Lab 7: Building Custom Routers

**Lab Time: 60 Minutes**

**Lab Directory: ECM401.CustomRouting**

**Lab Overview:**

In this lab, you will extend the record routing capability of the Litware Record Center to redirect, track and filter incoming records based on document metadata as well as custom business rules. This will require that you create several custom routers that will be installed by a feature that can be activated on a given record center site.

# Exercise 1: Creating a SharePoint Feature project

1. Start by creating a new **SharePoint Feature** project in Visual Studio. Give it the name **ECM401.CustomRouter** . Set the feature scope to **Web** so that it can be activated on any SharePoint website, as shown below.
2. Your feature will take advantage of utility code from the ECM401 class library located in the **student/resources/tools** folder. Add that project to your solution and then add a reference to it and to the **System.Configuration.Install** assembly that the ECM401 project depends on. Since you will also be relying on classes in the **Microsoft.Office.RecordsManagement** namespace, you will also need to add a reference to the **Microsoft.Office.Policy.dll** file. When you are finished, your project should resemble the following illustration.

# Exercise 2: Creating a Simple Filtering Router

The first router you create will simply filter incoming records based on incoming metadata. This router will examine selected properties associated with each incoming record and then decide whether to accept or reject the file based on their values. In this example, the filtering rules will be hardcoded. In actual practice, you would read the rules in from an external file or database system.

1. Add a new class to your project called **FilteringRouter** and open the new code file for editing. Add the following lines to the set of **using** statements at the top of the file.

using System.Diagnostics;  
using Microsoft.Office.RecordsManagement.RecordsRepository;  
using WSS=Microsoft.SharePoint;  
using ECM401.RecordsManagement;

Note: You need the **WSS=** prefix because there is a conflict between the **RecordsRepositoryProperty** object declared in the *Microsoft.SharePoint* namespace and an object of the same name declared in the *Microsoft.Office.RecordsManagement.RecordsRepository* namespace.

1. The **ECM401.RecordsManagement** namespace contains utility classes that simplify the construction of various SharePoint components. One of these is the **SharePointRouter** base class that understands how to install and uninstall custom routers and also provides a default implementation of the **IRouter** interface that all custom routers must implement. You will derive your custom router from this abstract base class and then override the **OnSubmitFile** method.
2. Modify the class declaration so that it matches the following code.

[Name("ECM401 Filtering Router")]  
public class FilteringRouter : SharePointRouter  
{  
}

Note: The **NameAttribute** is a custom attribute provided in the ECM401 utility library for attaching a name to any type. The **SharePointRouter** base class looks for this attribute and uses it to determine the router name when the custom router is installed.

1. The **IRouter** interface declares a single method called *OnSubmitFile* that will contain your custom routing logic. Insert the following code to the class definition.

Code Snippet: 'FilteringRouter - OnSubmitFile'

/// <summary>  
/// Custom implementation that validates the submitted files contents.  
/// </summary>  
protected override RouterResult OnSubmitFile(  
 string recordSeries,   
 string sourceUrl,   
 string userName,   
 ref byte[] fileToSubmit,   
 ref RecordsRepositoryProperty[] properties,   
 ref WSS.SPList destination,   
 ref string resultDetails)  
{  
 // setup the default result...  
 RouterResult result = RouterResult.SuccessContinueProcessing;  
 try  
 {  
 if (!(ValidateContent(ref resultDetails, ref fileToSubmit)  
 && ValidateMetadata(ref resultDetails, ref properties)))  
 {  
 result = RouterResult.RejectFile;  
 }  
 }  
 catch (Exception x)  
 {  
 EventLog.WriteEntry("FilteringRouter", String.Format("Exception occurred: {0}", x.Message));  
 // Cancel if we encounter problems.  
 result = RouterResult.SuccessCancelFurtherProcessing;  
 }  
 return result;  
}

1. Next, you will apply custom business rules to validate the content and metadata of each incoming record using two utility methods called **ValidateContent** and **ValidateMetadata** . Add the following code to the class definition.

/// <summary>  
/// Checks the file content for validity. This can be any algorithm you like.  
/// </summary>  
/// <param name="resultDetails"></param>  
/// <param name="fileToSubmit"></param>  
/// <returns></returns>  
bool ValidateContent(ref string resultDetails, ref byte[] fileToSubmit)  
{  
 return true;  
}

1. Complete the router implementation by adding the following code to the class definition.

Code Snippet: 'Filtering Router - ValidateMetadata'

/// <summary>  
/// Checks the metadata properties for validity and consistency.  
/// </summary>  
/// <param name="resultDetails"></param>  
/// <param name="properties"></param>  
/// <returns></returns>  
bool ValidateMetadata(ref string resultDetails, ref RecordsRepositoryProperty[] properties)  
{  
 foreach (RecordsRepositoryProperty property in properties)  
 {  
 if (property.Name.Equals("ContentType"))  
 {  
 // Only accept certain content types?  
 if (property.Value.Equals("Document"))  
 {  
 Log("Rejecting generic document.");  
 resultDetails = "Generic documents are not allowed.";  
 return false;  
 }  
 }  
 else if (property.Name.Equals("File\_x0020\_Type"))  
 {  
 // Only accept certain file extensions?  
 if (property.Value.Equals("xls"))  
 {  
 Log("Rejecting Excel file.");  
 resultDetails = "Excel Files are not allowed.";  
 return false;  
 }  
 }  
 else if (property.Name.Equals("\_IsCurrentVersion"))  
 {  
 // Only accept current versions of documents.  
 if (!property.Value.Equals("True"))  
 {  
 Log("Rejecting older version.");  
 resultDetails = "Only current versions of documents can be stored in the repository.";  
 return false;  
 }  
 }  
 }  
 return true;  
}

# Exercise 3: Installing the Router

1. In order for your router to be recognized within the SharePoint environment, it must be added to the global collection of routers for the web on which your feature is activated. To accomplish this, you will use a factory method of the **SharePointRouter** utility class provided. Open the **FeatureReceiver.cs** file and replace the **FeatureActivated** and **FeatureDeactivating** methods with the following code.

Code Snippet: 'FilteringRouter - FeatureReceiver'

/// <summary>  
/// Override the feature activation event to declare custom routers.  
/// </summary>  
/// <param name="properties"></param>  
public override void FeatureActivated(SPFeatureReceiverProperties properties)   
{  
 using (SPWeb web = properties.Feature.Parent as SPWeb)  
 SharePointRouter.AddRouter(web, typeof(FilteringRouter));  
}  
  
/// <summary>  
/// Override the feature deactivating event to remove custom routers.  
/// </summary>  
/// <param name="properties"></param>  
public override void FeatureDeactivating(SPFeatureReceiverProperties properties) {  
 using (SPWeb web = properties.Feature.Parent as SPWeb)  
 SharePointRouter.RemoveRouter(web, typeof(FilteringRouter));  
}

1. Close the file and build the project. This will run the **Install.bat** , which installs your feature into the local SharePoint farm.

# Exercise 4: Activating the Router

1. There are two steps you must complete before you custom router is available in the record center site. First, you must activate the feature. Second, you must associate the router with one or more record series types.
2. To activate the feature, navigate to the **Site Settings** page of the record center site and select the **Site Features** link. The **ECM401.CustomRouter** feature should appear in the list of features, as shown below.
3. Activate the feature.
4. Navigate to the home page of the record center site and select a record routing type from the **Record Routing** list, for example, **Contract** . Then, from the **Record Routing: Contract** page, scroll to the bottom of the page and select your custom router from the list as shown in the following illustration. Press **OK** to update the routing table.

Note: You can select any routing type you like. *Contract* was chosen here just to illustrate the process.

1. Now submit a variety of file types to the record center from any document library. When you try to submit an Excel spreadsheet, it will be rejected. The same will be true for generic documents and older versions of existing documents.

# Exercise 5: Creating a Tracking Router

1. The next router you create will be used to track incoming records by writing an entry to a custom list in the record center site. The same logic could be applied to write custom auditing records into the content database, or to an external SQL database.
2. Start by adding a new class to the project named **TrackingRouter** and then open the file for editing. As before, add the following **using** statements at the top of the file.

using System.Diagnostics;  
using Microsoft.Office.RecordsManagement.RecordsRepository;  
using WSS=Microsoft.SharePoint;  
using ECM401.RecordsManagement;

1. Next, modify the class declaration so it matches the following code.

[Name("ECM401 Tracking Router")]  
public class TrackingRouter : SharePointRouter  
{  
}

1. To simplify creating and updating the history list, you will use a nested wrapper class. Insert the following code inside the **TrackingRouter** class definition.

Code Snippet: 'TrackingRouter - RecordRouterHistoryList'

/// <summary>  
/// A custom wrapper class that facilitates creating a list  
/// and writing an entry to it.  
/// </summary>  
internal class RecordRouterHistoryList  
{  
 const string listTitle = "Record Routing History";  
 const string listDescription = "This list is used by the TrackingRouter to record incoming parameter values.";  
  
 const string colRecordSeries = "Record Series";  
 const string colSourceUrl = "Source Url";  
 const string colUserName = "User Name";  
 const string colFileSize = "File Size";  
 const string colDestination = "Destination";  
  
 private WSS.SPList m\_list = null;  
  
 public RecordRouterHistoryList(WSS.SPList sourceList)  
 {  
 // Create the list in the same web as the source list.  
 WSS.SPWeb web = sourceList.ParentWeb;  
  
 // Create or open a custom list  
 try { m\_list = web.Lists[listTitle]; }  
 catch { }  
 if (m\_list == null)  
 {  
 try  
 {  
 m\_list = web.Lists[  
 web.Lists.Add(listTitle, listDescription, WSS.SPListTemplateType.GenericList)  
 ];  
  
 m\_list.Fields.Add(colRecordSeries, WSS.SPFieldType.Text, false);  
 m\_list.Fields.Add(colSourceUrl, WSS.SPFieldType.URL, false);  
 m\_list.Fields.Add(colUserName, WSS.SPFieldType.Text, false);  
 m\_list.Fields.Add(colFileSize, WSS.SPFieldType.Number, false);  
 m\_list.Fields.Add(colDestination, WSS.SPFieldType.Text, false);  
 m\_list.OnQuickLaunch = true;  
 m\_list.Update();  
 }  
 catch { }  
 }  
 }  
  
 /// <summary>  
 /// Writes an entry to the list.  
 /// </summary>  
 public void WriteEntry(  
 string recordSeries,  
 string sourceUrl,  
 string userName,  
 int fileSize,  
 string destination)  
 {  
 try  
 {  
 WSS.SPListItem item = m\_list.Items.Add();  
 item[colRecordSeries] = recordSeries;  
 item[colSourceUrl] = sourceUrl;  
 item[colUserName] = userName;  
 item[colFileSize] = fileSize;  
 item[colDestination] = destination;  
 item.Update();  
 }  
 catch { }  
 }  
  
}

1. With your history list defined, you can override and implement the *OnSubmitFile* method. Add the following code to the **TrackingRouter** class definition.

Code Snippet: 'TrackingRouter - OnSubmitFile'

/// <summary>  
/// Custom implementation that writes incoming parameters  
/// to a custom list. Creates the list if it does not exist.  
/// </summary>  
protected override RouterResult OnSubmitFile(  
 string recordSeries,   
 string sourceUrl,   
 string userName,   
 ref byte[] fileToSubmit,   
 ref RecordsRepositoryProperty[] properties,   
 ref WSS.SPList destination,   
 ref string resultDetails)  
{  
 // setup the default result...  
 RouterResult result = RouterResult.SuccessContinueProcessing;  
 try  
 {  
 // Write an entry to the history list.  
 new RecordRouterHistoryList(destination).WriteEntry(  
 recordSeries, sourceUrl, userName, fileToSubmit.Length, destination.Title);  
  
 // Write an event log entry to capture the properties.  
 EventLog.WriteEntry("TrackingRouter Properties",  
 string.Format("Source Url = '{0}'\nProperties:\n{1}",  
 sourceUrl, ExtractProperties(properties)));  
 }  
 catch (Exception x)  
 {  
 // Cancel if we encounter problems writing to the list.  
 // (could reject with resultDetails)  
 EventLog.WriteEntry("TrackingRouter", String.Format("Exception occurred: {0}", x.Message));  
 result = RouterResult.SuccessCancelFurtherProcessing;  
 }  
 return result;  
}

1. This method performs the dual function of writing to the history list and also to the system event log, which it also uses to record any problems encountered while writing to the history list. To make it easier to deal with the custom properties that have been provided along with the incoming file, a separate helper method is used. Add the following code to the **TrackingRouter** class definition.

Code Snippet: 'TrackingRouter - ExtractProperties'

/// <summary>  
/// Returns a string containing all of the properties in the array.  
/// </summary>  
/// <param name="properties"></param>  
/// <returns></returns>  
string ExtractProperties(RecordsRepositoryProperty[] properties)  
{  
 StringBuilder sb = new StringBuilder();  
 foreach (RecordsRepositoryProperty property in properties)  
 sb.AppendFormat("{2}{0}={1}", property.Name, property.Value, sb.Length > 0 ? " ;" : "");  
 return sb.ToString();  
}

1. This completes the tracking router implementation. As a final step, you must add code to the **FeatureActivated** and **FeatureDeactivating** feature receiver methods to register and unregister the router in the SharePoint environment.
2. Open the **FeatureReceiver.cs** file and add the following line to the **FeatureActivated** method.

SharePointRouter.AddRouter(web, typeof(TrackingRouter));

1. Add the following line to the FeatureDeactiving method.

SharePointRouter.RemoveRouter(web, typeof(TrackingRouter));

1. Save your work and rebuild the project, then deactivate and reactivate the feature. Select a record series type and enable the "ECM401 Tracking Router" custom router. Now, when you send a document to the repository that matches the selected record series type, you should see a new list named "Record Routing History" with an entry for each incoming file.

# Exercise 6: Creating a Redirecting Router

1. The final router you create will redirect incoming records to different document libraries based on metadata associated with each record. This is a standard requirement for record routing and the code you write can easily be extended for use in your own solutions.
2. As with the other routers in this set, you will start by adding a new class to the project named **RedirectingRouter** . Open the code file for editing and add the appropriate using statements at the top of the file.

using System.Diagnostics;  
using Microsoft.Office.RecordsManagement.RecordsRepository;  
using WSS=Microsoft.SharePoint;  
using ECM401.RecordsManagement;

1. Modify the class declaration so it matches the following code.

[Name("ECM401 Redirecting Router")]  
public class RedirectingRouter : SharePointRouter  
{  
 // Write exceptions to the trace log.  
 void HandleException(Exception x)  
 {  
 Log(string.Format("Exception occurred: {0}",x.ToString()));  
 }  
}

1. Your implementation of **OnSubmitFile** will redirect incoming records based on the collection of metadata properties attached to the file. The metadata properties are provided in the **RecordsRepositoryProperties** array, which is passed to the method by SharePoint.
2. Add the following code snippet to the **RedirectingRouter** class definition.

Code Snippet: 'Redirecting Router - OnSubmitFile'

/// <summary>  
/// Custom implementation that redirects incoming records  
/// based on the metadata properties attached to the file.  
/// </summary>  
protected override RM.RouterResult OnSubmitFile(  
 string recordSeries,   
 string sourceUrl,   
 string userName,   
 ref byte[] fileToSubmit,  
 ref RM.RecordsRepositoryProperty[] properties,   
 ref SPList destination,   
 ref string resultDetails)  
{  
 // setup the default result...  
 RM.RouterResult result = RM.RouterResult.SuccessContinueProcessing;  
 try  
 {  
 // get the content type from the property array  
 string contentTypeName = string.Empty;  
 foreach (RM.RecordsRepositoryProperty property in properties)  
 if (property.Name.Equals("ContentType")) contentTypeName = property.Value;  
  
 // use the content type name and properties to determine the correct destination  
 SPList newDestination = destination;  
 if (AdjustDestination(contentTypeName, sourceUrl, userName, ref properties, ref newDestination))  
 {  
 string sourceFileName = Path.GetFileNameWithoutExtension(sourceUrl);  
  
 // store the file into the new destination  
 if (SaveDocument(contentTypeName, sourceUrl, userName, ref fileToSubmit,  
 ref properties, ref newDestination, ref resultDetails))  
 {  
 // succeeded in saving the document...  
 Log(string.Format("Saved '{0}' to '{1}'", sourceFileName, destination.Title));  
 }  
 else  
 {  
 // failed to save the document...  
 Log(string.Format("Document save failed for '{0}'", sourceFileName));  
 }  
  
 // return success but cancel further processing, since we  
 // are taking responsibility for storing the file...  
 result = RM.RouterResult.SuccessCancelFurtherProcessing;  
 }  
 else  
 {  
 // redirection is not required  
 // continue processing normally  
 Log(string.Format("Redirection not required for content type '{0}'", contentTypeName));  
 result = RM.RouterResult.SuccessContinueProcessing;  
 }  
 }  
 catch (Exception x)  
 {  
 // cancel processing if we encounter an error...  
 HandleException(x);  
 result = RM.RouterResult.SuccessCancelFurtherProcessing;  
 }  
 return result;  
}

1. The goal of a redirecting router is to change the target location of each incoming file based on a custom algorithm.

Note: The SharePoint API implies (by use of the *ref* keyword on the *destination* parameters) that you can change the destination target by changing this value to reference another list. In actual practice, this does not seem to work unless there is a problem storing the file into the original location.

1. Instead of relying on SharePoint to copy the file, you will use a helper method to copy the file depending on whether an adjustment to the destination is required. You will use another help method to make that determination based on the incoming content type and metadata properties.Add the following helper method to the **RedirectingRouter** class definition.

Code Snippet: 'RedirectingRouter - AdjustDestination'

// Attempts to adjust the destination list based on incoming metadata.  
bool AdjustDestination(string contentTypeName, string sourceUrl, string userName,  
 ref RM.RecordsRepositoryProperty[] properties,  
 ref SPList destination)  
{  
 // if no content type name was provided, then no special processing is possible...  
 if (string.IsNullOrEmpty(contentTypeName))  
 return false;  
  
 // if the content type name matches an existing document library, then use that  
 // as the new destination library...  
 SPList targetList = SharePointList.Find(destination.ParentWeb, contentTypeName);  
 if (targetList != null)  
 {  
 Log(string.Format("Reusing existing list '{0}'", targetList.Title));  
 destination = targetList;  
 }  
 else  
 {  
 // target list does not exist, so create a new document library...  
 destination = SharePointList.Create(destination.ParentWeb,  
 SPListTemplateType.DocumentLibrary, contentTypeName, "Created by the RedirectingRouter");  
 // no special fields added to the default "Document" content type  
 destination.ContentTypesEnabled = true;  
 destination.OnQuickLaunch = true;  
 destination.Update();  
 }  
  
 return true;  
}

1. To perform the actual copy operation, add the *SaveDocument* helper method to the class definition.

Code Snippet: 'RedirectingRouter - SaveDocument'

// Stores the document using metadata to determine where to place it within the target list.  
bool SaveDocument(string contentTypeName, string sourceUrl, string userName,  
 ref byte[] fileToSubmit, ref Microsoft.Office.RecordsManagement.RecordsRepository.RecordsRepositoryProperty[] properties,  
 ref SPList destination, ref string resultDetails)  
{  
 SPFolder targetFolder = null;  
 SPListItem item = null;  
 SPFile file = null;  
 string fileName = Path.GetFileNameWithoutExtension(sourceUrl);  
  
 try  
 {  
 // get a folder in the destination list using the user name  
 int pos = userName.LastIndexOf('\\');  
 string actualUserName = userName.Substring(pos >= 0 ? pos : 0);  
 targetFolder = SharePointList.CreateFolder(destination, actualUserName);  
 }  
 catch (Exception x1)  
 {  
 HandleException(new Exception("Failed to get target folder", x1));  
 }  
  
 try  
 {  
 // add the document to the folder  
 file = targetFolder.Files.Add(fileName, fileToSubmit);  
 item = file.Item;  
 }  
 catch (Exception x2)  
 {  
 HandleException(new Exception("Failed to add document to folder", x2));  
 }  
  
 // set the content type if recognized...  
 try  
 {  
 SPContentType ct = destination.ContentTypes[contentTypeName];  
 item["ContentTypeId"] = ct.Id;  
 item.Update();  
 }  
 catch (Exception x3)  
 {  
 HandleException(new Exception("Failed to update content type id", x3));  
 }  
  
 // copy any matching properties into the new item...  
 foreach (RM.RecordsRepositoryProperty property in properties)  
 {  
 if (item.Fields.ContainsField(property.Name))  
 {  
 try  
 {  
 // check the validity of the target field...  
 SPField field = item.Fields.GetField(property.Name);  
 if (field != null && !field.ReadOnlyField &&  
 (field.Type != SPFieldType.Invalid) &&  
 (field.Type != SPFieldType.WorkflowStatus) &&  
 (field.Type != SPFieldType.File) &&  
 (field.Type != SPFieldType.Computed) &&  
 (field.Type != SPFieldType.User) &&  
 (field.Type != SPFieldType.Lookup) &&  
 (!field.InternalName.Equals("ContentType")))  
 {  
 item[property.Name] = property.Value;  
 }  
 }  
 catch (Exception x)  
 {  
 resultDetails = string.Format("Exception occurred while saving '{0}': {1}", fileName, x.Message);  
 return false;  
 }  
 }  
 }  
  
 item.Update();  
 return true;  
}

1. Save your work. Finally, open the **FeatureReceiver.cs** file again and add the registration code to the feature receiver methods.

// Add this line to the FeatureActivated event receiver method.  
SharePointRouter.AddRouter(web,typeof(RedirectingRouter));  
  
// Add this line to the FeatureDeactivating event receiver method.  
SharePointRouter.RemoveRouter(web,typeof(RedirectingRouter));

1. Rebuild the project, then deactivate and activate the feature and associate the redirecting router for one or more record series types.

**This concludes the lab exercises.**