This feature can be deactivated so that you can customize your own <BODY> and <HEAD> tags. More on that in the language reference later in the manual.

Finally, although you can send lines of HTML directly to the browser via the LotusScript Print statement, features such as launching pages directly into MS Excel or MS Word on the visitor's workstation cannot be performed unless PrintHTML() is used to output the page. Auto tagging of page elements and carriage return insertion is also performed by the PrintHTML() method.

HTML Form/Agent interactions

Although Goodwin ePages can be used quite effectively to simply display pages, the real power in this tool is in transactional interactions. Transactions can best be viewed as conversations in HTML.

- The visitor arrives at the site
  - This invokes an agent that prepares and sends them a page or HTML form.
- The visitor responds in some way to the page or form.
  - This invokes an agent that processes the response and sends another page or HTML form based on that specific response received.
- And so on until the interaction is complete.

There are two important elements that must be considered for this kind of interaction to be effective.

a. The <FORM> tag.
   The <FORM> tag on HTML forms provides the browser with two very important details. First, it provides the browser with the method by which the data in the form is to be returned. There are two methods, GET and POST. The GET method tells the browser to return the form responses in the URL. Form responses will be encoded into the URL and passed to the location identified. URLs are limited in size and thus not widely used for Form data returns. The other method is POST. The POST method encodes the form responses into a stream that is sent to the location specified. The stream doesn't share the same limits as the GET method URLs do and is more commonly used for <Forms>.

   Finally, the <Form> tag contains the location where the return data is to be sent, in the "Action" clause of the <Form> tag. This is an URL and like all URLs has the same features and follows the same rules as any URL. If a POST method is chosen, as the data return method, the Action clause URL can be encoded to provide additional supporting information to go along with the Form Post data being returned. Here is a basic template as the minimum requirements for making a <Form> tags for use by your code;

   <FORM Method="Post" Action="MyAgentName?openagent">
   or
   <FORM Method="Get" Action ="MyAgentName?openagent">

b. Field Names
   In order to return useful data from the HTML form, all fields must have a unique name. This is accomplished through the NAME clause of the <INPUT> tag. Fields on your HTML form that are missing their names will not be properly returned and may cause unpredictable results.

The type of data being returned in the form, along with the design or source of the form is of little consequence. Goodwin ePages focuses on the data itself, leaving you to decide the rest. You could choose to use a JSP server in another part of the company to create the HTML forms which you will process and respond to. You could then choose to create

a response form that, when completed, sends the flow of the transaction back to the JSP server, effectively playing “ping-pong” the with the transactions. It’s doubtful that you would ever do this but I wanted to illustrate the freedom and flexibility that this approach affords you.

A good guideline to remember when developing in Goodwin ePages agents is that each response should be processed and responded to in the same agent. You may be able to have a single agent that processes multiple forms and responds to each individually, all in the same agent, but at minimum, data should be received, processed and responded to in a single agent or code element. It’s also useful to remember that hidden fields on the HTML form and embedded variables in the Action clause tag can be used to help your code keep track of the transaction and decide how best to proceed.

Technique

At first glance, it may seem that your application would become riddled with agents but the Goodwin ePages approach provides for a much more dynamic interaction than might otherwise be possible. With Goodwin ePages, a single agent can be coded to process and respond to a multitude of interactions using and/or reusing a collection of HTML pages, forms and other web page components. This is one of the many techniques that will be detailed here in this section.

The Goodwin ePages database contains several example agents and their associated documents. You’ll want to refer to these examples throughout this section. They will help further illustrate the discussion.

Before we get into the discussion, its worth going over the formal URL you enter to call upon any agent in your application. Here is an URL template you can use:

http://Server_Address/Database_directory/Database_Name.nsf/Agent_Name?openagent

Lesson 1- URLExamples

The URLExamples agent has been included to demonstrate the code mechanisms involved in reading URLs from the browser and processing them. It has been built with the absolute minimum code required to perform the task to make understanding the approach that much easier. It has also been built to provide you with a means to experiment with various URL variable/value combinations and see instant results. Any URL sent to the agent from your browser will be immediately processed and data formatted and sent back to the browser for review.

The object here is to understand how Variable/value pairs can be encoded into the URLs of your website to not only assist in code processing but also to provide you with your first taste of transactional interaction approach; whereby the browser sends a request to the agent who processes the request and sends a response back.

The code in this example is heavily commented for your benefit.

Lesson 2 - ReadFormExample

The ReadFormExample agent has been included to demonstrate the code mechanisms involved in reading HTML form data from the browser and processing it. It has been built with the absolute minimum code required to perform the task to make understanding the approach that much easier. It has also been built to provide you with a means to experiment with various values in the form and see instant results. Any HTML form sent to the agent from your browser will be immediately processed and data formatted and sent back to the browser for review.

Lesson 2 has a great deal to teach us. First and foremost, the ReadFormExample agent does nothing if called by itself from the browser URL. This agent represents the kind of agent you might have for operations in the middle of your user/application transactions. In this kind of scenario, the HTML form generated by one agent is submitted to another agent for processing and response. In this case, we use the OpenPage agent to launch our form for us which, upon submission, will be processed by the ReadFormExample agent. Here's how the calling URL will look.

http://server/directory/database.nsf/openpage?openagent&page=form+1

The OpenPage agent demonstrates the reusability aspect of Goodwin ePages. It's a utility agent that can be used to open any normal Web page from the database, from another site through site URLs or launch any page it sends from within the database to MS Word or MS Excel. It doesn't attempt any processing on the pages it handles although it does process URLs, and is a perfect tool for those little jobs like Help pages, static site maps, contact us pages, home pages and error or acknowledgement pages.

You can examine the HTML encoded into FORM 1 by opening it from the HTML view of the database itself. Here you will clearly see how the <FORM> tag has been encoded to call on the ReadFormExample agent. If we needed to, we could have encoded additional variables and values into the <FORM> tag URL which is the topic of our next lesson.

Finally, if you were observant, you would have noticed that aside from some minor HTML output formatting and comments, the ReadFormExample agent is identical to the URLExamples agent, further demonstrating the reading HTML form data is identical to reading URL data.

Lesson 3 – ReadFormAndUrlExamples

The ReadFormAndUrlExamples agent has been included to demonstrate the code mechanisms involved in reading HTML form data and form URL data from the browser and processing it. It has been built with the absolute minimum code required to perform the task to make understanding the approach that much easier. It has also been built to

provide you with a means to experiment with various values in the form and see instant results. Any HTML form sent to the agent from your browser will be immediately processed and data formatted and sent back to the browser for review.

To see this example in action, use OpenPage with the page variable as earlier to open Form 2 (Coded as form+2 in the URL)

Lesson 3 is really just a culmination of lessons 1 & 2. It combines the two browser read mechanisms to demonstrate how to perform both operations on the same inbound HTML form. The CheckCGI() method tracks and makes use of the <Form> tag method used to send the data. POST data takes precedence over all other forms of inbound data and as a result, CheckCGI() can't process POST and GET data returns together even though it has full access to and can see both. Additionally, returning a mix of URL and Form data to the same array could complicate form data processing by adding the additional need to parse out URL return data from the array prior to processing the form. This is counter productive to the Goodwin ePages objectives.. To deal with this issue, Goodwin ePages has implemented the CheckURL() method.
The CheckURL() method takes no parameters. It simply collects the URL information previously sent and parses it for variable/value pairs. It then normalizes the parsed data and presents it back to your code in the form of an array.

This is the approach you would use to read an HTML form as well as the encoded URL data within the HTML form.

Lesson 4 - AdvancedPublishExample

The AdvancedPublishExample agent represents an approach that can be used to produce some of the most highly reusable systems possible. In this example, the forms from Lessons 1 and 2 can both be published through the AdvancedPublishExample agent and associated HTML. There are some interesting tricks involved but they're easy to do so follow along carefully.

In this example, we're not going to focus as much on the inbound or outbound methods so much as the page creation process. The inbound and outbound methods are the same as those already demonstrated but the page creation process is completely new.

When examining the code, the first thing we notice is that it looks for a "Mode" parameter. The values it's prepared to process are "FORM1" or FORM2" so the URL parameter would look something like this; Openagent&Mode=FORM1
In this case, you will want to call the AdvancedPublishExample agent directly with its Mode parameter.

The AdvancedPublishExample agent utilizes the HTML document "Form 3" which can be found in the HTML view of the database. The first thing you'll notice is missing HTML elements and in their place, strange $$Name$$ elements all over. These are custom interpretation tags and are meant to act as placeholders for spots within the

HTML where dynamically generated content will be placed. In this case, the <FORM> tag, Page heading and </FORM> tag are all being generated by the agent just prior to publication, based on the Mode requested.

Here's how it works. Within the agent are two extra functions not found in other agents, ParseHTML() and ReplaceToken(). Both accept the HTML document being examined and the Web classes object as parameters. Once the agent has located the HTML document it's to publish, it passes the document pointer along with the Web classes object along to the ParseHTML() function which then begins searching the HTML for "$$" character strings. Once located, the function sends the chunk of HTML it has read, to this point, to the Web class, retrieves the string in between the "$$" pair and passes that along to the ReplaceToken() function that generates the appropriate HTML for that location, then sends it to the Web class and returns program flow back to the ParseHTML function. This process continues until the entire HTML document has been completely read.

Consider for a moment, the standard Web page. Common headline area/menu and bottom menu section and common side menus, unique announcement box and unique main area content including image. In many cases, the side menu items may change but the bulk of the HTML for the side menu remains the same. The content area and the little announcement box may change but the heading area and bottom menus will remain the same. In most cases, the layout doesn't change, just content in specific areas within the layout. The Goodwin ePages approach takes full advantage of this to further extend the reusability of the system. Using this approach any HTML document can be made reusable, while still maintaining it's uniqueness once published. In our example, the same HTML is used to create HTML forms that will be processed by two completely different agents. It would be just as easy to replace <input> tag field names, add additional fields, etc. In your sites, two or three agents with much of their code copy pasted from other locations may very well serve an entire transactional system.

Now you may have noticed that the ParseHTML() function is looking a little ratty and doesn't seem quite like it was originally designed for this task. In that you would be quite right. The ParseHTML() function was originally designed long before the existence of Goodwin ePages and has been copied from application to application ever since, a testament to its reusability. Future expansions of Goodwin ePages will include this functionality as a built in and simplified component.

Finally, one of the things you may have noticed in all of the examples provided is that none of them open a "DocumentContext". For those not familiar with this, "DocumentContext" is a specialized state that a document object can be set to for reading information from a browser. Because Goodwin ePages is handling browser to agent communications through the CheckCGI() method, it opens the "DocumentContext" as needed, once again freeing your code to focus on other things.

OpenPage – Utility agent

The OpenPage agent as previously mentioned is a utility agent. It has been provided as a tool to get you started and help with your more common and less specific transactions. Although it demonstrates much that has already been shown in the example agents, it also has a few other things to teach.

OpenPage demonstrates some of the flexibility available to you in the use of URL variable/value pairs and how to handle them. As we've seen in other examples, it opens any page we should happen to specify in the PAGE variable of the URL provided a page of that name is in the database's HTML view.

One of the things we haven't seen in other examples is referral techniques. OpenPage demonstrates how this can be accomplished. We have arbitrarily selected the word "URL" as our in url variable to hold the web page address to refer the interaction to. The address is specified as a value to the URL variable like any other web address but enclosed in quotes. The need to enclose the referral address in quotes is only really vital when the referral address contains special characters (i.e. & or %) such as those that Goodwin ePages is watching for in your own URLs. In such an instance, enclosing the referral address in a pair of quotes tells the Goodwin ePages class that everything inside the quote pair is for a completely different URL from the one it's processing. This causes the Web class to ignore the content of the URL and simply pass it as a value.

Once we have the referral address in our code, we need only remove the quote at either end, enclose it in square braces [URL] and print it to the browser. You'll note that I said print it to the browser rather then sending it to the Web class. The web class's services are designed for page interactions. Referrals act more like commands then pages in that the browser responds instantly to [URL] without the need for additional HTML tags or structure. Thus, a simple Print "[Your URL]" is all that is needed. The only caveat here is that the URL must be a complete URL. This includes the http:// reference.

The last area of interest in the OpenPage agent is the MS Office launch options. The OpenPage agent implements the "DOCTYPE" variable. We can specify either "WORD" or "EXCEL" as our DOCTYPE. When OpenPage receives this DOCTYPE value it sets the web page to launch into the specified MS Office application on the visitor's machine. If the visitor has the target MS Office application on their machine, it will be launched and the HTML will be sent directly to it. Both applications respond well to HTML. In fact entire spreadsheets including formulas can be built in HTML tables and sent to Excel who will treat table cells as worksheet cells and perform calculations as specified. Interactions with MS Word are very similar in compatibility. This should come as no surprise since HTML is in fact a document formatting language. Word can interpret the HTML document formatting and provides the same formatting in it's display of the same document.

Message loops – Interesting side note

You'll have noticed that almost every agent presented, that interprets received data, does so in a SELECT CASE statement embedded in a FOR-NEXT loop. Processing is either performed in place or processing of variables is passed off to other routines to complete before continuing the loop.

This methodology is called a Message loop and it may surprise you to learn that you've been working inside of a message loop almost identical to the ones coded here ever since you used MS Windows for the first time. Message loops are at the core of MS Windows. All clicks and drags or any other events that occur in the environment send messages back to the Windows environment which are handled through the associated message loop. This is a very common and proven technique.

Where do we go from here?

There are any number of ways Goodwin ePages could be used at this point. You could choose to code a separate agent for each stage of the web transaction or you could choose to encode stage information in the <Form> tag URL of your HTML forms and code a single agent to process and respond to each situation.

You could choose to implement applications requiring a user to log-in and which will respond differently for each user. Agents can be set to operate under the users name so as to maintain normal Domino security features and Goodwin ePages collects and maintains the user's log-in name within the persistent class for reference by your code as needed.

Launch controls can be built to inhibit site operation or refer any user based on specific criteria. You might choose to block certain IP addresses or ranges. Because the agent has the opportunity to interrogate browser details before proceeding, these things are all possible.

You'll be surprised at the number of doors that open and the level of reusability and control you'll have. The rest is up to your imagination.

Debugging

If you code it, they will come, BUGS. So it's useful to go over some of the debugging techniques used in web development and expose you to the debugging feature within the Goodwin ePages class.

First of all, when things go drastically wrong in your code, there's only one thing your user will see, "AGENT DONE". There is no link or explanation; these tasks are your job. We strongly recommend the use of "On Error" routines in your agents to deal with this eventuality. And NO, Resume Next is not considered, "Dealing with things". (I've actually seen this done by consultants)

Web agents operate free from any client. As such, they operate as backend agents. When errors do occur, the messages are actually displayed in the server's Miscellaneous Events

log just as they would if the agent was running some backend server processing. This is where you'll find your error messages.

Given that the agent is meant to run with a browser, this would leave you with no debugging tools whatsoever. The Goodwin ePages class has provisions to help with this situation.

You may have thought it curious that the CheckCGI() function is sent an empty string every time it is called in the examples. In fact, your code won't even save unless CheckCGI has been given a string, even an empty one. This string parameter is a critical one because it tells CheckCGI() whether you want normal operation or DEBUG operation. Placing any text, even a blank space, into the CheckCGI() string parameter will force it to stop accepting input from a browser and run as though it was running locally. This allows you to use standard Notes debugging features to analyze the situation and come up with a solution. The string placed into CheckCGI() will be parsed as though it came from the browser so as to facilitate recreation of the situation that produced the error condition.

For normal URL type interactions where the URL has come from the browser location bar or from within the Action clause of a <FORM> tag, simply place the URL inside the quotes in the CheckCGI() Parameter {i.e. CheckCGI("URL")} and run your agent through the LotusScript debugger.

For HTML Form problems, redirect the form output to an agent that will dump the raw output to the browser. This redirection can be done by pointing the Action clause of the form to the new agent. Once you have the raw form output, copy it and place it in the CheckCGI() string parameter. Once again, execute the agent in question through the LotusScript debugger.

Sometimes the amount of code or more commonly, the repetitions of a particular loop, make following the code though the script debugger a painful chore. Much of the work can be eliminated by either placing print statements in your code to identify how far the code got before failure. These print statements will output directly to the browser regardless of the CheckCGI() debug parameter. To report functionality to the Notes log, you can place MessageBox statements in your code. When not in debug mode and responding to a browser, these Messagebox statements display to the Miscellaneous Events in the Notes log rather then on any workstation. These can be helpful in tracking down intermittent problems or simply reporting on web site activity.

Conclusion

We could go on and on, providing you with techniques, approaches and code samples but all that would really do is make the manual bigger and give you more to read. At some point we have to stop talking about it and you have to go explore for yourself. We've reached that point.

The code samples are there for you to experiment with. I recommend you make a backup copy of the original database and play with the code and HTML samples provided, expanding them and exploring your new limits. We hope you have as much fun exploring the possibilities as we've had making them possible.

Appendix A

Command Reference

Browser Retrieval Methods & Properties

CheckCGI(*DebugURL as string***) - Method**

Description:
CheckCGI reads incoming CGI variables and performs the following;

- If CGI data is "POST" Data, CheckCGI reads from the "POST" CGI variable.
- If CGI data is "GET" data, CheckCGI reads the "Get" CGI variable.
- If the *DebugURL* variable is populated, CheckCGI ignores all variables and opens no incoming browser communication session. This is a Debug mode.
- Parses Variable/Value pairs into a 2 dimension array for code use.
- Parses incoming data for Hexadecimal values to be replaced with characters.
- Replaces all + signs with spaces.
- Collects the username (if any) of the connected visitor

Parameters:

DebugURL
DebugURL is intended to hold any URL string for use in debugging. *DebugURL* acts as a replacement for live Browser communication.

Usage:

Variant = WebObj.CheckCGI("") 'Normal operation
Variant = WebObj.CheckCGI("*DebugURL*") 'Debug mode

When using CheckCGI(), Debug mode overrides all browser communications. During normal operation, "Post" data takes precedence over "Get" data. For use in situations where both "Post" and "Get" data must be retrieved, see CheckURL()

When Calling CheckCGI() a variable of type Variant must be used to capture the array returned by the method. The array is a two dimensional array, the first of the subscript is the variable name received from the browser. The second subscript is the value represented by the variable name returned. All array elements are of type String.

Finally, CheckCGI() captures and stores the username of the logged in user and stores it for later retrieval by the agent as needed for security.

CheckURL() - Method

Description:

CheckURL evaluates previously read CGI variables and retrieves the URL portion of the browser communications. CheckCGI() must be run first.

CheckURL reads previously captured CGI variables and performs the following:

- Parses Variable/Value pairs into a 2 dimension array for code use.
- Parses incoming data for Hexadecimal values to be replaced with characters.
- Replaces all + signs with spaces.

Parameters:

None

Usage:

Variant = WebObj.CheckURL()

CheckURL() returns URL parameter information already gathered by CheckCGI(). CheckURL() is provided to allow for embedded Action URL data retrieval (Get Method) in conjunction with Form data retrieval (Post Method)

When Calling CheckURL() a variable of type Variant must be used to capture the array returned by the method. The array is a two dimensional array, the first of the subscript is the variable name received from the browser. The second subscript is the value represented by the variable name returned. All array elements are of type String.

WebUserName - Property

Description:

Returns the common name of the user currently logged into the application (If any). This is provided to assist in the creation of security routines without the need to pass username information from function call to function call.

Parameters:

None

Usage:

String = WebObj.WebUserName

WebObj.WebUserName = *String*

When retrieving WebUserName, a variable of type String must be used to capture the returned username.

Method - Property

Description:
Returns the type of return received from the browser, “Get” or “Post”

Parameters:
None
Usage:
String = WebObj.Method
WebObj.Method = *String*

When retrieving Method, a variable of type String must be used to capture the returned submission method.

Web Page Generation Methods & Properties

StoreHTML(*String*) - Method

Description:
Takes string information and adds it to the bottom of the web page instance currently being held within the Web class. StoreHTML adds a hard carriage return to the end of all lines it adds to the HTML to facilitate improved readability of generated HTML source. StoreHTML protects the developer from 32K string variable limit prevalent in the LotusScript language. Storage limits are dictated by the available memory on the machine. This method does not publish.

Parameters:
String
A *String* containing browser code you wish to add to your web page. This could be any string information desired as the method does not interrogate or modify the submitted data.
Usage:
Call WebObj.StoreHTML(*String*)

StoreHTML returns no data.

AutoTags - Property

Description:

Autotags sets or disables the publication of <BODY> , <TITLE> and <HEAD> tags when publishing web pages. This is used to facilitate full control over these tags and their content. When Autotags is disabled, the developer must ensure that these tags are in place where needed themselves. <HTML> tags and ContentType tags are still generated. Autotags defaults to enabled.

Parameters:

True/False

Usage:

WebObj.AutoTags=*True*

Integer = WebObj.AutoTags

ContentType - Property

Description:

ContentType is used to force the web page into a different context, thus causing specified applications on the visitor's workstation to launch and populate with the generated data.

Parameters:

Word/Excel

Usage:

WebObj.ContentType= *Word*

ContentType has no Property retrieval method. Settings can only be sent, not read. Setting this property to "Word" will cause MS Word to launch on the visitors machine. Setting this property to "Excel" will cause MS Excel to launch on the visitors machine. In both cases, the user must have these tools installed and working on their machine for the software to launch.

PrintHTML() - Method

Description:

PrintHTML() sends generated information to the web browser as a complete page. ContentType tags, <HEAD> data and <BODY> data that have been accumulated in the Class instance are all sent. By default, page components are wrapped in <HEAD> and <BODY> tag pairs but this can be disabled. See AutoTags property.
Web Page data is not destroyed by this method but remains in the class.

Parameters:

NONE

Usage:

Call WebObj.PrintHTML()

No return data is sent.

Head - Property

Description:

This property is used to store the HEAD data for the web page. Data sent to Head is not interrogated or modified in anyway. The HEAD property has a 32K limit.

Parameters:

String – Page HEAD code.

Usage:

WebObj.HEAD = *String*
String = WebObj.HEAD

Any string data is valid.

Title - Property

Description:

This property is used to store the TITLE data for the web page. Data sent to TITLE is not interrogated or modified in anyway. The TITLE property has a 32K limit.

Parameters:

String – Page Title

Usage:

WebObj.TITLE = *String*
String = WebObj.TITLE

Any string data is valid.

Utility Methods

ParseChunk(*Data as string, Delimiter as string***) - Method**

Description:

ParseChunk() separates a single string of data into an array of separate strings based on a supplied delimiter. If ParseChunk() is sent a sentence as data and a blank space as the delimiter, ParseChunk() will return an array of all words in the sentence listed in the array in the order in which they occurred. ParseChunk() is useful for situations where volumes of data elements are being provided in a single string, delimited throughout by a common delimiter.

Parameters:

Data as String – Original String
Delimiter as String – Delimiter between desired elements

Usage:

Variant = WebObj.ParseChunk(*Data, Delimiter*)

When Calling ParseChunk() a variable of type Variant must be used to capture the array returned by the method. The array is a single dimensional array. All elements within the array are of type string.

ParseReplace(*Data as string, Hunt as string, Replace as string***) - Method**

Description:

ParseReplace() is a global search and replace for strings and is widely used in the preparation of inbound browser data. Supplied with String data, a character to "hunt" for and a character to replace it with, ParseReplace() will return a string with the requested substitution.

Parameters:

Data as String – Original String
Hunt as String – Character to be replaced
Replace as String – Replacement character

Usage:

String = WebObj.ParseReplace(*Data as string, Hunt as string, Replace as string*)

When Calling ParseReplace() a variable of type String must be used to capture the String returned by the method.

ParseHex(*Data as string***) - Method**

Description:

ParseHex() is specifically written for browser return data where spaces and other special characters are often returned in an encoded Hexadecimal format (i.e. %20). ParseHex() seeks out these Hex representations and replaces them with the ASCII character they represent.

Parameters:

Data as String – Original String

Usage:

String = WebObj.ParseHex(*Data as string*)

When Calling ParseHex() a variable of type String must be used to capture the String returned by the method.
