
Enhance workflow efficiency with user-configured mail merge

by Ethann Castell



Ethann Castell
Senior Consultant
Caliton Innovations

Ethann Castell is a senior consultant with Caliton Innovations, an IBM business partner in Brisbane, Australia. Since 1995, he has consulted for some of Australia's largest Notes organizations on a wide variety of projects, ranging from project management to programming in the Notes C API, and encompassing just about everything in between. He has an MBA from Macquarie University, numerous Lotus certifications (dev. and admin), and is a member of MENSA. In his spare time, Ethann's passions include traveling, rugby union, reading, and cycling. You can contact Ethann at Ethann.Castell@caliton.com.

One of my early school teachers used to say that “a good mathematician is a lazy mathematician.” In other words, given a choice of solutions that work, the best solution is the one that requires the least effort. By extension, one of the best ways to be an efficient developer is to give users as much control over an application as possible by making the application as configurable as possible.

Workflow applications have been the bread and butter of Lotus Notes development for many years. A common feature of these applications is to send e-mail notifications to process participants to inform them of some relevant event, such as a new form requiring their approval. When developing these applications, we're often given requirements for the wording for the e-mail notifications. If the wording is static and does not contain any field values (e.g., for a status), then the application can simply store the text in a configuration document and look up the text as needed. However, this solution provides only very dry, static messages with no degree of personalization or customization.

Often, developers prepare mail merges by hard coding the e-mail text into LotusScript or @Formulas. The problem with this approach is that the text invariably needs to be changed during development and testing and even after the application has gone live. To create more customized messages that address the recipient by name or include information relevant to the workflow (e.g., Dear <Submitter of the form>), we need to have some application code that combines the standard message text with field values from the workflow form — a mail-merge facility of some kind.

In this article, I show you a better solution, one that allows designated users to completely control the content and format of e-mail notifications themselves, with no developer intervention required. You can allow them to format the rich text content of the memo body through a template, with as many fancy bells and whistles as they like. You can also define a list of fields from the workflow form and allow the user to set up a mail merge by

inserting field tags (placeholders for fields from the workflow form) into the template. When your application needs to send an e-mail notification, all it needs to do is take the user-defined template, merge in the field values from the workflow form, and then send the e-mail message. Your users will be in control and happy, and you won't have to lift a finger to make changes and then retest and redeploy the application.

A sample application called Software Purchase Approval demonstrates the key concepts. (You can download this database from THE VIEW Web site.¹) After a brief orientation guide to the example workflow, you'll see how an application administrator² can customize the contents of e-mail messages sent throughout the workflow process. Then I'll take you behind the façade and explain the nuts and bolts of the code, forms, and views from a developer's perspective. I'll end by suggesting some ways that you can extend the technique of enabling users to control document content and formatting using field tags in templates.

You may find it easier to download the database now and then work through the database in conjunction with reading this article. To understand the technique and extend the example functions requires intermediate Notes development skills, including a good grasp of LotusScript.

The sample database was created with Notes release 6.5. It will not work with earlier releases.

The Software Purchase Approval application

The sample database is a simple workflow application in which requests for software purchases go through a two-level approval process. Any user can fill out a form

¹ From the Home page, click Search the Archives → Browse Issue → January/February 2006 → the title of this article. Scroll to the bottom of the page for the download files.

² Do not confuse with Lotus Notes Systems Administrators. Application administrators are business users who look after the Notes application at the business level. They typically have special privileges to perform tasks such as editing and maintaining application settings, approving forms that get "stuck" in a workflow process, running reports from an application, and so forth.

in the Software Purchase Approval database to request that the organization purchase a software package. Figure 1 shows a sample Software Purchase Request form. The form contains the few fields necessary to represent the data for our sample workflow process; you can see the field descriptions in Figure 2. After an end user fills in the fields on the Software Purchase Request form and clicks the Submit for Approval button to save it, the approval workflow process starts.

Let's look briefly at the workflow and the e-mail messages sent in the application before delving into the configuration options and development techniques that fuel it. Please note that the Software Purchase Approval application is a sample designed to highlight the configuration of e-mail message contents and the

Figure 1 Software Purchase Request form

Request form field label	Description
Submitted By	The person who created the form. The value is computed automatically.
Status	Current status of the form.
Item	The item being requested.
Additional details	Some notes about the item purchase.
Approver 1	The name of the first-level approver.
Approver 2	The name of the second-level approver.

Figure 2 Fields on the Software Purchase Request form

mail-merge functionality. Many production-quality features are missing — the most notable being document security.

For a quick reference on the design elements in this database, please see the “Design elements” sidebar below.

The approval workflow process

The Software Purchase Approval application workflow consists of the approval process mentioned earlier. In the approval workflow, there are three user roles: the person who creates and submits the request (Creator/Submitter), the person who first approves or rejects a request (Approver1), and the person who has final

Design elements of the Software Purchase Approval database

Forms		
Name	Alias	Purpose
Email Template	EmailTemplate	Used by the application administrator to configure e-mail templates.
Field Mapping	FieldMap	Used to map form fields (from the Software Purchase Request form) to user-friendly field tags.
Memo	Memo	Slightly modified version of the Notes mail memo. This form is used for viewing mail messages in the sample database rather than actually sending messages to recipients.
Software Purchase Request	SoftApproval	Used to submit and approve software purchase requests.

Views		
Name	Alias	Purpose
Approval Requests	(none)	Displays all the Software Purchase Request documents within the application.
Config>Email Field Mapping	(none)	Displays all the field mappings.
Config>Email Templates	(none)	Displays all the Email Template documents.
Mail Messages	(none)	Displays all the mail messages (memos) which are saved, rather than sent.
(Lookup Email Templates)	LookupEmailTemplates	Used by LotusScript code to look up Email Template documents.
(Lookup Field Mapping)	LookupFieldMapping	Used by LotusScript code to look up field mappings.

Script libraries		
Name	Alias	Purpose
libFormWorkflow	n/a	Contains all the LotusScript code that responds to the user actions on the Software Purchase Request form.

User role	Workflow actions
Creator/Submitter	Submit
Approver1	Approve Reject
Approver2	Approve Reject

Figure 3 User roles in the approval workflow process

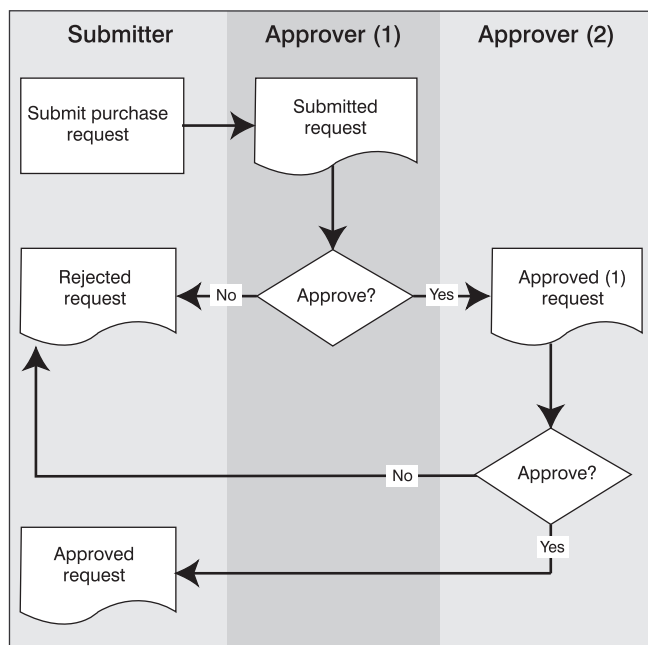


Figure 4 The software-purchase approval process

approval authority (Approver2). These user roles are defined in **Figure 3**.

Figure 4 represents the workflow approval process visually. At each stage in the flow, users take certain actions depending on their roles. These actions change the value in the Status field of the Software Purchase Request form and also trigger e-mail messages to be sent to other participants in the workflow. **Figure 5** describes these actions and their consequences.

Figure 6 shows an example of an e-mail message that the application automatically sent to the first approver (Miss Manager in this figure) after I submitted the Software Purchase Request form shown in Figure 1. Note that the message contains several pieces of information from the Software Purchase Request form, including the creator’s name (Ethann Castell), the software item being requested (MS BOB), and the name of the next approver in the process (CIO). You can see that the approver’s options are described in the body of the email and there is a doclink to take the approver to the Software Purchase Request form so that they can Approve or Reject the form.

The e-mail messages in the sample application

In the sample database, users create standard Notes mail messages with the Memo form, which is a copy of the standard Memo form from a Notes release 6 Mail database with a few fields removed for the sake of simplicity. I’ll come back to this form later.

Status on the Software Purchase Request form	User	Allowable action	Message sent To:	Status on the Software Purchase Request form becomes
Draft	Creator	Submit for Approval	Approver1	Submitted
Submitted	Approver1	Approve	Approver2	Approved1
Submitted	Approver1	Reject	Creator	Rejected
Approved1	Approver2	Approve	Creator	Approved2
Approved1	Approver2	Reject	Creator	Rejected

Figure 5 How the actions of application users change the Status field

By design, the sample application creates e-mail messages and stores them in the application database rather than sending them to real recipients. This design allows you to test the application without having to be hooked up to a mail system, and without the worry of sending test messages to real people. I'll show you later where you can change one line of code so that the application actually sends the e-mail messages to the recipients.

You can view all of the e-mail messages generated by the application in the Mail Messages view, shown in Figure 7.

Now that we're oriented to what the workflow application does, let's take a look at how an application administrator can configure the e-mail messages sent to approvers.

The Email Template form

Now we get really lazy and hand over the formatting of the e-mail messages to the application administrator. The format of the e-mail message displayed in Figure 6 is based on a user-defined template. The application

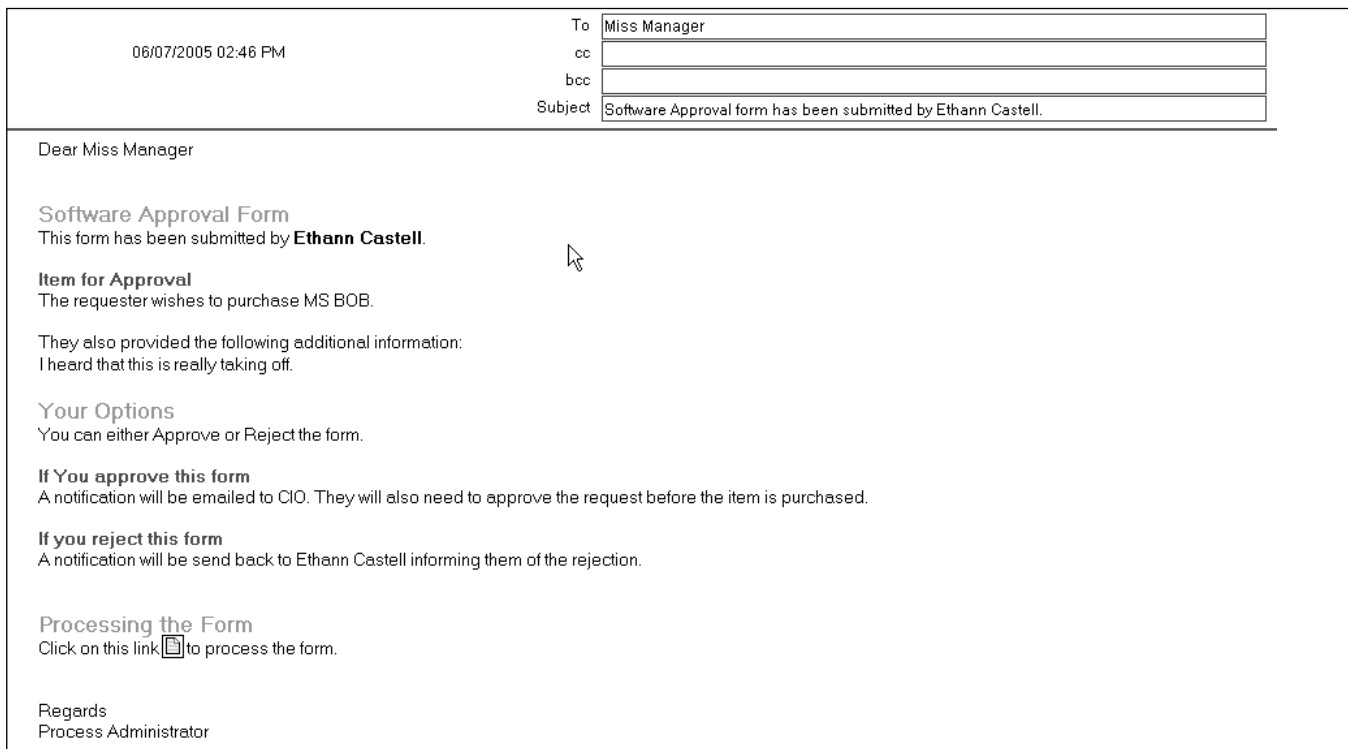


Figure 6 An example of an e-mail message generated for a first approver

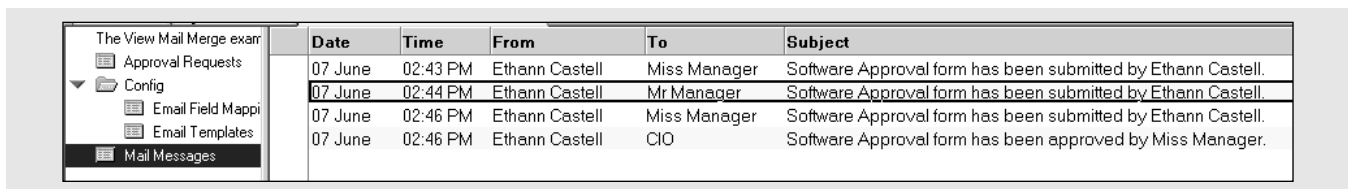


Figure 7 The Mail Messages view

administrator creates and accesses the configured template for each type of message from the Email Templates view under the Config menu in the Notes application (see Figure 8). As mentioned, there is no security on the sample application database, so anyone can access the views in this database. In a production environment, you would make configuration views accessible only to an application administrator or the application developers.

By scanning the Action column in the view, you can see that the four e-mail templates correspond to the four possible user actions throughout the approval process. The other thing to note about this view is that the templates are categorized under softApproval. This category is the Alias of the Notes form named “Software Purchase Request” used in the approval process.

The sample database includes only one approval process and therefore contains only one form, Software

Purchase Request, for approving software purchases. However, in a production environment, you may want to include several different types of approval processes and their related workflow forms in one database. In this case, you would create different Email Template documents for each stage of each of the other processes. The sample application is structured so that you can add other independent workflow forms into the one database — i.e., the categorization of the e-mail templates facilitates other templates to be linked to other forms. In applications with multiple workflow processes, the templates for each process and its related workflow form would appear categorized under the alias of the respective form.

The Email Template form for creating and configuring e-mail templates is quite simple and contains only four fields (Figure 9).

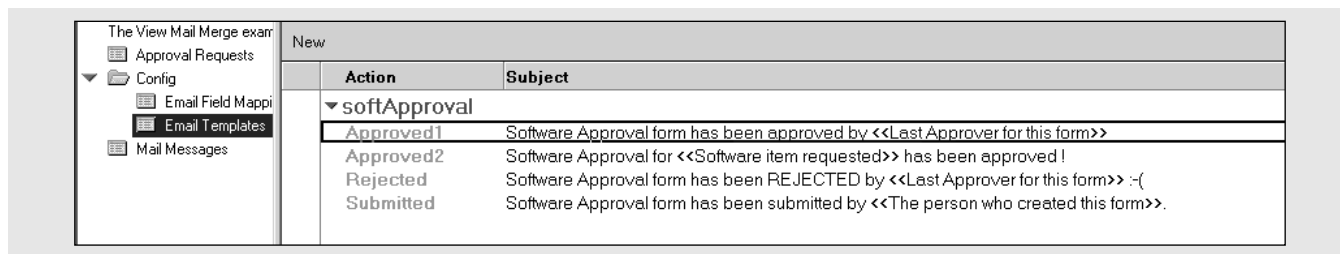


Figure 8 The Email Templates view shows the message templates currently configured for the application.

Label for field on Email Template form	Field name	Edited by	Description
Form alias	tempFormAlias	Developer	The alias of the Notes form to which this template applies.
Form Action	tempFormAction	Developer	The user action. The possible values are determined by the developer and used to identify each individual action that requires an e-mail message to be sent. These values are used in the LotusScript code that performs the mail merge and which is discussed later in this article.
Email Subject	tempEmailSubject	Business administrator/ End user	The subject of the message. The field can contain non-formatted text and field tags.
Email Body	tempEmailBody	Business administrator/ End user	The body of the message. The field can contain rich text formatting in addition to field tags.

Figure 9 Fields used on the Email Template form

Editing an e-mail template

In this technique for handling mail merge, the e-mail templates are created by the application developer and then edited by the application administrator. In the sample application, the application administrator has complete control over the format of the e-mail message body. The e-mail message body can contain any sort of rich text formatting, including tables, bold, colors, and so forth. The application administrator can also insert field tags into the format of the message.

For example, the e-mail message in Figure 6 was based on the e-mail template that is being edited in Figure 10. The fields labeled Email Subject and Email Body respectively contain the subject and body text for an e-mail message. Note that the text values within these fields also contain placeholder

tags (encased by the character strings << and >>) for fields from the Software Purchase Request form. Such tags allow the application administrator to select which fields are included and where they will appear in the e-mail message.

Note!

In a production environment, you would want to secure the editing of the Form alias and Form Action fields by using hide-when formulas. You wouldn't allow the application administrator to edit these fields as the specific values in these fields are used in LotusScript code.

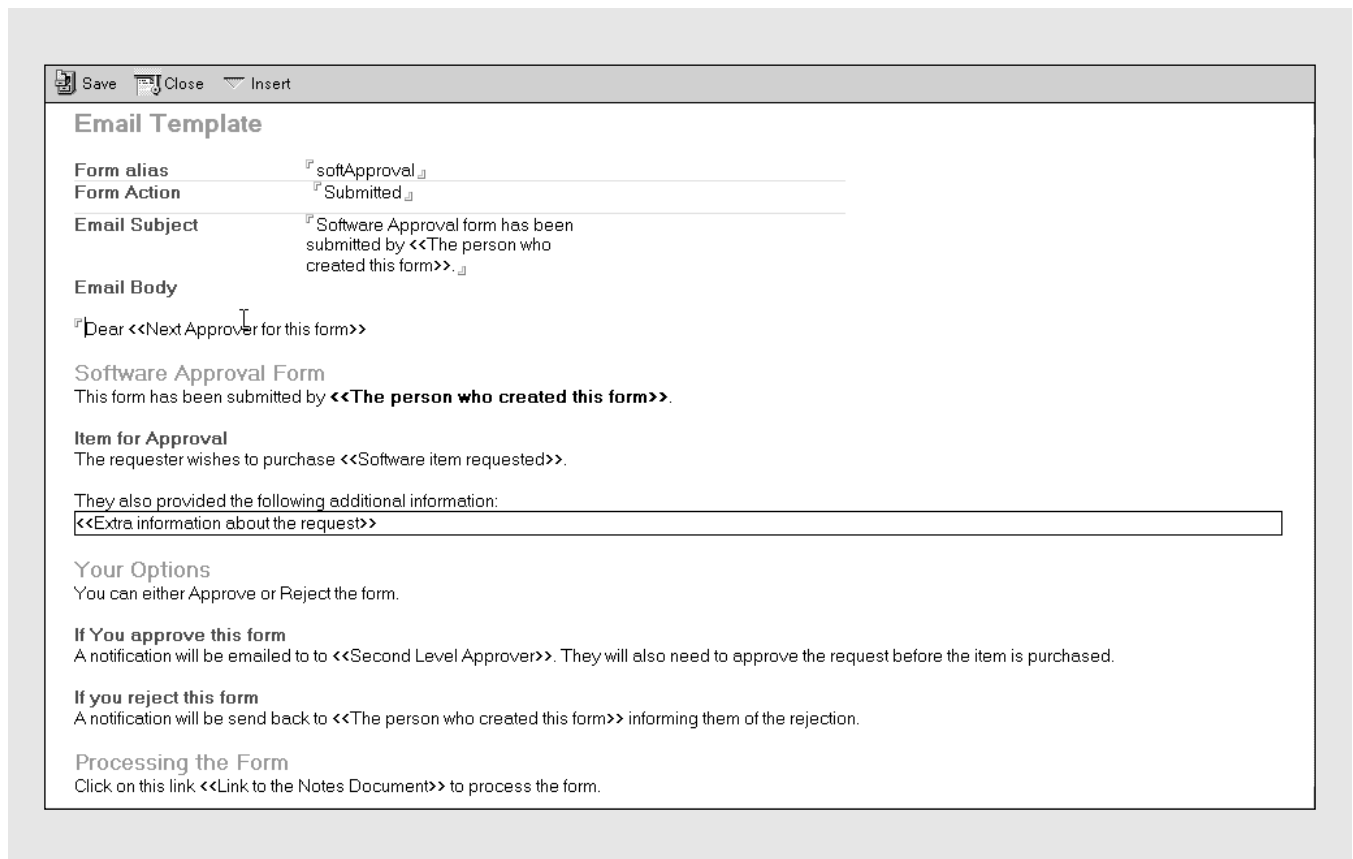


Figure 10 Editing an Email Template document

Inserting a field into the e-mail template

The application administrator can insert field tags by clicking Insert → Form field from the Email Template form’s menu (see Figure 11) and selecting from the Insert a Form Field dialog’s list of all available field tags, as shown in Figure 12. (In the next section, you’ll see how the developer creates this list.) In this case, you can see that the application administrator selected the <<FirstLevelApprover>> field tag. Figure 13 shows the e-mail template after the application administrator clicked OK and the application inserted this field tag.

The application administrator repeats the process for each field tag that needs to be included in an e-mail message. As you might expect from the Notes Doclink choice under Insert, an application administrator can also insert Notes doclinks into the Email Body field in a similar manner.

We’ve seen how to enable application administrators to control the format and contents of e-mail messages. Now let’s take look at how a developer provides the list of fields for application administrators to select for inclusion in the template.

Configuring the field tags

Developers maintain field tags via the Email Field Mapping view, which is shown in Figure 14. This view displays a list of the fields that the developer has configured for the application administrator to choose from the field-selection dialog box (as seen in Figure 12).

There are two important things to note about the field mappings. First, the application administrator can only select fields that have been mapped (i.e., fields for which the developer has created a Field Mapping document). So, developers can control which fields are “exposed” to the application administrator, and therefore, they can limit the fields that can be inserted into e-mail templates. Secondly, the application administrator selects the fields based on the field tag (as in Figure 12). The field tag can contain any text and does not have to

relate to the actual field name in any way at all. This feature is especially useful in applications with forms that contain a large number of

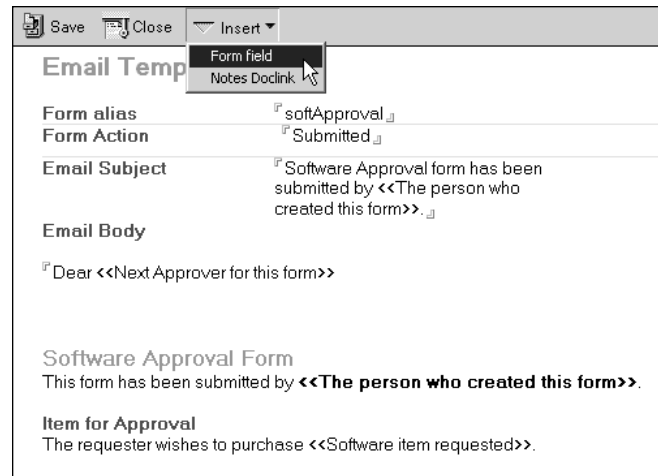


Figure 11 The Insert button

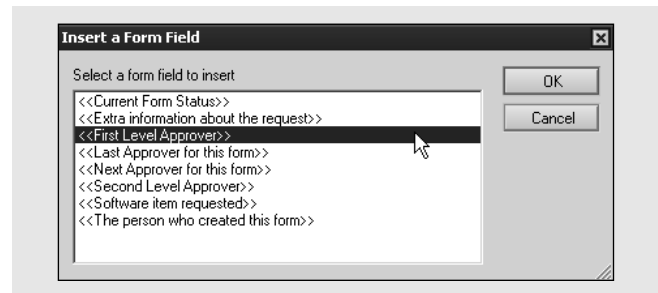


Figure 12 Selecting a field tag to insert

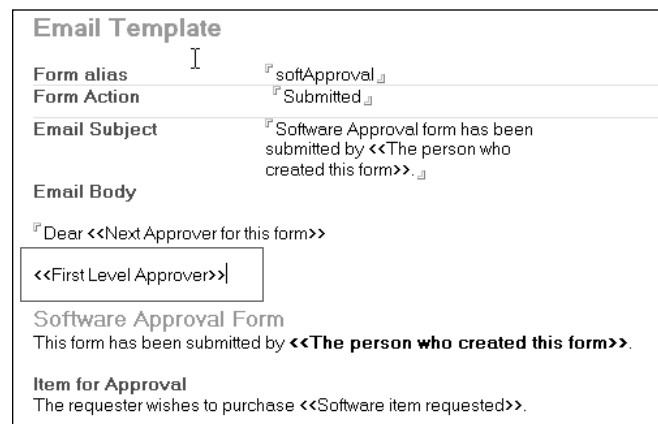


Figure 13 The Email Template document after the field insertion

Field Key	Form Field	Field Tag
Current Form Status	Status	<<Current Form Status>>
Extra information about the request	Details	<<Extra information about the request>>
First Level Approver	Approver1	<<First Level Approver>>
Last Approver for this form	LastApprover	<<Last Approver for this form>>
Next Approver for this form	NextApprover	<<Next Approver for this form>>
Second Level Approver	Approver2	<<Second Level Approver>>
Software item requested	Item	<<Software item requested>>
The person who created this form	Creator	<<The person who created this form>>

Figure 14 The Email Field Mapping view

Save Close

Field Mapping

Form field name	Status	The name of the field as it appears on the Notes form, or in the Notes document.
Field Tag	Current Form Status	This is the field tag that the user will select to insert into the e-mail template
Delimited Field Tag	<<Current Form Status>>	This is how the field tag will appear in the e-mail template after the user has inserted the field. i.e. the field tag contained within the tag delimiters.
Conversion Rules	<input checked="" type="radio"/> None <input type="radio"/> Common Name	How to convert the field value when we send the e-mail message.

Figure 15 The Field Mapping form

Field label on the Field Mapping form	Description
Form field name	The name of the field from the Notes form or document.
Field Tag	The user-friendly value used to represent this field to the end user.
Delimited Field Tag	Display only. The field tag as it will appear when inserted into an e-mail template.
Conversion Rules	The type of conversion that should be applied to the field value when the message is sent: <ul style="list-style-type: none"> - None: perform no conversion. - Common Name: convert this value to its common name.

Figure 16 Fields on the Field Mapping form

ambiguously named fields. For example, “The person who created this form” is much more user-friendly than “Creator.”

To add a new field mapping, click the New Field Mapping button. The Field Mapping form shown in **Figure 15** will open.

To configure a field mapping, fill out the form’s four fields, which are described in **Figure 16**; then click Save.

So far, we’ve looked at the elements of the database visible to users. Now, I’ll show you the inner workings of the database. We’ll examine the code for the Software Purchase Request form and the libFormWorkflow script library. Note that this script library makes use of two hidden views, (Lookup Email Templates) and (Lookup Field Mapping).

The Software Purchase Request form

The Software Purchase Request form contains a small amount of LotusScript code. The code that does most of the form's work is contained within a custom LotusScript class called formSoftware, which is stored in the libFormWorkflow script library. The formSoftware class encapsulates all the functions of the form and contains the code that performs the mail-merge operation. More on this later — first, let's examine the key elements of this form.

Globals section

The Globals section of the form includes the LotusScript code shown in **Figure 17**. The code performs three tasks:

- Includes the code from the libFormWorkflow script library (**line 4**)
- Declares an object variable of the class type formSoftware (**line 6**)
- Creates a new instance of the formSoftware class and assigns it to the variable thisForm (**line 8**)

Note that the global variable thisForm is used throughout the LotusScript code in the Software Purchase Request form.

PostOpen event

When the Software Purchase Request form opens, the PostOpen event calls the initForm method of the thisForm object (see **Figure 18**). This method simply initializes some of the class variables within the object.

Action buttons

The three actions buttons on the form correspond to the workflow actions that users can perform. As you can see in **Figure 19**, each button calls the ProcessForm

```

1. (Options)
2. Option Public
3. Option Declare
4. Use "libFormWorkflow"

5. (Declarations)
6. Dim thisForm As formSoftware

7. Sub Initialize
8.   Set thisForm = New formSoftware
9. End Sub

```

Figure 17 The code in the Globals section of the Software Purchase Request form

```

1. Sub Postopen(Source As Notesuidocument)
2.   Call thisForm.initForm()
3. End Sub

```

Figure 18 The PostOpen event of the Software Purchase Request form

'Submit For Approval button

```

1. Sub Click(Source As Button)
2.   thisForm.ProcessForm("Submit")
3. End Sub

```

'Approve button

```

4. Sub Click(Source As Button)
5.   thisForm.ProcessForm("Approve")
6. End Sub

```

'Reject button

```

7. Sub Click(Source As Button)
8.   thisForm.ProcessForm("Reject")
9. End Sub

```

Figure 19 The Action button code of the Software Purchase Request form

method of the thisForm object and passes a unique identifier corresponding to the user action that has taken place (lines 2, 5, and 8).

The libFormWorkflow script library

The libFormWorkflow script library contains two custom items, the formSoftware class and the ReplaceSubString function. As stated earlier, most of the action takes place using the formSoftware class, which encapsulates the approval processes associated with the form. The ReplaceSubString subroutine, as its name suggests, takes a string and replaces one of its substrings with another string. In other words, it's the LotusScript equivalent of @ReplaceSubString. Let's look at each of these custom items more closely.

The formSoftware class

The formSoftware class has three main methods:

- **ProcessForm** — determines which action the user has taken on the workflow form and performs the necessary steps for that action.
- **SendEmail** — creates and saves the new e-mail messages, with the help of MergeTemplate.
- **MergeTemplate** — populates the body of the message using fields from the Software Purchase Request form.

These methods are called in a hierarchical fashion, as shown in Figure 20. You can see the complete code for this class listed in Figure 21. I'll explain the parts of the code that create these three crucial methods next.

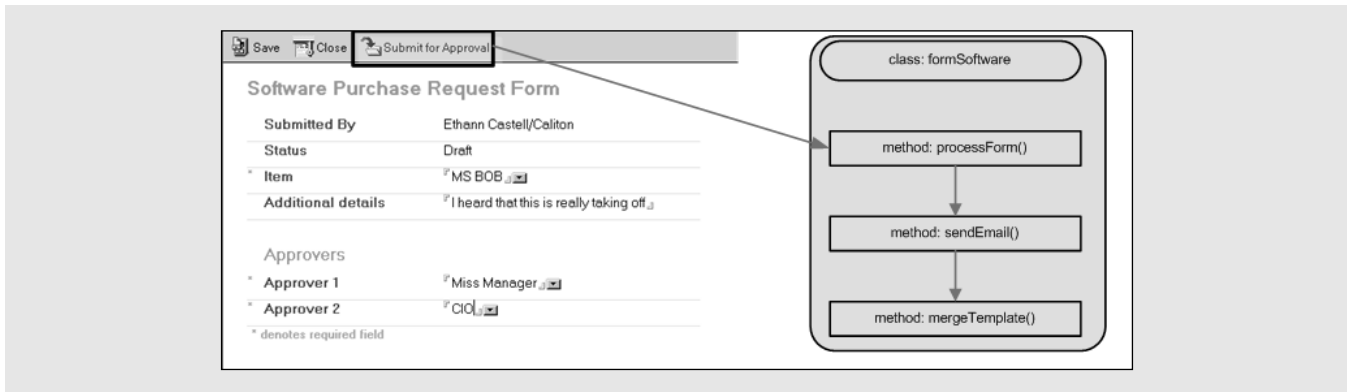


Figure 20 The method-calling hierarchy within the formSoftware class

```

1. Options
2. Option Public
3. Option Declare
4. %INCLUDE "Isconst.lss"

5. Declarations
6. '-----
7. ' class: formSoftware
8. ' =====

```

Figure 21 The source code for the libFormWorkflow script library

Continues on next page

Figure 21 continued

```

9. ' Date last updated: 12/20/2005
10. ' Updated by: Ethann Castell
11. '
12. ' Description
13. ' -----
14. ' This class encapsulates the functionality of Software Purchase Request form.
15. '-----
16. Class formSoftware
17. Private session As notessession
18. Private workspace As notesuiworkspace
19. Private dbForms As NotesDatabase
20. Private uidocForm As NotesUIDocument
21. Private docForm As notesdocument
22. Private sFormName As String
23. Private sFormAction As String

24. '-----
25. ' initForm()
26. '=====
27. '
28. ' Parameters
29. '-----
30. ' None
31. '
32. ' Return Value
33. ' -----
34. ' None
35. '
36. ' Description
37. ' -----
38. ' Initialize class-level variables and get handles to the workspace, session, uiForm,
    and notes document.
39. '-----
40. Sub initForm
41.     Set session = New notessession
42.     Set workspace = New notesuiworkspace
43.     Set dbForms = session.CurrentDatabase
44.     Set uidocForm = workspace.CurrentDocument
45.     Set docForm = uidocForm.document
46.     uidocForm.AutoReload= True
47. End Sub

48. '-----
49. ' processForm()
50. '=====
51. ' Parameters
52. '-----

```

```

53. ' pAction - a code for the user action that has occurred.
54. ' e.g., if pAction = "Submit" then the user has submitted the form for approval.
55. '
56. ' Return Value
57. ' -----
58. ' None.
59. '
60. ' Description
61. ' -----
62. ' This method is called by action buttons on the uiForm.
63. ' First we determine which action the user has taken.
64. ' Then, depending on the user action, we set the document status and next approver.
65. ' Finally we send an e-mail to the next approver based on the e-mail template for this
    action.
66. '-----
67. Sub processForm( pAction As String)
68.     Dim sResult As String

69.     ' Call a routine depending on the user action
70.     Select Case pAction

71.         ' User has submitted the form
72.         Case "Submit":
73.             ' If the form passes validation then
74.             If FormValidationOk( ) Then
75.                 docForm.Status = "Submitted"
76.                 docForm.LastApprover = ""
77.                 docForm.NextApprover = docForm.Approver1(0)
78.                 Call uidocForm.Save()
79.                 Call uidocForm.Close()
80.                 ' Send and e-mail to the next approver
81.                 Call SendEmail("Submitted" )
82.             End If

83.         Case "Approve":
84.             If docForm.Status(0) = "Submitted" Then
85.                 docForm.Status = "Approved1"
86.                 docForm.LastApprover = docForm.NextApprover(0)
87.                 docForm.NextApprover = docForm.Approver2(0)
88.             Else
89.                 docForm.Status = "Approved2"
90.                 docForm.LastApprover = docForm.NextApprover(0)
91.                 docForm.NextApprover = ""
92.             End If
93.             Call uidocForm.Save()
94.             Call uidocForm.Close()
95.             Call SendEmail(docform.Status(0) )

```

Continues on next page

Figure 21 continued

```

96. Case "Reject":
97.     docForm.Status = "Rejected"
98.     docForm.LastApprover = docForm.NextApprover(0)
99.     docForm.NextApprover = ""
100.    Call uidocForm.Save()
101.    Call uidocForm.Close()
102.    docForm.NextApprover = docForm.Creator(0)
103.    Call uidocForm.Save()
104.    Call uidocForm.Close()
105.    Call SendEmail("Rejected" )

106. End Select

107. End Sub

108. '-----
109. ' SendEmail()
110. '=====
111. ' Parameters
112. '-----
113. ' None
114. '
115. ' Return Value
116. ' -----
117. ' None
118. '
119. ' Description
120. ' -----
121. '
122. '-----
123. Sub SendEmail( EmailKey As String)
124.     Dim docMail As New notesdocument(session.CurrentDatabase )
125.     docMail.form = "Memo"
126.     Call MergeTemplate( docForm, docMail, EmailKey)
127.     docMail.SendTo = docForm.NextApprover(0)

128.     '*****
129.     ' Change the code here if you actually want to send the e-mail messages.
130.     '         Call docMail.send( False )
131.     Call docMail.Save(True, False)
132.     '*****

133. End Sub

134. '-----
135. ' MergeTemplate()
136. '=====

```

```

137. ' Parameters
138. '-----
139. ' None
140. '
141. ' Return Value
142. '-----
143. ' None
144. '
145. ' Description
146. '-----
147. '
148. '-----
149. Sub MergeTemplate( docForm As notesdocument, docEmail As
    notesdocument,Byval EmailKey As String)

150.   Dim viewMapping, viewTemplate As notesview
151.   Dim vecMapping As notesViewEntryCollection
152.   Dim veMapping As notesViewEntry
153.   Dim docTemplate As notesdocument
154.   Dim rtitem As Variant
155.   Dim sTemplateKey As String
156.   Dim sSearch As String
157.   Dim sReplace As String
158.   Dim sFieldName As String
159.   Dim i As Integer
160.   Dim sConvert As String
161.   Dim nameTemp As NotesName
162.   Dim itemTemplateBody As NotesItem
163.   Dim rtBody As notesrichtextitem
164.   Dim rtnav As NotesRichTextNavigator
165.   Dim rtrange As NotesRichTextRange

166.   Set viewMapping = dbForms.getView("LookupFieldMapping")
167.   Set vecMapping = dbForms.getView("LookupFieldMapping").AllEntries
168.   Set viewTemplate = dbForms.getView("LookupEmailTemplates")

169.   ' First get the e-mail template.
170.   sTemplateKey = docForm.Form(0) + "~" + EmailKey
171.   Set docTemplate = viewTemplate.getDocumentByKey( sTemplateKey )
172.   If (docTemplate Is Nothing ) Then
173.       ' Template could not be found. Add code to Log Error or take other
           action.
174.       Goto ExitSub
175.   End If

176. '-----
177. ' Process the Email Subject field.

```

Continues on next page

Figure 21 continued

```

178. ' We look for all the mapped fields even though they may not all be used in the
      Subject field.
179. '
180. docEmail.Subject = docTemplate.tempEmailSubject(0)
181. For i = 1 To vecMapping.count
182.     ' Get the mapped field.
183.     Set veMapping = vecMapping.GetNthEntry(i)
184.     ' Search value id the User Key i.e., the field wrapped i "<<" and ">>"
185.     sSearch = veMapping.ColumnValues(0)
186.     ' Get the actual field name.
187.     sFieldName = veMapping.ColumnValues(1)
188.     ' Get the field value for the document.
189.     sReplace = docForm.GetItemValue( sFieldName )(0)
190.     ' Convert the field if we need to.
191.     sConvert = veMapping.ColumnValues(2)
192.     If sConvert = "Common Name" Then
193.         Set nameTemp = New NotesName(sReplace)
194.         sReplace = nameTemp.Common
195.         Delete nameTemp
196.     End If

197.     ' Replace the field tag with the actual field value and store the result in
      the subject of the Mail memo.
198.     docEmail.Subject = ReplaceSubstring( docEmail.subject(0), sSearch,
      sReplace )
199. Next
200. '-----
201. ' Process Email Body.
202. ' Copy the template Body to the mail memo. NB the Body field will still
      contain all the tags at this stage.
203. Set itemTemplateBody = docTemplate.GetFirstItem( "tempEmailBody" )
204. Call itemTemplateBody.CopyItemToDocument( docEmail, "Body" )

205. Set rtBody = docEmail.GetFirstItem("Body")
206. Set rtnav = rtBody.CreateNavigator
207. Call rtnav.FindFirstElement(RTELEM_TYPE_TEXTPARAGRAPH)
208. Set rtrange = rtBody.CreateRange

209. ' Now replace all the tags in the Body field with the actual field values.
210. For i = 1 To vecMapping.count
211.     Set veMapping = vecMapping.GetNthEntry(i)
212.     sSearch = veMapping.ColumnValues(0)
213.     sFieldName = veMapping.ColumnValues(1)
214.     sReplace = docForm.GetItemValue( sFieldName )(0)
215.     sConvert = veMapping.ColumnValues(2)
216.     If sConvert = "Common Name" Then
217.         Set nameTemp = New NotesName(sReplace)

```

```

218.         sReplace = nameTemp.Common
219.         Delete nameTemp
220.     End If
221.     ' Need to set rrange here as it is reset after each call to FindAndReplace.
222.     Set rrange = rtBody.CreateRange
223.     Call rrange.FindAndReplace (sSearch, sReplace, RT_REPL_ALL)
224.     ' Update before looping.
225.     Call rtBody.Update
226. Next

227. ' Replace the Doclinks in the body, if there is one.
228. ' This is quite a long winded way of doing it, but I'm not sure that it can be
    optimized much more.
229. ' Basically we search for the tag and then append the Doclink to the rich text
    field at the point after the tag.
230. ' Next we reset all the RT objects and then do a search and replace on the tag,
    replacing it with blank.
231. ' The various RT classes interact with each other and often get reset after
    performing a function.
232. ' Be careful if changing this code because if you don't get it exactly right then
    Notes is likely to bring up the good old Red Screen of Death.
233. ' Read up the help documentation before jumping in.

234. ' Search for the DocLink tag.
235. If rtnav.FindFirstString("<<Link to the Notes Document>>",
    RT_FIND_CASEINSENSITIVE) Then
236.     Call rtBody.BeginInsert(rtnav, True)
237.     Call rtBody.AppendDocLink( docForm,"" )
238.     Call rtBody.EndInsert
239.     Call rtBody.Update

240.     Set rtnav = rtBody.CreateNavigator
241.     Call rtnav.FindFirstElement(RTELEM_TYPE_TEXTPARAGRAPH)
242.     Call rtnav.FindFirstString("<<Link to the Notes Document>>",
    RT_FIND_CASEINSENSITIVE)
243.     Set rrange = rtBody.CreateRange      ' Need to set rrange here as
    it is reset after each call to FindAndReplace.
244.     Call rrange.setbegin(rtnav)
245.     Call rrange.setend(rtnav)
246.     Call rrange.remove

247.     ' Insert the doclink into the rich text field.
248.     Call rtBody.Update

249. End If

250. Call rtBody.Compact

```

Continues on next page

Figure 21 continued

```

251. ExitSub:

252. End Sub

253. '-----
254. ' FormValidationOk()
255. '=====
256. ' Parameters
257. '-----
258. ' None
259. '
260. ' Return Value
261. ' -----
262. ' True or False:
263. ' True - All fields on the form passed validation
264. ' False - At least on of the fields on the form failed validation
265. '
266. ' Description
267. ' -----
268. ' This is a simple validation routine that checks that values have been entered for
    three fields;
269. ' Item, Approver1 and Approver 2.
270. ' If any of the fields are blank, then the routine displays a dialog box indicating the
    problem.
271. '-----
272. Function FormValidationOk() As Integer
273.     Dim iValid As Integer
274.     Dim sDialogTitle As String

275.     sDialogTitle = "Form field validation"

276.     ' Assume that the form will not pass validation.
277.     iValid = False
278.     If docForm.Item(0) = "" Then
279.         MsgBox "Please select an item", MB_OK, sDialogTitle
280.     Else
281.         If docForm.Approver1(0) = "" Then
282.             MsgBox "Please select an Approver1", MB_OK, sDialogTitle
283.         Else
284.             If docForm.Approver2(0) = "" Then
285.                 MsgBox "Please select an Approver2", MB_OK, sDialogTitle
286.             Else
287.                 ' We made it ! All required fields are entered.
288.                 iValid = True
289.             End If
290.         End If

```

```

291. End If

292. FormValidationOk = iValid
293. End Function

294. End Class
295. '
296. ' End of Class: formSoftware
297. '
298. Function ReplaceSubstring ( Byval fullString As String, oldString As String,
    newString As String )
299. Dim position As Integer, lenOldString As Integer

300. lenOldString = Len(oldString)
301. position = Instr ( fullString, oldString )
302. Do While position > 0 And oldString <> ""
    fullString = Left ( fullString, position-1) & newString & Mid (
        fullString, position + lenOldString )
    position = Instr ( position + Len(newString), fullString, oldString )
303. Loop
304. ReplaceSubstring = fullString
305. End Function

```

The ProcessForm method

The value for the action the end user makes in the Software Purchase Request form is passed as a String named pAction in **line 67**. The value comes from the button code shown in Figure 19 and can be one of the following: “Submit”, “Approve”, or “Reject”. In **line 70**, a Select Case statement determines which action has taken place and the steps that will be performed as appropriate for each action. The code for each of the three actions is very similar, so I’ll only walk through the code for the Submit action.

When the pAction value is Submit (**line 72**), the ProcessForm method calls a routine to validate the user input (**line 74**; see **lines 272 – 293** for the validation routine). If Validation fails, then an error message is displayed to the user and no further processing occurs.

On **line 75**, the ProcessForm method sets the approval status of the form to Submitted. On **line 76**, the code sets the LastApprover field to be blank

because, at this stage, there is no previous approver. Code on **line 77** sets the NextApprover field to the value the user entered into the Approver1 field on the Software Purchase Request form. **Lines 78 and 79** save and close the form.

Line 81 calls the SendEmail method to send the e-mail notification with the current user action (Submitted) as a parameter.

Lines 83 – 106 process the Approve and Reject actions in a similar manner.

The SendEmail method

On **lines 124 and 125**, the SendEmail method uses the Memo form to create a new e-mail message.

On **line 126**, SendEmail calls the MergeTemplate routine to merge the e-mail template with the fields from the Software Purchase Request form. It calls the MergeTemplate method with three parameters:

- **docForm** — the Software Purchase Request document where the buyer or approver is performing an action (Submit, Approve, or Reject)
- **docMail** — the mail memo
- **EmailKey** — a string representing the action that the user has performed

Line 127 sets the recipient of the mail memo to be the value contained in the NextApprover field of the Software Purchase Request form.

Finally, **line 131** saves the memo.

Note!

Line 131 saves the memo in the workflow database. If you want to actually send the memo, then comment out line 131 and uncomment line 130.

The MergeTemplate method

The MergeTemplate method populates the body of the e-mail message. It perform five broad steps:

1. Declares and initializes the variables used in the method
2. Gets the appropriate e-mail template based on the user action
3. Merges the Subject field of the memo with the fields from the Software Purchase Request form
4. Merges the Body field of the memo with the fields from the Software Purchase Request form
5. Inserts a doclink in the Body field of the memo

1. Declare and initialize the variables used in the method

Lines 150 – 165: Declare the variables used in this method.

Line 166: Open the Lookup Field Mapping view.

Line 167: Get all the entries from the Lookup Field Mapping view and store them in a ViewEntryCollection object.

I used a ViewEntryCollection object to improve performance. I could have just used the view, but it would have incurred the overhead of opening each document to get document values. ViewEntryCollection allows us to get values straight from the view.

Line 168: Open the view containing the e-mail templates (LookupEmailTemplates).

2. Get the e-mail template

Lines 169 – 175: Look up the e-mail template for the Software Purchase Request form. The lookup key is based on a combination of the form alias and the end user's action. If the e-mail template is not found, then this method exits. In a production environment, you would want to return an error to the calling method.

3. Populate the Subject field of the memo

Lines 180 – 199: Merge fields from the Software Purchase Request form with field tags in the e-mail template.

Line 180: Copy the Subject field from the e-mail template to the memo. This field will contain the user-defined text and field tags. We need to replace these field tags with values from the Software Purchase Request form.

Lines 181 – 199: Loop through all of the entries in the LookupFieldMapping view. We want to perform a search and replace for each field that could be mapped.

Line 183: Get the LookupFieldMapping entry from the ViewEntryCollection object.

- Line 185:** Get the field tag as it would appear in the text of the Subject field (e.g., <<The person who created this form>>).
- Line 187:** Get the corresponding name of the field on the Software Purchase Request form (e.g., Creator).
- Line 189:** Get the value of the field from the Software Purchase Request form. For example, if the current field mapping contains the value “Creator,” then we would get the value of the Creator field from the Software Purchase Request form.
- Lines 190 – 196:** If necessary, convert the field value to a common name.
- Line 198:** Replace the field tag (from the e-mail template) with the actual field value from the Software Purchase Request form.
- 4. Populate the Body field of the memo**
- Lines 203 – 226:** Merge the fields from the Software Purchase Request form with field tags in the body of the e-mail template. This is similar to the way we merged field tags for the Subject field, but because the Body field is a rich text field, we need to use NotesRichText classes.
- Lines 203 – 204:** Copy the Body field from the e-mail template to the memo. This field will contain the user-defined text and field tags. We need to replace these field tags with values from the Software Purchase Request form.
- Line 205:** Get a handle on the Body field of the memo.
- Lines 206 – 208:** Create a NotesRichTextNavigator object for the Body field. This class allows us to search (and replace) within the rich text field.
- Lines 210 – 226:** Loop through all of the entries in the LookupFieldMapping view. We want to perform a search and replace for each field that is mapped.
- Line 211:** Get the LookupFieldMapping entry from the ViewEntryCollection object.
- Line 212:** Get the field tag as it appears in the e-mail template (e.g., <<The person who created this form>>).
- Line 213:** Get the actual field name as it will be in the Software Purchase Request form (e.g., Creator).
- Line 214:** Get the value of the field from the Software Purchase Request form. For example, if the current field mapping contains the value “Creator,” then we would get the value of the Creator field from the Software Purchase Request form.
- Lines 215 – 220:** If necessary, convert the field value to be a common name.
- Line 222:** Set the range for the NotesRichText operation. This range has to be redefined each time as the FindAndReplace method will reset the range to null after it has completed.
- Line 223:** Perform the search and replace operations within the RichText item. The FindAndReplace method takes three parameters:
- The string to search for
 - The string to replace with
 - The scope of the replace operations. In this case we specify RT_REPL_ALL, which tells the method that we want to replace all occurrences of the search string.
- Line 225:** Call the Update method of the RichText item. This step is important; otherwise, the changes made by the FindAndReplace method may not be applied to the rich text field.

Important!

You must call the Update method of the NotesRichText item. This step is necessary because according to Domino Designer Help, “Operations on a RichText item are queued for efficiency. The order and time of completion are not predictable. Use this method to ensure that processing is complete at a certain point.”

5. Insert doclinks into the Body field

Inserting doclinks into the Body field of the memo is similar to replacing the field tags with field values. However, we cannot simply use the FindAndReplace method. Instead, we need to find the doclink tag in the body, insert a doclink following the tag, and then erase the tag.

Note!

Doclinks are hard-coded to use the tag “<<Link to the Notes Document>>”.

The example is set up to insert a maximum of one doclink per e-mail message.

First, we need to locate the doclink tag and then insert the doclink into the e-mail body:

- Line 235:** Search for the doclink tag within the Body field.
- Line 236:** Set an insertion point into the RichText item.
- Line 237:** Insert the doclink into the RichText item.
- Line 238:** Close the insertion point.
- Line 239:** Update the RichText item with the changes made by the RichTextNavigator object.

Next, we need to remove the doclink tag from the body:

- Line 240:** Create a new RichTextNavigator object.
- Lines 241 – 242:** Locate the doclink tag in the Body item.
- Line 243:** Create a RichTextRange object. We use this range to remove the tag.
- Lines 244 – 245:** Set the beginning and end of the range to match the beginning and end of the doclink tag.
- Line 246:** Remove the doclink tag.
- Line 248:** Tell Notes to process all outstanding operations on the RichText item.
- Line 250:** In rare cases, performing RichText operations can add extra space into the RichText item. The Compact method removes any extra space, which is not absolutely necessary but considered to be good practice.

Warning!

I use three classes to manipulate the rich text Body field: NotesRichTextItem, NotesRichTextNavigator and NotesRichTextRange. Used correctly, these classes work beautifully. However, the latter two classes are not the most robust LotusScript classes around. If you don't call exactly the right methods in exactly the right order, Notes will more than likely crash with the infamous “Red Screen of Death.” To get these classes to play nicely together, first you need to read thoroughly the class documentation in the Domino Designer Help file. The documentation of these classes is somewhat lacking, and you may need to perform some old-fashioned trial and error to get the result you are looking for.

Enhancements and other uses

This section lists enhancements you could consider making to the libFormWorkflow code, plus some other uses for the general approach to putting users in charge of configuring templates containing field tags.

Here are some of the enhancements that you could consider making for your applications:

URL doclinks

Allow the user to insert URL doclinks instead of, or in addition to, Notes doclinks.

Template preview function

Allow users to test the layout of the e-mail template as they are editing it. Similar to a print preview, a template preview would require merging the current e-mail template with a dummy set of data and then opening the resulting memo on the screen for the user to view.

Configuration of recipients

You could modify the Email Template form to allow the application administrator to select which persons/roles should be included on the To:, CC:, and BCC: fields of the memo.

Standard headers or footers

You could consider allowing the application administrator to maintain standard headers or footers (e.g., a standard disclaimer footer) for use across multiple e-mail templates.

Performance enhancements

In the example implementation, the MergeTemplate method processes all mapped fields regardless of whether or not they are in the template. A more efficient though slightly more complex way to process fields may be to locate the tags first and

then only process the tags that are actually in the template.

Error handling

In a production environment, you would want to add better error handling and error recovery to the application.

Logging and error handling

You may also want to add code that logs each e-mail message that is sent.

Additional data conversions

The example in this article only allowed for one type of data conversion: converting a name to a common name. You may wish to allow your users more conversion options (e.g., conversion to upper or lowercase, date formatting, etc.).

Other uses for this approach

Although we have focused on sending e-mail messages from a workflow application, you can apply the underlying technique (using fields tags in templates to merge the templates with real data) in several different ways, including:

E-mail newsletters

Allow users to send personalized e-mail newsletters to a subscription list.

Report layout

Allow your end users to create their own report formats.

Data extraction and exporting

Allow your end users to export data from applications based on a user-defined list of fields.

Conclusion

This article has demonstrated a general approach for providing solutions that involve three main components:

- **Field mapping** — to expose field values to the end user
- **User-defined templates** — for positioning and laying out the fields
- **A merge engine** — to merge the field values with the mail template

The main benefit of this approach is that it allows application administrators and other designated users

to configure and lay out templates the way they want them. In addition to putting the application administrator in control, it also leaves developers free to work on tasks that are more interesting than making changes to mail memo formats. Furthermore, when the system is live, application administrators can continue to change e-mail templates without the developer having to change any application code, and therefore, without having to get Domino administrators to roll out another version of the database every time a change is made. Developers do not totally lose control of the situation, as they only expose certain field values to application administrators. For the most part, however, when you have set up this sort of system, you're available for the next project that comes along.