

WC-ECM401 Lab Manual

Enterprise Content Management with SharePoint Server 2007

Schedule of Lectures

- 1) PowerShell Import
- 2) Content Types
- 3) Auditing
- 4) Information Management Policy
- 5) Custom Repository
- 6) Official File Submit
- 7) Custom Routing
- 8) Sales Proposal Schema
- 9) Site Provisioning

Lab 1: PowerShell Batch Import

Lab Time: 60 Minutes

Lab Directory: ECM401.PowerShellImport

Lab Overview:

In this lab, you will create a **Windows PowerShell** cmdlet that will enable you to import batches of documents into SharePoint from a file share. Your cmdlet will optionally read a manifest file located in a given folder that specifies the metadata you want to be associated with files in that folder. In addition to explicit metadata specified in the manifest file, your cmdlet will also extract a subset of document properties from each document so that they are available as columns in the target SharePoint lists.

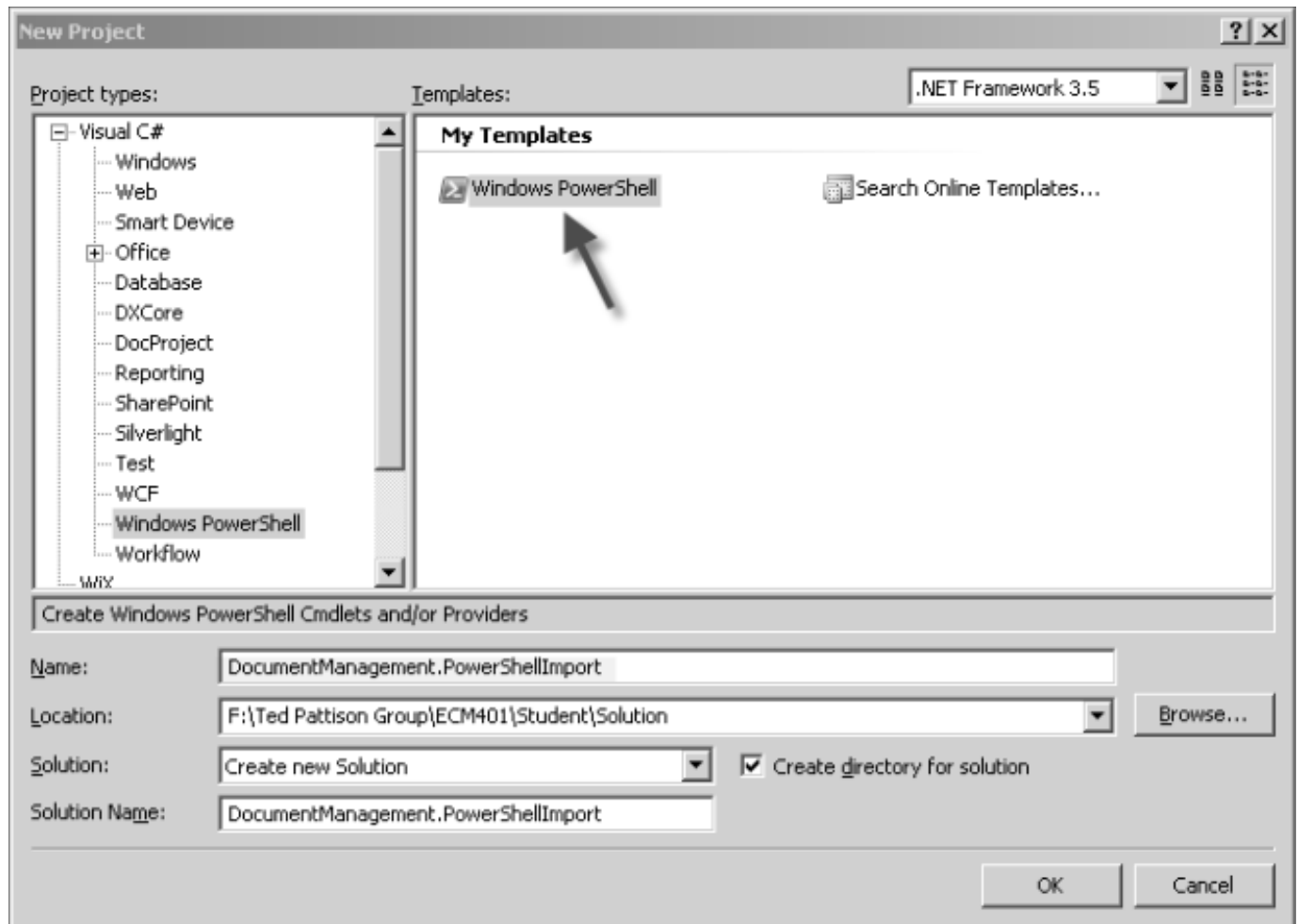
Exercise 1: Install the Windows PowerShell Templates

1. In order to create a **Windows PowerShell** cmdlet, you first need to setup your development environment. In your **student/resources/tools** folder, there is a set of Visual Studio templates. If you have not done so already, run the appropriate **VSI** file for your language of choice (CS or VB). This will install the PowerShell project and item templates into Visual Studio.

Note: The basic structure of every PowerShell command is that you have two parts to the command, separated by a hyphen. The first part is a verb and the second part is a noun. In this exercise, you will create two commands. "SPImport-Document" will import a specific set of documents using a file mask, and "SPImport-Folder" will import a folder and its subfolders.

Exercise 2: Create a Visual Studio Powershell Project

1. Start by creating a new Visual Studio project based on the **Windows PowerShell** project template, as shown in the following screenshot.

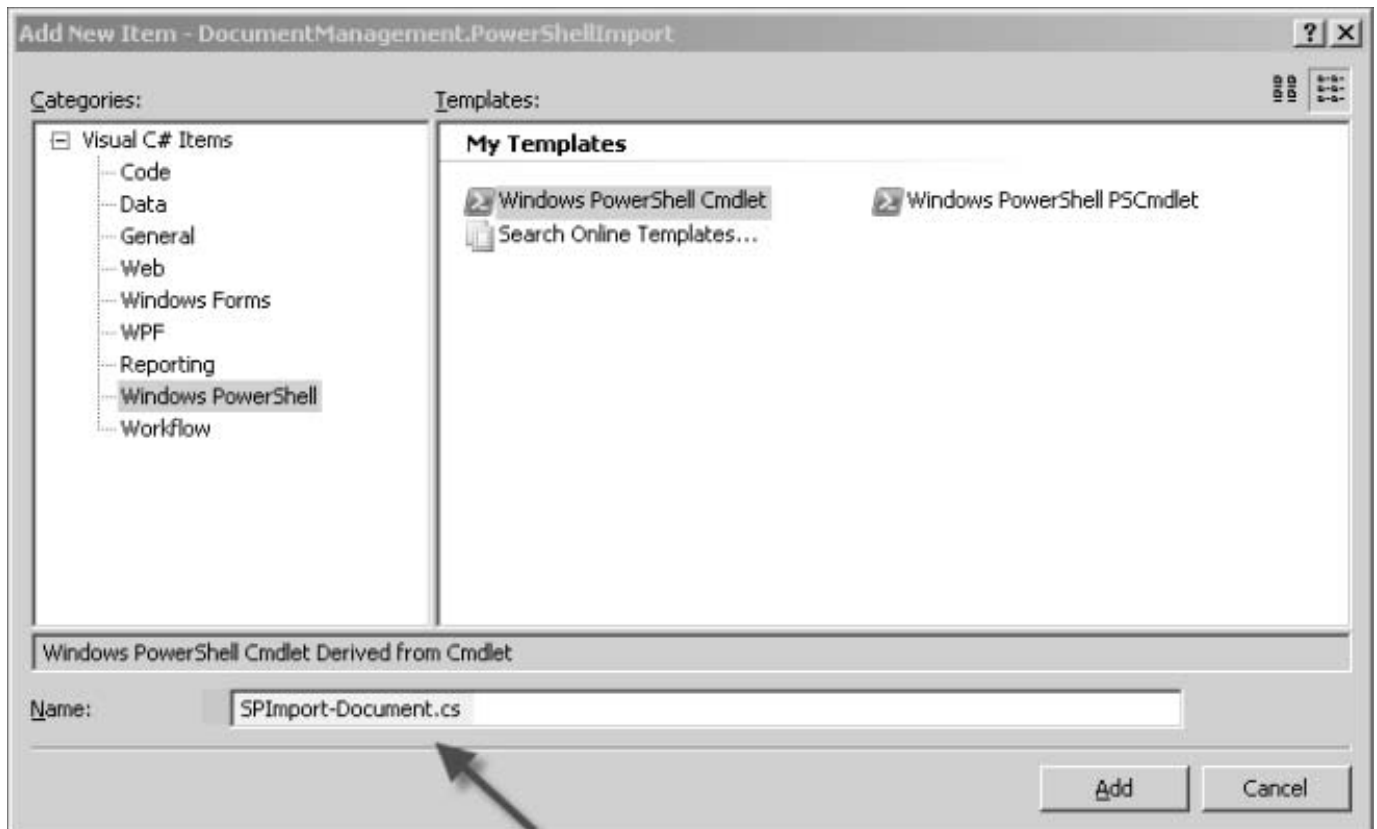


2. Select the **PSSnapin.cs** file and choose **View Code** from the context menu. Modify the **Name** property override so that it returns **"SPPowerShell"** instead of the default. You can leave the other overrides with their generated default values.

Note: You may also have to fix the generated class name if you included a '.' character in the project name. This is a bug in the PowerShell project template.

Exercise 3: Create the Import Document CmdLet

1. Right-click the project node and select **Add New Item**. From the **Add New Item** dialog, select the **Windows PowerShell CmdLet** item.



2. Change the generated **VerbsCommon.Get** parameter to the **Cmdlet** attribute to "**SPIImport**" and change the second parameter to "**Document**". The resulting code should look like the following:

```
[Cmdlet("SPIImport", "Document", SupportsShouldProcess = true)]
public class SPIImport_Document : Cmdlet
{
    #region Parameters
    /*
    [Parameter(Position = 0,
        Mandatory = false,
        ValueFromPipelineByPropertyName = true,
        HelpMessage = "Help Text")]
    [ValidateNotNullOrEmpty]
    public string Name
    {
        }
    }
    */
    #endregion

    protected override void ProcessRecord()
    {
        try
        {
            throw new NotImplementedException();
        }
        catch (Exception)
        {
        }
    }
}
```

```
}
```

3. To allow the caller to specify which documents to upload and where to put them, parameters are needed. **Windows PowerShell** will automatically parse the command line based on attributes you use within the CmdLet. The first parameter accepts a wildcard file mask that will be expanded to a list of files that will then be uploaded to the website or document library.

4. Add the following code to your CmdLet.

```
[ Parameter(Position = 0,
    Mandatory = true,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true,
    HelpMessage = "Enter the fully qualified path to the file(s) you wish to
upload. You can enter a standard wildcard mask.") ]

[ ValidateNotNullOrEmpty ]
public string FileMask { get; set; }
```

5. The second parameter specifies the url of the SharePoint site to which the document will be uploaded. If no document library is specified, then the file is uploaded directly into the site. If the site does not exist, an exception is thrown. Add the following code.

```
[ Parameter(Position = 1,
    Mandatory = true,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true,
    HelpMessage = "Enter the url of the target SharePoint site.") ]
public string SiteUrl { get; set; }
```

6. The third parameter specifies the name of the target folder or document library within the specified site. If the destination folder or document library does not already exist, then it is created automatically if the 'AutoCreate' parameter is also set to true. Add the following code.

```
[Parameter(Position = 2,
    Mandatory = false,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true,
    HelpMessage = "Enter the name of the folder or document library to which the
file(s) will be added. To specify a root folder within the site, precede the
name with a forward slash '/'") ]
    public string Destination { get; set; }
```

7. PowerShell includes the notion of a **switch** parameter, which is converted to a Boolean value. The next parameter is declared as a **SwitchParameter** that specifies whether to create the target folder or document library if it does not already exist within the specified site. The final parameter is another **SwitchParameter** that controls whether existing files with the same name are overwritten. Add the following two parameters to your CmdLet class.

```
[ Parameter(
    ValueFromPipeline=true,
    ValueFromPipelineByPropertyName = true,
    HelpMessage = "Specify whether to create the target library or folder if it
does not already exist. Default=false.") ]
```

```
public SwitchParameter AutoCreate { get; set; }

[ Parameter(
    ValueFromPipelineByPropertyName=true,
    HelpMessage="Specify whether to overwrite files that already exist in the
target library or folder. Default=false.") ]
public SwitchParameter Overwrite { get; set; }
```

8. Finally, you will add the main processing method to your cmdlet. Replace the generated **ProcessRecord** method with the following code snippet.

Code Snippet: 'PowerShell Import Document'

```
/// <summary>
/// This is the standard entry point for PowerShell commands.
/// It is called after the command line arguments have been
/// parsed and set into the corresponding properties of
/// the cmdlet class.
/// </summary>
protected override void ProcessRecord()
{
    try
    {
        // Expand the file mask into a list of FileInfo objects.
        FileInfo[] files = ExpandWildcards(FileMask);
        if (files.Length == 0)
            throw new Exception("No files specified.");

        // Locate the target SharePoint site collection.
        using (SPSite site = new SPSite(SiteUrl))
        {
            using (SPWeb web = site.OpenWeb())
            {
                // Check the destination to determine what to do.
                if (string.IsNullOrEmpty(Destination))
                {
                    // No destination provided, so copy
                    // the files into the site itself.
                    TransferFiles(web, files, null);
                }
                else
                {
                    // Find or create the target folder
                    // and transfer the files.
                    SPFolder targetFolder = FindFolder(
                        web, Destination);
                    TransferFiles(web, files, targetFolder);
                }
            }
        }
    }
    catch (Exception x)
    {
        Console.WriteLine("Unable to process files: ");
        Console.WriteLine(x.Message);
    }
}
```

Exercise 4: Add Helper Methods

1. Next, you will implement a set of helper methods that can be called from both cmdlets to perform the work of finding files, manipulating SharePoint folders and transferring files into the content database.
2. Add a new code file to your project named **"Helpers.cs"** and open it for editing.
3. The following helper method searches for a named folder within a SharePoint web site. If the folder does not exist, then the `AutoCreate` parameter controls whether the folder is created automatically. If the path includes multiple segments, then each segment is created with the specified name. If the path starts with a forward slash '/', then interpret the string as the fully qualified path of a folder within the site, otherwise treat the first segment as the name of a document library and the remainder as the path of a folder within the library. This gives the most flexibility when moving documents into the content database.

Code Snippet: 'PowerShell - FindFolder Method'

```
SPFolder FindFolder(SPWeb web, string path)
{
    SPFolder targetFolder = web.RootFolder;
    string[] names = path.Split("/\\\\".ToCharArray());
    if (path.StartsWith("/"))
    {
        // find or create the specified subfolder
        foreach (string name in names)
            if (!string.IsNullOrEmpty(name))
                targetFolder = FindOrCreateSubfolder(
                    targetFolder, name);
    }
    else
    {
        // check if the first name matches a list in the site
        SPList targetList = web.Lists[names[0]];

        // no match, so check for autcreate
        if (targetList == null)
        {
            if (AutoCreate)
            {
                // create a new document library in the site
                Guid guid = web.Lists.Add(names[0],
                    "Created from PowerShell",
                    SPListTemplateType.DocumentLibrary);
                targetList = web.Lists[guid];
            }
        }

        // check again if the list exists
        if (targetList == null)
            throw new Exception(string.Format(
                "Document library '{0}' not found.", path));

        // the default target is the root folder of the list
        targetFolder = targetList.RootFolder;

        // process the remaining names as folder name.
    }
}
```

```

        for (int index = 1; index < names.Length; index++)
            targetFolder = FindOrCreateSubfolder(targetFolder,
            names[index]);
    }
    return targetFolder;
}

```

4. Add the following helper method to transfer a collection of files into a folder within a SharePoint site. If the folder is null, then the files are copied directly into the site.

Code Snippet: 'PowerShell - TransferFiles Method'

```

void TransferFiles(SPWeb web, FileInfo[] files, SPFolder targetFolder)
{
    int nCopied = 0;
    int nSkipped = 0;
    SPList parentList = null;
    SPDocumentLibrary docLib = null;
    Guid parentListId = targetFolder.ParentListId;

    if (parentListId != Guid.Empty)
    {
        parentList = web.Lists[parentListId];
        if (targetFolder.ContainingDocumentLibrary != Guid.Empty &&
            targetFolder.ContainingDocumentLibrary != parentListId)
        {
            docLib = web.Lists[targetFolder.ContainingDocumentLibrary
            as SPDocumentLibrary];
        }
    }

    Console.WriteLine("Transferring files to folder '{0}' in site '{1}'",
        targetFolder.Name, web.Title);

    foreach (FileInfo fileInfo in files)
    {
        bool okToCopy = true;

        // Check if the file already exists in the target folder.
        SPFile targetFile = FindFile(fileInfo, targetFolder);
        if (targetFile != null)
        {
            if (this.Overwrite)
                targetFile.Delete();
            else
                okToCopy = false;
        }

        // Get the file bits and add them to the target folder.
        if (okToCopy)
        {
            nCopied++;
            using (FileStream fs = fileInfo.Open(FileMode.Open))
                targetFolder.Files.Add(fileInfo.Name, fs);
        }
        else
        {
            nSkipped++;
        }
    }
}

```



```
if (nCopied > 0)
    Console.WriteLine("{0} files transferred.", nCopied);
else
    Console.WriteLine("No files transferred.");

if (nSkipped > 0)
    Console.WriteLine("{0} files skipped.", nSkipped);
}
```

Exercise 5: Test Your Work

1. To test your work, open a **Windows Powershell** command window. While building PowerShell cmdlets, it is most convenient to run the PowerShell command window from within the Visual Studio debugger. To set this up, change the **Start Action** setting on the **Debug** page of the project properties to the following command:

```
c:\windows\system32\windowpowershell\v1.0\powershell.exe
```

2. In order for **Windows PowerShell** to recognize your snap-in, it must be registered on the system. The best way to do that is to use the **InstallUtil** utility. You can run **InstallUtil** from the post-build events in the project. Open the project property pages and navigate to the **Build Events** tab. Enter the following command into the **Post-build event command line** section.

```
InstallUtil "$(TargetPath)"
```

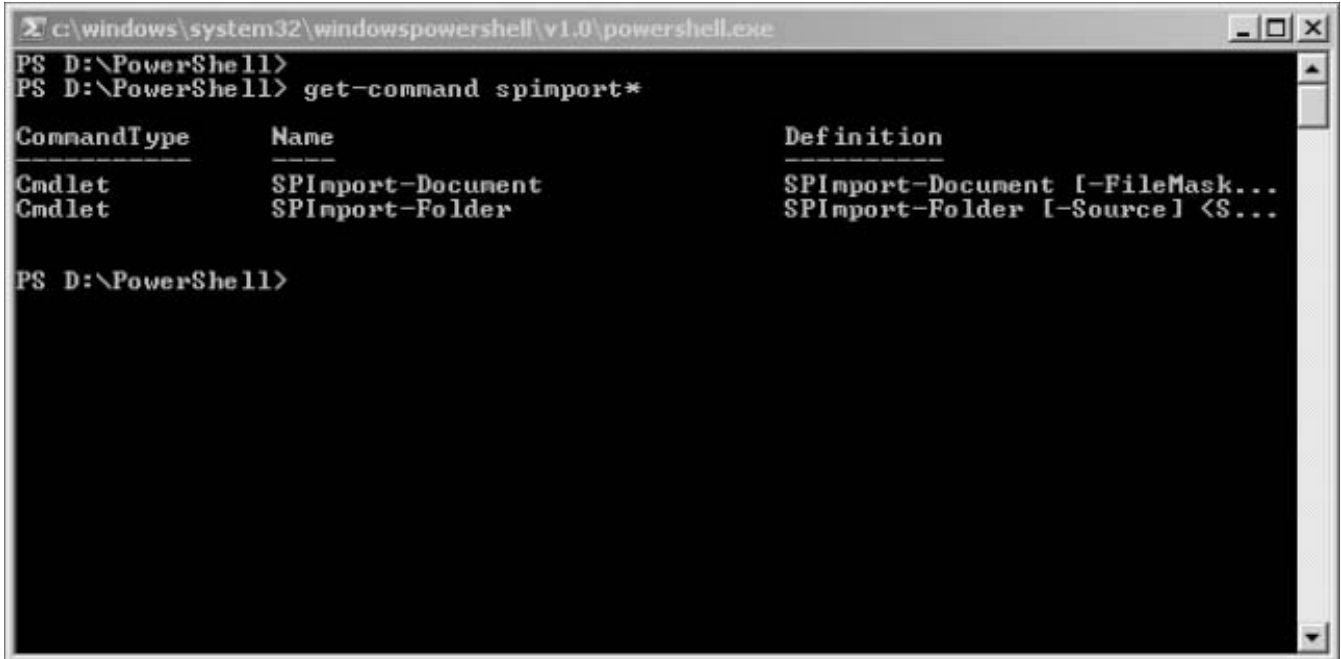
3. Each time PowerShell runs, you will need to add the SPPowerShell snapin so that your commands are recognized. Type the following command at the PowerShell command prompt.

```
add-psnapin SPPowerShell
```

4. To verify that the snapin was loaded successfully, enter the following command into the PowerShell command line.

```
get-command spimport*
```

5. You should see your custom commands listed on the display.



```

C:\windows\system32\windowspowershell\v1.0\powershell.exe
PS D:\PowerShell>
PS D:\PowerShell> get-command spinport*

CommandType      Name                Definition
-----
Cmdlet            SPIimport-Documen... SPIimport-Documen [-FileMask...
Cmdlet            SPIimport-Folder    SPIimport-Folder [-Source] <S...

PS D:\PowerShell>
  
```

6. Now you can enter a command to import a file into the SharePoint environment. Navigate to the **student/resources/documents** folder and enter the following command:

```
SPIimport-Documen Newsletter.doc "http://litwareinc.com"
```

Exercise 6: Create the Import Folder CmdLet

1. This cmdlet provides a more powerful and comprehensive way to move documents into a SharePoint site collection by letting the user add metadata to each document as it is being transferred. The source documents are first arranged into physical folders on the file system, and a manifest file is optionally added to a given folder to specify the metadata and other rules that should apply to the documents in that folder. Rules in the manifest control whether nested folders inherit or override the settings in the parent folder. This is similar to the way that ASP.NET handles web.config files.
2. Add a new class and source file to the project named **SPIimport-Folder.cs**, and insert the following code.

```

[ Cmdlet("SPIimport", "Folder", SupportsShouldProcess=true) ]
public class SPIimport_Folder : Cmdlet
{
    #region Parameters
    /// <summary>
    /// This parameter accepts a wildcard file mask that will be expanded
    /// to a list of folders to be imported to the SharePoint site.
    ///
    </summary>
    [Parameter(Position = 0,
        Mandatory = true,
        ValueFromPipeline = true,
        ValueFromPipelineByPropertyName = true,
  
```

```

        HelpMessage = "Enter the fully qualified path to the folder you wish
to import. You can enter a standard wildcard mask.")]

        [ValidateNotNullOrEmpty]
        [Alias("Folder", "Path", "SourceFolder")]
        public string Source { get; set; }

        /// <summary>
        /// This parameter specifies the url of the SharePoint site to which the
        /// documents will be imported. If no manifest is found or no document
        library is specified
        /// in the manifest, then the files are imported to a new library called
        "Imported Documents".
        /// If the specified site does not exist, an InvalidArgument exception is
        thrown.
        ///
        </summary>
        [Parameter(Position = 1,
            Mandatory = true,
            ValueFromPipeline = true,
            ValueFromPipelineByPropertyName = true,
            HelpMessage = "Enter the url of the target SharePoint site.")]
        [Alias("Site", "Url")]
        public string SiteUrl { get; set; }

        /// <summary>
        /// This parameter controls whether existing files with the same name are
        overwritten.
        ///
        </summary>
        [Parameter(
            ValueFromPipelineByPropertyName = true,
            HelpMessage = "Specify whether to overwrite files that already exist
in the target library or folder. Default=false.")]
        public SwitchParameter Overwrite { get; set; }

        /// <summary>
        /// This parameter controls whether subfolders are processed.
        ///
        </summary>
        [Parameter(
            ValueFromPipelineByPropertyName=true,
            HelpMessage = "Specify whether to process subfolders of the source
folder.")]
        public SwitchParameter Recursive { get; set; }
    #endregion
}

```

- Next, replace the generated **ProcessRecord** method with the following code snippet.

Code Snippet: 'PowerShell - Import Folder'

```

/// <summary>
/// This is the standard entry point for PowerShell commands.
/// It is called after the command line arguments have been parsed
/// and set into the corresponding properties of the cmdlet class.
/// </summary>
protected override void ProcessRecord()
{
    try
    {
        // Reduce the source path to a DirectoryInfo object.
    }
}

```

```

        if (!this.Source.EndsWith("\\\\"))
            this.Source += "\\\\";

        string folderName = Path.GetDirectoryName(Source);
        if (string.IsNullOrEmpty(folderName))
            folderName = Environment.CurrentDirectory;
        DirectoryInfo folder = new DirectoryInfo(
            Path.GetFullPath(folderName));

        // Locate the target SharePoint site collection.
        using (SPSite site = new SPSite(SiteUrl))
        using (SPWeb web = site.OpenWeb())
        {
            // Process the files in the folder
            // and its subfolders.
            Utilities.TransferFolderUsingManifest(
                web, folder, Overwrite, Recursive);
        }
    }
    catch (Exception x)
    {
        Console.WriteLine("Unable to process files: ");
        Console.WriteLine(x.Message);
    }
}

```

4. To process the folder, you will need a new helper method that understands how to process the manifest file. The manifest file is an XML file based on a schema you will create in the next exercise that describes the different elements that can be processed.
5. Open the **helpers.cs** file and add the following code snippet.

Code Snippet: 'PowerShell - TransferFolderUsingManifest'

```

/// <summary>
/// This method transfers all files in a given folder
/// using an optional manifest file located within the folder itself.
/// The manifest file name must be "_manifest.xml". This file is
/// deserialized into an ImportManifest object, which is then
/// used to perform the actual transfer.
/// </summary>
/// <param name="web">the target website into
/// which files will be transferred</param>
/// <param name="folder">the source folder from
/// which files will be transferred</param>
/// <param name="overwrite">whether to overwrite
/// existing files in the target library</param>
/// <param name="recursive">whether to process subfolders</param>
internal static void TransferFolderUsingManifest(SPWeb web,
    DirectoryInfo folder, bool overwrite, bool recursive)
{
    // create a default manifest
    ImportManifest manifest = new ImportManifest();
    FileInfo[] files = folder.GetFiles("_manifest.xml",
        SearchOption.TopDirectoryOnly);

    // search for the manifest file
    if (files.Length > 0)
        try

```

```

    {
        manifest = ImportManifest.Load(files[0].FullName);
    }
    catch (Exception x)
    {
        throw new Exception(
            string.Format("Failed to load manifest file: {0}",
                files[0].FullName), x);
    }

    // process the manifest
    if (manifest != null)
    {
        try
        {
            manifest.ImportFiles(web, folder, overwrite);
        }
        catch (Exception x2)
        {
            throw new Exception(
                string.Format("Failed to import files in folder: {0}",
                    folder.FullName), x2);
        }
    }

    // process the subfolders
    if (recursive)
    {
        foreach (DirectoryInfo subfolder in folder.GetDirectories())
        {
            TransferFolderUsingManifest(web, subfolder, overwrite, recursive);
        }
    }
}

```

Exercise 7: Create the Manifest XML Schema

1. Your **SPImport-Folder** cmdlet will optionally process a manifest file that can be placed in a given folder. You will use a schema to define the structure of this file and also to generate the code needed to process its elements.
2. Add a new **XML Schema** item to the project with the name **ImportManifest.xsd** Open the file for editing and replace the entire contents with the following code snippet.

XML Snippet: 'PowerShell - ImportManifest Schema'

```

<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="SPImportManifest"

targetNamespace="http://schemas.johnholliday.net/sharepoint/importmanifest.xsd"
elementFormDefault="qualified"
xmlns="http://schemas.johnholliday.net/sharepoint/importmanifest.xsd"
xmlns:mstns="http://schemas.johnholliday.net/sharepoint/importmanifest.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
>
  <xs:element name="ImportManifest">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Library" type="LibrarySpecification" minOccurs="0"
maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

```

```

</xs:element>

<xs:complexType name="LibrarySpecification">
  <xs:sequence>
    <xs:element name="Columns" type="ColumnSet" minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
  <xs:attribute name="Path" type="xs:string" use="required"/>
  <xs:attribute name="RootWeb" type="xs:boolean" use="optional"/>
  <xs:attribute name="InheritMetadata" type="xs:boolean" />
  <xs:attribute name="AutoCreate" type="xs:boolean" use="optional"
default="true"/>
  <xs:attribute name="FileMask" type="xs:string" use="optional"/>
  <xs:attribute name="Exclude" type="xs:string" use="optional"/>
</xs:complexType>

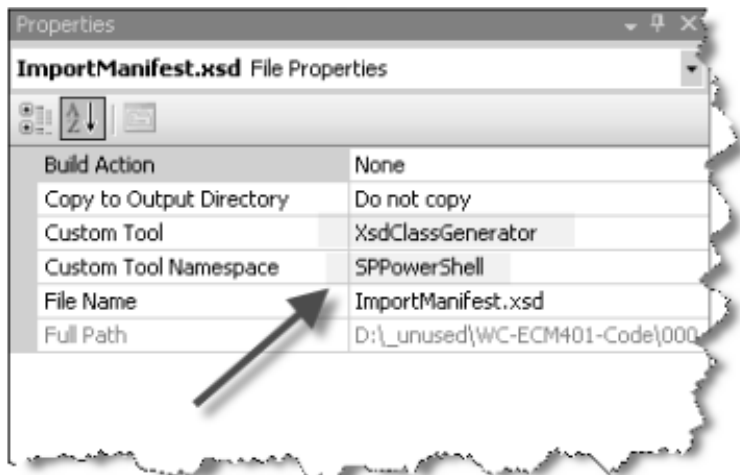
<xs:complexType name="ColumnSet">
  <xs:sequence>
    <xs:element name="Column" type="ColumnSpecification" minOccurs="0"
maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="ColumnSpecification">
  <xs:attribute name="DocProperty" type="xs:string" use="required"/>
  <xs:attribute name="DisplayName" type="xs:string" use="optional"/>
  <xs:attribute name="Required" type="xs:boolean" use="optional"/>
  <xs:attribute name="Type" type="xs:string" use="optional"/>
</xs:complexType>

<xs:complexType name="DocPropertyElement">
  <xs:simpleContent>
    <xs:extension base="xs:string">
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
</xs:schema>

```

3. In order to simplify the parsing of manifest files, you will generate wrapper classes from the schema using the XSD.EXE tool. Select the **ImportManifest.xsd** file in the Visual Studio **Solution Explorer** window. From the **Properties** window, set the **Custom Tool** and **Custom Tool Namespace** values to match the following diagram.



Note: The **XsdClassGenerator** assembly is located in your **Student/Resources/Tools** folder along with a batch file you can run to install the tool into your **Visual Studio** environment.

4. Next, you will extend the generated partial class with a static factory method to make it easy to load manifest files. Add a new class to the project named **ImportManifestEx** . Open the file for editing and add the following code snippet.

Code Snippet: 'PowerShell Import Manifest Extensions'

```

/// <summary>
/// Extends the generated ImportManifest class to support file transfer
/// operations into SharePoint sites.
/// </summary>
public partial class ImportManifest
{
    /// <summary>
    /// Loads a manifest from a specified file.
    /// </summary>
    /// <param name="fileName"></param>
    /// <returns></returns>
    public static ImportManifest Load(string fileName)
    {
        ImportManifest manifest = null;
        const string ns =
"http://schemas.johnholliday.net/sharepoint/importmanifest.xsd";
        XmlSerializer ser = new XmlSerializer(typeof(ImportManifest), ns);
        using (FileStream fs = new FileStream(fileName, FileMode.Open))
        {
            manifest = ser.Deserialize(fs) as ImportManifest;
        }
        return manifest;
    }

    /// <summary>
    /// Processes all files in the specified folder according to the rules
    /// specified in this manifest.
    /// </summary>
    /// <param name="web"></param>
    /// <param name="folder"></param>
    public void ImportFiles(SPWeb web, DirectoryInfo folder, bool overwrite)
    {
        // The manifest contains one or more <Library> nodes that target a
        specific
        // library. This means that the same files may be imported more than
        once.
        foreach (LibrarySpecification library in this.Library)
            library.ImportFiles(web, folder, overwrite);
    }
}

```

5. Finally, you will extend the generated **LibrarySpecification** and **ColumnSpecification** objects so that they understand how to transfer files into SharePoint and convert column specifications into SharePoint field references.
6. Add a new class to the project named **LibrarySpecification** . Open the **LibrarySpecification.cs** source file and replace the class declaration with the following code.

Code Snippet: 'PowerShell Library Specification'

```

/// <summary>
/// Extends the generated LibrarySpecification class to support importing of
files
/// into SharePoint.
/// </summary>
public partial class LibrarySpecification
{
    public void ImportFiles(SPWeb web, DirectoryInfo folder, bool overwrite)
    {
        int nCopied = 0;
        int nSkipped = 0;

        // determine the target web.
        SPWeb targetWeb = this.RootWeb ? web.Site.RootWeb : web;

        // find or create the target folder within SharePoint
        SPFolder targetFolder = Utilities.FindFolder(targetWeb, this.Path,
this.AutoCreate);

        if (targetFolder == null)
        {
            Console.WriteLine("Cannot find target folder at '{0}'", this.Path);
            return;
        }

        if (targetFolder != null)
        {
            Console.WriteLine("Importing files from '{0}' => '{1}'",
folder.FullName, targetFolder.Url);

            // get the list of files matching the file mask(s)
            Dictionary<string, FileInfo> files = new Dictionary<string,
FileInfo>();
            string maskSpec = string.IsNullOrEmpty(this.FileMask) ? "*.*" :
this.FileMask;
            string[] masks = maskSpec.Split(';');
            foreach (string mask in masks)
            {
                FileInfo[] matchingFiles = folder.GetFiles(mask);

                // add them to the list to be processed
                foreach (FileInfo match in matchingFiles)
                    files.Add(match.Name, match);
            }

            // remove the list of excluded files
            if (!string.IsNullOrEmpty(this.Exclude))
            {
                string[] excludePaths = this.Exclude.Split(';');
                foreach (string exclusion in excludePaths)
                {
                    FileInfo[] excludedFiles = folder.GetFiles(exclusion);
                    foreach (FileInfo exclude in excludedFiles)
                        if (files.ContainsKey(exclude.Name))
                            files.Remove(exclude.Name);
                }
            }

            // process each file...
            foreach (FileInfo file in files.Values)
            {

```



```

        bool okToCopy = true;
        if (file.Name.Equals("_manifest.xml"))
            continue;

        // check if the file already exists in the target folder
        SPFile targetFile = Utilities.FindFile(file, targetFolder);
        if (targetFile != null)
        {
            if (overwrite)
                targetFile.Delete();
            else
                okToCopy = false;
        }

        if (!okToCopy)
            nSkipped++;

        if (okToCopy)
        {
            try
            {
                // if there are any <Column> nodes, map them to columns
                // in the target document library so that SharePoint
                // any matching properties in the document.
                foreach (ColumnSpecification colSpec in this.Columns)
                {
                    colSpec.MapToLibrary(targetFolder);

                    // Get the file bits and add them to the target folder.
                    using (FileStream fs = file.Open(FileMode.Open))
                        targetFolder.Files.Add(file.Name, fs);
                }
            }
            catch (Exception x)
            {
                Console.WriteLine("Failed to transfer file: '{0}' to
'{1}', file.Name, targetFolder.Name);
                Console.WriteLine("Exception occurred: {0}",
x.ToString());
            }
            nCopied++;
        }
    }

    if (nCopied > 0)
        Console.WriteLine("{0} files transferred.", nCopied);
    else
        Console.WriteLine("No files transffered.");

    if (nSkipped > 0)
        Console.WriteLine("{0} files skipped.", nSkipped);
}
}
}

```

7. Add a second class to the project named **ColumnSpecification** , open the code file and replace the class declaration with the following code.

Code Snippet: 'PowerShell - Column Specification'

```
/// <summary>
```

```

/// Describes an individual column specification.
/// </summary>
public partial class ColumnSpecification
{
    /// <summary>
    /// This method ensures that a column exists in the containing library of the
    /// specified folder that matches a property by name in the document.
    /// </summary>
    /// <param name="targetFolder"></param>
    public void MapToLibrary(SPFolder targetFolder)
    {
        SPLList parentList = null;
        Guid parentListId = targetFolder.ContainingDocumentLibrary;

        if (parentListId == Guid.Empty)
            parentListId = targetFolder.ParentListId;

        if (parentListId != Guid.Empty)
            parentList = targetFolder.ParentWeb.Lists[parentListId];

        if (parentList != null)
        {
            SPField matchingField = null;
            SPFieldType fieldType = GetFieldType(this.Type);
            string fieldName = string.IsNullOrEmpty(this.DisplayName) ?
this.DocProperty : this.DisplayName;

            try
            {
                matchingField = parentList.Fields.GetField(fieldName);
            }
            catch { }
            if (matchingField == null)
            {
                try
                {
                    matchingField =
parentList.Fields.GetFieldByInternalName(fieldName);
                }
                catch { }
                if (matchingField == null)
                {
                    // field was not found, so add a new column to the list
                    parentList.Fields.Add(fieldName, fieldType,
this.RequiredSpecified ? this.Required : false);
                    parentList.Update();
                }
            }
        }
    }

    /// <summary>
    /// Determines the target field type based on a string.
    /// </summary>
    /// <param name="fieldType"></param>
    /// <returns></returns>
    SPFieldType GetFieldType(string fieldType)
    {
        SPFieldType type = SPFieldType.Text;
        try
        {
            type = (SPFieldType)Enum.Parse(typeof(SPField), fieldType);
        }
        catch
        {
        }
    }
}

```

```

    }
    return type;
}
}

```

8. Now your helper methods can call the wrapper classes to interpret the manifest file when processing a folder.

Exercise 8: (BONUS) Convert OLE Properties into SharePoint Metadata

1. When transferring documents from the file system into SharePoint, it is useful to have a way to automatically extract OLE properties and promote them to columns in the target list. This happens automatically for the custom properties, but not for the standard properties that most users are accustomed to.
2. If you would like to extend your cmdlet to handle the summary properties, you can use the following code to extract them.

Note: This code is a derivative work based on an article by Marcus Peters from the book "Inside Windows" published by Apress, May 4, 2008

Code Snippet: 'PowerShell - GetOleProperties'

```

namespace ECM401
{
    using DSOFile;

    /// <summary>
    /// Extract the OLE properties from a file using the DSOFile assembly.
    /// </summary>
    /// <remarks>
    internal static StringDictionary GetOLEProperties(string fileName)
    {
        StringDictionary dictionary = new StringDictionary();
        OleDocumentPropertiesClass oleDocument = null;
        try
        {
            oleDocument = new OleDocumentPropertiesClass();
            oleDocument.Open(fileName, true,
                dsoFileOpenOptions.dsoOptionOpenReadOnlyIfNoWriteAccess);

            // get the summary properties
            SummaryProperties properties = oleDocument.SummaryProperties;
            foreach (PropertyInfo property in
                typeof(SummaryProperties).GetProperties(
                    BindingFlags.Public | BindingFlags.Instance))
            {
                try
                {
                    object value = property.GetValue(properties, null);
                    if (value != null)
                        dictionary.Add(property.Name, value.ToString());
                }
                catch (Exception x)
                {
                    Debug.WriteLine(x.ToString());
                    continue;
                }
            }
        }
        catch { }
    }
}

```

```
    }  
    }  
  
    // get the custom properties  
    CustomProperties customProperties =  
        oleDocument.CustomProperties;  
  
    foreach (CustomProperty customProperty in customProperties)  
    {  
        string propertyValue = customProperty.get_Value().ToString();  
        if (!dictionary.ContainsKey(customProperty.Name))  
            dictionary.Add(customProperty.Name, propertyValue);  
        else  
        {  
            string name = string.Format(  
                "Custom({0})", customProperty.Name);  
            dictionary.Add(name, propertyValue);  
        }  
    }  
}  
finally  
{  
    // close the document  
    oleDocument.Close(false);  
    // release the unmanaged resource  
    Marshal.ReleaseComObject(oleDocument);  
  
}  
return dictionary;  
}  
}
```

This concludes the lab exercises.

Lab 2: Working with Content Types

Lab Time: 60 Minutes

Lab Directory: ECM401.ContentTypes

Lab Overview:

In this lab, you will write code that manipulates SharePoint Content Types via the SharePoint object model and API. The exercises in this lab take an imperative (code-based) rather than a declarative (xml-based) approach in order to familiarize you with the objects and methods provided by the SharePoint API for building solutions. Your code will focus both on discovering existing type definitions and creating new ones.

Exercise 1: Exploring the SharePoint API

1. In this exercise, you will write a console application that displays information about content types for a given site. Your utility will accept three parameters; the URL of the target site to scan, the name of a content type to be displayed and whether to recursively display all content types which inherit from the one specified.
2. Start by creating a new Console Application project in Visual Studio. Give it the name **ECM401.ContentTypeBrowser**.
3. In order to use the SharePoint API, you must add a reference to the Microsoft.SharePoint.dll file. Right-click the **References** node and select **Add Reference...**. Click on the **.NET** tab and scroll to the bottom of the list. Select **Windows® SharePoint® Services** and click OK.
4. Add the following line of code to the top of the **Program.cs** file.

```
using Microsoft.SharePoint;
```

5. Add three data members to the **Program** class as follows:

```
bool Recursive=false;  
string Url = "http://litwareinc.com";  
string TargetType = null;
```

6. Add the following code to the static **Main** method.

Code Snippet: 'Main Method'

```
static void Main(string[] args)  
{  
    bool isType = false;  
    bool recursive = false;  
    string baseType = null;  
    string url = "http://localhost:401/docman";
```

```

        foreach (string option in args)
        {
            switch (option.ToLower())
            {
                case "-type": isType = true; break;
                case "-recursive": recursive = true; break;
                default:
                    if (isType)
                    {
                        baseType = option.Replace("\\", "");
                        isType = false;
                    }
                    else
                    {
                        url = option.Replace("\\", "");
                    }
                    break;
            }
        }

        new Program(url, baseType, recursive).Run();

        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }

```

7. This handles the program arguments. Now you will implement the Program constructor and Run method. Add the following code to the Program class.

```

/// <summary>
/// Program constructor
/// </summary>
public Program(string url, string baseType, bool recursive)
{
    this.Url = url;
    this.TargetType = baseType;
    this.Recursive = recursive;
}

```

8. The Run method simply checks the data members to determine what to do. Enter the following code just after the Program constructor.

Code Snippet: 'Run Method'

```

/// <summary>
/// Executes the program.
/// </summary>
public void Run()
{
    // Open the site.
    using (SPSite site = new SPSite(this.Url))
    {
        // Get the root web.
        using (SPWeb web = site.OpenWeb())
        {
            // Prevent endless recursion.
            if (this.TargetType == null)
                this.Recursive = false;
        }
    }
}

```

```

        // Get the available content types.
        foreach (SPContentType contentType in
web.AvailableContentTypes)
        {
            if (this.TargetType == null)
                DisplayContentType(0,contentType);
            else if (contentType.Name.Equals(this.TargetType))
                DisplayContentType(0,contentType);
        }
    }
}

```

9. This method starts at the root web of the designated URL and obtains the catalog of available content types for the website. The AvailableContentTypes collection includes all content types that are visible from the target website up to the root web of the site collection. It does not include content types for website beneath the target web.
10. Now you will implement the **DisplayContentType** method, which simply displays the content type name at the specified indent level. If the recursive flag is turned on, then you will see a tree of content types showing the inheritance level for the selected type. Enter the following code beneath the Run method.

Code Snippet: 'DisplayContentType Method'

```

/// <summary>
/// Writes content type information to the console.
/// </summary>
/// <param name="indentLevel"></param>
/// <param name="contentType"></param>
public void DisplayContentType(int indentLevel, SPContentType
contentType)
{
    for (int i = 0; i < indentLevel; i++)
        Console.Write("  ");
    Console.WriteLine("{0}",contentType.Name);
    //Console.WriteLine(contentType.SchemaXml);
    if (this.Recursive)
    {
        foreach (SPContentType subType in
contentType.ParentWeb.AvailableContentTypes)
            if (DerivesFrom(contentType, subType))
                DisplayContentType(indentLevel + 1, subType);
    }
}

```

11. To determine if one content type derives from another, you can use the SharePoint API to walk up the inheritance tree. Enter the following method beneath the DisplayContentType method.

Code Snippet: 'DerivesFrom Method'

```

/// <summary>
/// Returns true if the child is derived from the parent.
/// </summary>

```

```
/// <param name="parentType"></param>
/// <param name="childType"></param>
/// <returns></returns>
private bool DerivesFrom(SPContentType parentType, SPContentType
childType)
{
    while (childType.Name != "System")
    {
        childType = childType.Parent;
        if (childType == parentType)
            return true;
    }
    return false;
}
```

12. Build the project and open a command prompt in the bin\debug folder. Enter the following command line:

```
ECM401.ContentTypeBrowser.exe -type Item -recursive
```

13. The resulting window should resemble the following diagram.



Exercise 2: Creating Content Types using XML

1. In the next exercise, you will create a SharePoint Feature that installs a custom content type called Promissory Note. The Promissory Note content type will include the following metadata fields:
 - Amount
 - Due Date
 - Interest Rate
2. Start by creating a new **SharePoint Feature** project in Visual Studio. Give it the name **ECM401.PromissoryNote** and delete the auto-generated **FeatureReceiver.cs** and **FeatureReceiverBase.cs** files. You won't be adding any code to the project, just XML.
3. Open the generated **feature.xml** file for editing. Remove the two highlighted lines shown below.

```
<?xml version="1.0" encoding="utf-8" ?>
<Feature xmlns="http://schemas.microsoft.com/sharepoint/"
Id="db85ec29-731a-4ace-810d-2675b3ef7b79"
Hidden="FALSE"
Title="ECM401.PromissoryNote"
Description="Creates a simple promissory note content type."
ImageUrl="Ted Pattison Group\bug_brwn_pith.gif"
ReceiverAssembly="ECM401.PromissoryNote, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=a9ca7666007df3c3"
ReceiverClass="ECM401.PromissoryNote.FeatureReceiver"
Scope="Site"
Version="1.0.0.0">

<ElementManifests>
<ElementManifest Location="elements.xml"/>
</ElementManifests>

</Feature>
```

Note: The actual feature **Id** value in your code will be different.

4. Now open the **elements.xml** file for editing. Select the entire contents of this file and replace the contents with the following XML code.

XML Snippet: 'Promissory Note Content Type'

```
<?xml version="1.0" encoding="utf-8" ?>
<Elements xmlns="http://schemas.microsoft.com/sharepoint/">
  <ContentType ID="0x01010070ECF29474514071B9E7F9D991D3DC05"
    Name="Promissory Note"
    Group="ECM401"
    Description="Defines fields for a simple promissory note."
    Sealed="FALSE"
    Version="0"
    ReadOnly="FALSE"
    BaseType="0x0101">

    <FieldRefs>
    </FieldRefs>

    <DocumentTemplate TargetName="/_layouts/ecm401/promissory note.docx" />
  </ContentType>
</Elements>
```

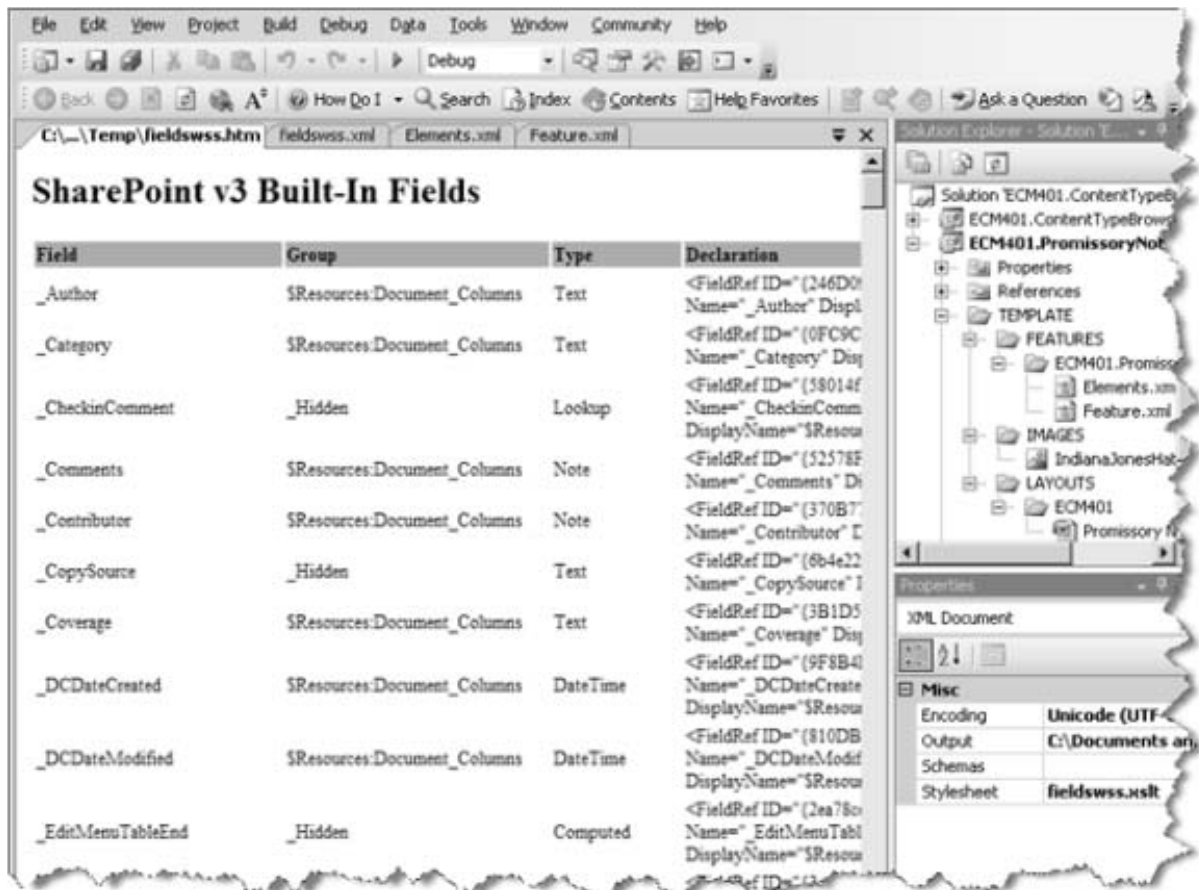
5. Working with content types in XML can be challenging, because in order to declare field references, you have to know the unique identifier of each field you want to use. Because of this, you will use a trick to obtain the field identifiers for the title and comments fields.
6. Right-click on the project node and select **Add -> Existing Item...** from the context menu. Browse to **C:\Student\Resources** and select the files **fieldswss.xml** and **fieldswss.xslt**. The fieldswss.xml file is a copy of the default field declaration file used by WSS. The fieldswss.xslt is a custom XSLT stylesheet you will use to transform the raw field declaration file into a table with field declarations you can simply copy into your content type definition.
7. The XSLT code you will reference looks like this:

XML Snippet: 'SharePoint Fields XSL Stylesheet'

```
<?xml version="1.0" encoding="utf-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:wss="http://schemas.microsoft.com/sharepoint/">
  <xsl:output method="html" version="1.0" encoding="utf-8" indent="yes" />
  <xsl:template match="wss:Elements">
    <html>
      <body>
        <h2>SharePoint 3.0 Built-In Fields</h2>
        <table border="0" width="100%" style="font-size:9pt;">
          <tr bgcolor="#9acd32">
            <th align="left">Field</th>
            <th align="left">Group</th>
            <th align="left">Type</th>
            <th align="left">Declaration</th>
          </tr>
          <xsl:apply-templates>
            <xsl:sort select="@Name" />
          </xsl:apply-templates>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="wss:Field">
    <tr>
      <td>
        <xsl:value-of select="@Name" />
      </td>
      <td>
        <xsl:value-of select="@Group" />
      </td>
      <td>
        <xsl:value-of select="@Type" />
      </td>
      <td>
        <!-- Emit The Full Declaration -->
        <xsl:element name="FieldRef">
          <xsl:attribute name="ID">
            <xsl:value-of select="@ID"/>
          </xsl:attribute>
          <xsl:attribute name="Name">
            <xsl:value-of select="@Name"/>
          </xsl:attribute>
          <xsl:attribute name="DisplayName">
            <xsl:value-of select="@DisplayName"/>
          </xsl:attribute>
        </xsl:element>
      </td>
    </tr>
  </xsl:template>
</xsl:stylesheet>
```

8. Double-click the **fieldswss.xml** file to open it for editing. Open the property pane and enter fieldswss.xslt into the Stylesheet property.
9. With the fieldswss.xml file still open, select **Show XSLT Output** from the **XML** menu in Visual Studio. When you are done, you should see a table like the one in the following diagram.



10. Browse the list of fields until you find the **Title** and **Comments** fields. Copy the contents of the **Declaration** column and paste the full declaration into the elements.xml file just inside the <FieldRefs> tag of the content type declaration.
11. In addition to the two built-in fields, you will also add three custom fields for the **Due Date** , **Interest Rate** and **Total Amount** of the promissory note. Add the following lines to the elements.xml file just under the opening <Elements> tag.

XML Snippet: 'Promissory Note Field References'

```
<!-- Declare some fields for use in the content type -->

<Field ID="{E718EA82-8127-481b-A704-41EA74F14E87}"
  Name="DueDate"
  SourceID="http://schemas.microsoft.com/sharepoint/v3"
  StaticName="DueDate"
  Group="ECM401"
  Type="DateTime"
  Sealed="FALSE"
  ReadOnly="FALSE"
  Hidden="FALSE"
  DisplayName="Due Date"
  StorageTZ="TRUE">
</Field>

<Field ID="{4BA9D3E0-ABA5-415d-89CD-CC34B1169071}"
```

```

Name="InterestRate"
SourceID="http://schemas.microsoft.com/sharepoint/v3"
StaticName="InterestRate"
Group="ECM401"
Type="Number"
DisplayName="Interest Rate"
ReadOnly="FALSE"
Sealed="FALSE">
</Field>

<Field ID="{4C736512-4E2A-4a7b-9C96-D4624B51995E}"
Name="TotalAmount"
SourceID="http://schemas.microsoft.com/sharepoint/v3"
StaticName="TotalAmount"
Group="ECM401"
Type="Number"
DisplayName="Total Amount"
ReadOnly="FALSE"
Sealed="FALSE">
</Field>

```

Note: When creating new field declarations, you need a unique GUID for each field identifier.

12. To complete your content type declaration, you must add field references to the three new fields you have just declared. When you are finished, the content definition should look like this.

```

<!-- Declare the content type -->

<ContentType ID="0x01010070ECF29474514071B9E7F9D991D3DC05"
Name="Promissory Note"
Group="ECM401"
Description="Defines fields for a simple promissory note."
Sealed="FALSE"
Version="0"
ReadOnly="FALSE"
BaseType="0x0101">

<FieldRefs>
<FieldRef ID="{fa564e0f-0c70-4ab9-b863-0177e6ddd247}" Name="Title"
DisplayName="$Resources:core,Title;" Required="FALSE"/>
<FieldRef ID="{9da97a8a-1da5-4a77-98d3-4bc10456e700}" Name="Comments"
DisplayName="$Resources:core,Description;" Required="FALSE"/>
<FieldRef ID="{E718EA82-8127-481b-A704-41EA74F14E87}" Name="DueDate"
DisplayName="Due Date" Required="TRUE"/>
<FieldRef ID="{4BA9D3E0-ABA5-415d-89CD-CC34B1169071}" Name="InterestRate"
DisplayName="Interest Rate" Required="TRUE"/>
<FieldRef ID="{4C736512-4E2A-4a7b-9C96-D4624B51995E}" Name="TotalAmount"
DisplayName="Total Amount" Required="TRUE"/>
</FieldRefs>

<DocumentTemplate TargetName="/_layouts/ecm401/promissory note.docx" />
</ContentType>

```

Note: The **DocumentTemplate** tag is used to specify the template that should be used for the content type.

13. Now you are ready to test your work. Save all files and build the project.

14. Although you are not actually writing any compiled code, the post-build event batch will copy the feature definition into the appropriate locations and then call **STSADM** to install the feature into the site collection.

Note: You will still need to manually activate the feature for any sub-site in which you wish to use the promissory note content type.

15. Open the browser and navigate to the <http://localhost:401/docmansite>. Click on Site Settings from the Site Actions menu and then choose Site Collection features. You should see **ECM401 - Promissory Note** next to an Indiana Jones hat. Click the **Activate** button and then return to the home page of the site.
16. Click on the **Shared Documents** link and then select **Document Library Settings** from the **Settings** menu. Click the **Advanced Settings** link under **General Settings** and select **Yes** under **Allow management of content types?** and then click **OK**.
17. On the **Customize Shared Documents** page, under the **Content Types** section, click the **Add from existing site content types** link. You should see **Promissory Note** in the list. Double-click it and then click the **OK** button at the bottom of the page.
18. Return to the **Shared Documents** page and click the **New** button. Select **Promissory Note** from the dropdown menu. **Microsoft Word** should open with a new document based on the promissory note document template.
19. Click the **Save** button to save the document back to the document library. When Word prompts you to select a document type, select **Promissory Note**. You will now see an error message saying that required fields are missing. This is because you specified that the **Due Date**, **Interest Rate** and **Total Amount** fields are required.

Note: If you see an error message when you try to save the document, it means that the **Event Notification** service is running. This service can sometimes interfere with network connectivity from within Word 2007. Disable the service by running **net stop sens** from the command line.
20. Open the **Document Information Panel** as directed, and enter some values for these fields. When you are done, continue saving the file to the **Shared Documents** library.

This concludes the lab exercises.

Lab 3: Content Auditing

Lab Time: 45 Minutes

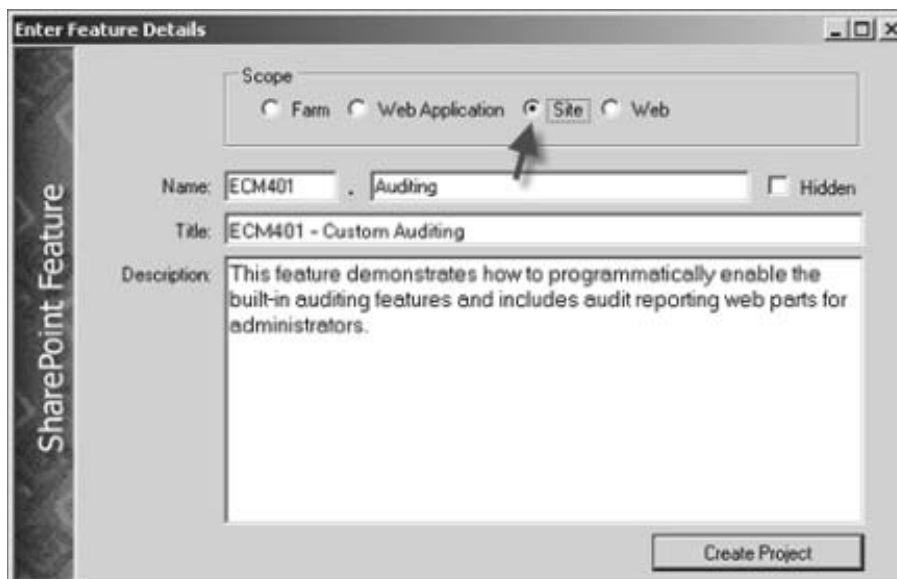
Lab Directory: ECM401.Auditing

Lab Overview:

In this lab, you will create a SharePoint feature to enable item-level auditing for any SharePoint list or document library. The built-in auditing support provided by Windows SharePoint Services must be enabled programmatically, so you will start by writing a feature receiver that enables auditing for all lists in the site collection. Then you will refine the feature to limit auditing to a specific list or document library.

Exercise 1: Create the SharePoint Feature project

1. Start by creating a **SharePoint Feature** project named **ECM401.Auditing** using the **SharePoint Feature** project template. When completing the **Feature Details** dialog, be sure to select the **Site** scope for the feature as shown below.



Note: You are setting the feature scope to "**Site**" so that it can be activated for all sites in the site collection. If you open the generated **feature.xml** file, you will notice that the **ReceiverAssembly** and **ReceiverClass** attributes are automatically set by the project template to reference the project assembly and the generated **FeatureReceiver** class.

2. Open the **FeatureReceiver.cs** file for editing. The generated **FeatureReceiver** class allows you to implement methods of the abstract base class **SPFeatureReceiver**.
3. Next, you will handle the **FeatureActivated** event. Replace the **FeatureActivated** method with the following code snippet.

Code Snippet: 'Auditing - FeatureActivated'

```
public override void FeatureActivated(SPFeatureReceiverProperties properties)
{
    SPSite site = properties.Feature.Parent as SPSite;
    if (site != null)
    {
        // Enable auditing for all items in the site collection.
        site.Audit.AuditFlags = SPAuditMaskType.All;
        site.Audit.Update();

        // Modify the top-level website title to indicate that auditing is on.
        // Save the title in the property bag for the site.
        SPWeb web = site.RootWeb;
        web.Properties[TitleKey] = web.Title;
        web.Properties.Update();
        web.Title = string.Format("{0} - Audited", web.Title);
        web.Update();
    }
}
```

4. Similarly, replace the generated **FeatureDeactivating** method with the following code snippet.

Code Snippet: 'Auditing - FeatureDeactivating'

```
public override void FeatureDeactivating(SPFeatureReceiverProperties
properties)
{
    SPSite site = properties.Feature.Parent as SPSite;
    if (site != null)
    {
        // Disable auditing.
        site.Audit.AuditFlags = SPAuditMaskType.None;

        // Restore the original title.
        SPWeb web = site.RootWeb;
        web.Title = web.Properties[TitleKey];
        web.Update();
    }
}
```

5. Save the file and build the project.
6. Test your work by navigating to **http://localhost** . Select **Site Settings** from the **Site Actions** dropdown menu and then click the **Site Collection Features** link.
7. On the **Site Collection Features** page, scroll down to the **ECM401.Auditing** feature and click the **Activate** button.
8. The root web title changes to a bracketed string indicating that the feature was successfully activated and that auditing is now enabled.

9. Verify this by navigating back to the **Site Settings** page and then click on **Configure Audit Settings** from the **Site Collection Administration** section. The **Configure Audit Settings** page should now have all items checked as shown below.

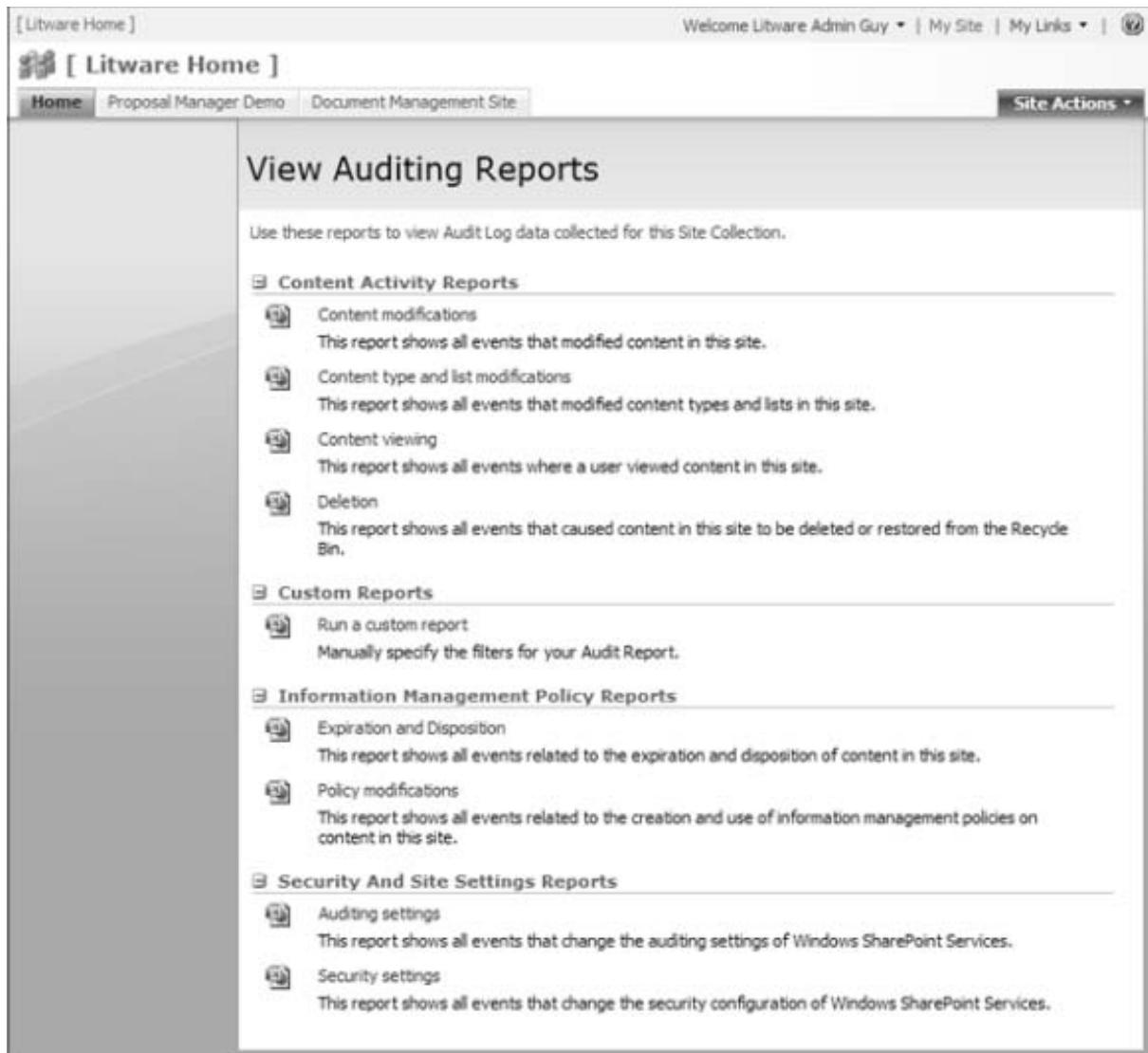
[Litware Home] > Site Settings > Configure Audit Settings

Configure Audit Settings

Documents and Items Specify the events that should be audited for documents and items within this site collection.	Specify the events to audit: <ul style="list-style-type: none"><input checked="" type="checkbox"/> Opening or downloading documents, viewing items in lists, or viewing item properties<input checked="" type="checkbox"/> Editing items<input checked="" type="checkbox"/> Checking out or checking in items<input checked="" type="checkbox"/> Moving or copying items to another location in the site<input checked="" type="checkbox"/> Deleting or restoring items
Lists, Libraries, and Sites Specify the events that should be audited for lists, libraries, and sites within this site collection.	Specify the events to audit: <ul style="list-style-type: none"><input checked="" type="checkbox"/> Editing content types and columns<input checked="" type="checkbox"/> Searching site content<input checked="" type="checkbox"/> Editing users and permissions

OK Cancel

10. To view the audit log reports, you have to activate the MOSS reporting feature. Navigate to the **Site Collection Features** page and scroll down until you see the **Reporting** feature. Click the **Activate** button and then return to the **Site Settings** page. In the **Site Collection Administration** section, there is now a link for **Audit log reports** . Click this link to open the **View Auditing Reports** page. Click any of the links to view a report in Microsoft Excel.



Exercise 2: Display the Audit Log in WSS

Since the Excel reports are available only in MOSS, another technique for viewing audit entries is to display them manually using a custom application page. In this exercise, you will extend the auditing feature to include a custom action that lets site administrators view the audit log. The custom action will be hidden from non-administrators.

1. From the Visual Studio **Solution Explorer**, open the **elements.xml** file for editing and enter the following code inside the <Elements> tag.

XML Snippet: 'Auditing - Custom Action'

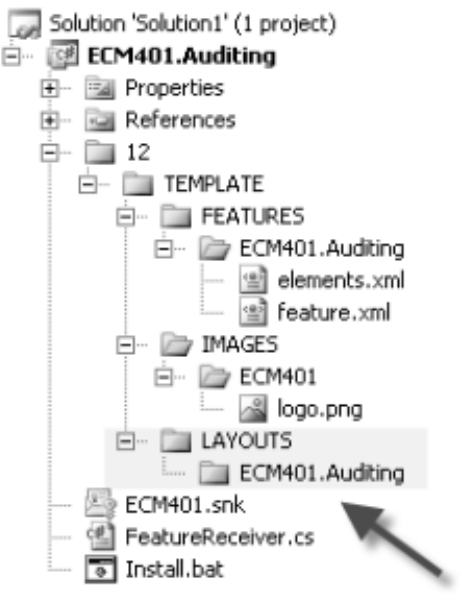
```
<CustomAction
  Id="SiteActions_ViewAuditLog"
  GroupId="SiteActions"
  Location="Microsoft.SharePoint.StandardMenu"
  Sequence="1001"
```

```

Title="View Audit Log"
Description="Display the audit log for this site collection."
RequiresSiteAdministrator="TRUE"
ImageUrl="/_layouts/images/LTTXTBOX.GIF">
<UrlAction Url="~sitecollection/_layouts/ECM401.Auditing/AuditLog.aspx" />
</CustomAction>

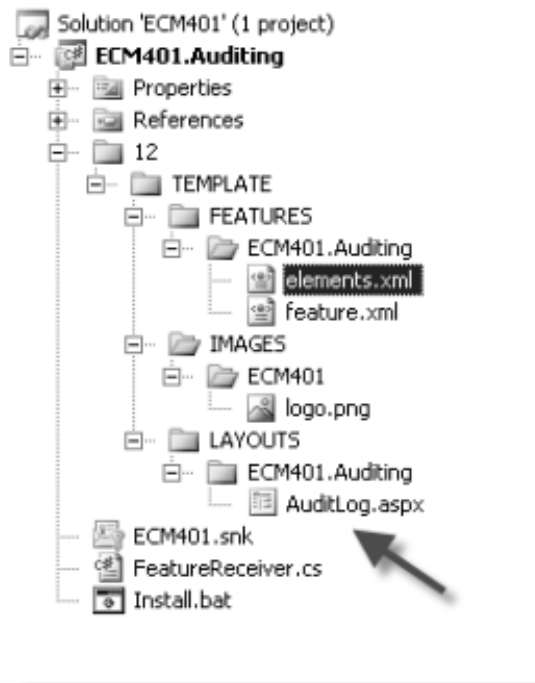
```

2. Next, you will create a custom application page to display the audit log entries.
3. Add a new folder to the project under the **12\TEMPLATE** folder called **LAYOUTS** . Within that folder, add another subfolder called **ECM401.Auditing** as shown below.



4. Right-click the folder and select **Add -> New Item...** from the context menu and create a new **TEXT** file with the name **AuditLog.aspx**

Note: Although you selected a text file type, Visual Studio will still open the file as a web page because of the **.ASPX** extension.



5. Enter the following code for the page. Since Visual Studio code snippets don't work well for ASPX pages, you will have to copy and paste.

```
<%@ Assembly Name="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce11e9429c" %>
<%@ Page Language="C#" MasterPageFile="~/_layouts/application.master"
Inherits="Microsoft.SharePoint.WebControls.LayoutsPageBase" %>
<%@ Import Namespace="Microsoft.SharePoint" %>
<%@ Import Namespace="System.Data" %>
<%@ Register TagPrefix="SharePoint"
Namespace="Microsoft.SharePoint.WebControls"
Assembly="Microsoft.SharePoint, version=12.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce11e9429c" %>

<script runat="server">
// *** code will go here ***
</script>
<asp:Content ID="contentMain" ContentPlaceholderID="PlaceholderMain"
runat="server">
<asp:Button ID="btnRefresh" runat="server" Text="Refresh"
OnClick="btnRefresh_Click" />
<asp:Button ID="btnClear" runat="server" Text="Clear Log"
OnClick="btnClear_Click" />
<hr />
<SharePoint:SPGridView ID="grid" runat="server" AutoGenerateColumns="False"
Width="100%">
<AlternatingRowStyle CssClass="ms_alternating" />
</SharePoint:SPGridView>
</asp:Content>

<asp:Content ID="contentTitle" ContentPlaceholderID="PlaceholderPageTitle"
runat="server">Audit Log</asp:Content>
<asp:Content ID="contentTitleTitle"
ContentPlaceholderID="PlaceholderPageTitleInTitleArea" runat="server">Custom
```

```
display of audit log entries.</asp:Content>
```

Note: To make your application page look like the other SharePoint landing pages, you will inherit from **LayoutsPageBase** and use the same **application.master** used by the other pages. Since the page is deployed to every Web front-end in the farm by an administrator, you can enter inline code.

6. This code sets up the page elements, which are an instance of the SPGridView control and two buttons - one to refresh the view and another to clear the log entries. Note: Clearing the log entries is a destructive action and might require more thought in a real application.
- 7.
8. Now you are ready to add some code inside the script block at the top of the page. The first routine you add will lock down the page so that only administrators can view it.

```
protected override bool RequireSiteAdministrator {
get { return true; }
}
```

9. Next you will handle the page load event to create a **DataTable** that can be bound to the **SPGridView** control.

Code Snippet: 'Auditing - AuditLog OnLoad'

```
protected override void OnLoad(EventArgs e) {
    // Access the site collection
    SPSite siteCollection = SPContext.Current.Site;
    SPWeb site = SPContext.Current.Web;

    // Create an Audit Query object.
    SPAuditQuery query = new SPAuditQuery(siteCollection);
    SPAuditEntryCollection entries = siteCollection.Audit.GetEntries(query);

    // Loop through the entries to create a data table
    DataTable table = new DataTable();
    table.Columns.Add("User", typeof(string));
    table.Columns.Add("Source", typeof(string));
    table.Columns.Add("Type", typeof(string));
    table.Columns.Add("ID", typeof(string));
    table.Columns.Add("Event", typeof(string));
    table.Columns.Add("Date", typeof(DateTime));

    DataRow row;

    foreach (SPAuditEntry entry in entries)
    {
        row = table.Rows.Add();
        row["User"] = site.SiteUsers.GetByID(entry.UserId).Name;
        row["Source"] = entry.DocLocation;
        row["Type"] = entry.ItemType.ToString();
        row["ID"] = entry.ItemId.ToString();
        row["Event"] = entry.Event;
        row["Date"] = entry.Occurred.ToLocalTime();
    }

    // Bind the columns to matching fields in the table.
    AddGridField("User", "User");
```

```

AddGridField("Source", "Source");
AddGridField("Type", "Type");
AddGridField("ID", "ID");
AddGridField("Event", "Event");
AddGridField("Date", "Date").ControlStyle.Width = new Unit(120);

// Bind the datasource to the grid.
grid.AutoGenerateColumns = false;
grid.DataSource = table.DefaultView;
grid.DataBind();
grid.AllowSorting=true;
grid.HeaderStyle.Font.Bold = true;
}

// Adds a bound field to the grid.
private SPBoundField AddGridField(string headerText, string fieldName)
{
    SPBoundField field = new SPBoundField();
    field.HeaderText = headerText;
    field.DataField = fieldName;
    grid.Columns.Add(field);
    return field;
}

```

10. Finally, add the button click event handlers.

Code Snippet: 'Auditing - AuditLog ButtonClick'

```

// Handle the refresh button click.
protected void btnRefresh_Click(object sender, EventArgs e)
{
    Response.Redirect(Request.RawUrl);
}

// Handle the clear button click.
protected void btnClear_Click(object sender, EventArgs e)
{
    SPSite siteCollection = SPContext.Current.Site;
    siteCollection.Audit.DeleteEntries(DateTime.Now.ToLocalTime().AddDays(1));
    siteCollection.Audit.Update();
    Response.Redirect(Request.RawUrl);
}

```

11. To test your work, build the project and then navigate to the **Site Settings** page and select the **Site Collection Features** link.
12. Deactivate and then re-activate the **ECM401.Auditing** Feature.
13. From the **Site Actions** menu, click the new **View Audit Log** link. Your page should resemble the one shown

below.

Custom display of audit log entries.					
<input type="button" value="Refresh"/>		<input type="button" value="Clear Log"/>			
User	Source	Type	ID	Event	Date
Litware Admin Guy	proposals/default.aspx	Document	2c999146-8b4d-446e-bd63-16a4f4d22b33	View	1/14/2008 5:41:36 AM
Litware Admin Guy	docman/default.aspx	Document	5d716f52-8e76-4e27-b5fc-242bc39596c7	View	1/14/2008 5:41:24 AM
Litware Admin Guy		Site	be50c034-ba5f-4299-8d81-e3991b432f9f	EventsDeleted	1/13/2008 11:39:12 PM
Litware Admin Guy	/default.aspx	Document	33aec399-1810-4a5f-ba76-ed47cbe468c7	View	1/14/2008 5:28:38 AM

14.

Exercise 3: Enable Item-Level Auditing

In this exercise, you will enable site users to display auditing event entries for individual list items. This will require a custom menu item added to the drop-down menu associated with each item. When the user selects the menu command, another custom application page will be displayed showing audit events for that item.

1. Start by adding another custom action to the **elements.xml** file.

XML Snippet: 'Auditing - ECB CustomAction'

```
<CustomAction
  Id="ECB_ViewAuditLog"
  RegistrationType="List"
  RegistrationId="101"
  ImageUrl="/_layouts/images/GORTL.GIF"
  Location="EditControlBlock"
  Sequence="401"
```

```

    Title="View Audit History">

<UrlAction
Url=~site/_layouts/ECM401.Auditing/itemAudit.aspx?ItemId={ItemId}&ListId={
ListId}"/>

</CustomAction>

```

Note: The **UrlAction** element includes special tokens that supply the current list and list item identifiers to the page. Also note that the url is specified relative to the current **site** and not the site collection.

2. Now you will create the **ItemAudit.aspx** page. Right-click on the **12\TEMPLATE\LAYOUTS\ECM401.Auditing** folder and select **Add -> New Item...** from the context menu.
3. Create a new text file named **ItemAudit.aspx** and enter the following code.

```

<%@ Assembly Name="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce11e9429c" %>
<%@ Page Language="C#" MasterPageFile=~/_layouts/application.master"
Inherits="Microsoft.SharePoint.WebControls.LayoutsPageBase" %>
<%@ Import Namespace="Microsoft.SharePoint" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.IO" %>
<%@ Import Namespace="System.Security" %>
<%@ Register TagPrefix="SharePoint"
Namespace="Microsoft.SharePoint.WebControls"
Assembly="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce11e9429c" %>
<%@ Register TagPrefix="Utilities" Namespace="Microsoft.SharePoint.Utilities"
Assembly="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce11e9429c" %>

<script runat="server">
// *** CODE GOES HERE ***
</script>

<asp:Content ID="Content6" ContentPlaceHolderID="PlaceHolderMain"
runat="server">
<SharePoint:SPGridView ID="grid" runat="server" AutoGenerateColumns="False"
Width="100%">
<AlternatingRowStyle CssClass="ms-alternating" />
</SharePoint:SPGridView>
</asp:Content>
<asp:Content ID="Content1" ContentPlaceHolderID="PlaceHolderPageTitle"
runat="server">
<asp:Label ID="ListTitle" runat="server" />
</asp:Content>
<asp:Content ID="Content2"
ContentPlaceHolderID="PlaceHolderPageTitleInTitleArea"
runat="server">
<asp:Label ID="ItemName" runat="server" />
</asp:Content>

```

4. This code sets up the page elements, which include an **SPGridView** to display the audit entries and two labels - one to show the list title and another to show the selected item name.

5. Now you will add inline code to handle the page load event. Insert the following code inside the <script> tag at the highlighted location.

Code Snippet: 'Auditing - ItemAudit PageLoad'

```
protected void Page_Load(object sender, EventArgs e) {

    SPSite siteColl = SPContext.Current.Site;
    SPWeb site = SPContext.Current.Web;

    string ListId = Request.QueryString["ListId"];
    SPList theList = site.Lists[new Guid(ListId)];

    string ItemId = Request.QueryString["ItemId"];
    SPListItem theItem = theList.Items.GetItemById(Convert.ToInt32(ItemId));

    // Append a custom audit event to record that this page has been viewed.
    theItem.Audit.WriteAuditEvent(SPAuditEventType.Custom,
    "CustomViewAuditEvent", "");

    /*** ADDITIONAL CODE GOES HERE ***/

}

private SPBoundField AddGridField(string headerText, string fieldName)
{
    SPBoundField field = new SPBoundField();
    field.HeaderText = headerText;
    field.DataField = fieldName;
    grid.Columns.Add(field);
    return field;
}

string GetUserNameById(int UserId, SPWeb site)
{
    try
    {
        return site.SiteUsers.GetByID(UserId).Name;
    }
    catch
    {
        return UserId.ToString();
    }
}

// Parse the version string for readability.
protected string ParseVersionNumber(string versionString) {
    try {
        int startMajor = versionString.IndexOf("<Major>") + 7;
        int endMajor = versionString.IndexOf("</Major>");
        int lengthMajor = endMajor - startMajor;
        int startMinor = versionString.IndexOf("<Minor>") + 7;
        int endMinor = versionString.IndexOf("</Minor>");
        int lengthMinor = endMinor - startMinor;

        string majorNumber = versionString.Substring(startMajor, lengthMajor);
        string minorNumber = versionString.Substring(startMinor, lengthMinor);

        if (majorNumber == "0" && minorNumber == "-1")
            return "N/A";

        return majorNumber + "." + minorNumber;
    }
}
```

```

    }
    catch {
        return "N/A";
    }
}

```

6. To retrieve audit log entries in the context of a non-administrator, elevated security privileges are required. To achieve this use the `SPSecurity` class by adding the following code to the `PageLoad` method. This code replaces the comment `"/*** ADDITIONAL CODE GOES HERE ***/"`

Code Snippet: 'Auditing - ItemAudit SPSecurity'

```

SPSecurity.RunWithElevatedPrivileges(delegate() {
    using (SPSite ElevatedSiteCollection = new SPSite(siteColl.ID)) {
        using (SPWeb ElevatedSite = ElevatedSiteCollection.OpenWeb(site.ID)) {

            SPList list = ElevatedSite.Lists[new Guid(ListId)];
            ListTitle.Text = "List: " + list.Title;

            SPLListItem item = list.Items.GetItemById(Convert.ToInt32(ItemId));
            ItemName.Text = "Item: " + item.Name;

            SPAuditQuery query;
            SPAuditEntryCollection auditEntries;

            query = new SPAuditQuery(ElevatedSiteCollection);
            query.RestrictToListItem(item);
            auditEntries = ElevatedSite.Audit.GetEntries(query);

            DataTable table = new DataTable();
            table.Columns.Add("User", typeof(string));
            table.Columns.Add("Event", typeof(string));
            table.Columns.Add("Date", typeof(DateTime));
            table.Columns.Add("Version", typeof(string));

            DataRow newRow;

            foreach (SPAuditEntry entry in auditEntries) {
                newRow = table.Rows.Add();
                newRow["User"] = GetUserNameById(entry.UserId, site);

                // check for our custom event
                if (entry.SourceName == "CustomViewAuditEvent") {
                    newRow["Event"] = "View Audit Log";
                }
                else {
                    newRow["Event"] = entry.Event;
                }
                newRow["Date"] = entry.Occurred.ToLocalTime();
                newRow["Version"] = ParseVersionNumber(entry.EventData);
            }

            AddGridField("User", "User");
            AddGridField("Event", "Event");
            AddGridField("Date", "Date").ControlStyle.Width = new Unit(140);
            AddGridField("Version", "Version");

            grid.AutoGenerateColumns = false;
            grid.DataSource = table.DefaultView;
            grid.DataBind();
        }
    }
}

```

```
        grid.AllowSorting = true;  
        grid.HeaderStyle.Font.Bold = true;  
    }  
};
```

7. Rebuild the project, re-activate the feature and test your work. Now, from the **Edit Control Block** of any document library, you should see an extra command to display the audit log entries for the selected item. Select the command to open a page showing the audit log entries for the item.

This concludes the lab exercises.

Lab 4: Information Management Policy

Lab Time: 60 Minutes

Lab Directory: ECM401.InformationPolicy

Back Story:

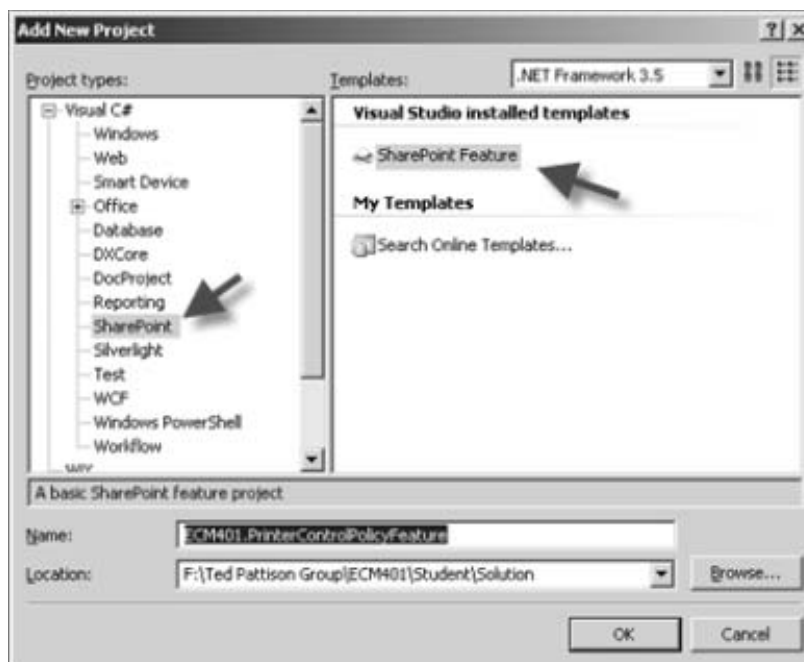
Litware management has decided to use Windows SharePoint Services 3.0 to manage sensitive company information. Because there are many different kinds of documents, the Litware IT department has decided to use information policy features to enable content administrators to apply different kinds of controls to various documents without having to create a separate feature for each document type.

Lab Overview:

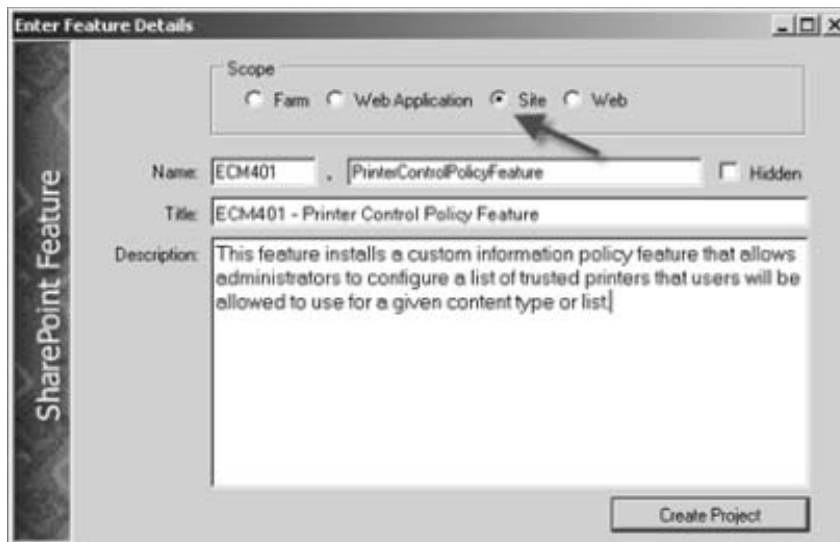
In this lab, you will create a custom information policy feature that controls the distribution of printed material by limiting the printing of certain documents to a specific set of secure printers. Your policy feature will contain a list of trusted printers, and will work in conjunction with a **VSTO** add-in that traps the Print event to determine whether a trusted printer is being used. If not, then the print job is cancelled.

Exercise 1: Create a SharePoint Feature Project

1. When working with policy features, it is useful to have a library of components that can be reused in several projects. In this exercise, you will employ a set of utility classes that greatly simplify the creation of custom policies, policy features and policy resources. Start by creating a new **SharePoint Feature** project named **ECM401.PrinterControlPolicyFeature** as shown below.



- Because Information Policy Features are installed globally, your feature could be scoped to any level. For convenience and in anticipation of adding additional components, such as content types and fields, to your solution, set the feature scope to **Site** as shown in the following illustration.



- In order to use the records management features, you will need to add a reference to the **Microsoft.Office.Policy** assembly. Right-click the References node and select Add Reference.... Click on the **Browse** tab and navigate to the **12\ISAPI** folder. Select the **Microsoft.Office.Policy.dll** file and click ok.
- Next, add a reference to the **ECM401** component library, which is located in the **student\solution\ecm401** folder. You can either add a reference to the assembly, or you can load the project and add a dependency reference to it.

Note: Browse through the source code to familiarize yourself with how the base classes are written. This project contains a number of classes that are designed to simplify the development of ECM solutions.

- The **ECM401** project includes some additional references you need to add in order for your project to build. First, add a reference to the **System.Configuration.Install** assembly from the **.NET** tab. You will also need the **System.Web** assembly.

Exercise 2: Create the Printer Control Policy Feature class

- Policy features allow administrators to select which parts of a policy should be applied to different items. In this exercise, you will create a custom policy feature that maintains a list of "trusted" printers that users are allowed to send output to. In the final exercise, you will create a **VSTO** add-in that uses this information to prevent the user from printing to an untrusted printer.
- Add a class to the project called **PrinterPolicyFeature** and open the source file for editing. Add the following **using** statements at the top of the file.

```
using System;
using System.ComponentModel;
```

```
using System.Diagnostics;
using System.Xml;
using Microsoft.Office.RecordsManagement.InformationPolicy;
using Microsoft.SharePoint;

using ECM401.ContentTypes;
using ECM401.InformationPolicy;
```

3. Delete the generated class declaration and replace it with the following code snippet.

Code Snippet: 'PrinterPolicyFeature - Class Declaration'

```
[Name("Document Print Controller")]
[Description("Maintains a list of 'trusted' printers so that administrators "
    + "can control where documents are printed.")]
[Publisher("John F. Holliday")]
public class PrinterPolicyFeature : SharePointPolicyFeature
{
    public const string TrustedPrintersFieldName = "TrustedPrinters";
    public const string PrintControlPolicyNamespace =
        "urn:ecm401:policy.printcontrol";

    public PrinterPolicyFeature()
    {
    }
}
```

4. Your policy feature class inherits from the **ECM401.InformationPolicy.SharePointPolicyFeature** abstract base class. This class provides a default implementation of the **IPolicyFeature** interface, which must be implemented to enable SharePoint to communicate with the policy feature. Using an abstract base allows you to implement only the methods you require.
5. The base class also uses **attributes** to generate the CAML code needed to register your component in the SharePoint environment. The *TrustedPrintersFieldName* and *PrintControlPolicyNamespace* constants will be used to setup the content types and list items affected by the policy.
6. First, create a static method for writing trace messages during development. Add the following code to the class definition.

```
/// <summary>
/// Helper method for logging trace messages.
/// </summary>
public static void Log(string message)
{
    Trace.WriteLine(message, "ECM401.PrinterPolicyFeature");
}
```

7. Policy features work by hooking into the event receiver mechanism for the lists and content types to which they are attached. When the printer control policy feature is registered for a content type, it will setup event receivers for the **ItemAdded** and **ItemUpdated** events so that it can copy the list of trusted printers into a special column on the target list item. To make it easier to register

and unregister event receivers, the **ECM401** utility library includes a class called **ItemEventReceiver**. In the next part of the exercise, you will inherit from this class to declare event receiver methods for these two event types.

8. Add a new *nested class* declaration inside the **PrinterPolicyFeature** class named **PrinterPolicyEventReceiver** that inherits from **ItemEventReceiver**. Your code should match the following code snippet.

Code Snippet: 'PrinterControlFeature - ItemEventReceiver'

```
/// <summary>
/// This class implements the event receivers for the policy feature.
/// </summary>
public class PrinterPolicyEventReceiver : ItemEventReceiver
{
    /// <summary>
    /// Handles the item added event to set the list of trusted printers.
    /// </summary>
    /// <param name="properties"></param>
    public override void ItemAdded(SPIItemEventProperties properties)
    {
        base.ItemAdded(properties);
        Log("PrinterPolicyFeature.ItemAdded - " + properties.ListTitle);
        AddPrintersToListItem(properties.ListItem);
    }

    /// <summary>
    /// Handles the item updated event to set the list of trusted printers.
    /// </summary>
    /// <param name="properties"></param>
    public override void ItemUpdated(SPIItemEventProperties properties)
    {
        base.ItemUpdated(properties);
        Log("PrinterPolicyFeature.ItemUpdated - " + properties.ListTitle);
        AddPrintersToListItem(properties.ListItem);
    }
}
```

9. Both of these routines delegate the job of adding the list of printers to the list item whenever a new item is added or an existing item is updated. The list of printers is extracted from the content type *payload* (a custom XMLDocument added by the policy feature) and placed into the *TrustedPrinters* field of the list item. Add the following code snippet to the *PrinterPolicyEventReceiver* class definition.

Code Snippet: 'PrinterPolicyEventReceiver - AddPrintersToListItem'

```
/// <summary>
/// Gets the list of trusted printers from the content type associated
/// with the item and places it into the appropriate field.
/// </summary>
/// <param name="item"></param>
private void AddPrintersToListItem(SPLListItem item)
{
    if (item == null || item.ContentType == null)
        return;
    Log("AddPrintersToListItem '" + item.Title + "'");
}
```

```

        string xml =
item.ContentType.XmlDocuments[PrinterPolicyFeature.PrintControlPolicyNamespace]
;
        if (string.IsNullOrEmpty(xml))
        {
            Log("-- content type payload is empty");
        }
        else
        {
            Log("-- loading content type payload");
            XmlDocument xmlDoc = new XmlDocument();
            xmlDoc.LoadXml(xml);
            XmlNamespaceManager nsmgr = new XmlNamespaceManager(xmlDoc.NameTable);
            nsmgr.AddNamespace("p",
PrinterPolicyFeature.PrintControlPolicyNamespace);
            XmlNode node = xmlDoc.SelectSingleNode("p:data/p:printers", nsmgr);
            Log("-- '" + node.InnerText + "'");
            SPField field =
item.Fields[PrinterPolicyFeature.TrustedPrintersFieldName];

            try
            {
                item[field.Id] = node.InnerText;
                item.SystemUpdate();
                Log("-- UPDATED!");
            }
            catch (SPException x)
            {
                Log(string.Format("-- FAILED to update item - {0}", x.Message));
            }
        }
    }
}

```

10. Now you are ready to override selected methods of the **SharePointPolicyFeature** base class. Add a region to the **PrinterPolicyFeature** class named *SharePointPolicyFeature Overrides* . Insert the following code snippet inside the region.

Code Snippet: 'SharePointPolicyFeature - Registration Routines'

```

/// <summary>
/// This method is called when the feature is registered for a content type.
/// Adds a "TrustedPrinters" field to the content type.
/// </summary>
/// <param name="ct"></param>
public override void Register(SPContentType ct)
{
    base.Register(ct);
    Log("Registering print control for content type: " + ct.Name);

    // Setup the item event receiver for the content type.
    ItemEventReceiver.Create(ct, typeof(PrinterPolicyEventReceiver));

    // Add the "TrustedPrinters" field to the content type.
    SPFieldLink fieldRef = SharePointContentType.AddOrCreateFieldReference(ct,
PrinterPolicyFeature.TrustedPrintersFieldName, SPFieldType.Text, false,
true, true,true,true);
}

/// <summary>
/// This method is called when the policy feature is removed.

```



```

/// Uninstalls the event receiver from the content type.
/// </summary>
/// <param name="ct"></param>
public override void UnRegister(SPContentType ct)
{
    base.UnRegister(ct);
    Log("Unregistering PrintControl for content type: " + ct.Name);
    ItemEventReceiver.Remove(ct, typeof(PrinterPolicyEventReceiver));
}

```

11. These routines handle the registration and un-registration of the feature for a given content type. The code first creates an ItemEventReceiver for the content type using the nested class you just created, and then it adds a "TrustedPrinters" field to the content type.
12. Next, you will override the **OnCustomDataChange** method, which is called whenever the custom data associated with the policy item changes. This happens when the policy administrator enters a new value into the custom UI you will provide for specifying the list of trusted printer names. Add the following code snippet to the class definition.

Code Snippet: 'SharePointPolicyFeature - OnCustomDataChange'

```

/// <summary>
/// This method is called when the custom data for a policy item changes.
/// </summary>
/// <param name="policyItem"></param>
/// <param name="ct"></param>
public override void OnCustomDataChange(PolicyItem policyItem, SPContentType
ct)
{
    base.OnCustomDataChange(policyItem, ct);
    Log("OnCustomDataChange for policy item: " + policyItem.Name
        + " and content type " + ct.Name + " where policy item custom data = "
        + policyItem.CustomData);

    try
    {
        // Get the custom data from the policy item.
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.LoadXml(policyItem.CustomData);
        XmlNamespaceManager nsmgr = new XmlNamespaceManager(xmlDoc.NameTable);
        nsmgr.AddNamespace("p",
PrinterPolicyFeature.PrintControlPolicyNamespace);

        // Add the custom data to the content type payload.

ct.XmlDocuments.Delete(PrinterPolicyFeature.PrintControlPolicyNamespace);
        ct.XmlDocuments.Add(xmlDoc);
        ct.Update();
    }
    catch (Exception x)
    {
        Log("OnCustomDataChange failed for item: " + policyItem.Name + " - " +
x.ToString());
    }
}

```

13. SharePoint monitors the lifecycle of items affected by a policy and calls the **ProcessListItem** method when updates are needed for a given item. Next, you will override this method to update the **TrustedPrinters** field with the updated list from the policy item. Add the following code snippet to the class definition.

Code Snippet: 'SharePointPolicyFeature - ProcessListItem'

```

/// <summary>
/// This method is called by the policy framework for list items that were
/// created before the policy was applied to the list.
/// </summary>
/// <param name="site"></param>
/// <param name="policyItem"></param>
/// <param name="listItem"></param>
/// <returns></returns>
public override bool ProcessListItem(SPSite site,
Microsoft.Office.RecordsManagement.InformationPolicy.PolicyItem policyItem,
SPListItem listItem)
{
    Log("Processing list item: " + listItem.Title);
    bool result = base.ProcessListItem(site, policyItem, listItem);

    // Add the policy item custom data to the TrustedPrinters field.
    try
    {
        // Get the policy data from the item payload.
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.LoadXml(policyItem.CustomData);

        XmlNamespaceManager nsmgr = new XmlNamespaceManager(xmlDoc.NameTable);
        nsmgr.AddNamespace("p",
PrinterPolicyFeature.PrintControlPolicyNamespace);
        XmlNode node = xmlDoc.SelectSingleNode("p:data/p:printers", nsmgr);

        // Store the list of trusted printers into the list item.
        listItem[PrinterPolicyFeature.TrustedPrintersFieldName] =
node.InnerText;
        listItem.Update();
    }
    catch (Exception x)
    {
        Log("ProcessListItem failed for item: " + listItem.Title + " - " +
x.ToString());
    }
    return result;
}

```

14. In order to test the policy feature, it must be registered in the site collection when the feature is activated. Open the **FeatureReceiver.cs** file for editing. Replace the *FeatureActivated* method with the following code snippet.

Code Snippet: 'Printer Control Policy Feature Activated'

```

/// <summary>
/// Override to install the printer control policy feature.
/// </summary>
/// <param name="properties"></param>

```

```
public override void FeatureActivated(SPFeatureReceiverProperties properties)
{
    using (SPSite site = properties.Feature.Parent as SPSite)
        if (!SharePointPolicyFeature.Install(typeof(PrinterPolicyFeature)))
            Trace.WriteLine("Printer Policy Feature installation failed.");
}
```

15. This method gets the *SPSite* object from the feature receiver properties and uses the static *Install* method of the *SharePointPolicyFeature* base class to register the policy feature within the SharePoint environment.
16. Similarly, replace the *FeatureDeactivating* method with the code snippet shown below.

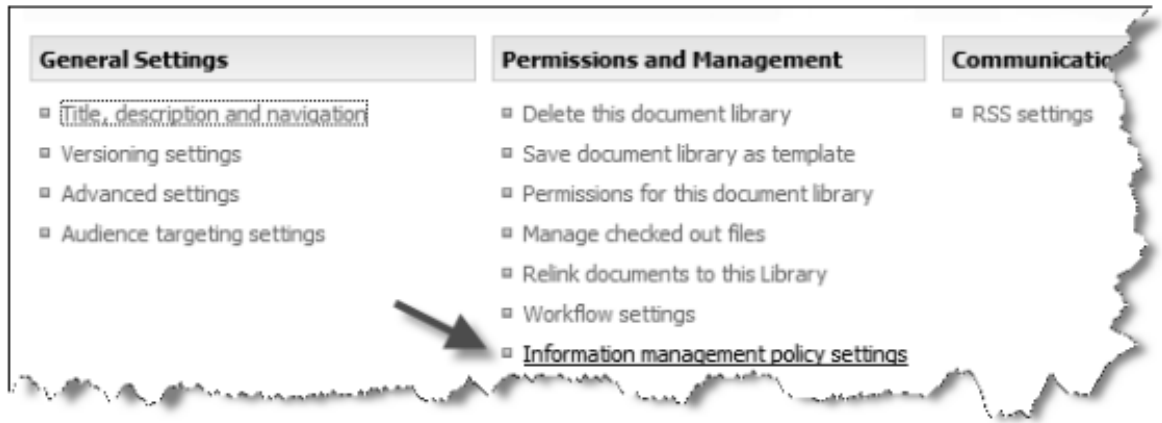
Code Snippet: 'Printer Control Policy Feature Deactivating'

```
/// <summary>
/// Override to remove the printer control policy feature.
/// </summary>
/// <param name="properties"></param>
public override void FeatureDeactivating(SPFeatureReceiverProperties
properties)
{
    using (SPSite site = properties.Feature.Parent as SPSite)
        if (!SharePointPolicyFeature.Uninstall(typeof(PrinterPolicyFeature)))
            Trace.WriteLine("Printer Policy Feature removal failed.");
}
```

17. Now you are ready to test your work so far. Build the project and navigate to any SharePoint site. From the *Site Settings* page, select *Site Collection Features* . You should see the your feature listed.



18. Navigate to any document library and then from the *List Settings* page, select the *Information management policy settings* link.



19. If the document library has content types enabled, then you will see a list of content types to choose from. Click one of the links to go to the policy settings page. If content types are not enabled for the list, you will go directly to that page.
20. From the policy settings page, select the *Define a policy...* link and click OK.



21. Now, you should see your policy feature displayed at the bottom of the list of available policy features to be enabled.



22. If you enable the feature, you will get a post-back, but nothing is displayed beneath the checkbox. In the next exercise, you will add a custom user interface to enable administrators to specify the list of trusted printers.

Exercise 3: Add Custom Policy Feature Settings

1. In this exercise, you will create a custom control for entering a list of trusted printers. This will be a simple textbox with instructions to enter a semicolon-delimited list of printer names.

Note: In actual practice, you would more likely retrieve the list of trusted printers from an external database or from a SharePoint list and present them as a dropdown list or combo-box.

- The first step is to create a control template that SharePoint can load when the administrator selects the checkbox next to the name of your policy feature.
- Right-click the *TEMPLATE* folder in the *Solution Explorer* window and select **Add -gt; New Folder** from the context menu. Name the new folder **LAYOUTS** . Beneath the LAYOUTS folder, add a sub-folder named **ECM401.PrinterControlPolicyFeature** . Within that folder, add a new **Text File** item and give it the name **PrinterPolicySettings.ascx** .

Note: Although you selected *Text File* as the item type, Visual Studio uses the file extension to determine which editor to open. Here, you are creating an **ASCX** file to use as a template for your custom user interface.

4. Add the following code to the ASCX file.

XML Snippet: 'Printer Policy Settings Control Template'

[illegible]

```

        Rows="5" MaxLength="1024" Columns="40" class="ms-input"
        ToolTip="Enter the list of trusted printers here." />
        <asp:RequiredFieldValidator ID="RequiredValidatorPrinters"
        ControlToValidate="TextBoxPrinters"
        ErrorMessage="At least one printer name is required."
        Text="Please enter a semicolon-delimited list of printer
names."
        EnableClientScript="false" runat="server" />
    </td>
</tr>
</table>
</p>

```

5. This template declares a Label, a TextBox and an associated FieldValidator control. It inherits from a code-behind class you will now implement in a separate file.
6. Add a new class to the project named **PrinterPolicySettings** and open the file for editing. Delete the generated class declaration and enter the following code snippet.

Code Snippet: 'PrinterPolicySettings - Class Declaration'

```

/// <summary>
/// This class implements a custom Information Policy settings control
/// that enables an administrator to enter the list of trusted printers.
/// </summary>
public class PrinterPolicySettings : CustomSettingsControl
{
    // holds the custom data associated with the control
    string m_customData = string.Empty;

    // automatically bound to the TextBox control in the template
    protected TextBox TextBoxPrinters;
}

```

7. Add the following using statements at the top of the file.

```

using System.Web.UI.WebControls;
using System.Xml;
using Microsoft.Office.RecordsManagement.InformationPolicy;
using Microsoft.SharePoint;

```

8. The *m_customData* string will hold the custom data being transferred to and from the TextBox control. The protected *TextBoxPrinters* control is automatically bound to the template by ASP.NET.
9. Your *PrinterPolicySettings* class inherits from a base class that is provided by the SharePoint Information Policy framework for implementing policy feature user interfaces. The *CustomSettingsControl* class is an abstract class that declares several abstract methods. Because they are abstract methods, you must provide an implementation for all of them in your derived class.

```

namespace Microsoft.Office.RecordsManagement.InformationPolicy
{
    public abstract class CustomSettingsControl :
        UserControl, IPostBackDataHandler
    {
        protected CustomSettingsControl();

        public abstract SPContentType ContentType { get; set; }
        public abstract string CustomData { get; set; }
        public abstract SPList List { get; set; }

        public abstract bool LoadPostData(string postDataKey,
            NameValueCollection values);
        public abstract void RaisePostDataChangedEvent();
    }
}

```

10. Several methods will not be used in this solution, so you can enter stubs for the overrides as follows.

Code Snippet: 'PrinterPolicySettings - Unused Methods'

```

/// <summary>
/// Not used - must be implemented because base class is abstract.
/// </summary>
public override void RaisePostDataChangedEvent(){}

/// <summary>
/// Not used - must be implemented because base class is abstract.
/// </summary>
public override SPList List { get; set; }

/// <summary>
/// Not used - must be implemented because base class is abstract.
/// </summary>
public override SPContentType ContentType { get; set; }

```

11. The custom data consists of a list of printer names that you will embed into an **XML** fragment so it can be added to the payload of a content type. Add the following method to the class definition.

Code Snippet: 'PrinterPolicySettings - CustomData'

```

/// <summary>
/// This accessor is called to get or set the custom data
/// associated with the control.
/// </summary>
public override string CustomData
{
    get
    {
        XmlDocument xmlDoc = new XmlDocument();
        string ns = PrinterPolicyFeature.PrintControlPolicyNamespace;
    }
}

```

```

        new XmlNamespaceManager(xmlDoc.NameTable).AddNamespace("p", ns);
        XmlElement rootNode = xmlDoc.CreateElement("p", "data", ns);
        xmlDoc.AppendChild(rootNode);
        XmlElement printersNode = xmlDoc.CreateElement("p", "printers", ns);
        rootNode.AppendChild(printersNode);
        printersNode.InnerText = TextBoxPrinters.Text;
        m_customData = xmlDoc.InnerXml;
        return m_customData;
    }
    set
    {
        m_customData = value;
    }
}

```

12. The *LoadPostData* method is called to set the custom data string after the text in the control is modified.

Code Snippet: 'PrinterPolicySettings - LoadPostData'

```

    /// <summary>
    /// This method updates the custom data associated with the control.
    /// </summary>
    /// <param name="postDataKey"></param>
    /// <param name="values"></param>
    /// <returns></returns>
    public override bool LoadPostData(string postDataKey,
    System.Collections.Specialized.NameValueCollection values)
    {
        string oldData = this.CustomData;
        string newData = values[postDataKey];
        if (oldData != newData)
        {
            this.CustomData = newData;
            return true;
        }
        return false;
    }
}

```

13. Finally, you need to initialize the TextBox with the initial data when the page is loaded.

Code Snippet: 'PrinterPolicySettings - OnLoad'

```

    /// <summary>
    /// Loads the custom data string into the textbox whenever the control is
    loaded.
    /// </summary>
    /// <param name="e"></param>
    protected override void OnLoad(EventArgs e)
    {
        base.OnLoad(e);
        if (!(base.IsPostBack || string.IsNullOrEmpty(m_customData)))
        {
            try
            {
                XmlDocument xmlDoc = new XmlDocument();
                xmlDoc.LoadXml(m_customData);
                XmlNamespaceManager nsmgr = new
                XmlNamespaceManager(xmlDoc.NameTable);
            }
            catch { }
        }
    }
}

```



```

        nsmgr.AddNamespace("p",
PrinterPolicyFeature.PrintControlPolicyNamespace);
        XmlNode node = xmlDoc.SelectSingleNode("p:data/p:printers", nsmgr);
        TextBoxPrinters.Text = node.InnerText;
    }
    catch (Exception x)
    {
        System.Diagnostics.Trace.WriteLine("Failed to load printer settings
- " + x.Message, GetType().Name);
    }
}
}

```

14. Save and close the file.
15. Now that you have created the control template and implemented its code-behind class, you need to tell SharePoint where it is located on the server. You also need to provide some instructions for the administrator describing the purpose of the feature and how it should be used.
16. Reopen the *PrinterPolicyFeature.cs* file for editing.

Note: Because the base *SharePointPolicyFeature* class inherits from the *Installer* class, Visual Studio treats the class file as a *Component*, causing the IDE to attempt to load a designer for it. To go directly to the code editor, you must right-click the file and choose *View Code*.

17. Add the following lines to the *PrinterPolicyFeature* constructor.

Code Snippet: 'PrinterPolicyFeature - Constructor'

```

        this.ConfigPage =
"ECM401.PrinterControlPolicyFeature/PrinterPolicySettings.ascx";
        this.ConfigPageInstructions = @"Add the names of trusted printers "
+ "here, separated by semicolons. This will prevent users from
printing "
+ "documents to which this policy is applied on unsecure printers. "
+ "NOTE: In order for this to work, you must also deploy the "
+ "PrintController add-on for Microsoft Office to all client
machines.";

```

18. Save your work and rebuild the project.
19. Navigate back to the *Policy Settings* page for the policy you created earlier. Now, your custom instructions appear beneath the *Document Print Controller* caption. When you click the checkbox next to the *Enable Document Print Controller* policy feature, you should see a labeled TextBox control where you can enter the list of trusted printers.

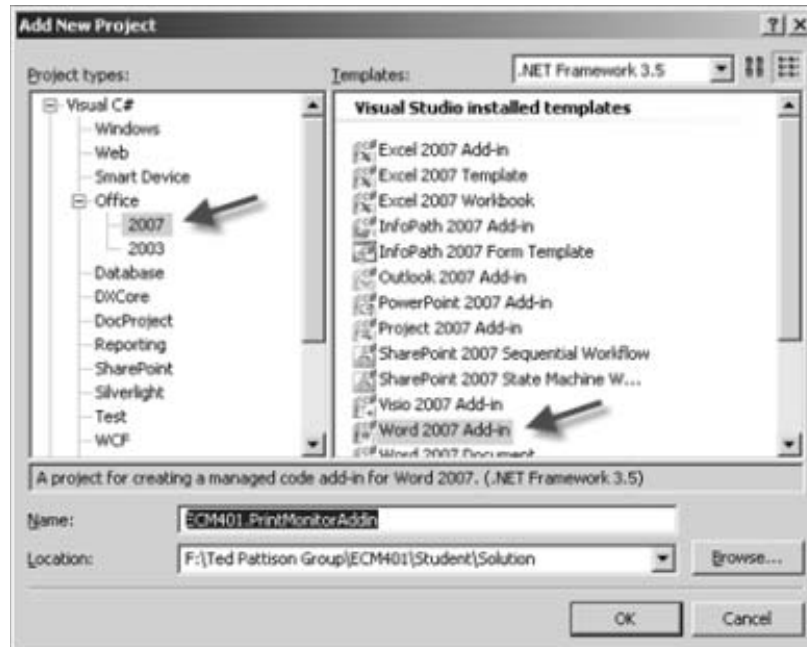


20. To test your work, enter some names into the TextBox control and save the policy. Whenever you add or update an item to which the policy is attached, you should see the list of printers in the *Trusted Printers* column of the item.

Note: It is not necessary to copy the list of trusted printers into a column of each list item. This was done in the lab only to illustrate the steps that are typically required when writing policy features. For the purposes of this scenario - managing trusted printers - you can access the policy definition along with its custom data directly from the content type payload. In the next exercise, you will use this approach to prevent users from printing to untrusted printers.

Exercise 4: Create a Print Monitor Add-in for Word 2007

1. To prevent users from violating the printer control policy, you need some additional code that will run on client machines. This code must be installed on each client and configured as an application-level add-in for the office client applications. In this exercise, you will use **Visual Studio Tools for Office** to create a print monitor add-in for Word 2007 that will read the trusted printer list from the document and compare them to the currently active printer. If the currently active printer name is not in the list, then any attempt to print the document is aborted with a message.
2. When an information policy is attached to a content type, a copy of the policy is embedded within any document that is associated with the type. This means that the entire policy definition travels with the document and is accessible from your add-in code.
3. Start by creating a new **VSTO** project called **ECM401.PrintMonitorAddin** . From the **New Project** dialog, select **Word 2007 Add-in** from the **Office/2007** section.



4. Open the **ThisAddIn.cs** file for editing. Add the following namespaces to the *using* statements at the top of the file.

```
using System.Xml;
using System.Windows.Forms;
```

5. The first bit of code to add will handle the *Startup* event to install a handler for the print event. This handler will be called whenever the user attempts to print the active document. Add the following code inside the **ThisAddIn_Startup** event handler that was auto-generated by the *New Project* wizard.

```
// install a handler for the print event
this.Application.DocumentBeforePrint
    += new Word.ApplicationEvent4_DocumentBeforePrintEventHandler(
        Application_DocumentBeforePrint);
```

6. Replace the generated event handler stub with the following code snippet.

Code Snippet: 'PrintMonitorAddin - DocumentBeforePrint'

```
/// <summary>
/// This event handler gets the name of the printer the user is
/// attempting to use. It then obtains the list of "trusted" printers
/// as defined by the PrintControl policy associated with the document.
/// If the current printer is not trusted, then a message is displayed
/// and the operation is cancelled.
/// </summary>
/// <param name="Doc"></param>
/// <param name="Cancel"></param>
void Application_DocumentBeforePrint(Word.Document Doc, ref bool Cancel)
{
    if (!PrinterIsTrusted(Application.ActivePrinter))
    {
```

```

        MessageBox.Show(String.Format(
            "Sorry. Big Brother does not want you to print this document " +
            "on printer '{0}'! Please select a different printer.",
            Application.ActivePrinter),
            "Printer Is Not Trusted");
        Cancel = true;
    }
}

```

7. The next routine will return true if the specified printer is a trusted printer as defined by the policy.

Code Snippet: 'PrintMonitorAddin - PrinterIsTrusted'

```

/// <summary>
/// Determines whether a specified printer is trusted.
/// </summary>
/// <param name="printerName"></param>
/// <returns>true if the specified printer is in the list of trusted
printers</returns>
bool PrinterIsTrusted(string printerName)
{
    // Retrieve the list of trusted printers from the attached policy.
    List<string> trustedPrinters = GetTrustedPrintersList();
    // Check whether the specified printer is in the list.
    if (trustedPrinters.Count == 0)
        return true;
    return trustedPrinters.Contains(printerName);
}

```

8. Finally, you need a routine to extract the list of trusted printers from the information policy attached to the document. The information policy is embedded as a *CustomXMLPart* object in the active document. The collection of custom xml parts is exposed as a property of the *Microsoft.Office.Interop.Word.Document* object. Add the following code snippet to the class definition.

Code Snippet: 'PrintMonitorAddin - GetTrustedPrintersList'

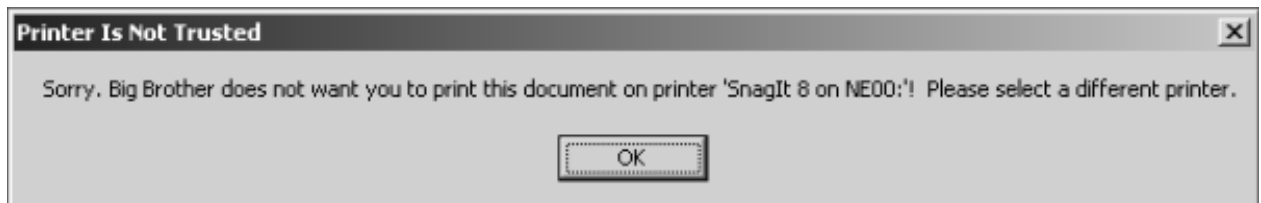
```

/// <summary>
/// Retrieves the list of trusted printers from the PrintControl policy
/// associated with the document.
/// </summary>
/// <remarks>
/// This method searches through the list of custom XML parts in
/// the document until it finds one matching the PrintControl policy URN.
/// </remarks>
/// <returns>the list of trusted printers from the attached policy</returns>
List<string> GetTrustedPrintersList()
{
    List<string> result = new List<string>();
    const string policyNamespace = "urn:ecm401:policy.printcontrol";
    Office.CustomXMLParts parts = Application.ActiveDocument.CustomXMLParts;
    foreach (Office.CustomXMLPart part in parts)
    {
        if (part.XML.Contains(policyNamespace))
        {

```

```
// extract the list of trusted printers.
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.LoadXml(part.XML);
XmlNamespaceManager nsmgr = new
XmlNamespaceManager(xmlDoc.NameTable);
nsmgr.AddNamespace("p", policyNamespace);
XmlNode node = xmlDoc.SelectSingleNode("p:data/p:printers", nsmgr);
if (node != null)
{
    string[] printers = node.InnerText.Split(";".ToCharArray());
    foreach (string printer in printers)
        result.Add(printer);
    break;
}
}
return result;
}
```

9. To test your work, simply run the project in the debugger and open a document from the document library you used to test the information policy. Try to print the document. You should see a dialog like the one shown below.



This concludes the lab exercises.

Lab 5: Building a Records Repository

Lab Time: 45 Minutes

Lab Directory: ECM401.CustomRepository

Lab Overview:

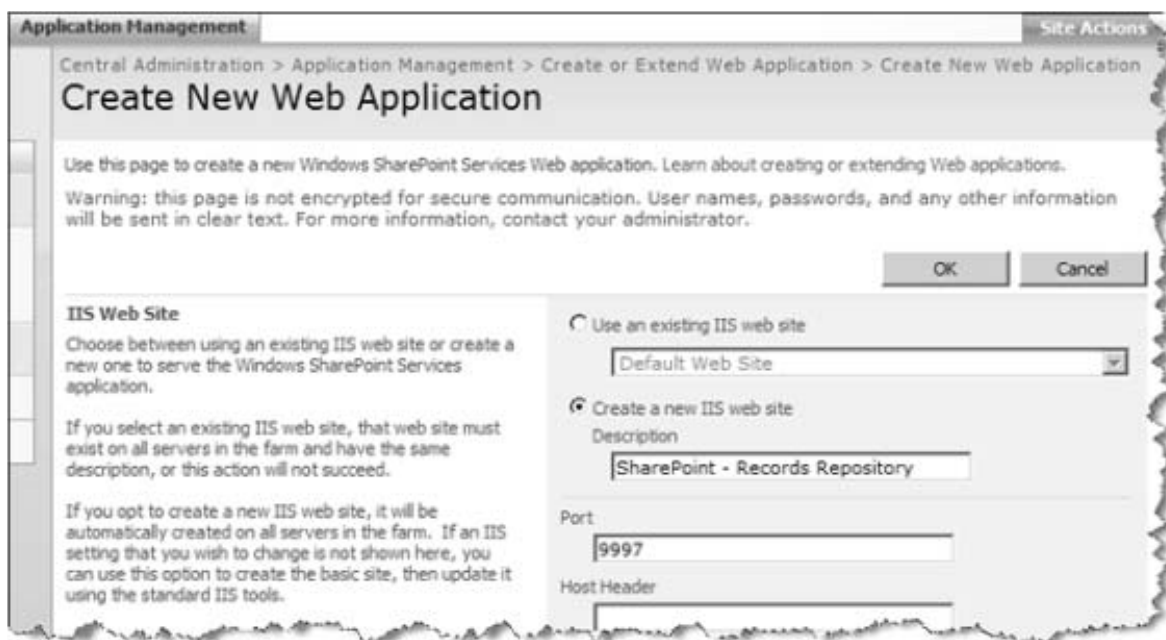
In this lab, you will create a controlled records repository for the Litware legal and accounting departments. The repository will be used to store contracts and legal documents related to ongoing litigation involving Litware and its suppliers. It will also be used to house annual reports produced by the accounting department.

Exercise 1: Creating a Record Center Site Collection

1. In this exercise, you will setup a separate site collection for official records. In order to facilitate greater administrative control over the contents of the repository, you will create the site collection in a new web application.

Note: Using a separate web application makes it easy to partition official records from other types of content. The web application you create will have its own content database and will be used exclusively to store and manage official documents.

2. Open the web browser and navigate to the **Central Administration** website located at **http://litwareserver:9999** . When the page opens, click the **Application Management** tab. From the Application Management page, click **Create or extend Web application** in the **SharePoint Web Application Management** section.



Application Management **Site Actions**

Central Administration > Application Management > Create or Extend Web Application > Create New Web Application

Create New Web Application

Use this page to create a new Windows SharePoint Services Web application. Learn about creating or extending Web applications.

Warning: this page is not encrypted for secure communication. User names, passwords, and any other information will be sent in clear text. For more information, contact your administrator.

IIS Web Site

Choose between using an existing IIS web site or create a new one to serve the Windows SharePoint Services application.

If you select an existing IIS web site, that web site must exist on all servers in the farm and have the same description, or this action will not succeed.

If you opt to create a new IIS web site, it will be automatically created on all servers in the farm. If an IIS setting that you wish to change is not shown here, you can use this option to create the basic site, then update it using the standard IIS tools.

☐ Use an existing IIS web site

Default Web Site

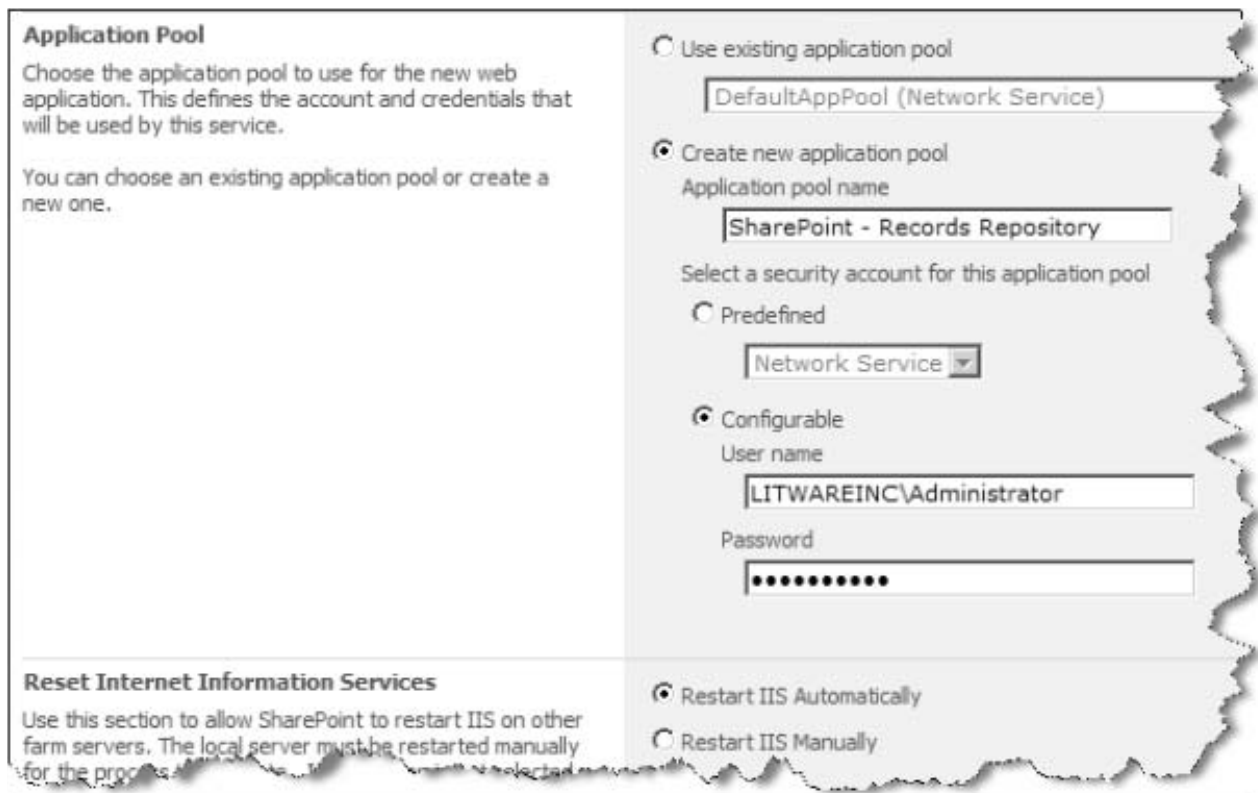
☒ Create a new IIS web site

Description
SharePoint - Records Repository

Port
9997

Host Header

3. On the **Create New Web Application** page, click **Create a new Web application** . In the **IIS Web Site** section, select the default **Create a new IIS web site** . Enter **SharePoint - Records Repository** into the **Description** field. Enter **9997** into the **Port** field. Leave the **Host Header** field blank.
4. Scroll down to the **Application Pool** section. Select **Create new application pool** and enter **SharePoint - Records Repository** as the Application pool name.
5. Select **Configurable** under **Select a security account for this application pool** . Enter **LITWAREINC\Administrator** as the user name. Enter **pass@word1** as the password.
6. Click the **Restart IIS Automatically** radio button.



Application Pool

Choose the application pool to use for the new web application. This defines the account and credentials that will be used by this service.

You can choose an existing application pool or create a new one.

☐ Use existing application pool

DefaultAppPool (Network Service)

☒ Create new application pool

Application pool name

SharePoint - Records Repository

Select a security account for this application pool

☐ Predefined

Network Service

☒ Configurable

User name

LITWAREINC\Administrator

Password

pass@word1

Reset Internet Information Services

Use this section to allow SharePoint to restart IIS on other farm servers. The local server must be restarted manually for the process to be selected.

☒ Restart IIS Automatically

☐ Restart IIS Manually

7. Scroll down to the **Database Name and Authentication** section. Enter **WSS_OfficialRecords** into the **Database Name** field.

Database Name and Authentication

Use of the default database server and database name is recommended for most cases. Refer to the administrator's guide for advanced scenarios where specifying database information is required.

Use of Windows authentication is strongly recommended. To use SQL authentication, specify the credentials which will be used to connect to the database.

Database Server

Database Name

Database authentication

☒ Windows authentication (recommended)

☐ SQL authentication

Account

Password

- Click **OK** to create the new web application. On the **Application Created** page, click the **Create Site Collection** link.

Central Administration > Application Management > Create or Extend Web Application > Create New Web Application


Application Created

Application Created

The Windows SharePoint Services Web application has been created.

If this is the first time that you have used this application pool with a SharePoint Web application, you must wait until the Internet Information Services (IIS) Web site has been created on all servers. By default, no new SharePoint site collections are created with the Web application. To create a new site collection, go to the [Create Site Collection](#) page.

To finish creating the new IIS Web site, you must run "iisreset /noforce" on the local server.



- On the **Create Site Collection** page, ensure that **http://litwareserver:9997** is showing in the **Web Application:** dropdown in the **Web Application** section.

Central Administration > Application Management > Create Site Collection

Create Site Collection

Use this page to create a new top-level Web site.

OK Cancel

Web Application Select a Web application.	Web Application: http://litwareserver:9997/ ▼
Title and Description Type a title and description for your new site. The title will be displayed on each page in the site.	Title: <input type="text" value="Litware Records"/> Description: <input type="text" value="Official records repository for the Litware group of companies."/> ▼
Web Site Address Specify the URL name and URL path to create a new site, or choose to create a site at a specific path. <small>To add a new URL Path on to the Refine Managed Paths page.</small>	URL: http://litwareserver:9997/ ▼

10. Enter **Litware Records** in the Title field. Enter any description you like and then scroll down to the **Template Selection** section. Click the **Enterprise** tab and select the **Records Center** site template.
11. In the **Primary Site Collection Administrator** section, enter **Administrator** and press **ctrl+k** to resolve the name. Enter **AngelaB** as the **Secondary Site Collection Administrator**. Press **ctrl+k** again to resolve the name.

Template Selection

This template creates a site designed for records management. Records managers can configure the routing table to direct incoming files to specific locations. The site prevents records from being modified after they are added to the repository.

Primary Site Collection Administrator
Specify the administrator for this Web site collection.

Secondary Site Collection Administrator
Specify the secondary administrator for this Web site collection.

Quota Template
Select a predefined quota template to limit resources used for this site collection.

Select a template:

Collaboration Meetings Enterprise ProSharePoint2007

Publishing

- Document Center
- Records Center**
- Site Directory
- Report Center
- Search Center with Tabs
- My Site Host
- Search Center

User name:

User name:

Select a quota template:

Storage limit:

Number of invited users:

12. Press **OK** to create the top-level site. From the **Top-Level Site Successfully Created** page, click the link to open the site and verify the site URL **http://litwareserver:9997**.

Litware Records

This Site: Litware Records

Home Site Actions

View All Site Content

Documents

- Unclassified Records

Lists

- Record Routing
- Holds
- Tasks

Sites

People and Groups

- Recycle Bin

Official records repository for the Litware group of companies.

Records Center

Records Center Setup! NEW 11/14/2007 3:08 AM
by System Account
This Records Center was created successfully.
This site is not e-mail enabled.
Please contact your administrator to configure Windows SharePoint Services to receive e-mail.

☐ Add new announcement

Record Routing

Title	Description	Location	Aliases	Default
Unclassified Records! NEW	This Record Routing is an example that stores records submitted to the Records Center in the "Unclassified Records" document library.	Unclassified Records		Yes

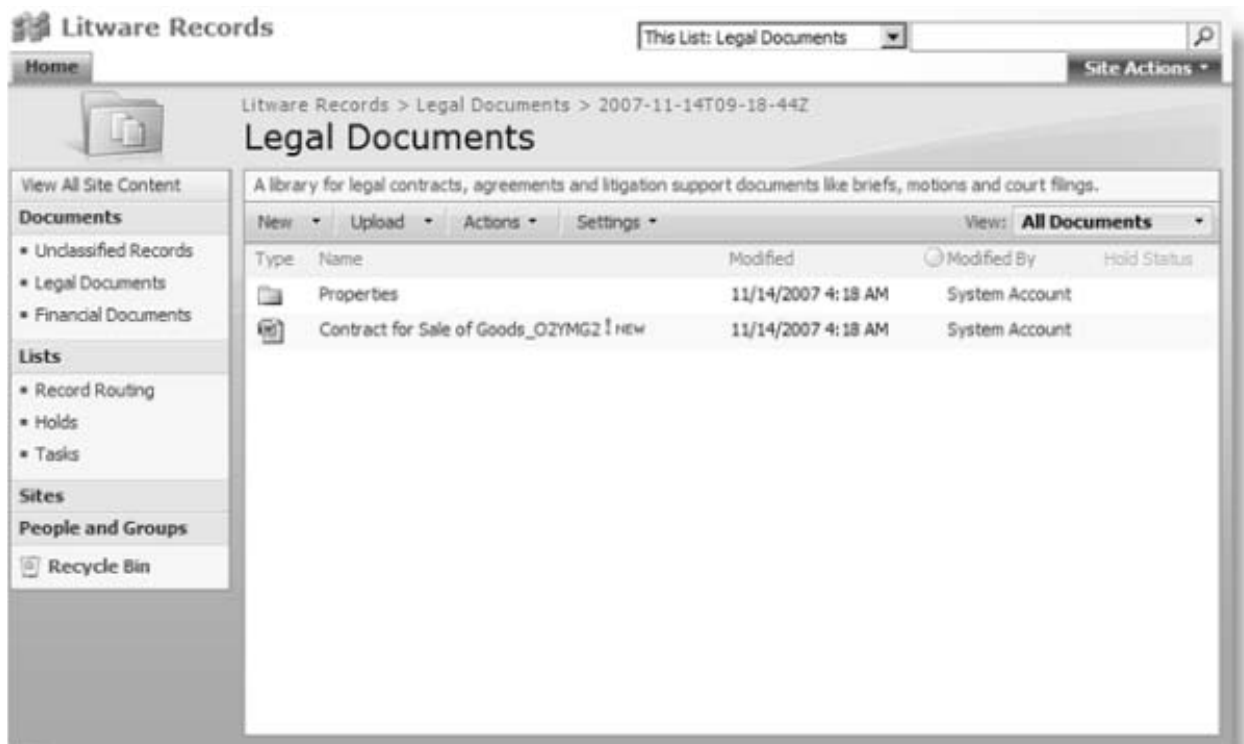
Links

- Configure Record Routing
- Manage Holds
- View Unclassified Records
- Add new link

13. You have now created a records repository site. In the next exercise, you will configure the repository to accept official records.

Exercise 2: Setting up Routing Types

1. In this exercise, you will configure the record center to accept specific kinds of documents. For the legal department, you will setup a document library to hold legal documents like contracts and service agreements, as well as litigation support documents like motions and other court filings. For the accounting department, you will setup a separate document library to hold annual reports and financial statements.
2. From the quick launch navigation bar on the left side of the record center home page, click the **Documents** link. From the **All Site Content** page, click **Create** . On the **Create** page, click **Document Library** . Enter **Legal Documents** as the library name, along with a description and click **Create** .



3. Now create a second document library called **Financial Documents** .
4. Navigate back to the home page and click **Configure Record Routing** from the Links list at the top right of the page. On the **Record Routing** page, click **New** to create a new routing type. Enter the information you see below.

Litware Records > Record Routing > New Item

Record Routing: New Item

OK Cancel

Attach File | Spelling... * indicates a required field

Title *	Legal Document
Description	A professional services agreement or other evidence of a contractual relationship or obligation.
Location *	Legal Documents The title of the library where records matching this record routing item should be stored. Libraries used to store submitted records cannot be deleted.
Aliases	Contract/Agreement/Brief/Motion A '/' delimited list of alternative names that represent this record routing entry.
Default	<input type="checkbox"/> If checked, this routing item will be used for submitted records that do not match the title or aliases of any other record routing item.

OK Cancel

5. Now create a second routing type for financial records.

Litware Records > Record Routing > New Item

Record Routing: New Item

OK Cancel

Attach File | Spelling... * indicates a required field

Title *	Financial Document
Description	A library to hold financial records such as Annual Reports, financial statements and related materials.
Location *	Financial Documents The title of the library where records matching this record routing item should be stored. Libraries used to store submitted records cannot be deleted.
Aliases	Annual Report/Financial Statement A '/' delimited list of alternative names that represent this record routing entry.
Default	<input type="checkbox"/> If checked, this routing item will be used for submitted records that do not match the title or aliases of any other record routing item.

OK Cancel

6. When you are finished, your **Record Routing** page should look similar to the following:

Litware Records > Record Routing

Record Routing

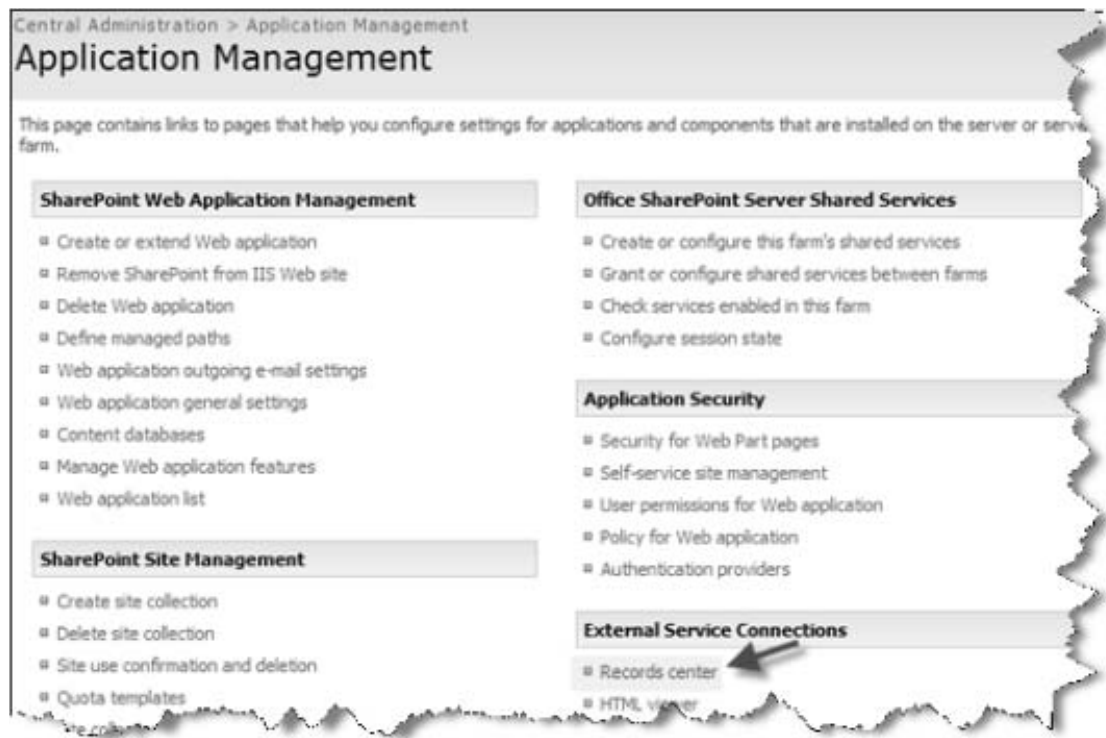
Use this list to route incoming records to the appropriate document library. For each type of record in the list, specify the title and description of the record type, the location where you want to store records of that type, and the names of other record types (aliases) that you want to store in the same location. You can specify any record type to be the default. If an incoming record doesn't match any of the record type in the list, it will be stored in the location you specified for the default record type.

New ▾ Actions ▾ Settings ▾ View: All Items

Title	Description	Location	Aliases	Default
Unclassified Records ! new	This Record Routing is an example that stores records submitted to the Records Center in the "Unclassified Records" document library.	Unclassified Records		Yes
Legal Document ! new	A professional services agreement or other evidence of a contractual relationship or obligation.	Legal Documents	Motion/Agreement/Brief/Contract	No
Financial Document ! new	A library to hold financial records such as Annual Reports, financial statements and related materials.	Financial Documents	Financial Statement/Annual Report	No

7. Now that you have created your record center site and configured the record routing types, you can complete the configuration of the SharePoint Farm so that users can send documents to the repository easily from the SharePoint UI.

8. Open the **Central Administration Web** site again. Navigate to the **Application Management** page and select **Records center** in the **External Service Connections** section.



9. On the **Configure Connection to Records Center** page, click **Connect to a Records Center**. Enter **http://litwareserver:9997/_vti_bin/officialfile.asmx** as the URL and enter **Litware Records** as the title.

Exercise 3: Submitting Documents

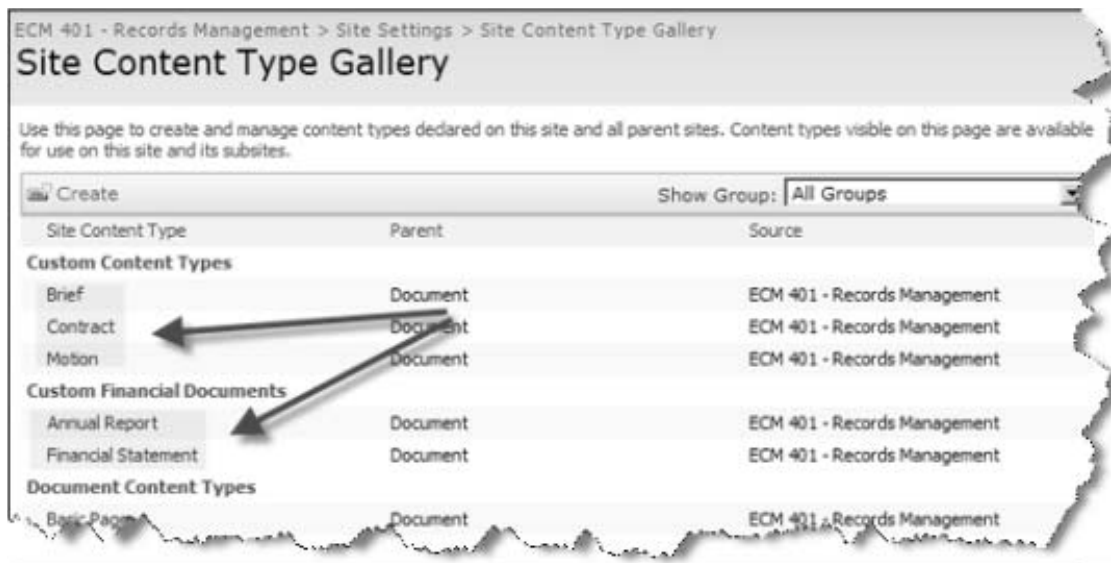
1. In this exercise, you will test your work by submitting documents from various places to the records repository.
2. Open the browser and navigate to the sample records management site at **http://litwareinc.com/sites/recman**.

Note: If this site does not exist, simply create a new site at that location before continuing with the exercise.

3. Select **Site Settings** from the **Site Actions** menu and open the content types gallery for the site.
4. Create the following content types:

Name	Parent	Group
Brief	Document	Custom Content Types
Contract	Document	Custom Content Types
Motion	Document	Custom Content Types
Financial Statement	Document	Custom Financial Documents
Annual Report	Document	Custom Financial Documents

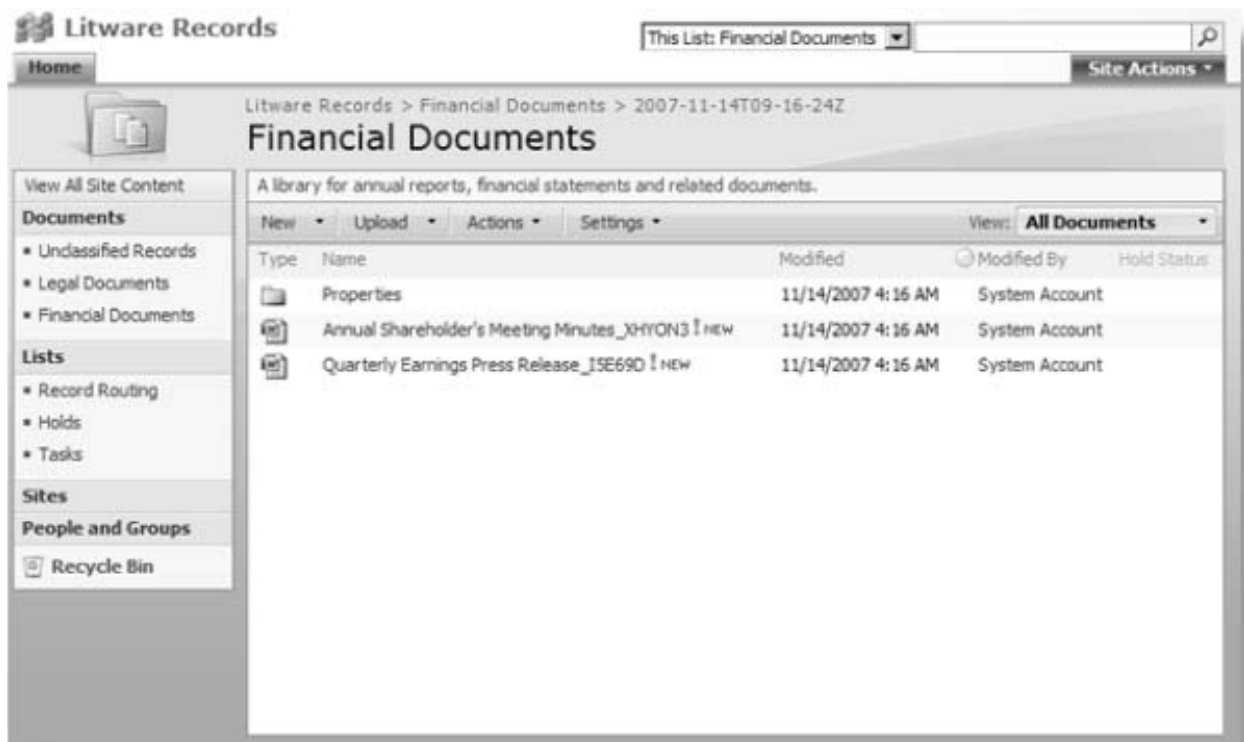
5. When you are finished, the content gallery should look like the following illustration.

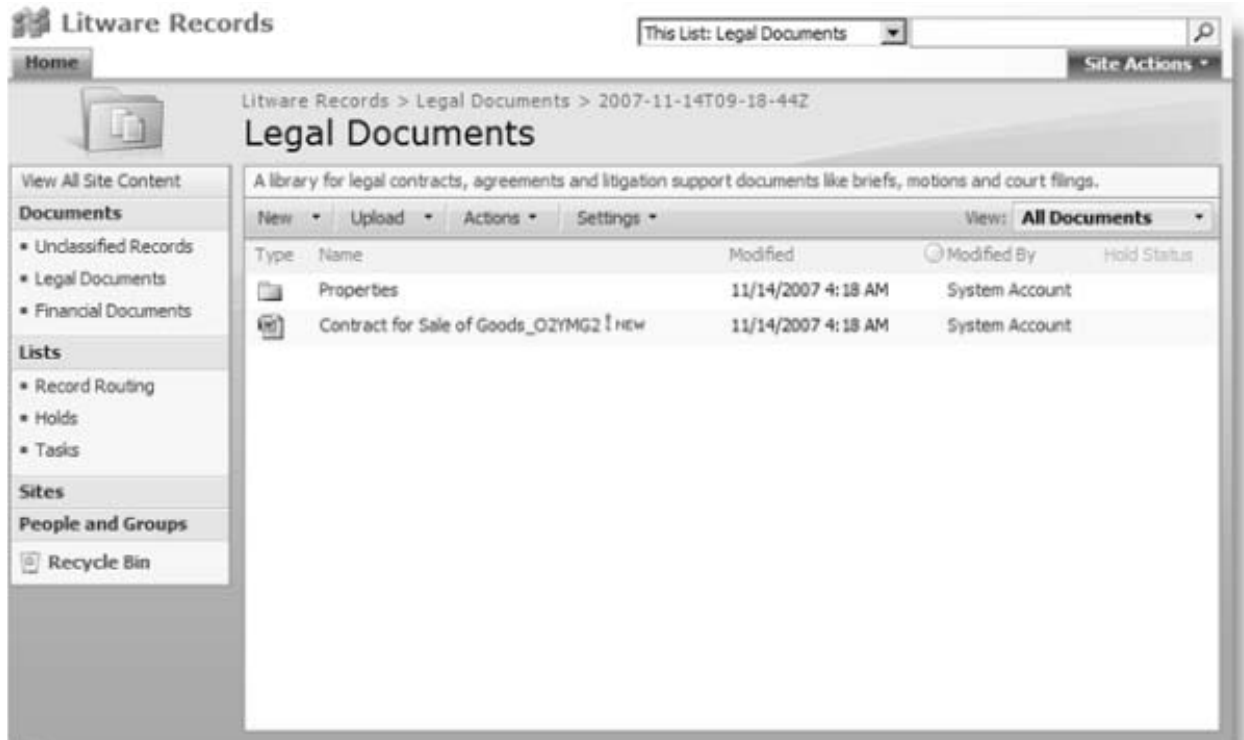


- Click **Site Actions -> Create** and create a new **Document Library** called **Shared Documents** . Open the **Document Library Settings** page and click the **Advanced Settings** link. Select **Allow management of content types** and click **OK** .
- On the **Custom Shared Documents** page, click **Add from existing site content types** in the **Content Types** section.
- On the **Add Content Types: Shared Documents** page, select the content types you just created from the list and click the **Add** button. Now you can navigate back to the **Shared Documents** page and upload according to the type of document it is.



9. Select any document and then choose **Send To -> Litware Records** from the context menu. When the operation has completed, navigate to the record center at <http://litwareserver:9997>. You should see the document in the appropriate library, depending on its content type.





The screenshot displays the 'Litware Records' web application. The top navigation bar includes a 'Home' button and a search bar with the text 'This List: Legal Documents'. The main content area is titled 'Legal Documents' and contains a description: 'A library for legal contracts, agreements and litigation support documents like briefs, motions and court filings.' Below this, there is a table with columns for 'Type', 'Name', 'Modified', 'Modified By', and 'Hold Status'. The table lists two items: 'Properties' and 'Contract for Sale of Goods_O2YMG2 NEW', both modified on 11/14/2007 at 4:18 AM by the 'System Account'. The left sidebar contains navigation links for 'Documents', 'Lists', 'Sites', 'People and Groups', and a 'Recycle Bin'.

Litware Records

Home

This List: Legal Documents



Site Actions

Litware Records > Legal Documents > 2007-11-14T09:18-44Z

Legal Documents

A library for legal contracts, agreements and litigation support documents like briefs, motions and court filings.

New • Upload • Actions • Settings • Views: **All Documents**

Type	Name	Modified	Modified By	Hold Status
	Properties	11/14/2007 4:18 AM	System Account	
	Contract for Sale of Goods_O2YMG2 NEW	11/14/2007 4:18 AM	System Account	

Documents


- Unclassified Records
- Legal Documents
- Financial Documents

Lists

- Record Routing
- Holds
- Tasks

Sites

People and Groups

 Recycle Bin

This concludes the lab exercises.

Lab 6: Using the Official File API

Lab Time: 45 Minutes

Lab Directory: ECM401.OfficialFileSubmit

Lab Overview:

In this lab, you will create a custom **STSADM** command that will allow content administrators to send files to the Litware records repository from the command line. It will also enable them to enumerate the available routing types in the repository.

Exercise 1: Create a command to enumerate routing types

1. Start by creating a new **Class Library** project in Visual Studio. Give it the name **ECM401.OfficialFileSTSADMCommand**.
2. Add a reference to the **Microsoft.SharePoint** assembly. Right click on the References node and select **Add Reference...** from the context menu. From the Add Reference dialog, scroll to the bottom and select **Windows SharePoint Services** from the list.
3. Add a second reference to the **Microsoft.Office.Policy** assembly. Click **Add Reference...** again and click **Browse** at the top of the dialog. Navigate to the SharePoint **ISAPI** folder and double-click the **Microsoft.Office.Policy.dll** file.

Note: The ISAPI folder is located at **c:\Program Files\Common Files\Microsoft Shared\web server extensions\12\ISAPI**

4. Rename the auto-generated **class1.cs** file to **EnumRoutingTypes.cs** and open the file for editing.
5. Add the following lines of code at the top of the file.

```
using Microsoft.SharePoint;  
using Microsoft.SharePoint.StsAdmin;  
using Microsoft.Office.RecordsManagement.RecordsRepository;
```

6. Delete everything inside the **ECM401.OfficialFileSTSADMCommand** namespace and create a new class named **EnumRoutingTypes** derived from **ISPStsadmCommand**.
7. Enter the following code to provide a help message for users.

Code Snippet: 'ISPStsadmCommand.GetHelpMessage'

```
string ISPStsadmCommand.GetHelpMessage(string command)  
{  
    string msg = "Enumerates the available routing types in a records repository  
site."  
    msg += "\n-url\t&lt;url&gt;  
\t\tthe url of the record center site";  
    return msg;  
}
```

```
}

```

8. Implement the Run method as follows:

Code Snippet: 'ISPStsadmCommand.Run'

```
int ISPStsadmCommand.Run(string command,
System.Collections.Specialized.StringDictionary keyValues,
out string output)
{
    int result = 1;
    string sRepositoryUrl = string.Empty;
    try
    {
        // validate the arguments
        if (null == (sRepositoryUrl = keyValues["url"])
            || sRepositoryUrl.Length == 0)
            throw new ApplicationException("No url specified.");

        // open the website
        using (SPSite site = new SPSite(keyValues["url"]))
        {
            using (SPWeb web = site.OpenWeb())
            {
                // Enumerate the routing types using
                // the RecordSeriesCollection wrapper.
                RecordSeriesCollection routingTypes =
                new RecordSeriesCollection(web);

                if (null == routingTypes || routingTypes.Count == 0)
                {
                    output = "No routing types were found.";
                }
                else
                {
                    output = string.Format(
                        "Found {0} routing types:", routingTypes.Count);
                    for (int i=0; i < routingTypes.Count; i++)
                    {
                        RecordSeries routingType = routingTypes[i];
                        output += string.Format(
                            "\n{0}\n\t{1}\n", routingType.Name,
                            routingType.Description);
                    }
                }
            }
        }
    }
    catch (Exception x)
    {
        output = x.ToString();
    }
    return result;
}
```

9. Sign the assembly using the **ECM401.snk** file located in your **Student\Resources** folder.

10. Add a new **XML File** item to the project named **stsadmcommands.records.xml** . This file will contain the CAML code needed to tell STSADM that your custom command exists and where to load the ISPSsadmCommand implementation you have provided. Open the file and add the following lines of XML code.

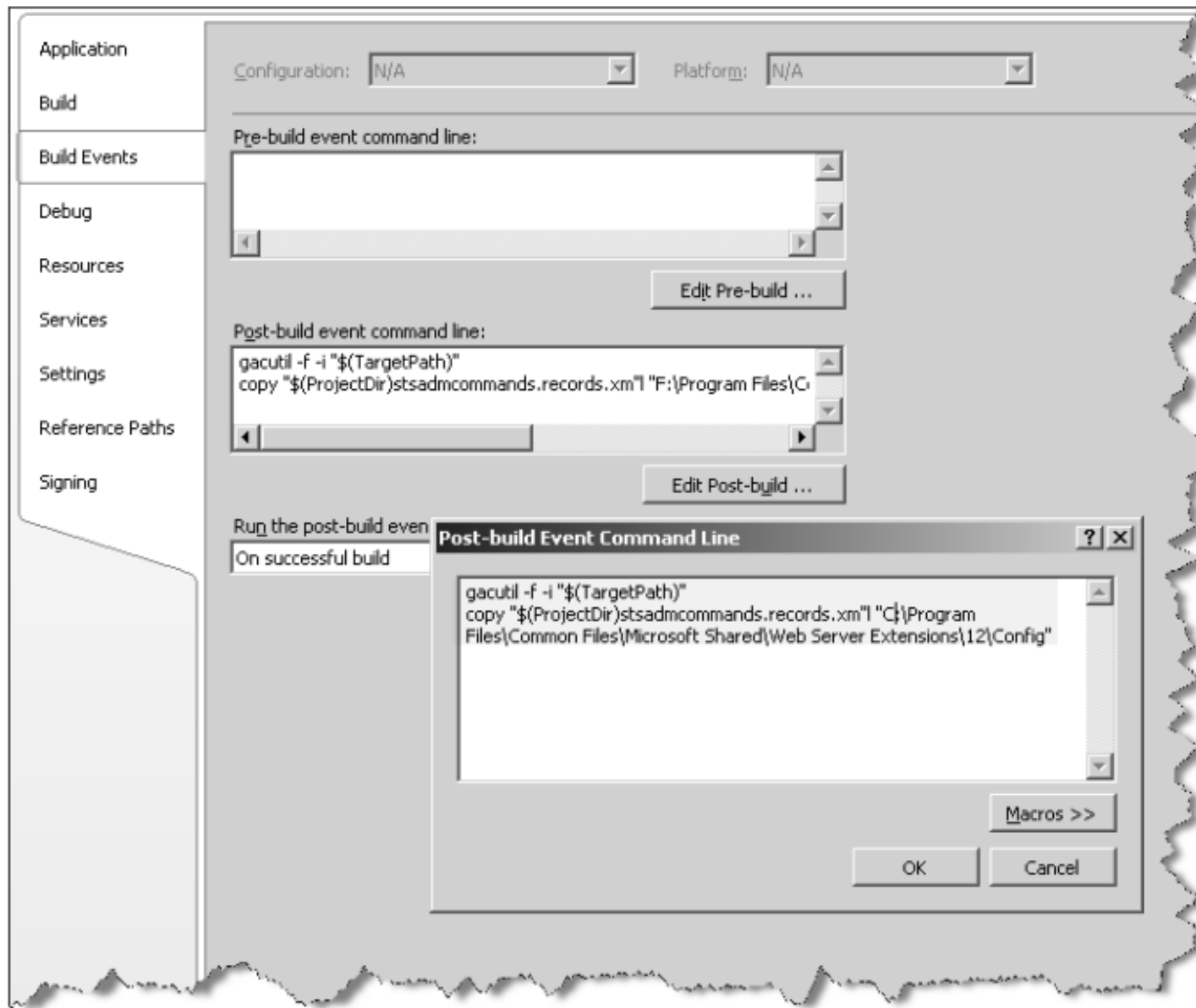
XML Snippet: 'EnumRoutingTypes Command Definition'

```
PublicKeyToken***INSERT PUBLIC KEY TOKEN***
  <?xml version="1.0" encoding="utf-8" ?>
  <commands>
    <command name="enumroutingtypes"
      class="ECM401.OfficialFileSTSADMCommand.EnumRoutingTypes,
ECM401.OfficialFileSTSADMCommand, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=$PublicKeyToken$"/>
  </commands>
```

Note: The PublicKeyToken value must match the actual public key token of the assembly, or STSADM will be unable to locate your command implementation.

11. Save the file and open the project property pages. Navigate to the **Build Events** page and enter the following commands into the **Post-Build Events** window.

```
gacutil -f -i "$(TargetPath)"
copy "$(ProjectDir)stsadmcommands.records.xml" "C:\Program Files\Common
Files\Microsoft Shared\Web Server Extensions\12\Config"
```



12. These commands will register your assembly in the Global Assembly Cache and copy your custom command definition file into the SharePoint **Config** folder. Now you are ready to build the project.
13. Open a command prompt window and type the following command: `stsadm -help enumroutingtypes`
14. You should see the following screen:

```

C:\>stsadm -help enumroutingtypes

stsadm -o enumroutingtypes
Enumerates the available routing types in a records repository site.
-url      <url>          the url of the record center site

C:\>_

```

15. Now type in the command to enumerate the routing types in the Litware Record Center: stsadm -o enumroutingtypes -url http://localhost:9997

16. You should see the following screen:

```

C:\>stsadm -o enumroutingtypes -url http://litwareserver:9997

Found 3 routing types:
Unclassified Records
    This Record Routing is an example that stores records submitted to the R
ecords Center in the "Unclassified Records" document library.

Legal Document
    A professional services agreement or other evidence of a contractual rel
ationship or obligation.

Financial Document
    A library to hold financial records such as Annual Reports, financial st
atements and related materials.

C:\>

```

17. Next, you will extend the solution to include another command to submit files to the record center.

Exercise 2: Add a command to submit files to the records center

1. In this exercise, you will add a second **STSADM** command to enable administrators to select files and send them to the records center directly from the command line.
2. Create a new class file and give it the name **SubmitOfficialFile.cs**. Add the following lines of code at the top of the file.

```

using System.IO;
using Microsoft.SharePoint;
using Microsoft.SharePoint.StsAdmin;
using OfficialFile=Microsoft.Office.RecordsManagement.RecordsRepository;

```

3. Implement the **GetHelpMessage** method as follows to provide guidance for the user.

Code Snippet: 'SubmitOfficialFile.GetHelpMessage'

```

string ISPStsadmCommand.GetHelpMessage(string command)
{
    string msg = "stsadm -o submitfiles";
    msg += "\nSubmits one or more files to an official records repository
site.";
    msg += "\n-url\t<url> \t\tthe url of the record center site";
    msg += "\n-path\t<file mask=> \tthe wildcard mask for the files to be
submitted";
    msg += "\n-type\t<routing type=> \tthe routing type name";
    return msg;
}

```

4. Next, implement the run command as follows:

Code Snippet: 'SubmitOfficialFile.Run'

```

int ISPStsadmCommand.Run(string command,
    System.Collections.Specialized.StringDictionary keyValues,
    out string output)
{
    int result = 1;

    string sRoutingType=string.Empty;
    string sFileMask=string.Empty;
    string sRepositoryUrl=string.Empty;
    StringBuilder sbResult = new StringBuilder();

    try
    {
        // validate the arguments
        if (null == (sRepositoryUrl = keyValues["url"]))
            || sRepositoryUrl.Length == 0)
            throw new ApplicationException("No url specified.");

        if (null == (sFileMask = keyValues["path"]) || sFileMask.Length == 0)
            throw new ApplicationException("No files specified.");

        if (null == (sRoutingType = keyValues["type"]))
        {
            sbResult.AppendLine("No routing type specified - using default.");
            sRoutingType = string.Empty;
        }

        // open the website
        using (SPSite site = new SPSite(keyValues["url"]))
        {
            using (SPWeb web = site.OpenWeb())
            {
                OfficialFile.RecordsRepositoryProperty[] props =
                    new OfficialFile.RecordsRepositoryProperty[1];

                // Process each file specified.
                foreach (FileInfo info in ExpandWildcards(sFileMask))
                {
                    string strResultInfo="";
                    sbResult.AppendLine(info.Name);

                    OfficialFile.OfficialFileResult rDisposition =

```

```

        OfficialFile.OfficialFileCore.SubmitFile(web,
            File.ReadAllBytes(info.FullName),
            props, sRoutingType, info.FullName,
            out strResultInfo);

        sbResult.AppendFormat("--> {0}\n", rDisposition.ToString());
    }
}
}
output = sbResult.ToString();
result = 0;
}
catch (Exception x)
{
    output = x.ToString();
}
return result;
}

/// <summary>
/// Reduces a wildcard mask to an array of FileInfo objects.
/// </summary>
static FileInfo[] ExpandWildcards(string mask)
{
    string folderName = Path.GetDirectoryName(mask);
    if (null == folderName || folderName.Length == 0)
    {
        folderName = Environment.CurrentDirectory;
    }
    DirectoryInfo folder = new DirectoryInfo(Path.GetFullPath(folderName));
    return folder.GetFiles(Path.GetFileName(mask));
}

```

5. Save the file.
6. Now you need to go back and edit the STSADM **command definition file** you created earlier. Open the **stsadmcommands.records.xml** file for editing. Add the following XML code above the closing </commands> tag.

XML Snippet: 'SubmitFiles STSADM Command Definition'

```

PublicKeyTokenReplace with the actual public key token for the assembly
<command name="submitfiles"
    class="ECM401.OfficialFileSTSADMCommand.SubmitOfficialFile,
    ECM401.OfficialFileSTSADMCommand, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=$PublicKeyToken$"/>

```

7. Save the file and rebuild the project.
8. Open a command prompt and type the following command: stsadm -help submitfiles
9. You should see the following screen:


```
C:\>stsadm -help submitfiles

stsadm -o submitfiles
        Submits one or more files to an official records repository site.
-url      <url>           the url of the record center site
-path     <file mask>     the wildcard mask for the files to be submitted
-type     <routing type>  the routing type name

C:\>
```

10. Now submit some files using the command line. The following screen shows the result of sending two new contracts to the record center:

```
C:\Sample Documents>dir *.docx
Volume in drive C has no label.
Volume Serial Number is 10B1-B1F4

Directory of C:\Sample Documents

11/09/2007  01:33 PM                13,248 LoanAgreement.docx
11/09/2007  12:33 PM                16,846 SalesAgreement.docx
                2 File(s)                30,094 bytes
                0 Dir(s)            479,547,392 bytes free


C:\Sample Documents>stsadm -o submitfiles -url http://litwareserver:9997 -path *.docx -type Contract


LoanAgreement.docx
--> Success
SalesAgreement.docx
--> Success


C:\Sample Documents>_
```


11. Open the browser and navigate to the record center site to verify the results.

Litware Records

Welcome System Account | My Site | My Links | 

 Litware Records






This List: Legal Documents 





Home 


Litware Records > Legal Documents > 2007-11-14T09-18-44Z

Legal Documents

A library for legal contracts, agreements and litigation support documents like briefs, motions and court filings.

New  Upload  Actions  Settings  View: All Documents 

Type	Name	Modified	Modified By	Hold Status
	Properties	11/14/2007 4:18 AM	System Account	
	Contract for Sale of Goods_OZYMG2 ! NEW	11/14/2007 4:18 AM	System Account	
	LoanAgreement_RI7CLJ ! NEW	11/14/2007 8:44 AM	System Account	
	SalesAgreement_WDIY8Z ! NEW	11/14/2007 8:44 AM	System Account	



View All Site Content

Documents


- Unclassified Records
- Legal Documents
- Financial Documents

Lists

- Record Routing
- Holds
- Tasks

Sites

People and Groups

 Recycle Bin

This concludes the lab exercises.

Lab 7: Building Custom Routers

Lab Time: 60 Minutes

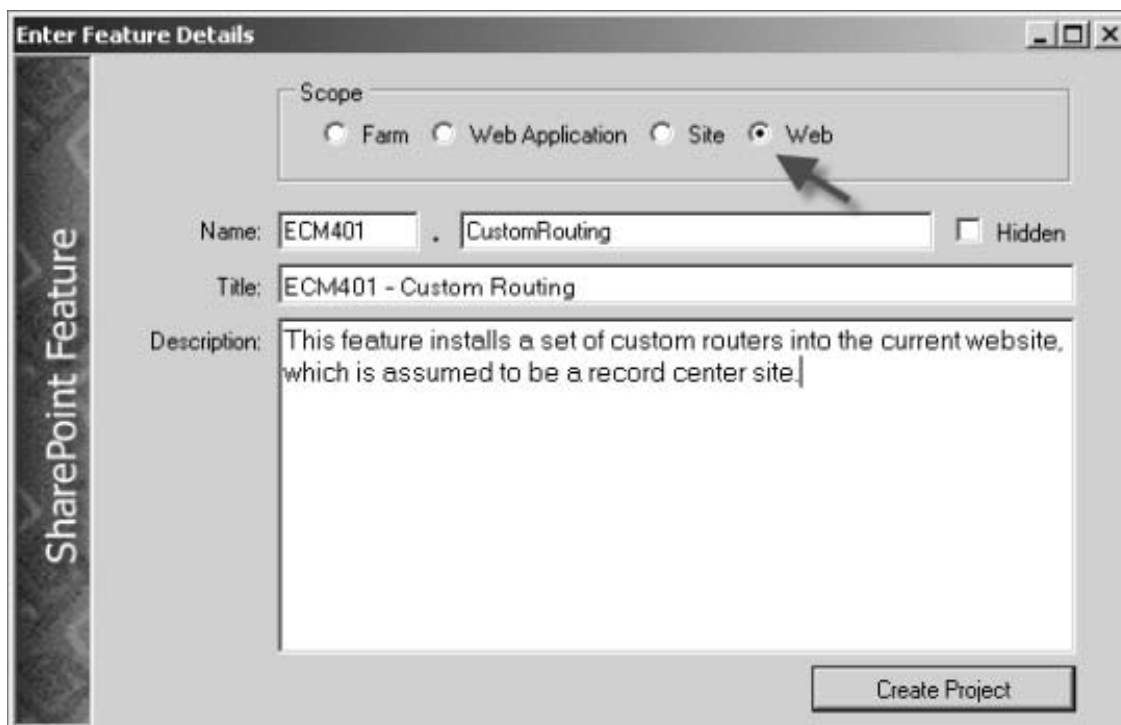
Lab Directory: ECM401.CustomRouting

Lab Overview:

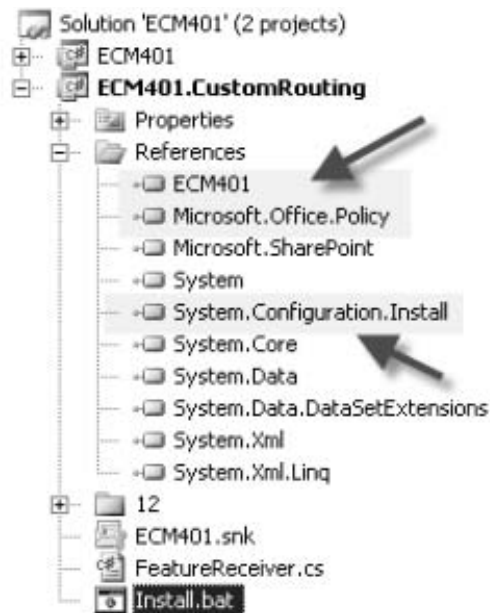
In this lab, you will extend the record routing capability of the Litware Record Center to redirect, track and filter incoming records based on document metadata as well as custom business rules. This will require that you create several custom routers that will be installed by a feature that can be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

2. 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.
3. 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.

4. 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;
}
```

5. 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;
}
```

```
}
```

6. 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));
}

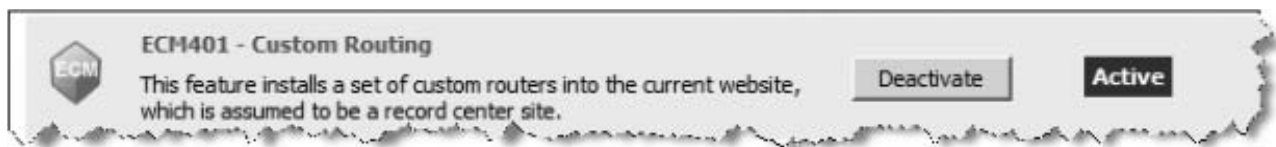
/// <summary>
/// Override the feature deactivating event to remove custom routers.
/// </summary>
/// <param name="properties"></param>
public override void FeatureDeactivating(SPFeatureReceiverProperties
properties) {
    using (SPWeb web = properties.Feature.Parent as SPWeb)
        SharePointRouter.RemoveRouter(web,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

```

2. 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 your 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.



Intranet > Record Center

Welcome System Account | My Site | My Links |

Record Center

Record Center > Record Routing > Contract > Edit Item

Record Routing: Contract

Attach File | Delete Item | Spelling... * indicates a required field

Title * Contract

Description Binding legal agreements.

Location * Legal Documents
The title of the library where records matching this record routing item should be stored. Libraries used to store submitted records cannot be deleted.

Aliases Agreement/Promissory Note
A '/' delimited list of alternative names that represent this record routing entry.

Default ☐
If checked, this routing item will be used for submitted records that do not match the title or aliases of any other record routing item.

Router ECM401 Filtering Router
The custom router to use for records matching this item.

Created at 10/30/2008 3:11 PM by System Account
Last modified at 10/30/2008 3:12 PM by System Account

- 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

- 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.
- 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;
```

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


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

4. 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.SPLListItem item = m_list.Items.Add();
        item[colRecordSeries] = recordSeries;
        item[colSourceUrl] = sourceUrl;
        item[colUserName] = userName;
        item[colFileSize] = fileSize;
        item[colDestination] = destination;
        item.Update();
    }
    catch { }
}

```

5. 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.SPLList 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;
}

```

6. 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();
}

```

7. 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.

8. Open the **FeatureReceiver.cs** file and add the following line to the **FeatureActivated** method.

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

9. Add the following line to the **FeatureDeactivating** method.

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

10. 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;
```

3. 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()));
    }
}
```

4. 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.
5. 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;
}

```

6. 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.

7. 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;
}

```

8. 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;
}

```

9. 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));
```

10. 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.

Lab 8: Gathering Schematized Data

Lab Time: 60 Minutes

Lab Directory: ECM401.SalesProposalSchema

Lab Overview:

In this lab, you will develop a reusable object model and API for managing sales proposals. Your object model will be based on an XML schema that you will use to validate the metadata that will be associated with every sales proposal document. You will also use the schema to develop an InfoPath form for gathering the metadata and storing it in a SharePoint form library.

Exercise 1: Create a Sales Proposal Schema

1. Start by creating a new **Class Library** DLL project in Visual Studio. Give it the name **ECM401.SalesProposal** and delete the auto-generated **Class1.cs** file.
2. Right-click the **ECM401.SalesProposal** project node and select **Add -> New Item...** from the context menu. Choose the **XML Schema** item template and give it the name **SalesProposal.xsd**.

Note: You will be using the built-in schema editor within Visual Studio. This means you will develop the schema using the "raw" xml-based schema definition language.

3. Add the following code **inside** the **xs:schema** element.

Note: Be careful not to overwrite the generated **targetNamespace** and **xmlns** attributes when you insert the code snippet.

4. First, you will define the root **Sales Proposal** object. This object will contain all the other parts of the sales proposal data definition.

XML Snippet: 'Sales Proposal Schema - Root'

```
<!-- Root Object -->
<xs:element name="SalesProposal">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="CompanyInfo" type="CompanyInfo" minOccurs="1"
maxOccurs="1"/>
      <xs:element name="ClientInfo" type="ClientInfo" minOccurs="1"
maxOccurs="1"/>
      <xs:element name="ProposalInfo" type="ProposalInfo" minOccurs="1"
maxOccurs="1"/>
    </xs:sequence>
    <xs:attribute name="DateCreated" type="xs:date" use="optional"/>
  </xs:complexType>
</xs:element>
```

- Next, you will define an object to describe the company offering the proposal, as well as the people involved in developing the proposal. This is a separate type that will correspond to a C# class when the schema is transformed. This object includes a sub-component that holds the company address information.

XML Snippet: 'Sales Proposal Schema - Company Info'

```
<!-- Company Information -->
<xs:complexType name="CompanyInfo">
  <xs:sequence>
    <xs:element name="Name" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="Author" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="Manager" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="SignedBy" type="xs:string" minOccurs="1"
maxOccurs="1"/>
    <xs:element name="SignedByTitle" type="xs:string" minOccurs="1"
maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>

<!-- Address Block -->
<xs:complexType name="AddressBlock">
  <xs:sequence>
    <xs:element name="Street" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="City" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="State" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="Zip" type="xs:string" minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>
```

- The next component describes the client who receives the proposal.

Note: The **CompanyInfo** and **ClientInfo** components both share the **AddressBlock** definition.

XML Snippet: 'Sales Proposal Schema - Client Info'

```
<!-- Company Information -->
<!-- Client Information -->
<xs:complexType name="ClientInfo">
  <xs:sequence>
    <xs:element name="Name" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="Description" type="xs:string" minOccurs="1"
maxOccurs="1"/>
    <xs:element name="Address" type="AddressBlock" minOccurs="1"
maxOccurs="1"/>
    <xs:element name="Contacts" type="ContactList" minOccurs="1"
maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>
```

- To facilitate communicating with the client, you will need a list of contacts. For maximum flexibility when dealing with collections of objects, you will declare a **Contact** object as well as a **ContactList** object.

XML Snippet: 'Sales Proposal Schema - Contacts'

```

<!-- Contact List -->
<xs:complexType name="ContactList">
  <xs:sequence>
    <xs:element name="Contact" type="Contact" minOccurs="0"
maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

<!-- Contact (describes an individual contact) -->
<xs:complexType name="Contact">
  <xs:sequence>
    <xs:element name="Name" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="Title" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="Email" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="Phone" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="Fax" type="xs:string" minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>

```

8. The **ProposalInfo** component describes the actual proposal, including title and description, starting and ending dates and a list of deliverables.

XML Snippet: 'Sales Proposal Schema - ProposalInfo'

```

<!-- Proposal Info -->
<xs:complexType name="ProposalInfo">
  <xs:sequence>
    <xs:element name="Title" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="Description" type="xs:string" minOccurs="1"
maxOccurs="1"/>
    <xs:element name="StartingDate" type="xs:date" minOccurs="1"
maxOccurs="1"/>
    <xs:element name="EndingDate" type="xs:date" minOccurs="1"
maxOccurs="1"/>
    <xs:element name="Deliverables" type="DeliverablesList" minOccurs="1"
maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>

<!-- Deliverables List -->
<xs:complexType name="DeliverablesList">
  <xs:sequence>
    <xs:element name="Deliverable" type="Deliverable" minOccurs="0"
maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

```

9. Finally, each **Deliverable** includes an identifier, an amount, and a **ServiceItem** which is defined as an enumeration of the kinds of services that the company provides.

XML Snippet: 'Sales Proposal Schema - Deliverables'

```

<!-- Deliverable (describes an individual deliverable) -->
<xs:complexType name="Deliverable">
  <xs:simpleContent>
    <xs:extension base="xs:string">

```

```

        <xs:attribute name="Name" type="xs:string" use="optional"/>
        <xs:attribute name="Type" type="ServiceItem" use="optional"/>
        <xs:attribute name="Amount" type="xs:decimal" use="optional"/>
    </xs:extension>
</xs:simpleContent>
</xs:complexType>

<!-- Service Item Enumeration -->
<xs:simpleType name="ServiceItem">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Design"/>
        <xs:enumeration value="Consulting"/>
        <xs:enumeration value="Mentoring"/>
        <xs:enumeration value="Development"/>
        <xs:enumeration value="Installation"/>
        <xs:enumeration value="SystemAdministration"/>
        <xs:enumeration value="SecurityAssessment"/>
        <xs:enumeration value="Programming"/>
        <xs:enumeration value="ProjectManagement"/>
        <xs:enumeration value="BusinessDevelopment"/>
        <xs:enumeration value="Marketing"/>
        <xs:enumeration value="Sales"/>
        <xs:enumeration value="Support"/>
    </xs:restriction>
</xs:simpleType>

```

Note: The **Deliverable** object is declared using the **xs:simpleContent** declaration so that the description can be conveniently placed inside the **<Deliverable>** tags.

Exercise 2: Generate Serialization Classes

1. In this exercise, you will generate wrapper classes for each of the components defined above. To do this, you will use the **XsdClassGenerator** custom tool which is located in the **Student\Resources\Tools** directory.

Note: If you have not done so already, you will need to run the **XsdClassGeneratorSetup** batch file in order to register the assembly as a **custom tool** within Visual Studio.

2. Make sure the **Properties** window is open in Visual Studio, then in the **Solution Explorer** window, right-click the **SalesProposal.xsd** file. Enter **XsdClassGenerator** in the **Custom Tool** property and press the **Enter** key.
3. A new node appears beneath the **SalesProposal.xsd** file. Double-click the generated **SalesProposal.cs** file to open it for editing. Your file should resemble the following illustration.

```
//-----
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:2.0.50727.3031
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//-----

//
// This source code was auto-generated by xsd, Version=2.0.50727.42.
//
namespace Sales_Proposal {
    using System.Xml.Serialization;

    /// <remarks/>
    [System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
    [System.SerializableAttribute()]
    [System.Diagnostics.DebuggerStepThroughAttribute()]
    [System.ComponentModel.DesignerCategoryAttribute("code")]
    [System.Xml.Serialization.XmlTypeAttribute(AnonymousType=true,
        Namespace="http://tempuri.org/SalesProposal.xsd")]
    [System.Xml.Serialization.XmlRootAttribute(
        Namespace="http://tempuri.org/SalesProposal.xsd", IsNullable=false)]
    public partial class SalesProposal {

        private CompanyInfo companyInfoField;

        private ClientInfo clientInfoField;

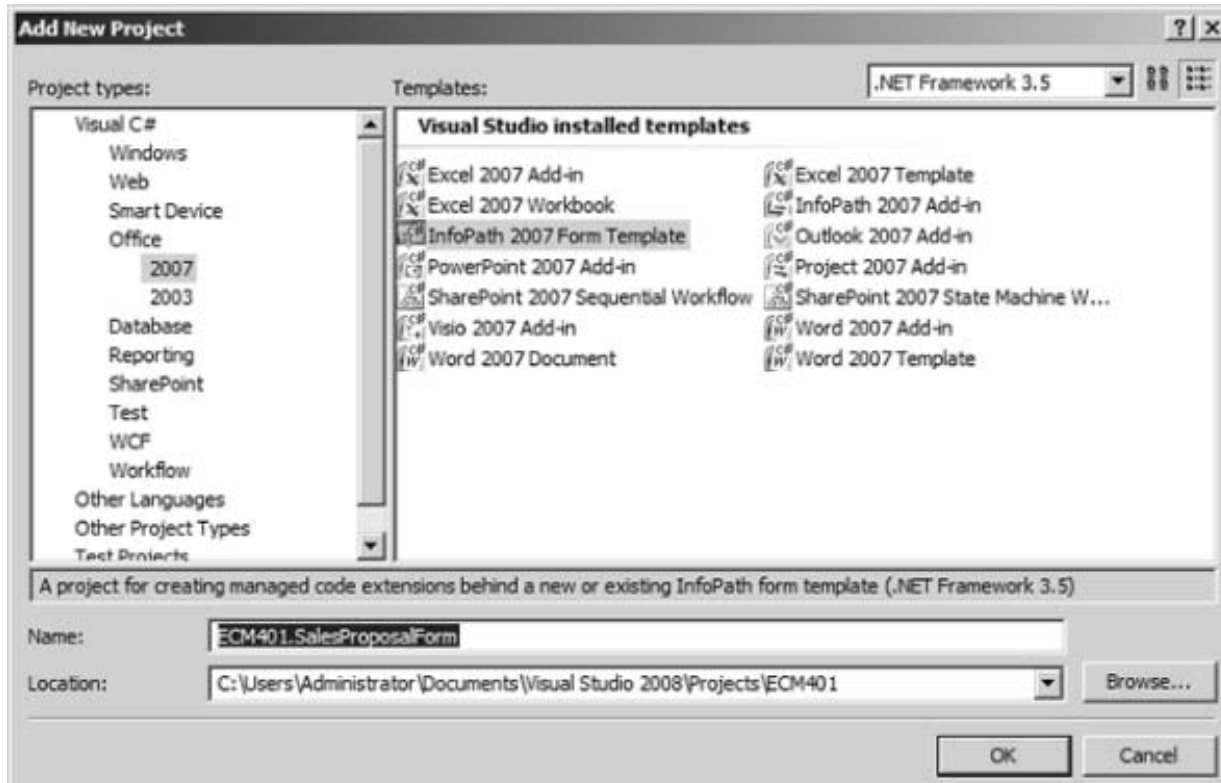
        private ProposalInfo proposalInfoField;
    }
}
```

4. In later exercises you will add code to extend the functionality of the generated classes. At this point, you can build the project to create a compiled assembly. This will ultimately become a custom **Sales Proposal API** that can be applied to different kinds of solutions.
5. In the next exercise you will use the schema to develop an InfoPath form for gathering Sales Proposal data.

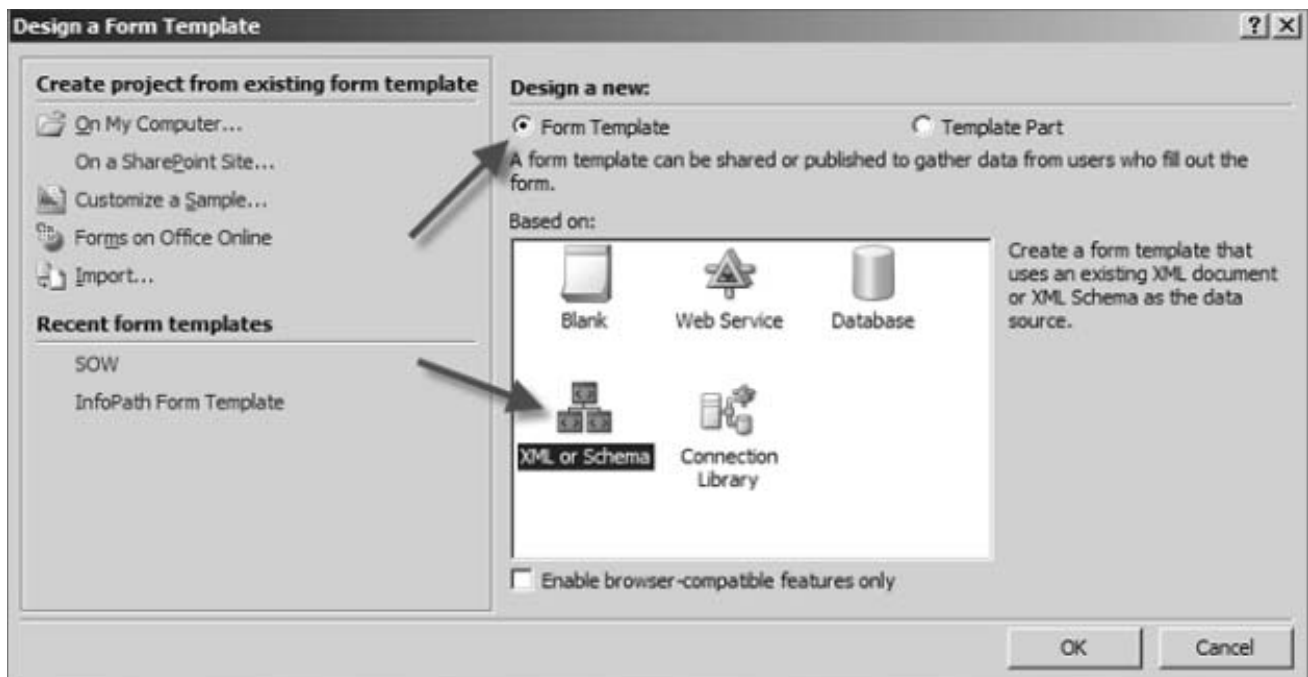
Exercise 3: Create an InfoPath Form for Gathering Data

In order to test the Sales Proposal API, you will need sample data. You have the option of entering the data manually, using the built-in schema validation provided by the Visual Studio XML editor or you can use InfoPath to create a richer user experience. In this exercise, you will use **Visual Studio Tools for Office (VSTO)** to create an InfoPath form to generate your sample data.

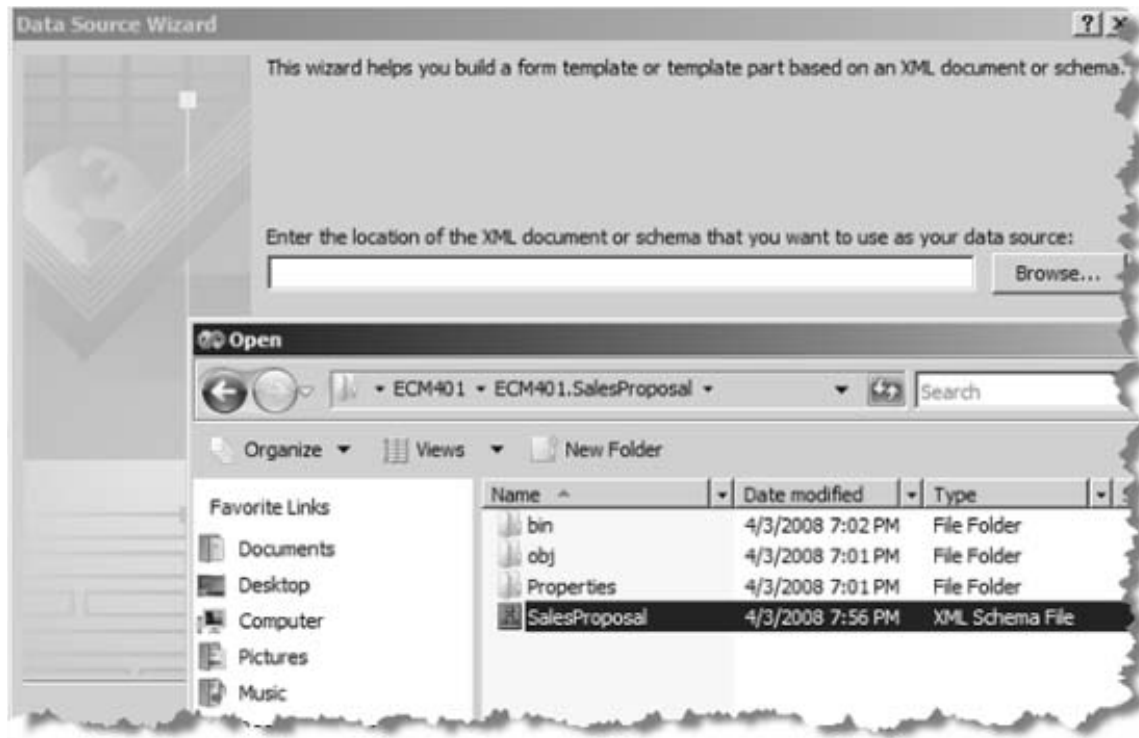
1. Start by creating a new **InfoPath Form Template** project in Visual Studio. Give it the name **ECM401.SalesProposalForm** as shown below.



2. In the next dialog, select **Form Template** based on **XML or Schema**. This allows you to automatically define the data source for the form based on the schema you have already created.



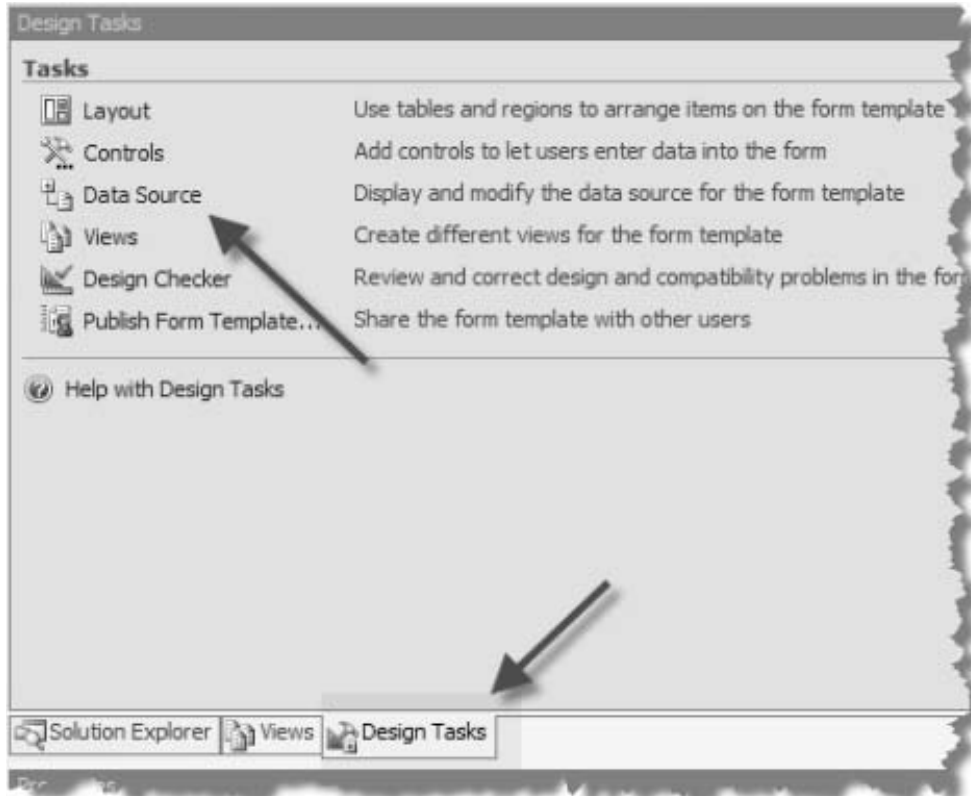
3. In the next dialog, browse to the **ECM401.SalesProposal** project folder and select the **SalesProposal.xsd** file you created above.



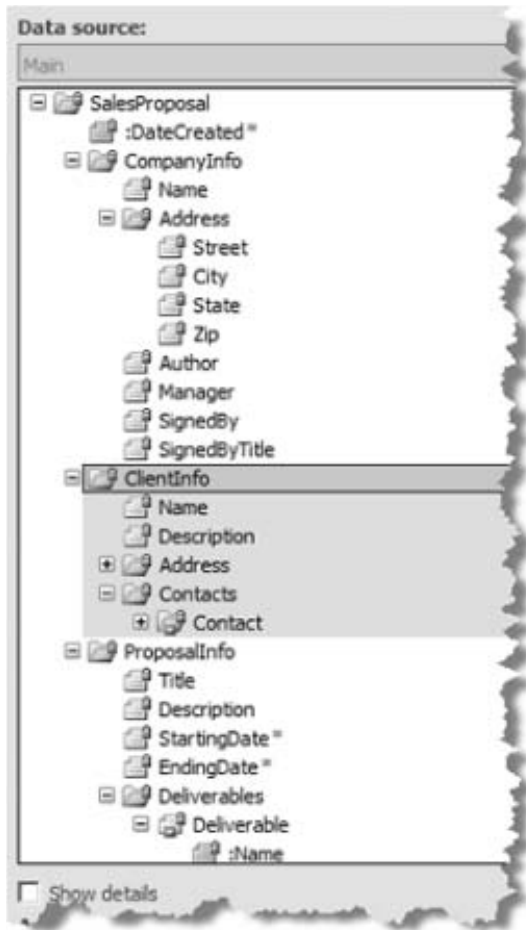
4. Press OK to generate the form template and open it in the Visual Studio **InfoPath Form Designer**.

Note: The InfoPath form designer is integrated into the Visual Studio IDE.

5. Open the **Design Tasks** window to begin designing your form.



6. Next, click the **Data Source** link to examine the **SalesProposal** data source definition that InfoPath has created. Notice that the data source is **locked** to prevent inadvertent modifications of the schema.



Note: By locking data sources based on external schemas, InfoPath ensures that the schema will not be changed when controls are dragged onto the form designer surface.

7. At this point, you can start dragging data elements onto the form. Use the following illustration as a guide as you drag and drop your controls. The most important point is to ensure you have included all of the data elements. One way to do that is to simply drag the entire data source onto the design surface and then rearrange the controls. Another approach (used here) is to create the basic layout using **layout tables** and then drag individual data elements or element groups into the layout table cells.

ECM401 Sales Proposal	
Date Created:	Title:
Starting Date:	Description:
Ending Date:	
	Author:
	Manager:
Client Information	Company Information
Name:	Name:
Description:	
Street	Street
City	City
State	State
Zip	Zip
Client Contacts	

8. When you finish designing your form, you are ready to test it. To do this, right-click the **manifest.xsf** file in the **Solution Explorer** and select **Open With...** from the context menu. Choose **Microsoft Office InfoPath 2007** and press **OK** . This will open a new instance of **InfoPath** with your form ready for editing. Enter some data as shown below.

Client Contacts				
Name	Title	Email	Phone	Fax
James Madison	Trainer	jmadison@nexient.c	122-399-4808	908-388-3884
William Penn	Sales Agent	wpenn@nexient.coi	390-298-3840	374-273-3777

☒ Insert item

Schedule of Deliverables			
Name	Type	Amount	Deliverable
M101	Mentoring	1,200	Deliver training materials
M202	Consulting	3,500	Provide consulting services
M303	Design	4,500	SharePoint branding

☒ Insert item

9. After entering your sample data, save the file by browsing to the **ECM401.SalesProposal** project directory. Name the file **Sample Data** . InfoPath will save the input data to a new file named **Sample Data.xml** . Double-click the file in the **ECM401.SalesProposal** project to view it within Visual Studio.

```

<?xml version="1.0" encoding="UTF-8"?><?mso-infoPathSolution
solutionVersion="1.0.0.8" productVersion="12.0.0" PIVersion="1.0.0
name="urn:schemas-microsoft-com:office:infopath:ECM401-
SalesProposalForm:http---tempuri-org-SalesProposal-xsd" language="
" ?><?mso-application progid="InfoPath.Document" versionProgid=
"InfoPath.Document.2"?><msns:SalesProposal DateCreated="2008-04-0
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:msns="
://tempuri.org/SalesProposal.xsd" xmlns:xd="http://schemas.microsof
com/office/infopath/2003">
  <msns:CompanyInfo>
    <msns:Name>Ted Pattison Group</msns:Name>
    <msns:Address>
      <msns:Street></msns:Street>
      <msns:City>Tampa</msns:City>
      <msns:State>Florida</msns:State>
      <msns:Zip></msns:Zip>
    </msns:Address>
    <msns:Author>John F. Holliday</msns:Author>
    <msns:Manager>Ted Pattison</msns:Manager>
    <msns:SignedBy></msns:SignedBy>
    <msns:SignedByTitle></msns:SignedByTitle>
  </msns:CompanyInfo>
  <msns:ClientInfo>
    <msns:Name>Nexient Learning</msns:Name>
    <msns:Description>The Learning Center</msns:Description>
    <msns:Address>
      <msns:Street>30 Eglinton Avenue</msns:Street>
      <msns:City>Mississauga</msns:City>
      <msns:State>Ontario</msns:State>
      <msns:Zip></msns:Zip>
    </msns:Address>
    <msns:Contacts>
      <msns:Contact>

```

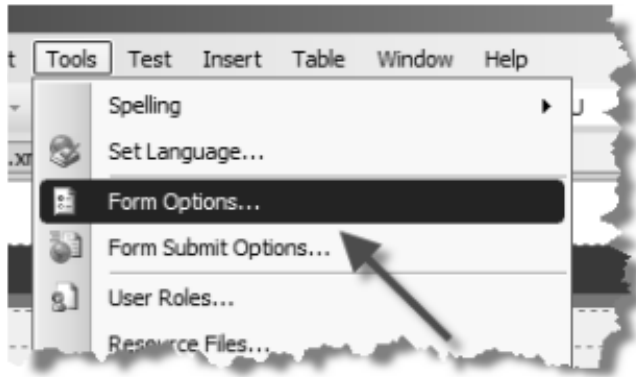
10. Now you have schematized data that can be serialized and deserialized using a custom API and that can also be created easily using a rich user interface that ensures the data will conform to the schema. In the next exercise, you will publish the form to SharePoint and use it to gather sales proposal data.

Exercise 4: Publish the Form to SharePoint

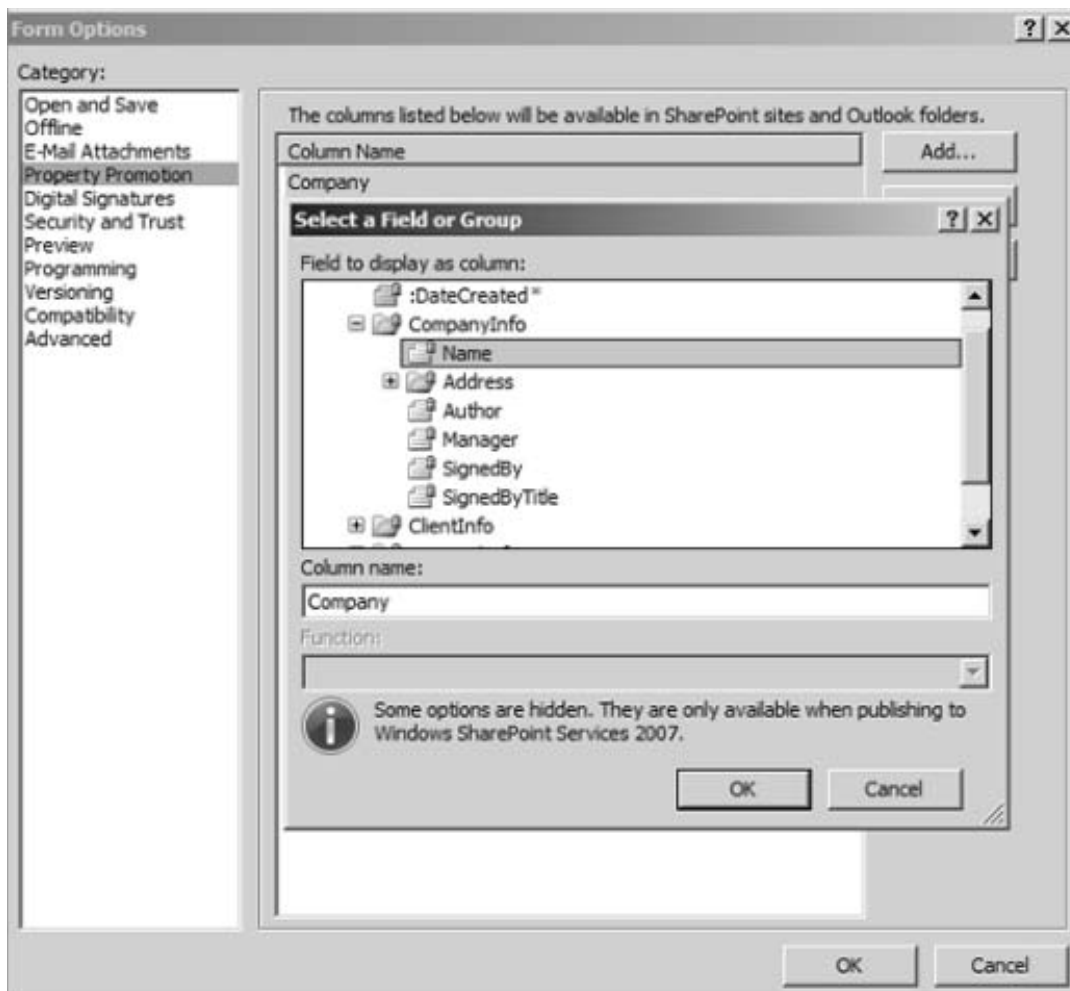
In this exercise, you will publish the **Sales Proposal** form to a SharePoint **Form Library** . This will form the basis for extending the functionality of your solution in later labs and exercises. Instead of publishing the form directly into SharePoint, you will package the form into a **Feature** so it can be deployed easily to many sites.

1. Start by creating a new **SharePoint Feature Project** named **ECM401.SalesProposalFeature** . Set the **Feature Scope** to **Web** .

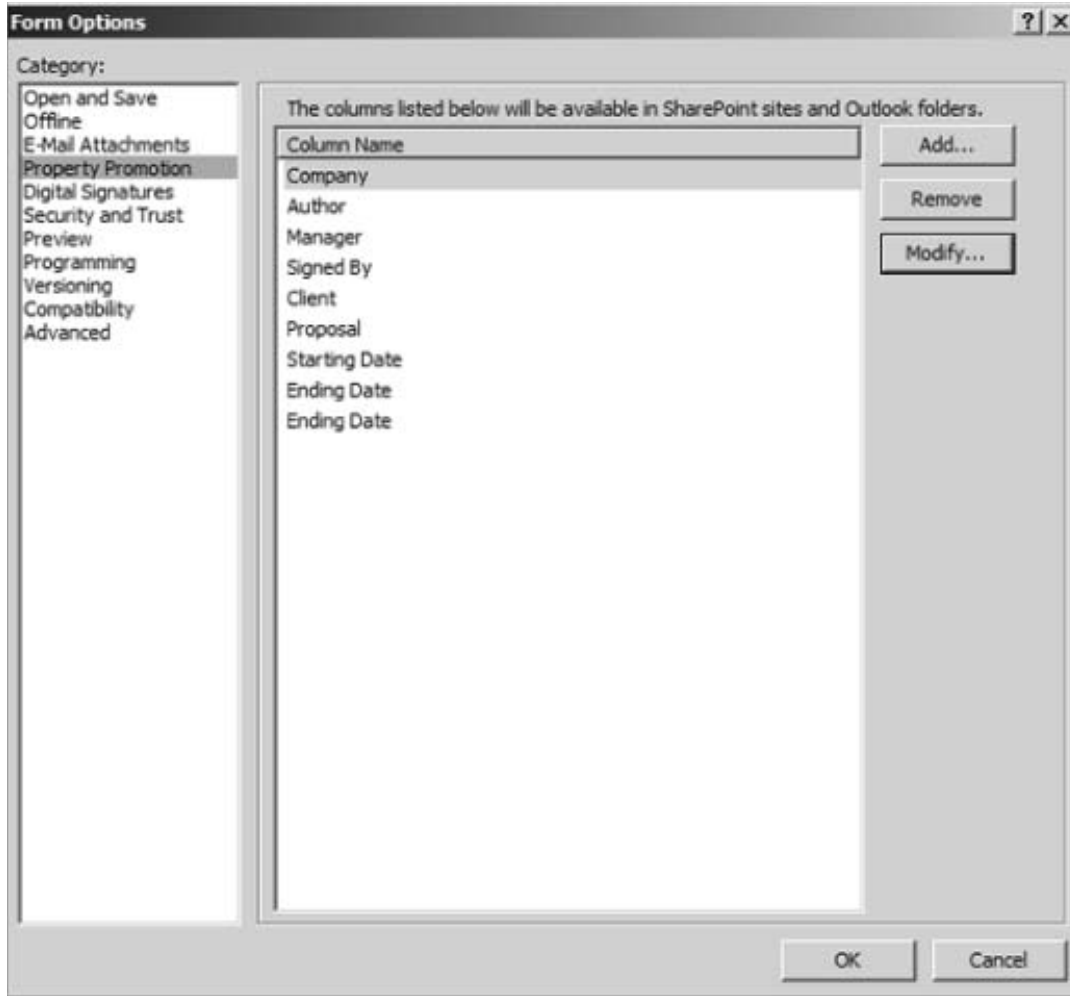
- Next, re-open the **manifest.xsf** file in **design mode** by double-clicking in the **Solution Explorer** window. Select **Form Options...** from the **Tools** menu.



- Select **Property Promotion** on the left side of the **Form Options** dialog. From here, you will select the data elements you wish to be promoted to SharePoint list columns. Starting with the **Company Name** field, navigate to the **Name** element inside the **Company Info** element and give it the name **Company** as shown below.

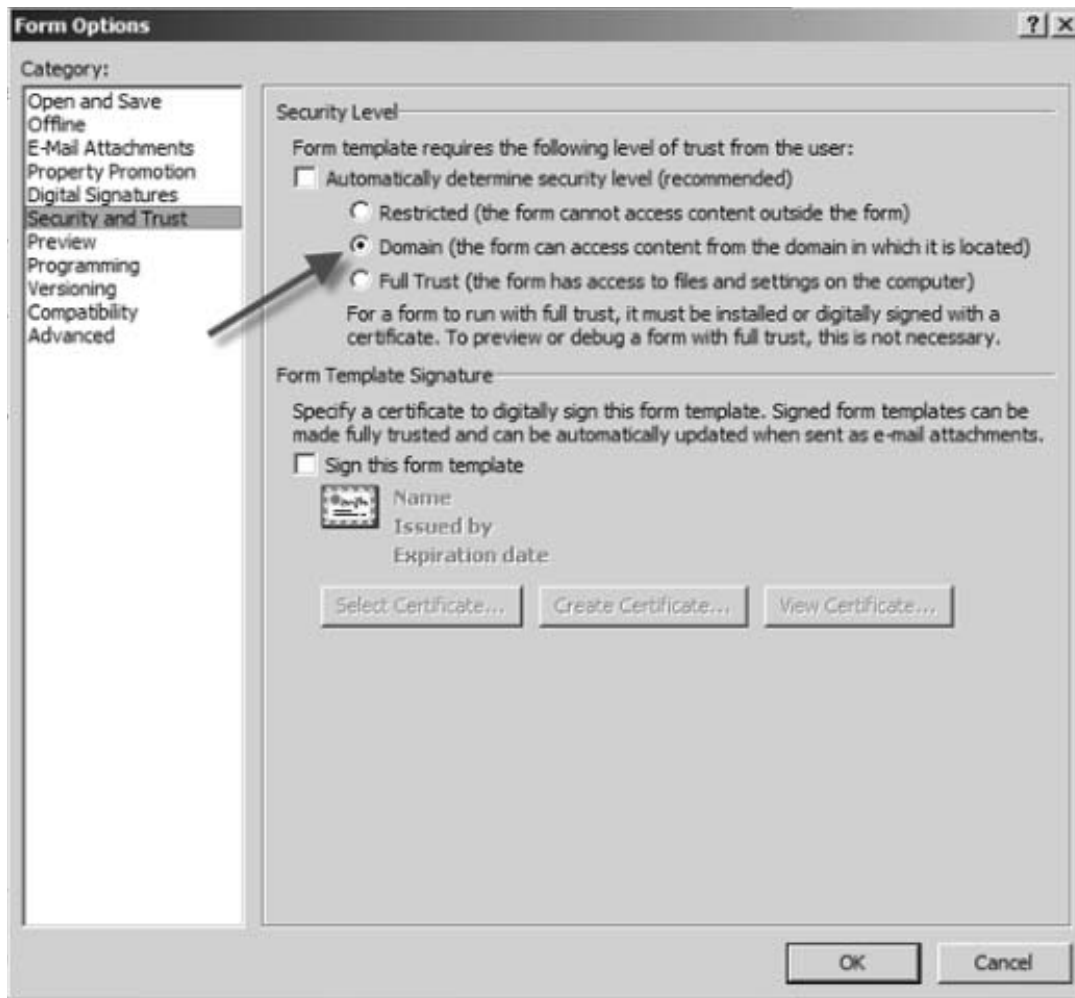


4. Continue adding the remaining fields shown below.



The dialog box is titled "Form Options" and has a standard Windows window frame with a question mark and close button in the top right corner. On the left, under the heading "Category:", there is a list of categories: "Open and Save", "Offline", "E-Mail Attachments", "Property Promotion", "Digital Signatures", "Security and Trust", "Preview", "Programming", "Versioning", "Compatibility", and "Advanced". The "Security and Trust" category is currently selected. The main area of the dialog contains the text "The columns listed below will be available in SharePoint sites and Outlook folders." followed by a list of column names: "Column Name", "Company", "Author", "Manager", "Signed By", "Client", "Proposal", "Starting Date", "Ending Date", and "Ending Date". To the right of this list are three buttons: "Add...", "Remove", and "Modify...". At the bottom right of the dialog are "OK" and "Cancel" buttons.

5. Select **Security and Trust** on the left side of the dialog. Specify **Domain** security for the form. This will allow it to be opened from within a SharePoint form library.

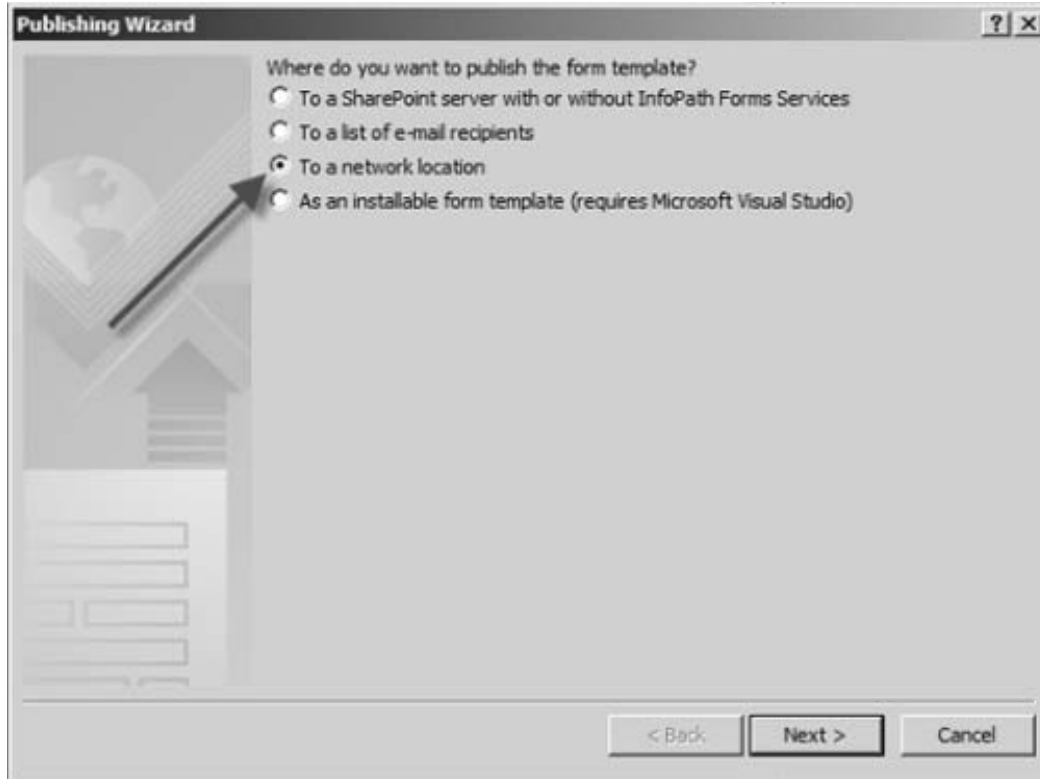


6. Click **OK** to save your changes and return to the design surface. From the **Design Tasks** window, click the **Publish Form Template...** link.

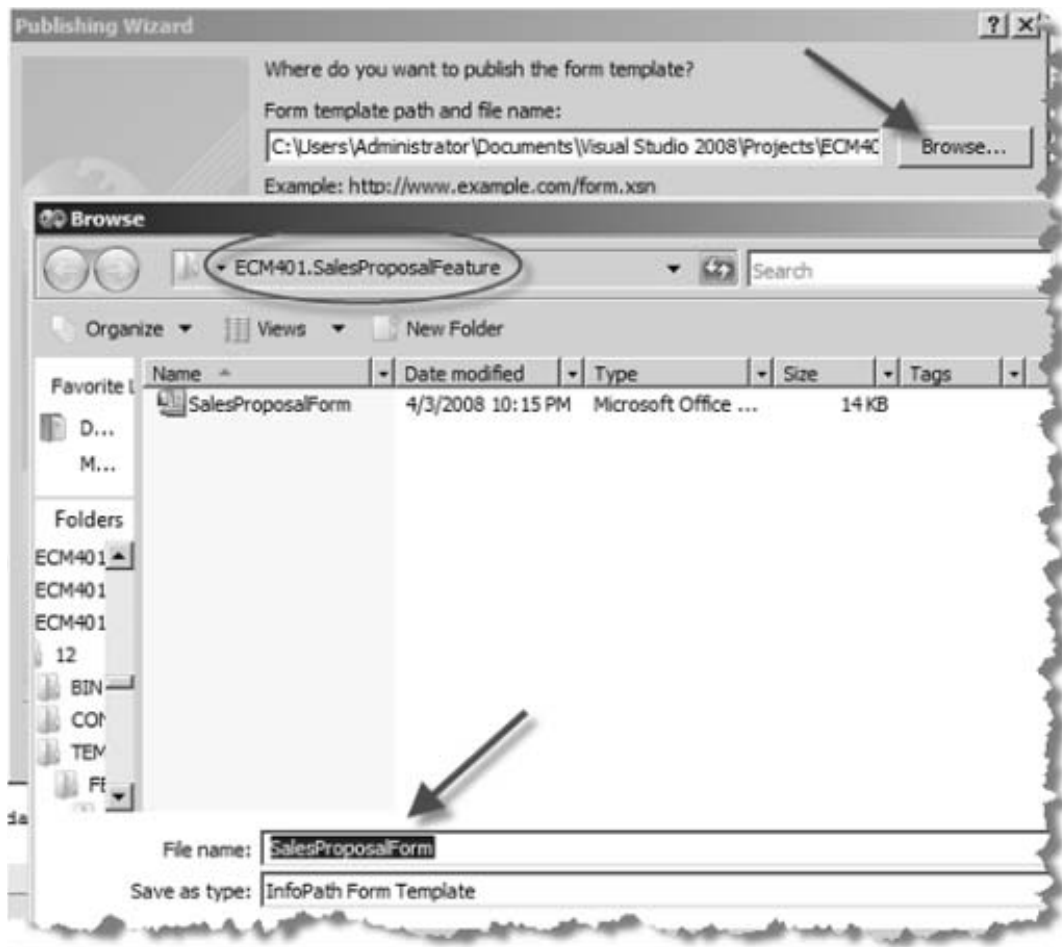


Note: "Publishing" is **NOT** the same as "saving" a form template. The form template you are working on is saved as part of the SalesProposalForm project. In this next step, you will publish the form into the **12\TEMPLATE\FEATURES\ECM401.SalesProposalFeature** folder so it can be packaged along with the feature.

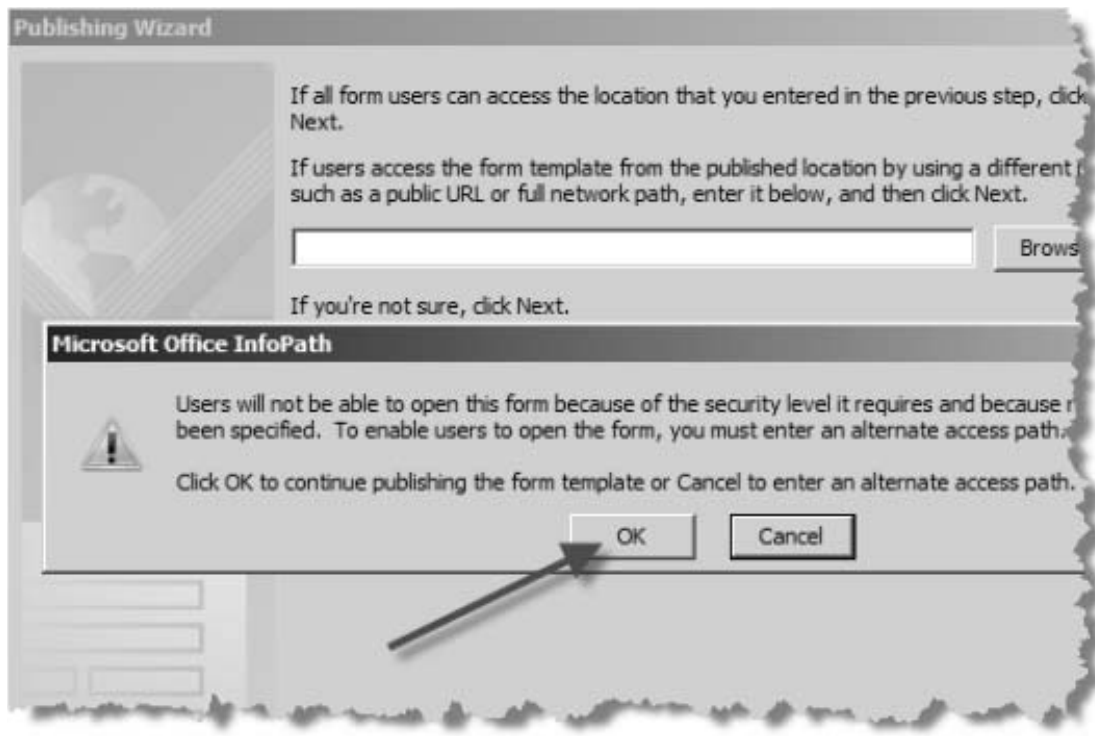
7. When you click the **Publish Form Template...** link, Visual Studio responds by building your project. This is necessary because any managed code you might write would have to be compiled and packaged within the form template you are publishing. In this case, you are not writing any managed code, but the assembly is part of the form template anyway. After the project is built, you are presented with series of dialogs. Choose the options shown in the next series of illustrations.



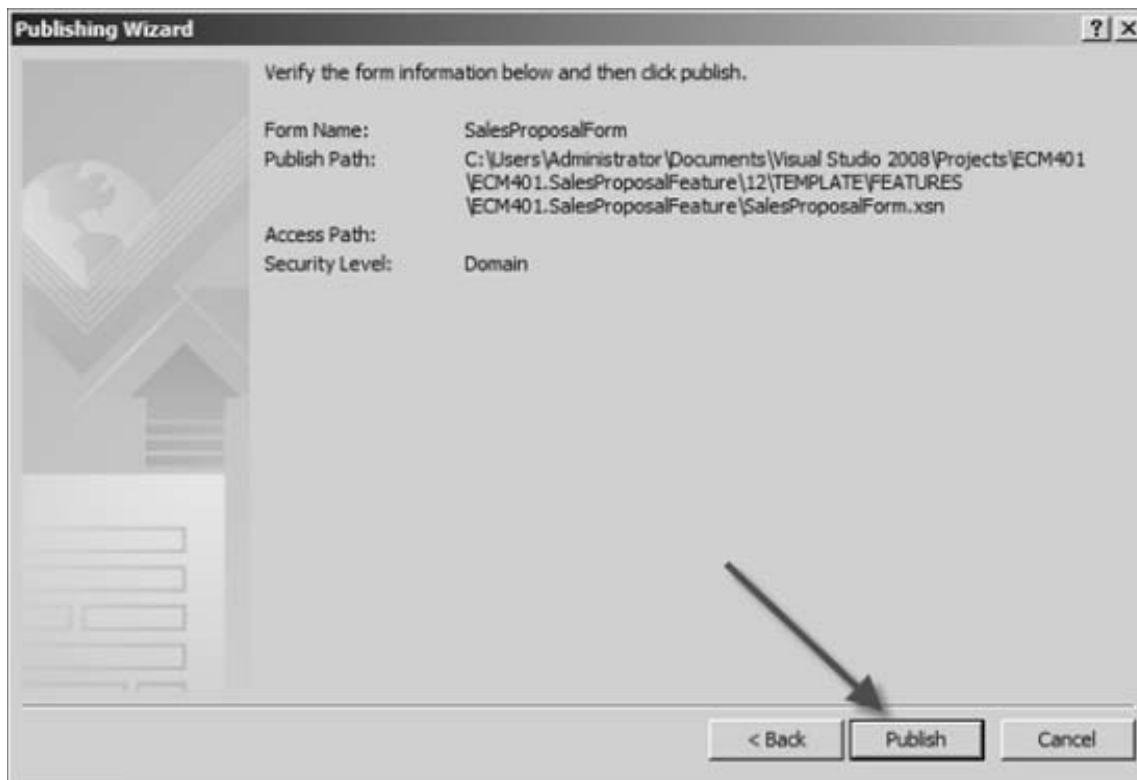
8. Browse to the **ECM401.SalesProposalFeature** project folder. Within that, navigate down to the **12\TEMPLATE\FEATURES\ECM401.SalesProposalFeature** folder. Give the form the name **SalesProposal** .



9. Clear the default access path, and then click **OK** to continue in the popup dialog box.



10. Review your options and then click **Publish** to generate the published form.



11. Now that you have a form template in the feature folder, you can proceed to package it into your solution. Double-click the **elements.xml** file in the **ECM401.SalesProposalFeature** project and insert the following code inside the **<Elements>** tag.

XML Snippet: 'Sales Proposal Feature Elements'

```
<ListInstance
  FeatureId="00BFEA71-1E1D-4562-B56A-F05371BB0115"
  TemplateType="115"
  Id="SalesProposals"
  Title="Sales Proposals"
  Description="Proposal data for clients."
  Url="SalesProposals"
  OnQuickLaunch="True"
/>

<Module Name="FormTemplate" List="115" Url="SalesProposals/Forms">
  <File Url="SalesProposalForm.xsn" Type="GhostableInLibrary"/>
</Module>
```

Note: These elements declare a form library with the name "Sales Proposals" and instruct SharePoint to load the form into a specific location within the form library. In the next step, you will write code that attaches the form to the library as a document template.

12. Double-click the **FeatureReceiver.cs** file to open it for editing. Add an override for the **OnActivated** virtual method. Be sure to choose the version that is appropriate for the feature scope. In this case, choose the method that accepts an **SPWeb** parameter. Replace the body of the method with the following code.

Code Snippet: 'Sales Proposal List'

```
try
{
    // Setup the document template for the form library.
    SPDocumentLibrary formLib = web.Lists["Sales Proposals"] as
SPDocumentLibrary;
    if (formLib != null)
    {
        formLib.DocumentTemplateUrl =
"SalesProposals/Forms/SalesProposalForm.xsn";
        formLib.Update();
    }
}
catch
{
}
```

13. Build the project, open the web browser and navigate to the **localhost:401/bpm/part1** site. From the **Site Actions** menu, choose **Site Settings** and then choose **Site Features**. Scroll to the **ECM401.SalesProposalFeature** and activate it. Go back to the home page of the site and verify that the **Sales Proposal** form library was created.



14. Open the form library and click the **New** menu command to create a new form. Verify that InfoPath opens with your form.

This concludes the lab exercises.

Lab 9: Using the WSS Provisioning Engine for Publishing

Lab Time: 45 Minutes

Lab Directory: ECM401.SiteProvisioning

Lab Overview:

In this lab you will create a SharePoint Feature that will provision a set of site columns, content types and page layouts for web publishing. The content type will be based on the default Page content type installed by the publishing framework.

Exercise 1: Creating Site Columns using a SharePoint Feature

1. Start by creating a Class Library project named **ECM401.Publishing** using the **SharePoint Feature** project template. Accept the default name and title, enter a description such as *"This feature creates and deploys custom publishing resources."*, and select **Site** as the scope for the feature. Press **Create** to close the **Feature Wizard** dialog box.

Note: You must select **Site** because you will be provisioning **Field** elements, which are not allowed at **Web** scope.

2. Open the feature.xml file located in the **12\TEMPLATE\FEATURES\ECM401.Publishing** folder. Replace the contents of the **<ElementManifests>** section with the highlighted lines shown below.

```
<ElementManifests>
<ElementManifest Location="SiteColumns.xml"/>
<ElementManifest Location="ContentTypes.xml"/>
</ElementManifests>
```

3. Save and close the **feature.xml** file. Next, you will create two element manifest files - one for site columns and another for content types.
4. First, rename the existing **elements.xml** file to **SiteColumns.xml**. Replace the file contents with the following XML code.

```
<Elements xmlns="http://schemas.microsoft.com/sharepoint/">
<!-- new site column off the "site line of text" field type -->
<Field SourceID="http://schemas.microsoft.com/sharepoint/"
ID="{DC172FD7-CC9D-493c-A1EB-231B8AF29F3E}"
Name="Division2"
StaticName="Division2"
DisplayName="Division"
Group="ECM401"
Type="Text"
Required="FALSE"
Sealed="FALSE"
```

```
Hidden="FALSE" />

<!-- new site column based off the "publishing html" field type -->
<Field SourceID="http://schemas.microsoft.com/sharepoint/"
ID="{8DDDC83E-625C-4d5e-B4EF-0CD8A3A434C2}"
Name="DivisionDescription"
StaticName="DivisionDescription"
DisplayName="Division Description"
Group="ECM401"
Type="HTML"
Required="FALSE"
Sealed="FALSE"
Hidden="FALSE" />
</Elements>
```

- Now add a second XML file to the same folder named **ContentTypes.xml** and replace its contents with the following XML code.

Note: YOU WILL **MODIFY THE CONTENT TYPE ID IN THE NEXT STEP**

```
<Elements xmlns="http://schemas.microsoft.com/sharepoint/">
<!-- this content type inherits from the "article page" content type -->
<ContentType
ID="0x010100C568DB52D9D0A14D9B2FDCC96666E9F2007948130EC3DB064584E219954237AF390
0242457EFB8B24247815D688C526CD44D00976A5C165AB04065B1F9AFAE0E4658B5"
Name="Division Article Feature"
Group="ECM401">
<FieldRefs>
<FieldRef ID="{DC172FD7-CC9D-493c-A1EB-231B8AF29F3E}" Name="Division" />
<FieldRef ID="{8DDDC83E-625C-4d5e-B4EF-0CD8A3A434C2}"
Name="DivisionDescription" />
</FieldRefs>
<DocumentTemplate TargetName="/_layouts/CreatePage.aspx" />
</ContentType>
</Elements>
```

Note: The content type ID you use here will cause this content type to inherit its metadata from the **Article Page** content type created by the MOSS Publishing framework. To get the correct identifier, perform the following steps:

- Navigate to the **C:\Program Files\Common Files\Microsoft Shared\web server extensions\12\TEMPLATE\FEATURES\PublishingResources** folder in **Windows Explorer**.
 - Copy the **PublishingContentTypes.xml** file to **C:**.
 - Open the **C:\PublishingContentTypes.xml** file in Visual Studio and search for the string **"\$Resources:cmscore,contenttype_articlepage_name;"**.
 - Copy the content type ID into the ID attribute of the content type you are creating.
 - Append **"00"** to your new ID.
 - Select **Create GUID** from the **Tools** menu in Visual Studio. Copy and paste the new GUID after the **"00"**.
 - Remove the curly braces and hyphens from the identifier.
- You now have a feature that will create new site columns and a new Page content type containing those columns. In the next exercise, you will add a page layout to the feature.

Exercise 2: Adding a Page Layout

In this exercise you will update the Feature created in the previous exercise to include a custom page layout with an associated preview image. Once this is done, you can deploy the solution, activate the feature, and begin testing.

1. Start by creating the master page template file. Right-click on the **ECM401.Publishing** folder in the Visual Studio **Solution Explorer** and select **Add > New Item...** from the context menu. Create a new text file named "ECM401ArticleLayout.aspx".
2. Replace everything in the file using the "ECM401ArticleLayout" code snippet.

XML Snippet: 'ECM401 Article Layout'

```
<%@ Page language="C#"

Inherits="Microsoft.SharePoint.Publishing.PublishingLayoutPage,Microsoft.ShareP
oint.Publishing,Version=12.0.0.0,Culture=neutral,PublicKeyToken=71e9bce111e9429
c" meta:progid="SharePoint.WebPartPage.Document" %>

<%@ Register Tagprefix="SharePointWebControls"
Namespace="Microsoft.SharePoint.WebControls" Assembly="Microsoft.SharePoint,
Version=12.0.0.0, Culture=neutral, PublicKeyToken=71e9bce111e9429c" %>

<%@ Register Tagprefix="WebPartPages"
Namespace="Microsoft.SharePoint.WebPartPages" Assembly="Microsoft.SharePoint,
Version=12.0.0.0, Culture=neutral, PublicKeyToken=71e9bce111e9429c" %>

<%@ Register Tagprefix="PublishingWebControls"
Namespace="Microsoft.SharePoint.Publishing.WebControls"
Assembly="Microsoft.SharePoint.Publishing, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce111e9429c" %>

<%@ Register Tagprefix="PublishingNavigation"
Namespace="Microsoft.SharePoint.Publishing.Navigation"
Assembly="Microsoft.SharePoint.Publishing, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce111e9429c" %>

<asp:Content ContentPlaceholderID="PlaceHolderAdditionalPageHead"
runat="server">
  <PublishingWebControls:editmodepanel runat="server" id="editmodestyles">

    <!-- Styles for edit mode only-->
    <SharePointWebControls:CssRegistration name="<% $SPUrl:~sitecollection/Style
Library/~language/Core Styles/zz2_editMode.css %>" runat="server"/>

  </PublishingWebControls:editmodepanel>

  <SharePointWebControls:CssRegistration name="<% $SPUrl:~sitecollection/Style
Library/~language/Core Styles/rca.css %>" runat="server"/>

  <SharePointWebControls:FieldValue id="PageStylesField"
FieldName="HeaderStyleDefinitions" runat="server"/>
</asp:Content>

<asp:Content ContentPlaceholderID="PlaceHolderPageTitle" runat="server">
  <SharePointWebControls:FieldValue id="PageTitle" FieldName="Title"
runat="server"/>
</asp:Content>
```



```

<asp:Content ContentPlaceholderID="PlaceHolderPageTitleInTitleArea"
runat="server">
  <SharePointWebControls:TextField runat="server" id="TitleField"
FieldName="Title" />
</asp:Content>

<asp:Content ContentPlaceholderId="PlaceHolderTitleBreadcrumb" runat="server">
  <div class="breadcrumb">
    <asp:SiteMapPath ID="siteMapPath" Runat="server"
SiteMapProvider="CurrentNavSiteMapProviderNoEncode"
RenderCurrentNodeAsLink="false" SkipLinkText="" CurrentNodeStyle-
CssClass="breadcrumbCurrent" NodeStyle-CssClass="ms-sitemapdirectional"/>
  </div>
</asp:Content>

<asp:Content ContentPlaceholderID="PlaceHolderMain" runat="server">

  <SharePointWebControls:CssRegistration name="<% $SPUrl:~sitecollection/Style
Library/~language/Core Styles/pageLayouts.css %>" runat="server"/>

  <div style="clear:both">&nbsp;</div>

  <table class="floatLeft" cellpadding=0 cellspacing=0>
    <tr>
      <td class="image">
        <PublishingWebControls:RichImageField id="ImageField"
FieldName="PublishingPageImage" runat="server" />
      </td>
    </tr>
    <tr>
      <td class="caption">
        <PublishingWebControls:RichHtmlField id="Caption"
FieldName="PublishingImageCaption" AllowTextMarkup="false" AllowTables="false"
AllowFonts="false" PreviewValueSize="Small" runat="server" />
      </td>
    </tr>
  </table>

  <table class="header">
    <tr>
      <td class="dateLine">
        <SharePointWebControls:datetimefield FieldName="ArticleStartDate"
runat="server" id="datetimefield3"></SharePointWebControls:datetimefield>
      </td>
      <td width=100% class="byLine">
        <SharePointWebControls:TextField FieldName="ArticleByLine" runat="server" />
      </td>
    </tr>
  </table>

  <div class="pageContent">
    <PublishingWebControls:RichHtmlField id="Content"
FieldName="PublishingPageContent" runat="server" />
  </div>

  <div style="clear:both"></div>

  <div>
    <strong>Division Name</strong>:
    <SharePointWebControls:TextField FieldName="Division" runat="server"
id="TextField1"></SharePointWebControls:TextField>
    <br /><PublishingWebControls:RichHtmlField
FieldName="Division_x0020_Description" runat="server"
id="RichHtmlField1"></PublishingWebControls:RichHtmlField>
  </div>

```

```

</div>

<PublishingWebControls:editmodepanel runat="server" id="editmodepanel1">
<!-- Add field controls here to bind custom metadata viewable and editable in
edit mode only.-->
<table cellpadding="10" cellspacing="0" align="center" class="editModePanel">
<tr>
<td>
<PublishingWebControls:RichImageField id="ContentQueryImage"
FieldName="PublishingRollupImage" AllowHyperLinks="false" runat="server" />
</td>
<td width="200">
<asp:Label text="<%=Resources:cms,Article_rollup_image_text%>"
runat="server" />
</td>
</tr>
</table>
</PublishingWebControls:editmodepanel>
</asp:Content>

```

- Now copy the image file **ECM401ArticlePageLayout.gif** from the **C:\Student\Resources\Images** folder into the **ECM401.Publishing** folder of the project.



- Create a new XML file in the same folder and give it the name **PageLayouts.xml** . Replace its contents with the following XML code.

```

<Elements xmlns="http://schemas.microsoft.com/sharepoint/">
<!-- add the page layout to the Master Page Gallery -->

<Module Url="_catalogs/masterpage"
RootWebOnly="TRUE">

<!-- provision the page layout into the Master Page Gallery -->
<File Url="ECM401ArticleLayout.aspx"
Name="ECM401ArticleLayoutFeature.aspx"
Type="GhostableInLibrary">
<!-- specify the content type associated this page layout is associated
with -->

<Property Name="PublishingAssociatedContentType"
Value=";#Division Article
Feature;#0x010100C568DB52D9D0A14D9B2FDCC96666E9F2007948130EC3DB064584E219954237
AF3900242457EFB8B24247815D688C526CD44D00976A5C165AB04065B1F9AFAE0E4658B5;#"
/>

<!-- specify the URL to the preview image, provisioned below -->
<Property Name="PublishingPreviewImage"
Value="~SiteCollection/_catalogs/masterpage/Preview
Images/Litware/ECM401ArticlePageLayoutFeature.gif,
~SiteCollection/_catalogs/masterpage/Preview
Images/Litware/ECM401ArticlePageLayoutFeature.gif" />
<!-- specify the page layout content type -->
<Property Name="ContentType"
Value="<%=Resources:cmscore,contenttype_pagelayout_name%>" />
<!-- specify the title of the page layout -->

```

```

    <Property Name="Title"
    Value="Litware Division Article" />
  </File>
</Module>

<!-- add the preview image to the Master Page Gallery -->
<Module Url="_catalogs/masterpage/Preview Images/Litware"
RootWebOnly="TRUE">
  <!-- provision the preview image into the Master Page Gallery -->
  <File Url="ECM401ArticlePageLayout.gif"
    Name="ECM401ArticlePageLayoutFeature.gif">
    <!-- specify the title of the preview image -->
    <Property Name="Title"
    Value="ECM401ArticlePageLayoutFeature.gif" />
  </File>
</Module>
</Elements>

```

Note: You need to edit the highlighted attribute of the property named **PublishingAssociatedContentType** so that it matches the content type name and ID you constructed previously. **TAKE CARE TO ENSURE THAT THE ";" DELIMITER REMAINS IN TACT. IT OCCURS THREE TIMES, AS IN ";#name;#id;#"**

5. Open the **feature.xml** file for editing and insert the following highlighted lines.

```

<Feature xmlns="http://schemas.microsoft.com/sharepoint/"
Id="fcafc2f0-2a2d-42b7-bb6f-3a83a8f90819"
Hidden="FALSE"
Title="ECM401 - Publishing Feature"
Description="This feature creates custom publishing resources."
Scope="Site"
ReceiverAssembly="ECM401.Publishing, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=6dc3927da2abcbba"
ReceiverClass="ECM401.Publishing.FeatureReceiver"
Version="1.0.0.0">

  <ElementManifests>
    <ElementManifest Location="SiteColumns.xml"/>
    <ElementManifest Location="ContentTypes.xml"/>
    <ElementManifest Location="PageLayouts.xml"/>
  </ElementManifests>

</Feature>

```

6. Now you are ready to build the project and deploy the solution. Select the **DebugDeploy** configuration and build the project, then navigate to the **Litware Publishing** site to activate the feature. Check the **Site Column Gallery** to ensure that the new site columns were created and then check the **Content Type Gallery** to ensure that the new content type was created. Finally, create a new page using the new page layout.

Note: You have successfully created and deployed custom publishing resources using a SharePoint Feature instead of SharePoint Designer.

This concludes the lab exercises.