Extending Workflow Functionality

A White Paper for OpenText's Livelink

Version	3.0
Release Date	15 May 2003
Author	Greg Griffiths
Contact Email	livelink-whitepapers@greggriffiths.org
Website	http://www.greggriffiths.org/livelink/whitepapers

Introduction

This White Paper discusses one possible option for extending the functionality of Livelink workflows – mainly Adobe or WebForms based ones – by using standard web CGI (Common Gateway Interface) languages such as ASP, JSP and Perl alongside some client side scripting languages such as Javascript and VBScript.

This White Paper seeks to provide an option to those who require additional functionality in their workflows in addition to or instead of that of the Livelink Builder SDK toolkit. This White Paper illustrates the functionality that can be provided within a workflow without the use of the Livelink Builder SDK.

The server side code presented here is written in VBScript for use in an Active Server Page (ASP) file, but it is easily portable to other languages such as Perl or PHP. The client side code presented here is written mainly in JavaScript – to ensure a wide browser support – although it can easily be converted to any other scripting language.

About the Author

The white paper's author is **Greg Griffiths,** a Livelink Certified Developer. He has several years experience as Senior Livelink Administrator & Developer for several large installations of Livelink with multinational user bases. He has had exposure to the majority of Livelink modules and is a frequent contributor to the discussions in the Knowledge Centre, as well as several other groups on web development.

More Information

More Information about the Author can be found at :
http://www.greggriffiths.org/livelink/authors/greggriffiths.html

More tools, customisations, white papers etc can be found at :
http://www.greggriffiths.org/livelink/

More information about Livelink and OpenText can be found at :
http://www.opentext.com/livelink/

Contents

Livelink Workflow – An Introduction

Introduction

This section provides a basic overview of Livelink workflow to familiarise the reader with some of the concepts that will be discussed later. It will also illustrate some concepts in workflow design that may provide enhancement to your existing workflows.

Workflow Steps

Livelink Workflow as featured in Livelink 8.1.5. provided a selection of Workflow steps :

	Initiator	This step is to be performed by the initiator of the workflow
	User / Group	This step is to be performed by the named user or group
	Sub Workflow	This step is a link to another workflow
	Evaluate	This is a step that evaluates a set of values and takes a path according to the result.
	Milestone	This step represents a point of achievement in your process.

Livelink 9 provided many enhancements to the existing selection of step types, but also added two new ones :

	Process	This step is an automate process, such as send an email, update attributes or save an attachment.
	XML Data Interchange	This is a link for an XML data export or import.

These workflow steps provide the basis of the system that can be combined to produce a 'map' representing the process, similar to a flow chart or diagram.

Now that you have the process, you need some people to perform the steps within your Workflow. Livelink offers two steps types for user interaction, *initiator* and *user*. The first sends the step back to the Livelink user who initiated that workflow, whereas the latter sends the step to the Livelink user or group that you have specified.

The *evaluate* step allows you to add routing into your workflow, allowing you to take one branch if a condition is met, another if it is not. For example, if your expenses are under a certain value, then your manager can approve them, over that threshold then the department manager must approve them instead. In Livelink 8, only a single Yes / No evaluation was possible, in Livelink 9 however, multiple true paths can be selected – similar to a Case statement used in programming.

At certain points in your Workflow, you may have reached a point of achievement – for example, request rejected, process completed etc – *Milestones* provide this functionality.

As a single process can produce a huge workflow map, the *subworkflow* step allows the developer of the workflow to break the workflow down into more manageable sections. This also allows you to modularise commonly used functions – for example bulk notification activities – rather than having them repeated throughout the main map or even several workflows maps or systems. This also makes them easier to manage and maintain.

Subworkflow steps can be very useful to split your main process into its constituent elements – in a large process there is likely to produce a quicker initiation and processing. However, in some cases, the overhead of moving from a 'parent' workflow into a 'child' subworkflow can be a lot higher than the performance benefit of doing so, therefore think an plan the workflow carefully.

Livelink 9 offers the addition of the *process* step, this allows you to perform some simple automated functions, for example update the value of attributes in your workflow. You can also send an email to one or more users or save an object attached to the workflow into the main folder structure of Livelink.

Workflow Form Types

Livelink 8.1.5 provides one in-built type of workflow – *attributes* – a second – *Adobe Forms* – can be added as an optional module to provide a much nicer user interface among other benefits. Livelink 9 offers an additional module, the *WebForms* module, which uses HTML templates containing a HTML or PDF form, which contain a subset of the elements from the template.

Of these three options, the first – attributes – is the quickest to design and develop, but you have very little control over the look and feel of the display to any user. Advanced functionality such as totalling a list of values for example is also not possible.

Despite these drawbacks, for simple uncomplicated workflows or prototypes of workflows with complex data or interface requirements, attributes can be a useful asset. It may also prove to be a good starting point as any transition to the other methods – which would take longer to develop – is greatly reduced as the process and its data is already organised and the users would perceive the change as an upgrade.

PDF forms allow the designer much more control over the user interface than attributes permit. It also allows the developer access to the functionality provided by Adobe Acrobat in addition to allowing the use of Javascript functionality.

However, PDF Forms are much more complex to develop – a 'template' must be prepared in another tool such as Microsoft Word, before being imported into Adobe Acrobat to have the form fields and custom code layered on top. Therefore any major change to this base template can be a long and laborious process, although a variety of tools such as Cardiff's Teleforms Designer tool which is part of the PDF Forms Professional module can assist in this area.

Also, despite the fact that the 'reader' component – which all users will need to have installed in their browsers to handle PDF files – is free, the 'writer' tool can be quite expensive to purchase.

The webforms module, only available since Livelink 9, allows you to create a base 'template' in a similar fashion to that of attributes. However, this 'template' can be exported into your favourite HTML editor – from notepad or VI to Dreamweaver or Frontpage. Then you can take as many of the fields form there as you need to create each of your 'views' as well as making the layout more appropriate – for example adding the company logo – and adding some client side code to add functionality such as totalling and validation.

Database Connectivity

Introduction

In this section we will look at the reasons for using database connectivity with Livelink workflow, then we shall look at the options available with Livelink and then those available for a little extra work via the use of CGI scripts such as ASP, Perl or PHP.

Why Database Connectivity

As you begin to use Livelink workflow functionality there may come various scenarios in which it would be beneficial to have access to at least one database for either getting or saving data, some examples include :

- Getting of information relating to a given product from a central reference database.
- Retrieving the latest value of regularly changing data such as exchange rates, product costs, address or phone numbers.
- Getting data from another system's database.
- Saving a copy of workflow form data to a separate isolated database for review and reporting.
- Updating a central reference database after using Livelink Workflow to manage the change control process.
- Population of other systems databases with workflow data.

There may be some workflows that you may already have developer or be planning to develop that would benefit from this kind of connectivity.

Provided Database Connectivity

Livelink does provide some database storage functionality. At the most simple level data is stored in Livelink's own proprietary database – a schema for which is available after signing a Non Disclosure Agreement with your OpenText Account Manager.

For PDF Form based workflows you have the option to save workflow data to either a table created by Livelink – based on the field definition on the form – or a custom designed database. The database you create in the latter must share the same area of the database as the Livelink system tables. In both cases the entire form is dumped to or read from a single table, there is no ability to get or save data from multiple tables.

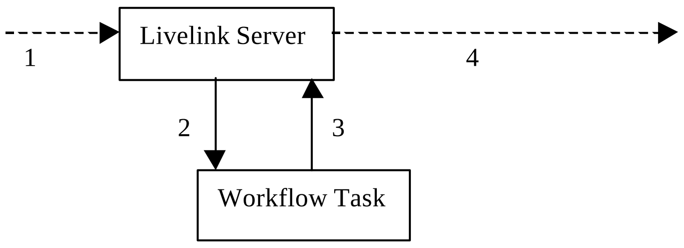
Webforms and attributes are encoded and saved into Livelink's database. For more information on retrieving this data I suggest that you get the Livelink Schema and look at the Livelink Database and Builder Courses.

OpenText are responding to the increased use of Livelink workflow that would benefit from this sort of connectivity. There is a module available from the OpenText *FirstLook* site[1] called *Form DB Lookup* which although a beta module provides a lot of functionality in this area, and the expectation is that subsequent releases with offer improvements in this area among others.

[1] This is a new OpenText site contains examples of beta concepts and modules which may be included in future releases, contact your OpenText Account Manager for more information and access.

Including the custom with the standard

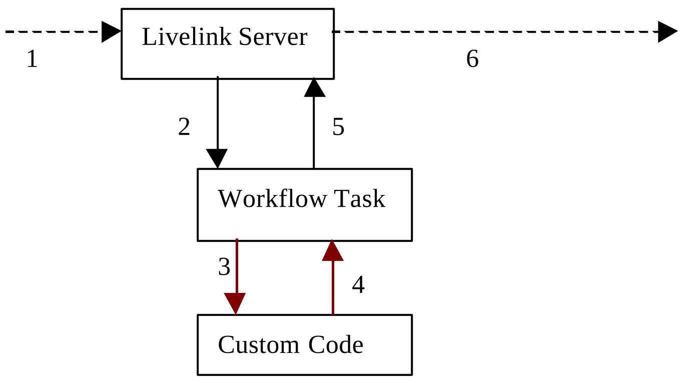
How do we include this kind of connectivity within the Livelink Workflow process ? To understand where this fits in we need to look at how Livelink processes a single step :



The previous task is completed (1) and Livelink assigns the next task, the user then accepts the task (2). Having completed the task, the user sends the task back to Livelink (3) where it continues along the workflow (4)

Our Custom code is hung off the *Workflow Task* component and occurs between steps 2 and 3. Thus we have something that looks more like this :

The first two steps as before, however the user has the extra interaction with the server side script (3) with any changes being made to the form as a result of the script being made (4) before the Form can be completed and the task returned to Livelink as before.



By introducing the custom components here and simply updating the values on a standard Livelink form, we are ensuring that our code is separated out from Livelink.

By compartmentalising out the code, we are protecting it from any problems that could be caused when we upgrade any of the components such as Livelink, our server side script, any databases our server side scripts use etc.

This method also allows the additional functionality to be added without risking problems arising out of the workflow using a mixture of standard Livelink and any custom code. This is because the code is only updating the form and not directly interacting with Livelink, so as long as valid values are placed onto the form you should be able to eliminate your server side code from any Livelink problems.

However, this does mean that you will have code sitting outside your immediate Livelink environment, which you will need to manage and consider in your overall support plan. Also, by using this approach you will have to provide the OpenText support team with all the code and settings to recreate this environment if they are working on an issue with you as opposed to simply sending them a self created Builder module.

PDF Forms Connectivity

Introduction

In this section we shall look at ways of providing access to a datasource for a PDF workflow form for retrieving and saving information for a selection of objects, from text fields to drop down lists.

Preparing the Form

Once you have your standard form, you need do little more than make a few minor amendments and additions to achieve connectivity.

There are two methods that are commonly used on the forms themselves, the first is to add a 'submit' like button in the appropriate place on the form. Both of these approaches will result in the form being submitted to your specified script, where it can be processed just like any other HTML submitted form.

This can be achieved by selecting the *Form* mode and then drawing a box for your submit button. Once the field is created, set the *Type* to *Button*, and then set a *Mouse Up* action on the *Actions* tab to *Javascript* and then add an appropriate script followed by a call to submit the form which should look like :

```
this.submitForm(“http://myserver.com/scripts/formhandler.asp#FDF”,false,false);
```

The second is to add a submit call using Adobe Forms Javascript in as a *Document Javascript*, this is done by selecting *Set Page Action* from the *Document* menu, and then highlighting the *Page Open* choice, press the *Add* button and select *Javascript*. A simple Javascript can then be inserted as required, as described above. This will submit the form as soon as the form is opened by any user.

Unlike submitting a HTML form, the Acrobat PDF remains open on the screen in this case. We then use the script that is the target of the submission to create a Forms Data Format (FDF) file which is then sent back to the PDF Form and then updates it according to the contents of the FDF file.

Adobe Forms Javascripts and Livelink

Some of you may have noticed that there is also a *Document Javascript* option in Adobe Acrobat, the reason that we are recommending that you use a *Page Javascript* is because of the sequence of events that occur when you open a PDF form, they are as follows :

1. Open PDF Form
2. Run any defined Document Javascripts
3. Inherit fields from a Master Form
4. Run Page Javascripts.
5. Allow user interaction.

As you can see, if you need to use any of the form fields (for example a reservation number or process id) you need to do it after the field value has been inherited from the Master Form.

The Server Side

Now that we have your form submitted to your server side script, you can process it just like any other web-based form, this may include validation of data, calculations of totals or results or database interaction.

If you are on a Windows platform, then you can use ODBC to provide connectivity to almost any datasource you need, from Microsoft Excel to Oracle. This is well documented in many books and websites, so I shall not cover it here.

The problem comes when you wish to push your new data back onto the form, which is still open in the browser window. Adobe have produced a freeware toolkit known as the *FDF Toolkit* which contains a series of Library files for Java, Perl and VBScript ASP's to allow scripts written in these languages to interact with PDF forms. The interaction is done using FDF – *Forms Data Format* – files, which is simply an encoded data stream sent to the PDF file. The toolkit can be found by searching on Adobe's website.

A tutorial on using PDF files instead of standard HTML files as a web based front end forms solution can be found at :

http://www.greggriffiths.org/webdev/serverside/asp/asp_fdf/

This site contains a worked example, demonstration code and other resources.

Taking it a little further

The only problem with using *Page Javascript* functions is that they are run every time the page they are assigned to is viewed by the user, which could result in it being run several times rather than the once you require.

This, however is easily resolved. Simply create a variable in a *Document Javascript* and then set its value, test this value in the *Page Javascript* and if it is still set to the same then run the code and then change the value of the field. For example

Document Javascript :

```
var runonce=0;
```

Page Javascript :

```
If (runonce==0)
{
    runonce=1;
    this.submitForm(“http://myserver.com/scripts/formhandler.asp#FDF”,false,false);
}
```

In some cases, the processing may take a while to complete, so you may not want to display the submit button that saves back to Livelink immediately, but wait until you get a 'completed' signal – or return value – from your script.

Adobe comes to the rescue with the use of its *Hide/Show* field options, which can be used to hide and show buttons on the form, these actions can be added in the same way as the Javascript described above, or even within a Javascript as shown below :

```
Var LLButton=this.getField("submit2Livelink");

LLButton.hidden="true";
```

Once your script has completed you are likely to need to send something back to the form to let it know what to do next, for example show the submit to Livelink button, display an error message etc. This is known as a *return code* in programming, and can be achieved by creating another field – usually a *hidden field* - on the form.

This code is then included in the FDF response from the server to the client where it updates the hidden field on the PDF Form. This change causes any Javascript in that fields *Calculate* tab to be executed.

We can achieve this by inserting some code into both the *Document Javascript* and the fields *Custom Calculation* box – which can be selected from the *Calculate* tab of the field properties window - as shown below :

Document Javascript

```
// reset the status value
var status=this.getField("status");
status.value=0;
```

Custom Calculation

```
var status=this.getField("status")

// if the value is one, then save was good
if (status.value==1)
{
        // show the submit to Livelink button
        var LLButton=this.getField("submit2Livelink");
        LLButton.hidden="false"
}
// else inform the user
else
{
        // display a popup alert window to the user
        app.alert("There has been a problem. Please contact the HelpDesk",0,0)
}

// reset the status value
status.value="0";
```

More on Adobe PDF specific Javascript can be found in the *Form Javascript Guide* on the *Help* menu in Adobe Acrobat. The code here is not limited to PDF Workflow forms, it can be used to create a PDF Form that is added as a Livelink *document* object to provide an enquiry form over a database, or even on any PDF that is not in Livelink.

WebForms

Introduction

In this section we shall examine how to achieve similar results to those described above for Livelink WebForms. The main problem with WebForms is that they are simply HTML forms, this makes them accessible to anyone with a copy of Notepad, VI, FrontPage or Dreamweaver for example and a little HTML knowledge.

Web Form Field names

When we interact with Web Forms, we need to provide a reference to each field that we wish to interact with. With PDF forms, the developer names each field as you create it, however Web Form field names created in a slightly different way. If you view the source code of a WebForm, you will notice that each of the fields has a unique name, which bears little resemblance to the label associated with it, for example :

```
<INPUT TYPE="text" NAME="_1_1_2_1" VALUE="[LL_FormTag_1_1_2_1 /]" SIZE="32"
MAXLENGTH="32" ONCHANGE="markDirty();">
```

The important part of this HTML snippet that we will be using later on is the *Name* value of the form, in this case *_1_1_2_1*.

The newWin approach

The main problem is that when submitted to a server side script the form does not remain in the browser window, as can happen with PDF forms, making interaction with server side scripts a little more complex, without recreating the entire form as the output from your server side script. One solution is to use a little bit of Javascript and a new window.

The best way to get access to server side scripts without submitting the Web Form is to use the Javascript *window.open* call to open a new browser window. Into which we load a HTML file that uses Javascript to copy the values it needs from the WebForm into a form hidden within itself before submitting itself to our server side script. Once processed on the server side, the page sent back by the server side script contains Javascript to update the WebForm and then close the window.

At the appropriate place in our WebForm, we add our button :

```
<input type="button" value="Retrieve"
onclick="window.open('http://www,myserver.com/dataloader3.html','newwin','menubar=no,stat
usbar=no,scrollbars=yes,resizable=yes')">
```

Clicking on this button will open the page *dataloader3.html* in the new window that we have defined. This file would look something like this :

```
<html>
<head>
<script language="javascript">
function loader()
{
        document.specwin.idno.value=opener.document.myForm._1_1_4_1.value;
        document.specwin.submit();
}
</script>
</head>
<body onload="loader()">
<form name="specwin" action="http://www.myserver.com/scripts/dataloader.asp"
method="post">
<input type="hidden" name="idno">
</form>
<center><font face="verdana,comic sans ms,arial" size="+3">Getting Data<p>Please
Wait</font></center>
</body>
</html>
```

This page copies the value of the field *_1_1_4_1* into a hidden field *idno* and then submits itself to the script at *http://www.myserver.com/scripts/dataloader.asp*. Notice also that the body of the page displays a please wait message, which remains until replaced by the response from our server side script.

The *dataloader.asp* file simply processes the data and then sends back a page containing Javascript to put any new values back into the correct fields on the form, you may also want to close the window once this has been done. An example could be something like :

```
<%
Dim idno
Dim textone
Dim texttwo

idno=request.form("idno")

if (Len(idno)>0) then

        textone=”Hello”
        texttwo=”World”

        Response.Write("<html><body>")
        Response.Write("<script language='Javascript'>")
        Response.Write("opener.myForm._1_1_2_1.value='" & textone & "';")
        Response.Write("opener.myForm._1_1_3_1.value='" & texttwo & "';")
        Response.Write("</script>")
        Response.Write(“<center><font size=’+3’>Data Loaded</font></center>")
        Response.Write("<form>”)
        Response.Write(“<input type='button' value='Close' onclick='window.close()'>”)
        Response.Write(“</form></body></html>")
%>
```

This script sets the value of the field *_1_1_2_1* to the value of the variable *textone* and sets the value of the field *_1_1_2_1* to the value of the variable *texttwo*.

If you want to remove the first form - *dataloader3.html* in this case – you could replace it with a call to the server side script, but you would have to pass the parameter(s) in the Query String of the browser, for example :

```
http://www.myserver.com/scripts/dataloader.asp?idno=2
```

This also would require you to amend your server side script to get the value from the Query String rather than a form submission – in technical terms a *Get* submission rather than a *Post* submission.

The IFrame approach

In some cases, you may not wish to open a new window that is visible to, and can be closed by, the user. Some web browsers support the concept of Inline Frames – commonly called IFrames – which work in the same way as a normal frame, but exist as an object on a given HTML page.

Just like a standard Frame, the URL in the IFrame can be different from that of the 'parent' or 'container' form. This allows us to use the IFrame in the same way as we used the new window in the previous section.

Population of an IFrame, just like the new window, can be in one of two methods – *push* or *pull*. In the former, the main page drives the copying of values from the main page to the new window or IFrame. In the latter, the page that is loaded drives the loading of data into the new window or IFrame.

Cascading Style Sheets (CSS) can be used to make the IFrame invisible to the user of the form, by using the *visibility* or *display* attributes, so the definition of our IFrame could look something like :

```
<IFRAME SRC="foo.html" WIDTH="300" HEIGHT="100" STYLE="visibility:none">
```

Using these methods, you will be able to provide the same level of access to server side scripts, as you can with the previous methods for PDF forms, also Web Forms are a lot cheaper to produce – not requiring Adobe Acrobat – and maintain as they are basically HTML web pages.

Displaying Workflow Data Differently

WebForms are also simpler to customise, as far as look and feel and additional client side coding are concerned as they are simply HTML forms, thus anything that a web developer could do with a web page can be done on a Web Form. So you can arrange the form fields alongside a company standard layout, add text, company logos and the like until you are happy with the final layout.

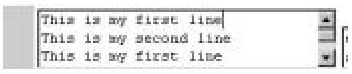
Additionally, you may want to display the contents of a field without the restriction of the form field. For example, if you have a field defined as follows :

```
<INPUT TYPE="text" NAME="_1_1_2_1" VALUE="[LL_FormTag_1_1_2_1 /]" SIZE="30"
MAXLENGTH="50">
```

This field will only display the first 30 characters of the field and you would lose the rest from your on screen display or print out. The same issue can arise if you are using a *Text: MultiLine* – called a *textarea* in HTML – for example if you are storing comments or descriptions on a form you will only see / print what is visible in the box.

You could remove the HTML tags and just leave the Livelink tag, which works will display the value of the field. This will work in many cases, but in some cases, it will not be suitable as it will not translate control characters such as carriage return, so a field with the content of :

```
This is my first line
This is my second line
This is my first line
This is my second line
```



This is my first line
This is my second line
This is my first line

Would be displayed as :

```
This is my first lineThis is my second line
```

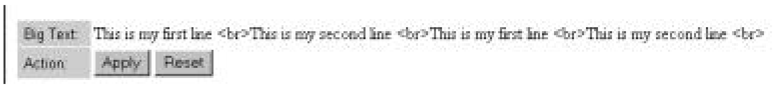
If you changed the tag from :

```
<TEXTAREA NAME="_1_1_2_1" WRAP="soft" ROWS="3" COLS="32"
ONFOCUS="" ONCHANGE="markDirty();">[LL_FormTag_1_1_2_1 /]</TEXTAREA>
```

To :

```
[LL_FormTag_1_1_2_1 /]
```

The solution is to use Javascript to convert the control characters into their HTML equivalents. If you do this conversion before you save the value back to Livelink, Livelink will convert your newly added HTML tags when it saves it back, so they appear as text, for example :

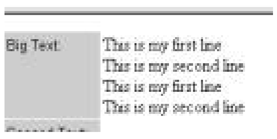


Big Text: This is my first line
This is my second line
This is my first line
This is my second line

The solution is to turn the field into a *hidden* field and then where you want the text to appear you use Javascript to display it, using a Regular Expression to convert the carriage return into the HTML control characters :

```
<input type="hidden" name="_1_1_3_1" value="[LL_FormTag_1_1_3_1 /]">
<script language="javascript">
        // use a RE to replace the \n with a <br> in the text field
        var newtext=document.myForm._1_1_3_1.value.replace(/\n/g,"<br>");
        document.write(newtext);
</script>
```

This would display the field on the form as follows :



Big Text:	This is my first line This is my second line This is my first line This is my second line

Replacement Variables

Livelink 9 provides a selection of replacement variables that can be used to bring Livelink content onto the forms. In addition to those detailed in the manual, several other variables exist that may be of help to a Workflow developer.

Variable	Description
[LL_UserLogin /]	Users Login name
[LL_UserFirstName /]	Users First Name
[LL_UserLastName /]	Users Last Name
[LL_UserMiddleName /]	Users Middle Name
[LL_UserTitle /]	Users Title
[LL_UserMailAddress /]	Users email Address
[LL_UserContact /]	Users Contact Details
[LL_UserGroupName /]	Users Default Group Name
[LL_UserFullName /]	Users Full Name

These values are sourced from the user profile defined for the current logged in user. The data can be amended using the *Users & Groups* option. At present, these values are only available within the workflow subsystem. These values can be incorporated into a workflow form, for example, the value of the following field is set to the Login name of the current user :

```
<INPUT TYPE="text" NAME="_1_1_2_1" VALUE="[LL_UserLogin /]" SIZE="32"
MAXLENGTH="32" ONFOCUS="" ONCHANGE="markDirty();">
```

Possible uses include using the users full name as part of a header in a comments box, or on form audit trail, providing an option to display certain content items depending on the username or default group.

Read Only fields

In some cases you may want to display workflow data as read only rather than editable on a given form. In some cases, using the approaches defined above in *Displaying Workflow Data Differently* above, in other cases you may wish to preserve the form field elements on the screen while keeping the content read only.

Some browsers support the *disabled* attribute for a form field. However Internet Explorer, and some other browsers, will not pass a value back to the server for fields defined in this way. While this may work for text elements, it has the effect of unsetting any check or radio boxes that you may be using on your form. In these situations the *readonly* attribute could be used, for more information on the difference between these two and their uses can be found on the internet, for example :

http://www.htmlcodetutorial.com/forms/_INPUT_DISABLED.html

One of the alternatives is to use client side coding such as Javascript to act as an event handler on the form field, for example :

```
<INPUT TYPE="text" NAME="_1_1_2_1" onfocus="blur()">
```

This will call the Javascript blur event when this field receives focus. This approach will work for most form elements, with the notable exception of check box fields.

The combination of the lack of an onfocus event for checkbox elements and the problem previously highlighted for the *disabled* attribute make using checkboxes in this way difficult.

While the checkbox form element does not have an onfocus event, it does it have an onclick event. You can use this with some Javascript to recheck or reuncheck the field, for example :

```
function manage_xboxes(theElement)
{
        // if the element is now checked, it should be unchecked
        if (theElement.checked)
        {
                theElement.checked=false;
        }
        else
        {
                theElement.checked=true;
        }
}
```

this would be called as follows

```
<INPUT TYPE="checkbox" NAME="_1_1_3_1" onclick="manage_xboxes(this)">
```

The use of the parameter *this* will pass the a reference to the field that calls the function, if you wish to write the value “longhand” it would appear as :

```
document.myForm._1_1_3_1
```

An alternative is to use a Javascript function which will parse the form when the submit button is pressed, store the field names in a hidden field and then use a handler on the form to decode and then check as appropriate

Managing Active Workflows

Introduction

This section will cover some approaches to managing the data values and presentation of those values in an active workflow. This includes the amending of data values, changes in the business logic, changing the contents of the form depending on a specified criteria and mass amendment of a given value or values in all or many of your active workflows. This section is specific to the WebForms approach.

What could we need to manage ?

We may need to interact with the forms and their data from a global point of view for many different reasons. I intend to use the term *patch* to cover all of these changes that are implemented after the project has gone live. The main areas of changes are :

- Presentation Layer effects
  criteria determining field colouration or visibility or edibility of a given field has changed, or a popup box is now required if some criteria is met.
- Data Changes / Mass Updates
  VAT calculations need to be redone if VAT rate changes, customer name changes
- Business Logic changes
  Validation criteria changes, routing requirements change.
- Patches / Bug Fixes that need to be applied to many active workflows

In the first case could be handled simply by adding one or more Javascript functions and some CSS directly onto the affected forms. However, this may result in large-scale repetition of code throughout your workflow system which would be difficult to manage and any changes would mean loading in new versions of the affected views.

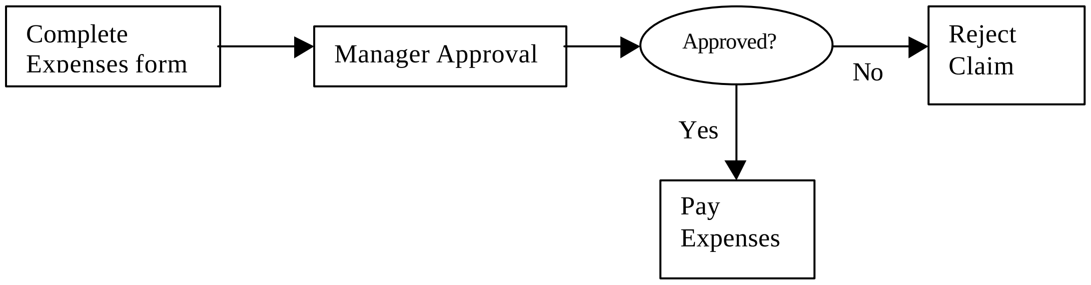
In the latter cases, we could rely on the users to make the appropriate changes or ensure that the data was correct according to the latest validation requirements, but this is unlikely to be acceptable to the users.

One approach is to suspend, correct and restart each workflow in sequence, although for large numbers of workflows this would be very time consuming. However, the risk of missing some of the workflows is greater the number of workflows that are involved.

We need a simpler, quicker, programmatic and more generic approach. The best approach is to use external client side scripting 'library' files to each of the workflow forms. These files can be amended in isolation of the workflow forms and the change would only need to be made once.

A single file could be used, but may quickly become very large and difficult to manage as well as taking up bandwith with functionality that is not required by the current form. Therefore it is better to create a *Globals* and a *Locals* file. Any variable, constant of function that will be used by more than one form / step is placed in the *globals* file, for example VAT rate, an 'is the field value numeric ?' function. Anything that is specific to only one form / step should be in the *locals* file, for example a validate employee id function.

The *globals* file will be loaded with every form in the workflow, whereas the *locals* file is specific to a given form or step, for example in the workflow below :



The inclusion of the *library* files would be as follows :

	Globals	Locals			
Library name / Step Name	Globals	Complete Expenses form	Manager Approval	Pay Expenses	Reject Claim
Complete Expenses form	✓	✓	✗	✗	✗
Manager Approval	✓	✗	✓	✗	✗
Pay Expenses	✓	✗	✗	✓	✗
Reject Claim	✓	✗	✗	✗	✓

Thus we can provide functionality on the client side as appropriate within a workflow environment.

As client side library files are loaded when required, we can make a change and the next time a form is opened that requires that file the new file is loaded. Using this ability, we can add functions into either the *globals* or *locals* file that will update data on the form the next time the form is opened.

For example, if we decided to use a four digit employee id rather than the current three digit version and change the current ones to the new format by adding a leading zero, we could use the following Javascript snippet. As we want this patch to run the next time any of the forms in the workflow are opened, we need to place it in the *globals* library file.

```
// convert Employee ID to a four digit version by adding a leading zero if required.
function patch_empID()
{
        // get the current value
        var cur_empID=document.myForm._1_1_13_1.value;

        // if the length is 3 characters
        if (cur_empID.length==3)
        {
                // update the value on the form
                document.myForm._1_1_13_1.value='0' + document.myForm._1_1_13_1;
        }
}
```

Linking the library file to the WebForm

Linking the file to the form is relatively simple. As a WebForm is basically a HTML page, we can attach external files containing client side scripting code in the same way as we would to a normal web page.

Thus, for our *Manager Approval* step in the example workflow we could use the following to attach two Javascript client library files as part of the HTML document header, i.e. between the <*head*> and </*head*> tags :

```
<script language="javascript" src="\expenseslive\expensesglobals.js"></script>
<script language="javascript" src="\expenseslive\mgrapproval.js"></script>
```

All we need to do then is to have a function in the *globals* file that can be called by the *onload* event of every form. This function will then setup the form for use by the user, this may for example involve updating values on the form, recalculating totals, setting the colour of fields using CSS.

Running a patch only once

Many patches need only to be run once. Running them more than that may be a waste if they have nothing to do, as our example patch would if the employee id was four digits, but that may not be an issue.

However if you are giving a certain client a 10% reduction, you only want to apply that to the order once and not every time someone looks at the order form. The more patches that you have, the more likely it is that other issues may arise. For example, they may cause performance issues, use more bandwith than really needed and / or raise the complexity of the library file management, especially on large projects with large numbers of patches, which in themselves may be complex or performance intensive.

To resolve this we create another field on the form and call it something like *patched_version*. This field can be hidden on each form and stores an integer representing the patch level of the current form, we can then query this to determine which patches have already been run, and which still need to, after which we can update the value to the current patch level.

Affecting routing of a Workflow

The *Evaluate* step is used to route the workflow to different steps depending on a certain number of conditions. In many cases these are simple decisions such as *is total over 10,000 dollars* for example. However, sometimes they can be a lot more complex, for example consider the following :

```
If CustomerA and total > 10,000 dollars then goto step1
Else if CustomerB and total > 150,000 then goto step 2
Else if CustomerB and total <100,000 and salesman is Tony then goto step 1
Else if CustomerC then goto step 3
Else goto step 4
```

This routing information is more complicated than the last and only works for a limited subset of customers and order totals. The other big issue is that if you wanted to change the change a rule or add a new rule for a customer then you would need to amend the workflow template (to make the change for any new workflows) and then stop, amend and restart existing workflows. This could become a major piece of work if there were a significant number of *Evaluate* steps to amend or workflows in progress.

A better way is to move this business logic into the *globals* file and use that to set routing flags stored as hidden fields on the forms, which can be used by the workflow. Thus, any change in the business logic simply requires a change to the function that sets the routing flags and then either a call to that function to reset them or a direct change to one or more routing flags. Therefore our previous routing statement becomes much simpler and more generic :

```
If nextStep = 1 then goto step 1
Else if nextStep = 2 then goto step 2
Else if nextStep = 3 then goto step 3
Else goto step 4
```

Now, adding a new rule or amending a current one is just a change to the client side code and can be immediately implemented across all new and currently active workflows.

Managing dropdowns

Drop down lists ensure that the user can only pick from a predefined set of values, thus ensuring consistent and valid data entry. We could simply create the drop down list within the Form Template, but if we wished to add or remove one of the options, or the data was being sourced from somewhere else ? In these cases we would need to amend the template and then correct and reload all of the workflow forms where that dropdown appeared.

Two solutions to this issue present themselves, in both cases we simply create the field as a text field. In the first approach we make the field read only (as discussed previously) and then provide a button next to it which opens a new window containing the dropdown list. Once we have selected an option, its value is populated back onto the form and the pop up window closed.

In the second approach we create the drop down as a drop down with no options on the form that it is used on. Then we can have a function in our client side scripting library files – one function in which is called on page load – to populate the current set of options. There was a problem in some of the earlier releases of the WebForms module, which caused the form to error and only show the content as far as the start of the drop down list, this has, been fixed in later releases.

Validation

So far, we have looked at functionality related to setting up the form for use by an end user. However, the same concepts can apply to form validation as well. Now that we have divorced the client side code from the individual forms, we can look at more generic approach to data validation.

We can place our validation routines in the library files, this reduces the repetition of these functions throughout our workflow application, making changing them a much simpler process which does not require us to reload one or more affected WebForm views.

However, we can go one stage further and create a validation handler function in the global library file. The purpose of this function is to ensure that the correct validation functions are called depending on the form that is currently open, the validation functions themselves can rest in either the global or form specific library files. This makes updating the validation requirements of a given form much simpler.

Rather than displaying an error message for every error that occurs when we validate the form, it would be nicer to wrap them all up into a single message. To support this we can use the following Pseudo Code :

```
Create an integer to store ‘number of errors’ and set it to zero
Create a string to store the ‘error messages’
For each validation function
        If the form fails then
                Increment the ‘number of errors’
                Append a the appropriate error message to our string
        End if
Next validation
If ‘number of errors’ > 0 then
        Display a message to the user telling them how many errors and the messages
Else
        Submit the form
End if
```

This allows us to present a consistent user interface for errors to our user and also show all of the errors in one dialog rather than several or failing on the first error every time the form is submitted.

Conclusion

We now have all of our client side code in a single location it is easier to apply a whole host of changes to our workflow application in a simple, consistent and manageable way. In addition we have an approach which allows us to amend the data in a given subset of our active workflows without missing any out or making any mistakes.

Adding an Attachments Screen to a Form Step

If you use a Workflow Form step in your workflow, then he user simply see the Form View that you have selected and does not see the rest of the Workflow package, so they do not have an option to interact with the other screens such as the Attachments screen. In some cases you may need to use a Form step, but would like to provide a link to the Attachments section as well.

We start by copying the URL of the attachments screen from another step in the workflow which is a normal workflow step. Then we change the *TaskId* value in the URL, this appears in the main URL and the *NextURL* Query String attribute. These need to be changed to the TaskID value of the step that we want the new screen to appear on. A sample change is shown below change the link form Task 2 to Task 16 :

Original URL	New URL
Livelink.exe?func=work.frametaskright&workid=81010&subworkid=81010&taskid=2&nextURL=%2Flivelink9001sql%2Flivelink%2Eexe%2F%3Ffunc%3Dwork%2EframeTaskLeft%26WorkID%3D81010%26SubWorkID%3D81010%26TaskID%3D2%26NextURL%3Djavascript%3Awindow%2Eclose%28%29%3B&paneindex=3	livelink.exe?func=work.frametaskright&workid=81010&subworkid=81010&taskid=16&nextURL=%2Flivelink9001sql%2Flivelink%2Eexe%2F%3Ffunc%3Dwork%2EframeTaskLeft%26WorkID%3D81010%26SubWorkID%3D81010%26TaskID%3D16%26NextURL%3Djavascript%3Awindow%2Eclose%28%29%3B&paneindex=3

The other values that we need to change are those of the *workid* and *subworkid* which also appear in both the URL and the *NextURL* attribute. Unfortunately, we can't simply change these as we did the others as these are unique to an active workflow instance. However, we can get the values for the current workflow out of one of the hidden values in our WebForm View – ll_func.

We will need to get the values out of the field – using a substring function or similar - and then concatenate them with the rest of the URL that we already have to form a unique URL for each workflow. I would suggest that you also use the *target* attribute of the link, so that the linked screen will open in a new window.

The only other remaining problem is on the screen that we are calling. The Attachments screen has several Javascript functions that interact with functions and variables that exist in other frames. While this is fine when the screen is displayed as part of the Assignments step Frameset, it will cause Javascript errors when used on its own. This can be resolved by adding a check to see if we are in the Assignments Frameset or not before each is run. The file to change is *tgenericrightframe.html*, an amended snippet of which appears below, with the changes in red :

```
function markDirty()
{
        if (parent.FrameTop)
        {
                parent.FrameTop.markTaskEditDirty();
        }
}
```

You'll need to make this change to the markTaskEditDirty and clearDirty functions as well.

Increasing Workflow Performance

Introduction

In this section, we shall investigate ways to increase the performance of your Livelink workflow in general, this will include the workflow map(s) as well as to the forms used in the workflow.

Subworkflows

As discussed above, the use of subworkflows allows you to separate out the components of your workflow as well as to reuse certain workflows in several places within a single workflow system or across several workflow systems.

However, as mentioned above, you need to be aware of the cost of a subworkflow, as they need to be initiated, transfer any data or attachments before they begin and then perform a similar process when completed. This overhead may be more than the overhead of including those steps into their parent workflows, you will need to experiment with this to find the correct balance.

Speeding up the 'Send On' process

This process is what happens when a user presses the *Send On* button after completing a step and can sometimes be a long wait for the user or even time out the browser. If you are using a server side script to 'save' your workflow data to a datasource at each step, then you may be able to significantly reduce the amount of time this takes.

By now you should be able to produce some server side code that given a 'key' value – such as a change control number, product identifier, SKU etc – you could retrieve all the data relating to that from your custom datasource. You can use this code to populate the form when opened.

Now that we have this loading we do not need to pass the values of the fields back from the form to Livelink at every step and then from Livelink to the next form. Thus, we can incorporate a 'reset' function into the 'submit' button to reset all the fields except the Livelink required fields and your key fields. As there are now only a handful of fields with data in on your form, the time taken to process this by Livelink will be significantly shorter.

However, this method places a higher load the server side script server and the time taken for the save and load to take place may impact the benefit you gain from this method.

Custom Datasources

If you are taking advantage of the *Speeding up the 'Send On' process* tip, then you can also use it to 'correct' any workflows. By simply putting the correct information into the custom datasource and then opening one of the workflow forms – causing the load handler to run, thereby overwriting any field on the form with the latest values from the datasource.

Conclusion

Livelink provides a very powerful workflow engine that works with multiple types – attribute, PDF and Web Forms. The latter two provide an easily configurable interface for users. With some simple effort additional functionality can be easily added to the forms to provide anything from a multi datasource lookup to calculating a total on the form.