Introduction to SharePoint Workflows

Developing SharePoint Workflow Templates with Visual Studio

Agenda

- What is a Reactive Program?
- Why is workflow beneficial
- Introduction to Windows Workflow Foundation
- Introduction to Workflow in SharePoint

Reactive Programming

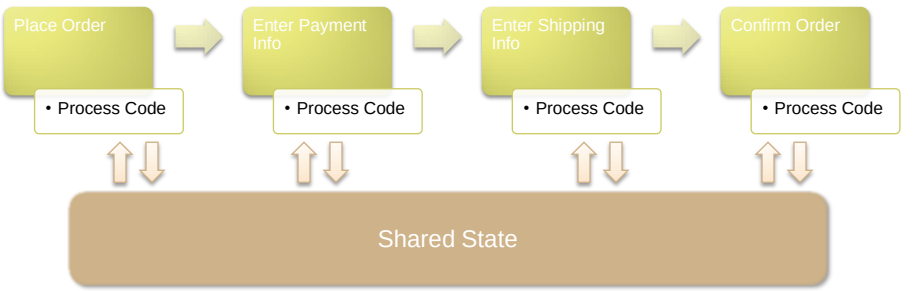
- Processes requiring human interaction
  - Process defined centrally
  - Asynchronously interact with other systems

```
Users
Web Services
Other external services
```

Scenario – Online Order Pipeline

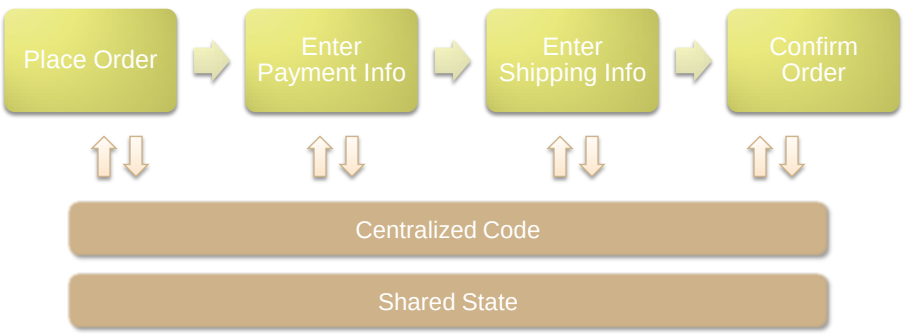
Where does the code live?

- Each transition's code exists in step
  - Decentralized process



A Centralized Model

- Centralize code in central controller
  - Decouples interface from process

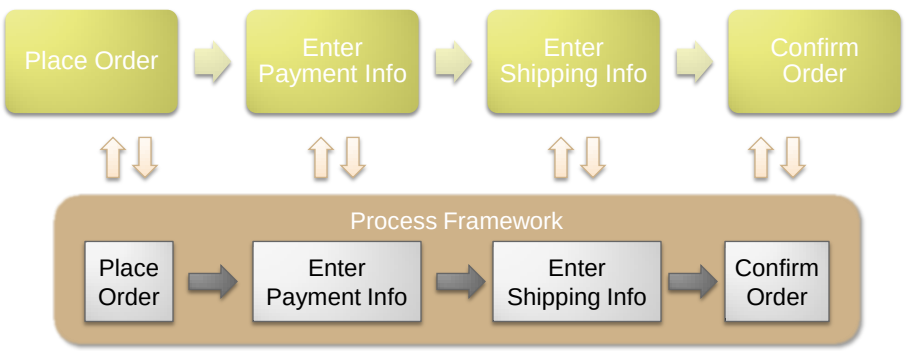


Common Process Functionality

- Centralized process leads to common behaviors
  - Track the state of the process
  - Move between “states”
  - Communicate with the outside world

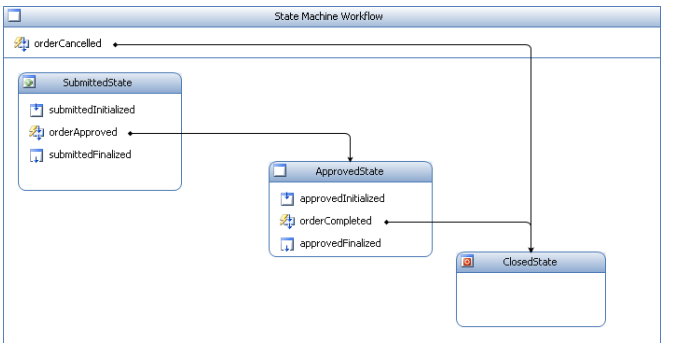
A Process Management Framework

- Decoupling allows generic process framework
  - Stores state as .NET objects
  - Defines process as an interconnected set of steps
  - Define a well known interface for communication



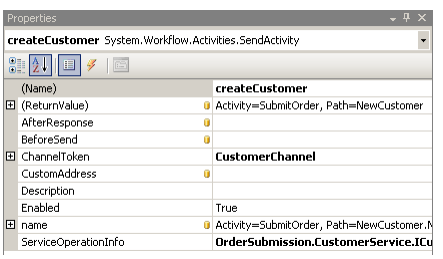
Windows Workflow Foundation

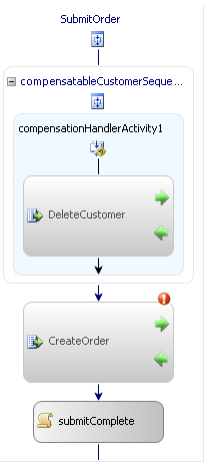
- Provides processing framework
- Shipped with .NET 3.0
  - Design environment in VS2005, VS2008



What is WinWF?

- Workflows are made up of Activities
  - Processes = Workflows
  - Steps = Activities





Hosting the WinWF Runtime

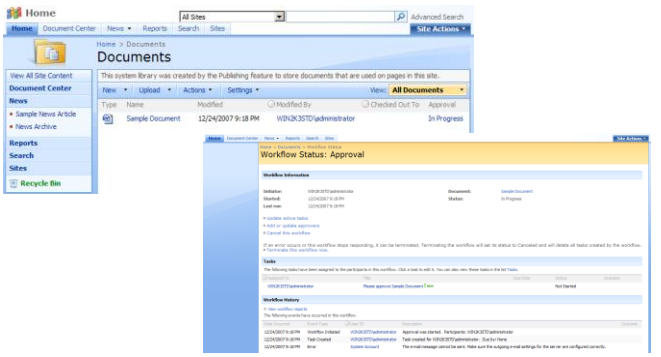
- Can be hosted in any application
  - Windows Forms
  - ASP.NET
  - SharePoint
- Host/Framework interaction via services
  - ExternalDataExchange
  - Persistence
  - Threading
  - Etc...

SharePoint Integration

- SharePoint is another WinWF Host
  - SharePoint provides it's own services
  - Can execute any WinWF workflow
- Ships with several built in workflows
  - Three State
  - Approval – MOSS only
  - Translation – MOSS only

Human Interaction in SharePoint

- Human interaction defined via Tasks
  - Assigned to users
  - Workflow “pauses” while waiting for task completion
  - Think of SharePoint as UI for WinWF



Student Questionnaire

- What’s Your Name?
- What Company are you with?
- How have you evolved as a Developer?
- Do you have experience with…
  - The .NET Framework and Visual Studio
  - WSS 2.0 and SPS 2003
  - WSS 3.0 and MOSS
  - WinWF or BizTalk

Day 1

- Introduction to SharePoint Workflows
- Windows Workflow Foundation Primer
- Understanding the Windows Workflow Foundation Runtime

Day 2

- Windows Workflow Foundation Integration with SharePoint 2007
- Programming with SharePoint Workflow API
- Developing SharePoint Workflow Templates with Visual Studio 2008

Day 3

- Creating and Waiting on SharePoint Tasks
- Creating Workflow Association Forms
- Creating Workflow Instantiation and Modification Forms

Day 4

- Integrating InfoPath Forms into a SharePoint Workflow
- Developing Custom Activities for a SharePoint Workflow
- Extending SharePoint Designer with Custom Activities

Summary

- What is a Reactive Program?
- Why is workflow beneficial
- Introduction to Windows Workflow Foundation
- Introduction to Workflow in SharePoint

Windows Workflow Foundation Integration with SharePoint 2007

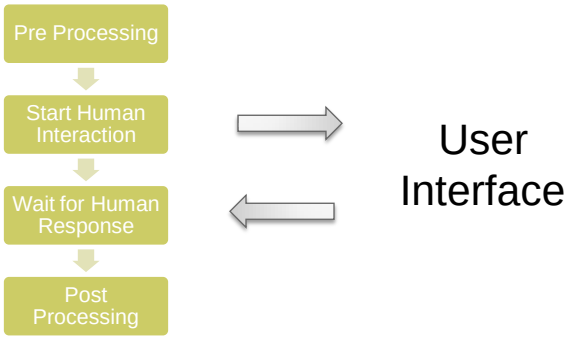
Developing SharePoint Workflow Templates with Visual Studio

Agenda

- WSS as a Workflow UI
- Association Workflow to Lists and Content Types
- Executing Workflow in WSS
- WSS/WinWF Communication
- Integration with Office 2007

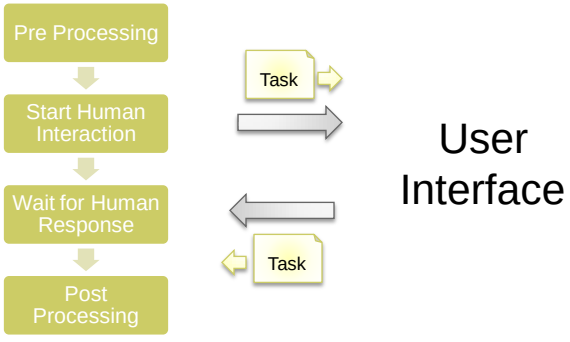
Workflow UI

- Human workflow requires process and UI
  - WinWF provides process, but no UI
  - WinWF provides external communication mechanism
  - Traditionally it's the developers job to provide UI



Workflow UI Patterns

- UI in human workflow centers around tasks
  - Process defines tasks and waits for result
  - Humans complete tasks and provide result
  - Process continues to execute

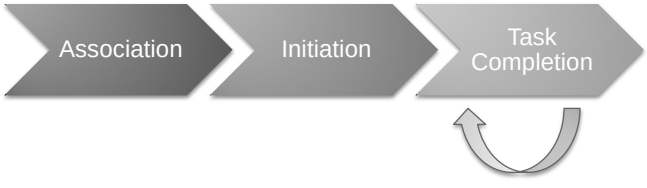


WSS as Workflow UI

- Tasks are very similar to list items
  - In fact WSS 3.0 has a list type called tasks
  - Could the tasks list be our Workflow UI?
  - Can the tasks list hold multiple types of tasks with custom UIs?
- What useful functionality do lists have in WSS
  - Custom display and edit forms
  - Potential for multiple content types with custom forms
  - Filtering to allow display of a single type of task

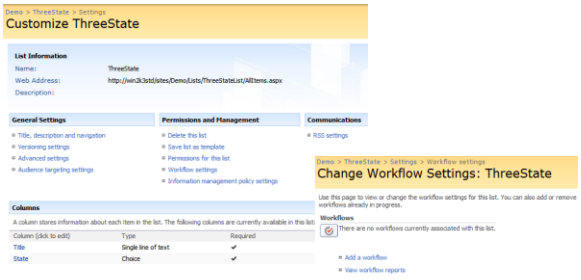
Using Workflow in WSS

- There are three primary interaction points
  - Workflow Association
  - Workflow Initiation
  - Task Completion

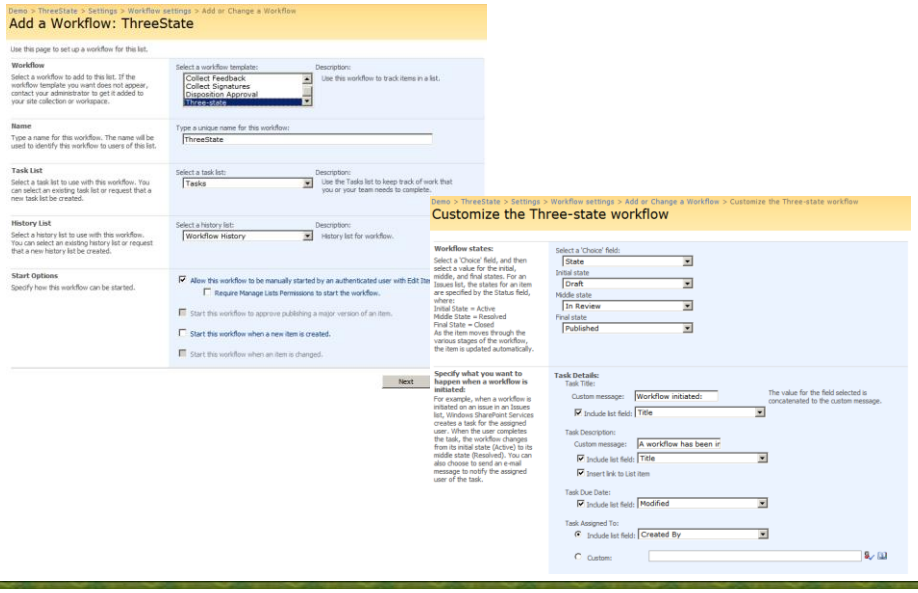


WSS Workflow Association

- Workflows must be associated with containers
  - Can be associated with lists or content types
  - Association ties a type of workflow to a set of items
  - Requires entry of workflow type specific data

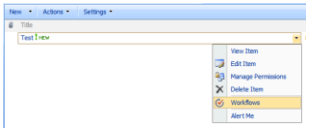


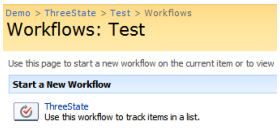
WSS Workflow Association



WSS Workflow Initiation

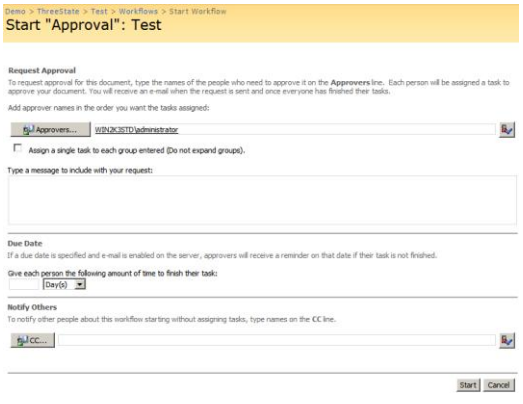
- Initiation occurs at the list item level
  - Workflow types associated with its container allowed
    `Ex. WF Types associated with content type or list`
  - Initiation can be automatic (when item created edited)
  - Initiation can be manual (anyone or only editors)





WSS Workflow Initiation UI

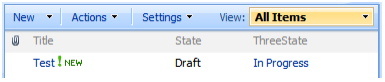
- When a workflow instance is initiated
  - User has a chance to override instance settings
  - Done using UI specified by workflow type

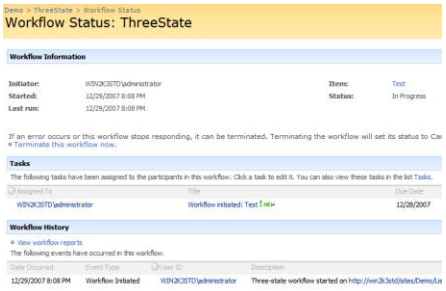


WSS Workflow Status

- Once workflow is running, status accessed via list
  - List column added showing WF instance current state
  - Links to a special status page showing

    `Tasks related to workflow instance`
    `History information logged by workflow instance`



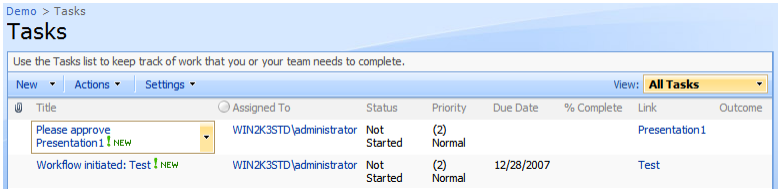


WSS Workflow Modification

- What if a workflow needs to be changed?
  - The list of approvers changes
  - The approval process becomes more urgent?

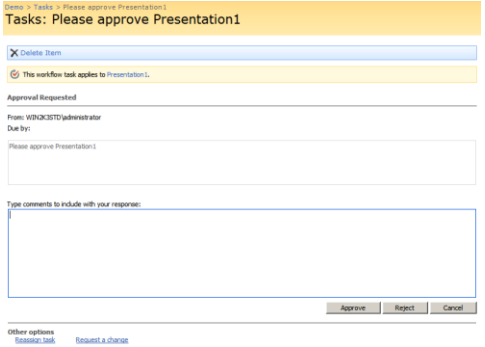
Using WSS Tasks

- Tasks are the key WF/UI interaction point
  - Process defines tasks that the user can complete
  - Tasks are stored in a standard list with custom forms
  - The status UI displays a filtered view of the tasks



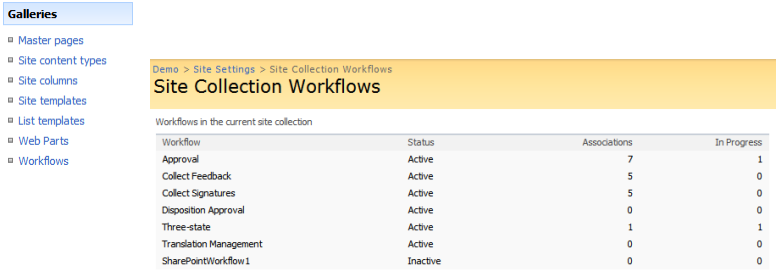
Completing WSS Tasks

- Content types allow multiple forms per list
  - Each workflow can have its own task content type
  - Each content type has it's own custom forms
  - Custom forms have 1 to 1 relationship with aspx pages



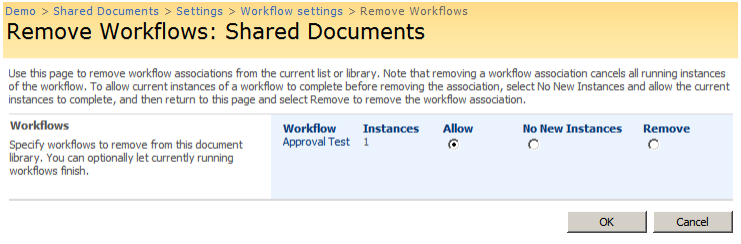
Viewing Current Workflow Tasks

- Report page displays workflow types
  - Located in site settings at site collection level
  - Displays list of all workflow types
  - Displays number of associations and instances



Un-associating a Workflow Type

- Removal of a workflow type done in three steps
  - Remove the ability to start new instances
  - Wait until all instances are complete
  - Remove the workflow association
- It is possible to terminate current instances

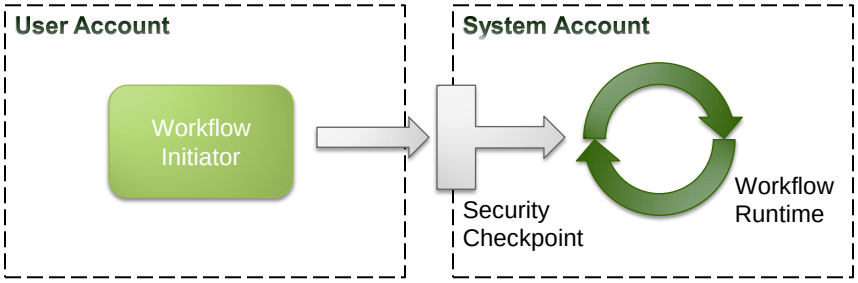


WSS Workflow Security

- Workflow runs under worker process identity
  - Instances may have more permissions than initiator

    `Initiator may also be an event, not a user`
  - Security is necessary on the workflow initiation

    `Tight controls on who can start a workflow`



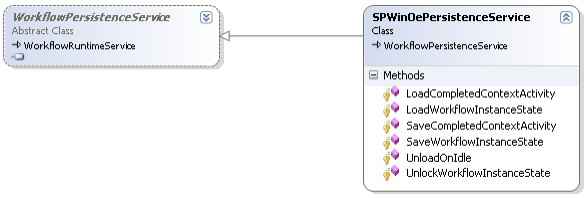
How does this all work?

- WSS integrates with WinWF two ways
  - WSS hosts the WorkflowRuntime class
  - WSS provides custom services for the runtime

    `SPWinOePersistenceService`
    `SPWinOETaskService`
    `SPWinOEWSSService`

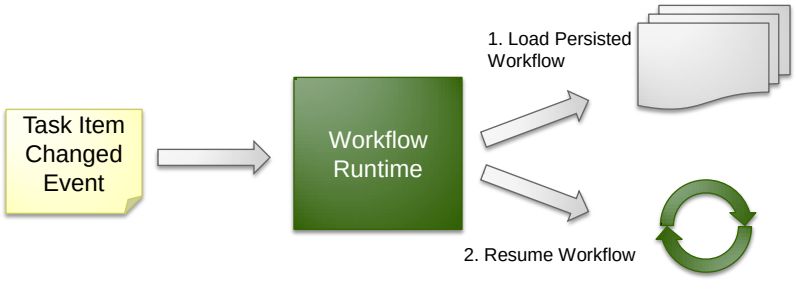
Dehydration of workflow instances

- Workflow persisted while waiting on tasks
  - Running workflow is serialized into content database
  - Persistence service chosen in code
  - Can't be overridden in config file



Rehydration of workflow instances

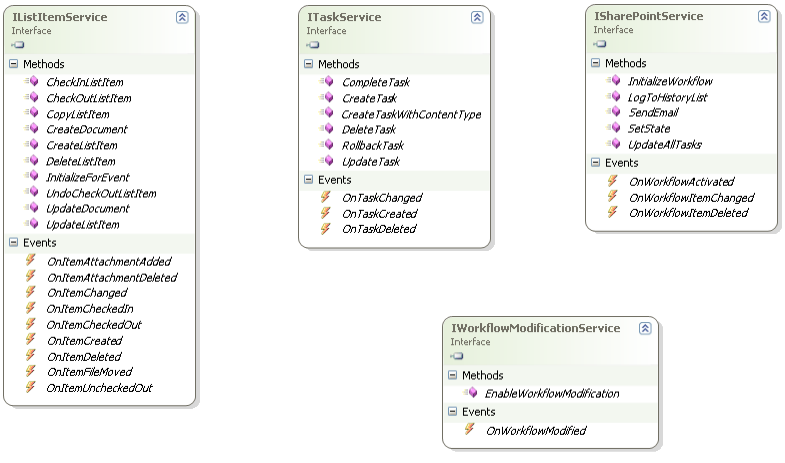
- Workflow deserialized when tasks change
  - Tasks are just list items
  - SPItemEventReceiver fires the onUpdated event
  - Workflow event receivers deserialize and start workflow



How are events delivered?

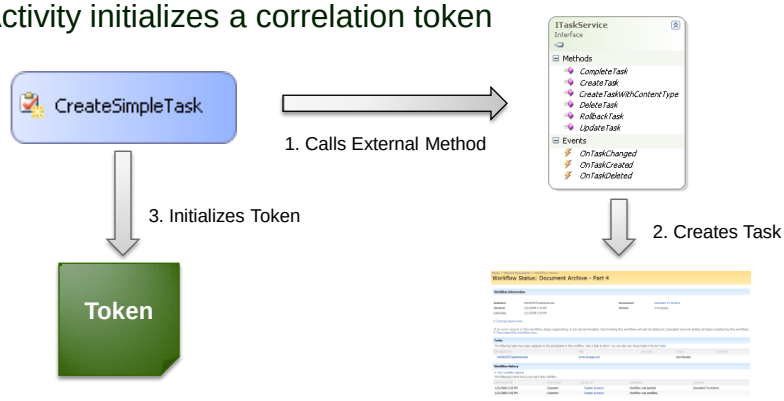
- Done using ExternalDataExchange
  - WSS Defines a set of ExternalDataExchange interfaces
  - Implements the interfaces in special service
  - WSS provides a set of activities to use interfaces

ExternalDataExchange Interfaces



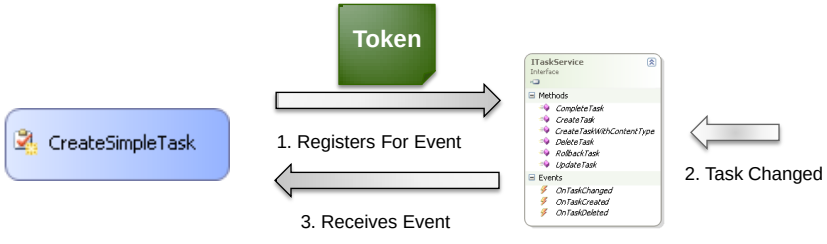
Task Creation Scenario

- Workflow instance uses CreateTask activity
  - Workflow calls method in ITaskService interface
  - Task is created in the Tasks list
  - Activity initializes a correlation token



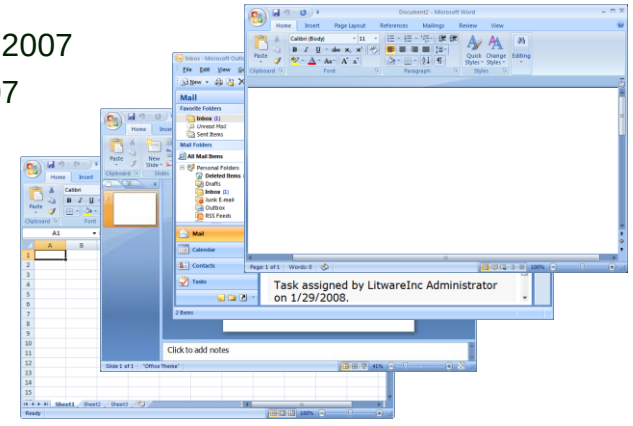
Task Completion Scenario

- WSS Event handler sends event to workflow
  - Event is converted into an ITaskService event
  - Message is correlated to a previous CreateTask call
  - Event is routed back to the correct activity and workflow



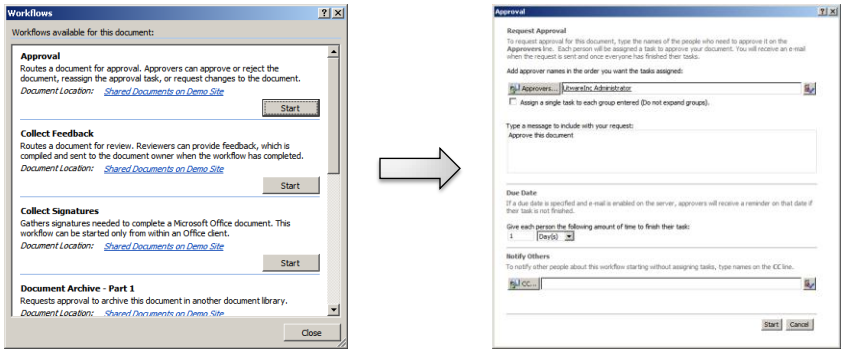
Office 2007 Integration

- Office Applications that support workflow
  - Word 2007
  - Excel 2007
  - PowerPoint 2007
  - Outlook 2007



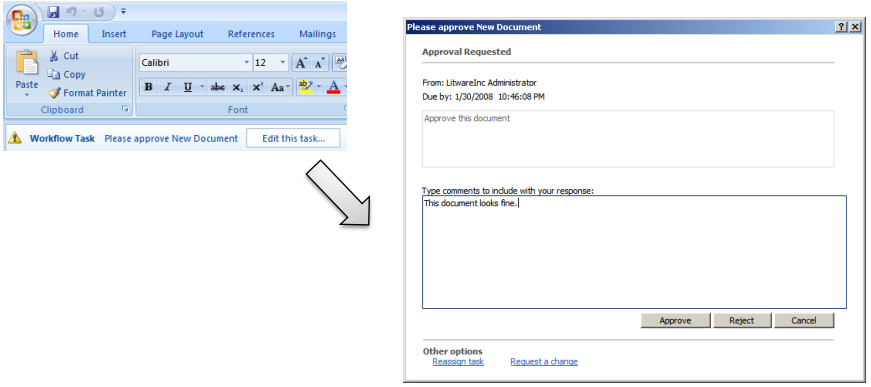
Starting Workflow via Office 2007

- Office applications allow workflow to be started
  - Can only start workflows previous associated
  - Office hosts initiation page in embedded browser



Acting on Tasks via Office 2007

- Office can find any tasks for open item
  - Display notification for the user
  - Can host task edit form in embedded host



Receiving Tasks via Outlook 2007

- Workflow status received via Email
  - Tasks sent as an email to all recipients
  - Hosts task edit form in embedded host

Summary

- WSS as a Workflow UI
- Association Workflow to Lists and Content Types
- Executing Workflow in WSS
- WSS/WinWF Communication
- Integration with Office 2007

Programming with SharePoint Workflow API

Developing SharePoint Workflow Templates with Visual Studio

Agenda

- Overview of Workflow API
- Associating workflow templates
- Initiating workflows
- Accessing running instances using the API

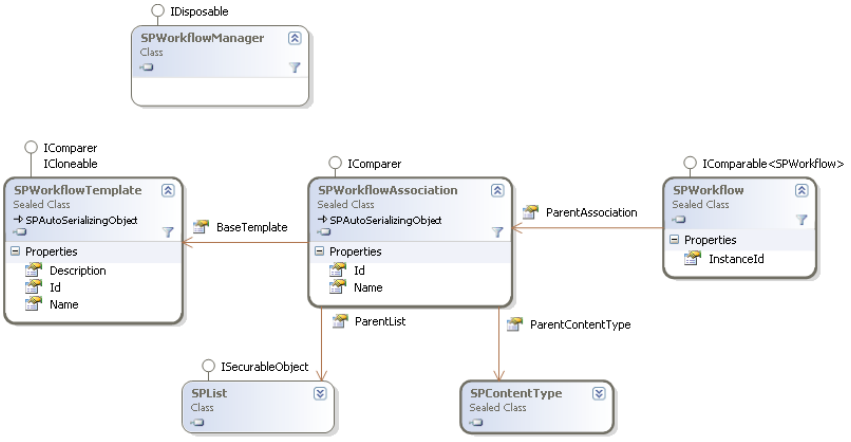
Why do we need Worflow APIs?

- SharePoint allows custom workflows and forms
  - This means developers must create custom UIs
  - Custom UIs require access to workflow concepts
- APIs used by SharePoint UIs are available to developers

Managing Workflow via Code

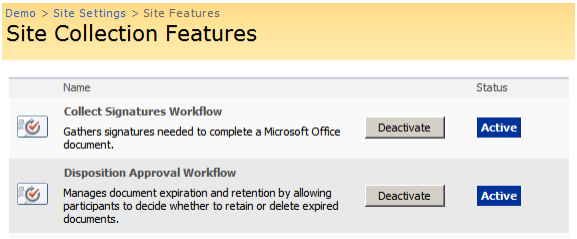
- SharePoint's workflow UIs all use standard API
  - Necessary to allow third party extensions to workflows
  - Used in custom management forms and web parts
  - Used in custom workflow forms
- These APIs allow:
  - Manual workflow template association
  - Manual workflow initiation
  - Manual workflow status
  - Manual workflow modification

Workflow API Objects



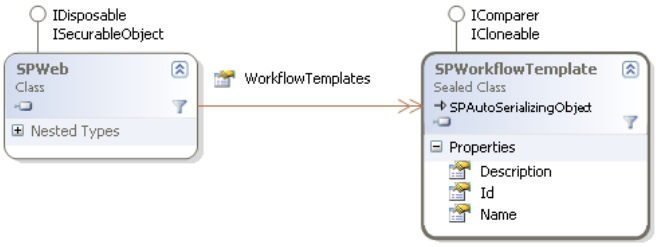
Workflow Templates

- Define a type of workflow with no relationships
  - Installed via features and associated with site collection
  - Used as the template when associating workflows
  - Can be attached to lists or content types



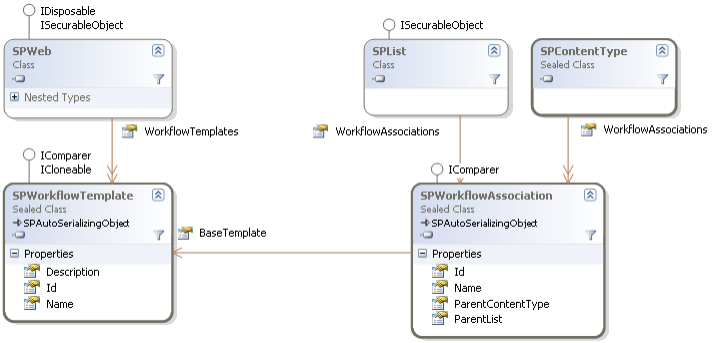
Workflow Templates

- Can be referenced from the SPSite object
  - Can be listed using SPWeb.WorkflowTemplates
  - Can be filtered using SPWorkflowManager
  - No ability to add or remove



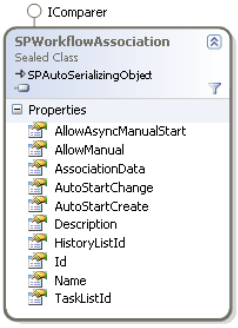
Associating Workflow Templates

- Workflows can't be started until associated
  - Association ties SPWorkflowTemplate to container
  - Association connects SPList or SPContentType
  - Holds association specific data



SPWorkflowAssociation

- Association defined by SPWorkflowAssociation
  - Defines workflow association name
  - Defines startup conditions
    `Auto/Manual Startup`
  - Defines association data
  - Defines the tasks and history lists



Association Data

- Associated Workflows need parameters
  - Differentiates one association to another
    `Ex. Who is the default approver for this process?`
  - Set using a custom string called association data
    `Often formatted as XML`
  - SPWorkflowAssociation.AssociationData stores string

Association Data Example

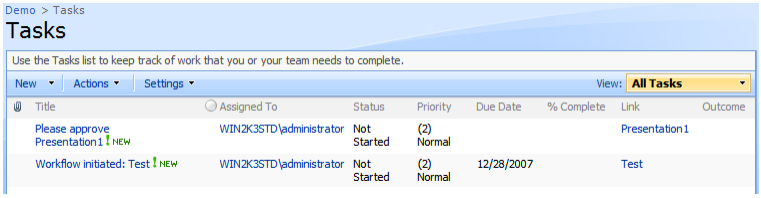
```
<my:myFields xml:lang="en-us" xmlns:xsi="…" xmlns:my="…">
  <my:Reviewers>
    <my:Person>
      <my:DisplayName></my:DisplayName>
      <my:AccountId></my:AccountId>
      <my:AccountType></my:AccountType>
    </my:Person>
  </my:Reviewers>
  <my:CC></my:CC>
  <my:DueDate xsi:nil="true"></my:DueDate>
  <my:Description></my:Description>
  <my:Title></my:Title>
  <my:DefaultTaskType>1</my:DefaultTaskType>
  <my:CreateTasksInSerial>true</my:CreateTasksInSerial>
  <my:AllowDelegation>true</my:AllowDelegation>
  <my:AllowChangeRequests>true</my:AllowChangeRequests>
  <my:StopOnAnyReject xsi:nil="true"></my:StopOnAnyReject>
  <my:WantedTasks xsi:nil="true"></my:WantedTasks>
  <my:SetMetadataOnSuccess>false</my:SetMetadataOnSuccess>
  <my:MetadataSuccessField></my:MetadataSuccessField>
  <my:MetadataSuccessValue></my:MetadataSuccessValue>
  <my:ApproveWhenComplete>false</my:ApproveWhenComplete>
  <my:TimePerTaskVal xsi:nil="true"></my:TimePerTaskVal>
  <my:TimePerTaskType xsi:nil="true"></my:TimePerTaskType>
  <my:Voting>false</my:Voting>
  <my:MetadataTriggerField></my:MetadataTriggerField>
  <my:MetadataTriggerValue></my:MetadataTriggerValue>
  <my:InitLock>false</my:InitLock>
  <my:MetadataStop>false</my:MetadataStop>
  <my:ItemChangeStop>false</my:ItemChangeStop>
  <my:GroupTasks>false</my:GroupTasks>
</my:myFields>
```

Tasks and History lists

- Running workflow needs to store information
  - Tasks to define user interaction
  - History to log key events
  - All stored in special SharePoint lists

Task Lists

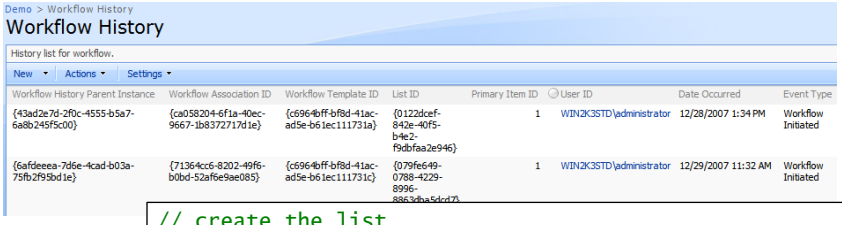
- List defined using the Tasks list template
  - List type of SPListTemplateType.Tasks
  - List template Id of 107




```
// create the list
Guid listId = m_web.Lists.Add(
    name, string.Empty, SPListTemplateType.Tasks);

// commit the changes
m_web.Update();
```

History Lists

- List defined using the WorkflowHistory template
  - List type of SPListTemplateType.WorkflowHistory
  - List template Id of 140




```
// create the list
Guid listId = m_web.Lists.Add(
    name, string.Empty, SPListTemplateType. WorkflowHistory);

// commit the changes
m_web.Update();
```

Creating Workflow Associations

- Done using SPList.AddWorkflowAssociation
  - Verify tasks and history lists are available
  - Find the workflow template to use
  - Create a new SPWorkflowAssociation

    Optionally assign new association data

```
// create the association
SPWorkflowAssociation result =
    SPWorkflowAssociation.CreateListAssociation(template,
txtName.Text,
        taskList, historyList);
result.AssociationData = associationData;

// add the association to the list
list.AddWorkflowAssociation(result);
list.Update();
```

Starting workflow instances

- All running workflows are tied to a list item
  - Workflows available tied to item's list or content type
  - Start using SPWorkflowManager.StartWorkflow

    Requires list item, association, and init data

```
SPListItem listItem = ...;
SPWorkflowAssociation association =
    list.WorkflowAssociations.
        GetAssociationByName("Approval",
            CultureInfo.CurrentCulture);
SPWorkflow workflow =
    site.WorkflowManager.StartWorkflow(
        listItem, association, initiationData);
```

Initiation Data

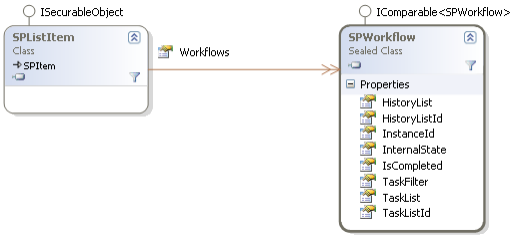
- Workflow instances require parameters as well
  - Just like association data in concept
  - Custom string passed directly to the workflow
    `Often formatted XML`

Initiation Data Example

```
<my:myFields xml:lang="en-us" xmlns:xsi="…" xmlns:my="…">
  <my:Reviewers>
    <my:Person>
      <my:DisplayName>WIN2K3STD\administrator</my:DisplayName>
      <my:AccountId>WIN2K3STD\administrator</my:AccountId>
      <my:AccountType>User</my:AccountType>
    </my:Person>
  </my:Reviewers>
  <my:CC></my:CC>
  <my:DueDate xsi:nil="true"></my:DueDate>
  <my:Description></my:Description>
  <my:Title></my:Title>
  <my:DefaultTaskType>1</my:DefaultTaskType>
  <my:CreateTasksInSerial>true</my:CreateTasksInSerial>
  <my:AllowDelegation>true</my:AllowDelegation>
  <my:AllowChangeRequests>true</my:AllowChangeRequests>
  <my:StopOnAnyReject xsi:nil="true"></my:StopOnAnyReject>
  <my:WantedTasks xsi:nil="true"></my:WantedTasks>
  <my:SetMetadataOnSuccess>false</my:SetMetadataOnSuccess>
  <my:MetadataSuccessField></my:MetadataSuccessField>
  <my:MetadataSuccessValue></my:MetadataSuccessValue>
  <my:ApproveWhenComplete>false</my:ApproveWhenComplete>
  <my:TimePerTaskVal xsi:nil="true"></my:TimePerTaskVal>
  <my:TimePerTaskType xsi:nil="true"></my:TimePerTaskType>
  <my:Voting>false</my:Voting>
  <my:MetadataTriggerField></my:MetadataTriggerField>
  <my:MetadataTriggerValue></my:MetadataTriggerValue>
  <my:InitLock>false</my:InitLock>
  <my:MetadataStop>false</my:MetadataStop>
  <my:ItemChangeStop>false</my:ItemChangeStop>
  <my:GroupTasks>false</my:GroupTasks>
</my:myFields>
```

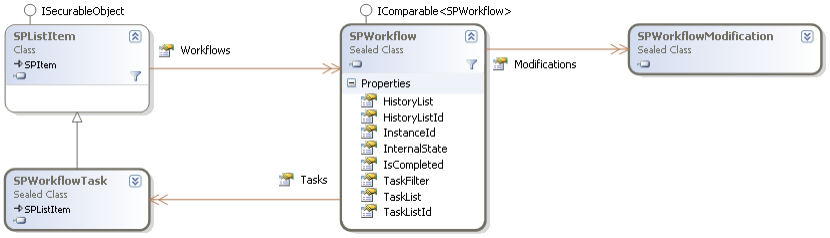
SPWorkflow Object

- Running workflows represented by SPWorkflow
  - Can be found in SPListItem.Workflows list
  - Can also be found using SPWorkflowManager
  - Provides access to state, tasks, and other details



Workflow Status

- Once the workflow is running, status is available
  - SPWorkflow object allows status access
  - Can provide basic workflow stats
  - Can provide a list of tasks and history events
  - Can provide access to available modifications

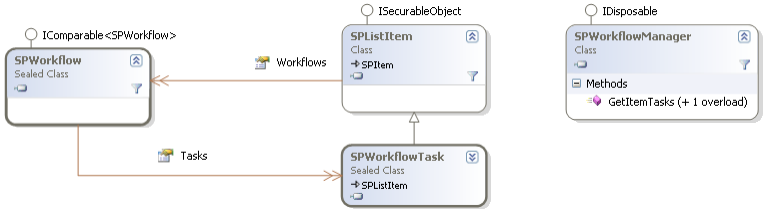


Current Tasks

- Tasks related to the workflow are provided
  - Accessible using tasks list and filter
  - List of tasks available via

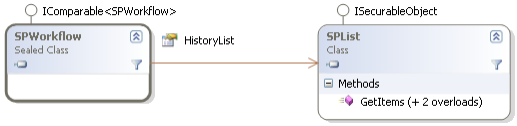
    `SPWorkflow.Tasks`

    `SPWorkflowManager.GetItemTasks`



Current History

- History related to the workflow is provided
  - No simple property that filters the history list
  - Accessible using history list and query




```
<Where>
  <Eq>
    <FieldRef Name="WorkflowInstance“ />
    <Value Type="Text">
     {00000000-0000-0000-0000-000000000000}
    </Value>
  </Eq>
</Where>
```

Current History Query

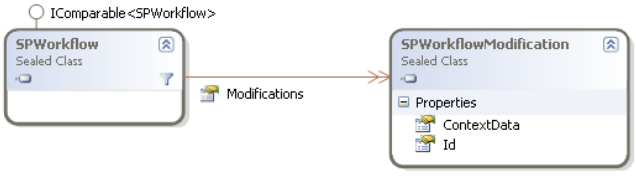
- Get history for workflow using SPList.GetItems

```
SPQuery query = new SPQuery();
query.Query =
    "<Where>" +
      "<Eq>" +
        "<FieldRef Name=\"WorkflowInstance\" />" +
        "<Value Type=\"Text\">" +
          workflow.InstanceId.ToString() +
        "</Value> " +
      "</Eq>" +
    "</Where>";
SPListItemCollection history =
    workflow.HistoryList.GetItems(query);
```

Workflow Modifications

- Available modifications are stored in SPWorkflow
  - Accessed via SPWorkflow.Modifications
  - Stored in SPWorkflowModification class

    Only contains ID and ContextData



Displaying Available Modifications

- SPWorkflowModification only contains an Id
  - Id represents a predefined modification point
  - Metadata describing modification stored in template
  - Accessed using specially formatted property names

    `Name – Modification_`*`ModID`*`_Name`

```
SPWorkflow workflow = …;
SPWorkflowTemplate template =
    workflow.ParentAssociation.BaseTemplate;

SPWorkflowModification modification;
foreach (modification in workflow.Modifications)
    Console.WriteLine(
        template[
            "Modification_" + modification.Id.ToString() + "_Name"]);
```

SPWorkflowModification ContextData

- Once a modification is made changes are stored
  - SPWorkflowModification.ContextData stores changes
  - Up to the form implementer to store changes
  - Up to the workflow implementer to use changes

Updating Workflow Associations

- Done using UpdateWorkflowAssociation method
  - Available in SPList and SPContentType

```
SPWorkflowAssociation association = …;

// update the association
association.ParentList.
  UpdateWorkflowAssociation(association);
```

Removing Workflow Association

- Done using RemoveWorkflowAssociation method
  - Available in SPList and SPContentType

```
SPWorkflowAssociation association = …;

// remove the association
association.ParentList.RemoveWorkflowAssociation(association);
association.ParentList.Update();
```

Summary

- Overview of Workflow API
- Associating workflow templates
- Initiating workflows
- Accessing running instances using the API

Developing SharePoint Workflow Templates with Visual Studio 2008

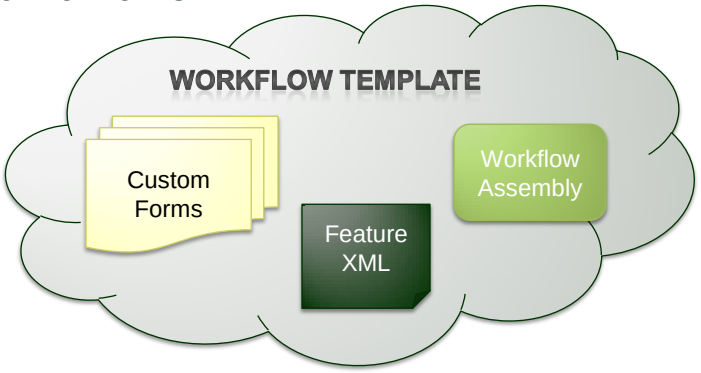
Developing SharePoint Workflow Templates with Visual Studio

Agenda

- Create a simple workflow in VS 2008
  - Installing workflow using features
- Pass startup parameters to workflows
- Build state machine workflows in WSS
- Handle item change events in workflow

WSS Workflow Templates

- Made up of several parts
  - Workflow assembly
  - Feature definition
  - Workflow forms



What makes up a workflow assembly?

- Workflow assembly contains all used by workflow
  - Workflow classes
  - Feature activation handlers
  - Code behind for forms
  - Any other supporting code
- Workflow assembly must be deployed in GAC

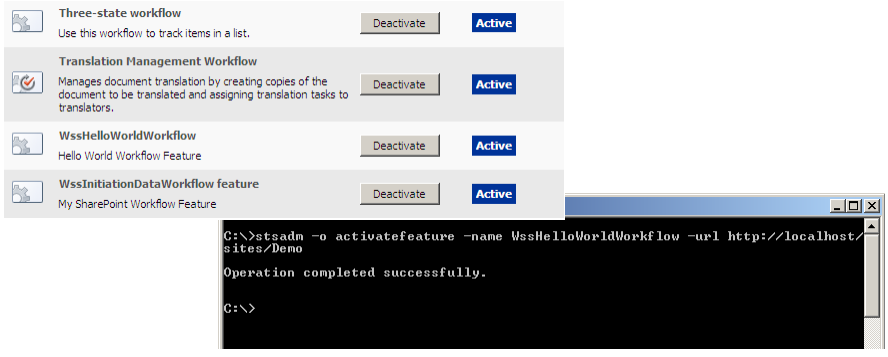
Packaging Workflow Templates

- Feature definition packages assembly and forms
  - Feature scoped at Site level
  - Workflow element ties a name to workflow class
  - Also Defines any form overrides

```
<Workflow
   Name="Hello World Workflow"
   Description="My Hello World Workflow"
   Id="59dd0352-c8be-4e08-9e0e-3c6a91fa18ec"
   CodeBesideClass="WssHelloWorldWorkflow.HelloWorld"
   CodeBesideAssembly="WssHelloWorldWorkflow, ...">
  <Categories />
  <MetaData />
</Workflow>
```

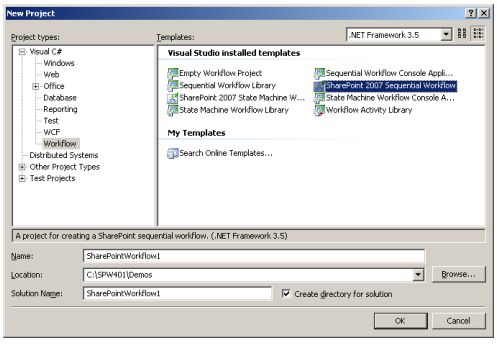
Installing Workflow Template

- Feature activation adds workflow templates
  - Activated via UI or stsadm
  - Represented as SPWorkflowTemplate
  - Related to SPWebs within the site feature activated in



Visual Studio 2008 WSS Workflow

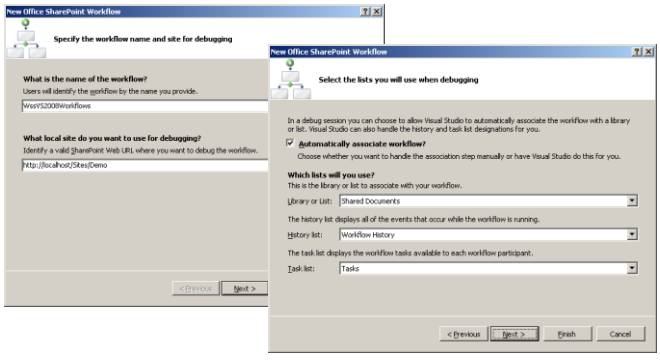
- Visual Studio 2008 adds WSS Workflow Project
  - Provides feature related files
  - Provides facilities to automatically deploy workflows
  - Automatically associates workflow to a list



Creating VS2008 Workflow Project

- Requires key information about workflow
  - Workflow name and site path required
  - Allows automatic association to an existing list

```
Requires list name, task list, history list
```



VS2008 Workflow Project

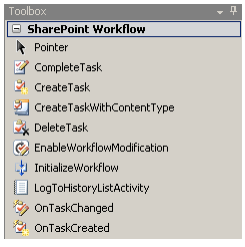
- Completed project is WSS Specific
  - Contains Feature.xml and Workflow.xml
  - Contains user properties in Project.csproj.user

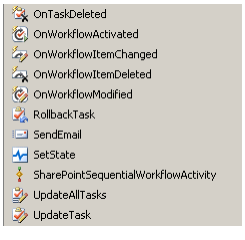
    Used to deploy and associate the workflow

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ProjectExtensions>
    <VisualStudio>
      <FlavorProperties GUID="{…}" xmlns="">
        <DisplayName>WSSHelloWorldWorkflow</DisplayName>
        <SiteURL>http://…/Docs</SiteURL>
        <ListURL>http://…/Docs/Documents/Forms/AllItems.aspx</ListURL>
        <TargetList Id="{…}">Documents</TargetList>
        <HistoryList Id="{…}">Workflow History</HistoryList>
        <TaskList Id="{…}">Tasks</TaskList>
        ...
      </FlavorProperties>
    </VisualStudio>
  </ProjectExtensions>
</Project>
```

SharePoint Workflow Activities

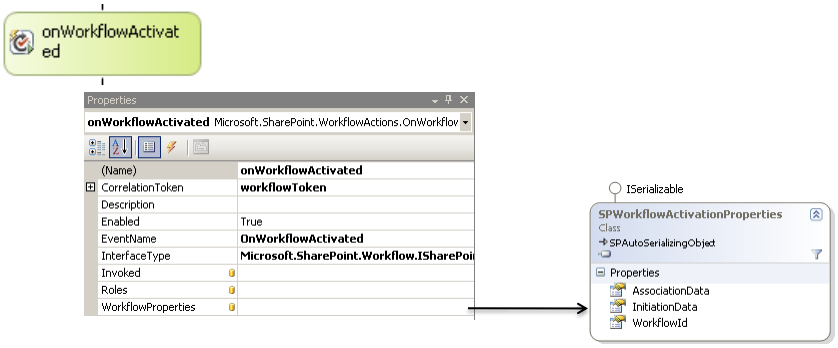
- Integration with SharePoint done via activities
  - Communicate with host via ExternalDataExchange
  - Focused around workflow, task, and workflow item





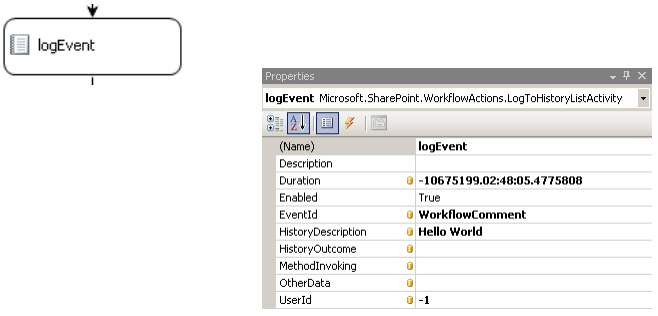
OnWorkflowActiviated Activity

- All SharePoint workflows start with this activity
  - Derived from HandleExternalEventActivity
  - Used to receive SPWorkflowActivationProperties



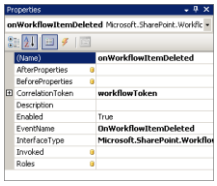
LogToHistoryList Activity

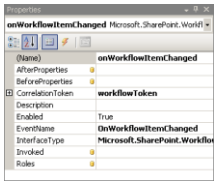
- Used to log status information to the host
  - Implemented like CallExternalMethod
  - Used to log an entry to the workflow's history list



Other Workflow Item Events

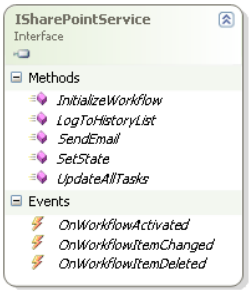
- Workflow can also wait for the item to change
  - OnWorkflowItemChanged received on item change
  - OnWorkflowItemDeleted received on item delete
  - Both contain a Hashtable of before and after properties





How Workflow Communication Works

- Communication done using ISharePointService
  - Implemented by SPWinOEWSSService
  - Provides events needed for workflow and item activities
  - Also provides methods for other communication



Passing Data to Workflow Instances

- Done using AssociationData and InitiationData
  - AssociationData assigned at workflow association
  - InitiationData assigned at workflow start
  - Both strings, often passed as xml

```
<my:myFields xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:my="http://schemas.microsoft.com/office/infopath/2003/myXSD">
  <my:Reviewers>
    <my:Person>
      <my:DisplayName>WIN2K3STD\administrator</my:DisplayName>
      <my:AccountId>WIN2K3STD\administrator</my:AccountId>
      <my:AccountType>User</my:AccountType>
    </my:Person>
  </my:Reviewers>
  <my:CC></my:CC>
  <my:DueDate xsi:nil="true"></my:DueDate>
  ...
</my:myFields>
```

Building InitiationData

- Often formatted using serialized classes
  - Define classes storing initiation data
  - Use XmlSerializer to convert classes to XML

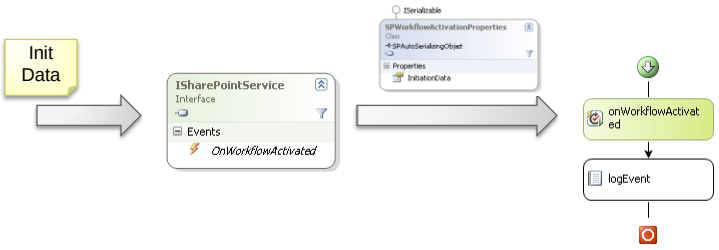
```
[Serializable]
public class StartupData
{
    public string Message { get; set; }
    public static string Serialize(StartupData value)
    {
        XmlSerializer serializer =
            new XmlSerializer(typeof(StartupData));
        using (StringWriter writer = new StringWriter())
        {
            serializer.Serialize(writer, value);
            return writer.ToString();
        }
    }
}
```

Sending InitiationData

- Sent using SPWorkflowManager.StartWorkflow
  - String representing serialized initiation data sent
  - Starts workflow instance and sends data via event

Receiving InitiationData

- Received via OnWorkflowActivated activity
  - First activity in workflow is OnWorkflowActivated
  - Receives event from the host containing initiation data



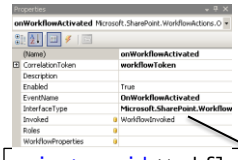
Deserializing InitiationData

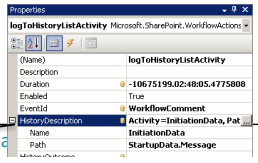
- InitiationData received as string and deserialized
  - String data deserialized via XmlSerializer
  - Converts string xml into classes

```
[Serializable]
public class StartupData
{
    public string Message { get; set; }

    public static StartupData Deserialize(string value)
    {
        XmlSerializer serializer =
            new XmlSerializer(typeof(StartupData));
        using (StringReader reader = new StringReader(value))
            return serializer.Deserialize(reader) as StartupData;
    }
}
```

Verifying InitiationData

- Write initiation parameters to the event log
  - Receive and deserialize initiation data
  - Write data to the event log

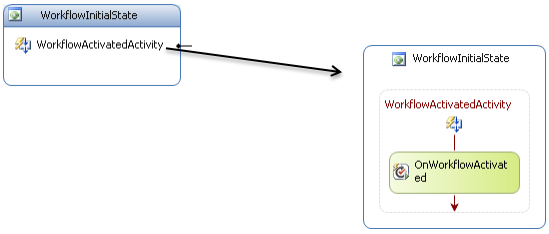




```
private void WorkflowInvoked(object sender, ExternalDa
    try {
        this.StartupData = StartupData.Deserialize(
            onWorkflowActivated.WorkflowProperties.InitiationData);
    }
    catch {
        StartupData = new StartupData();
        this.StartupData.Message = "No startup data provided.";
    }
}
```

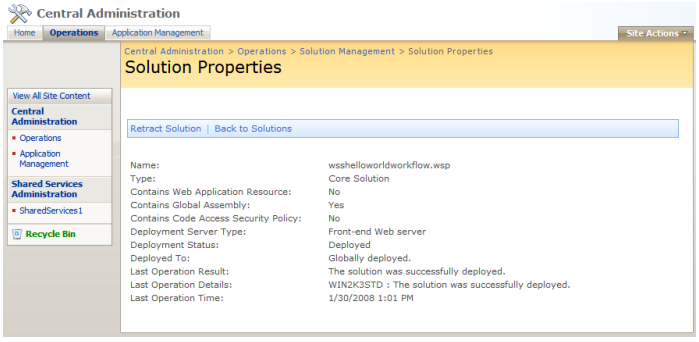
Creating WSS State Machines

- State Machines can be used in WSS
  - Same requirements as SequentialWorkflows exists
  - Workflows start with OnWorkflowActivated activity
  - Usually done with an EventDriven activity in start state



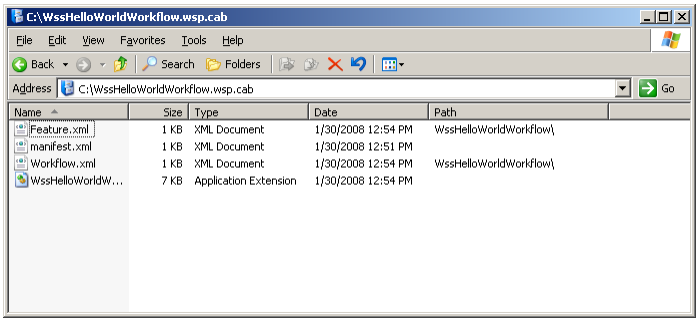
Deploying Workflows

- Features don't handle entire deployment
  - Need a way to copy files, register files in GAC
  - Solution packages are the solution
  - Installed using stsadm



Solution Package structure

- Solution packages are single .wsp files
  - Just renamed .cab files
  - Cab files contain all needed files and a manifest file
  - Manifest file contains instructions to install package



Workflow Solution Package

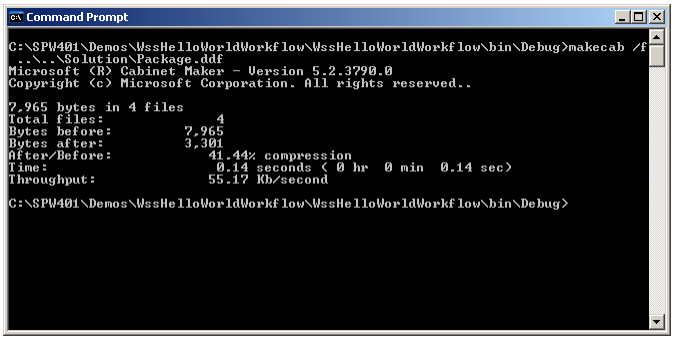
- Manifest contains references to all files
  - Installs DLL in the GAC
  - Copies .aspx files to Layouts folder
  - Copies features into Features folder
  - Installs features

Workflow Solution Manifest

```
<?xml version="1.0" encoding="utf-8" ?>
<Solution SolutionId="189c9a36-a1a1-4f76-a36c-fbba6cb99fdb"
          xmlns="http://schemas.microsoft.com/sharepoint/">
  <FeatureManifests>
    <FeatureManifest Location="WssHelloWorldWorkflow\Feature.xml" />
  </FeatureManifests>
  <Assemblies>
    <Assembly Location="WssHelloWorldWorkflow.dll“
              DeploymentTarget="GlobalAssemblyCache"/>
  </Assemblies>
</Solution>
```

Creating the solution Package

- Packaged into cab using makecab.exe
  - Requires DDF file
  - DDF file contains list of files to package
  - Each file contains a source and destination path



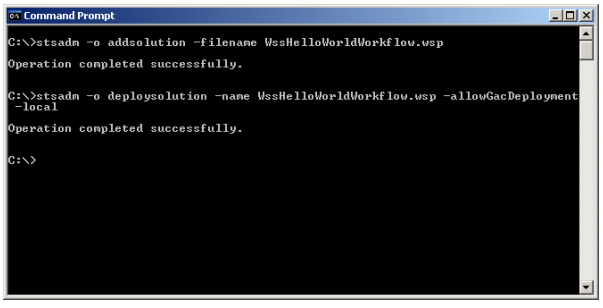
Workflow Solution DDF

```
.OPTION EXPLICIT     ; Generate errors
.Set CabinetNameTemplate=WssHelloWorldWorkflow.wsp
.set DiskDirectoryTemplate=CDROM ; All cabinets go in a single directory
.Set CompressionType=MSZIP;** All files are compressed in cabinet files
.Set UniqueFiles="ON"
.Set Cabinet=on
.Set DiskDirectory1=Package

..\..\Solution\Manifest.xml manifest.xml

..\..\Template\Features\WssHelloWorldWorkflow\Feature.xml
   WssHelloWorldWorkflow\Feature.xml
..\..\Template\Features\WssHelloWorldWorkflow\Workflow.xml
   WssHelloWorldWorkflow\Workflow.xml

WssHelloWorldWorkflow.dll WssHelloWorldWorkflow.dll
```

Installing the solution package

- Installation is made up of two steps
  - stsadm –o addsolution –filename Package.wsp
  - stsadm –o deploysolution –name Package.wsp

    `Use -allowgacdeployment to authorize GAC install`
    `Use -local or -immediate to deploy now`



Integrating Solution into Build

- Often integrated into post build steps
  - Post build runs makecab.exe
  - Deploy configuration runs stsadm to install and deploy
- Some open source packages under development
  - WSPBuilder - http://www.codeplex.com/wspbuilder
    `Generates solution based on files in folders`
  - WSP Proj - http://www.codeplex.com/wspprojecttemplate
    `Project template that generates manifest and ddf`
  - Stsdev - http://www.codeplex.com/stsdev
    `Generates simple VS Project files`

Summary

- Create a simple workflow in VS 2008
  - Installing workflow using features
- Pass startup parameters to workflows
- Build state machine workflows in WSS
- Handle item change events in workflow

Creating and Waiting on SharePoint Tasks

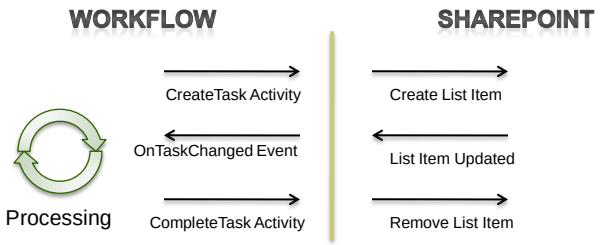
Developing SharePoint Workflow Templates with Visual Studio

Agenda

- Discuss workflow/task interaction
- Create a simple workflow using tasks
- Add a custom form a task

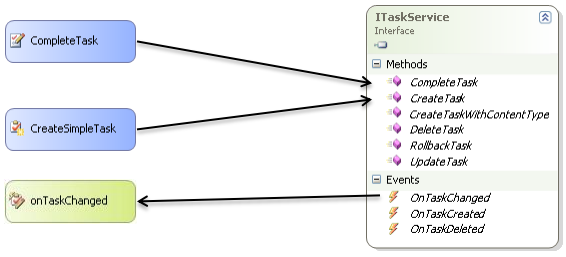
Task Responsibilities

- Tasks are SharePoint's way of user interaction
  - WSS responsible for managing tasks
  - Workflow responsible for passing data to tasks
  - Workflow responsible for reacting to task changes



Workflow/WSS Task Interaction

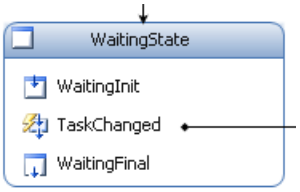
- WF\WSS interaction done via activities
  - CreateTask and CompleteTask used to manage tasks
  - OnTaskChanged used to wait for task changes
  - All methods use ITaskService interface



Common Task Scenario

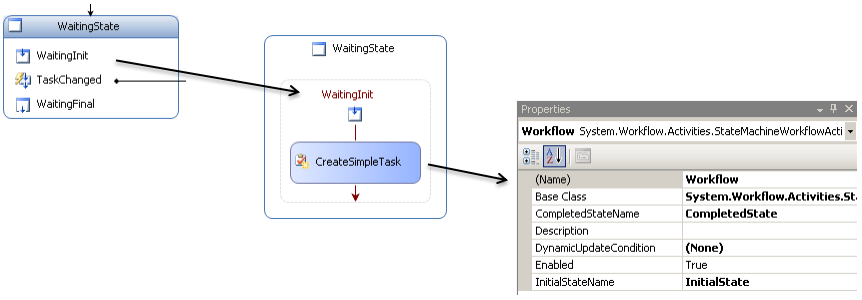
- Waiting State
  - State init creates a task
  - Waits for task changed event

    `Transitions out of state if task was completed`
  - State finalization completes or deletes a task



Creating the Task

- Done using CreateTask activity
  - Requires correlation token
  - Requires task ID
  - Requires task specific properties



CreateTask Correlation Token

- Correlation token defines activity relationships
  - Token initialized in CreateTask activity
  - Used later to wait or act on the task created
- Often scoped at the state
  - Can't be used outside the state
  - Allows state to be re-enterable

(Name)	**CreateSimpleTask**
⊟ CorrelationToken	**taskToken**
OwnerActivityName	**WaitingState**

CreateTask TaskId

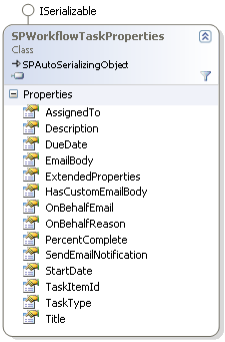
- TaskId required for task related activities
  - Stored as a Guid
  - Used to initialize the correlation token
- Often stored as a workflow property
  - Initialized prior to CreateTask execution
  - CreateTask activity bound to the property

⊟ TaskId	**Activity=Workflow, Path=Ta**
Name	**Workflow**
Path	**TaskId**

```
public Guid TaskId { get; set; }

this.TaskId = Guid.NewGuid();
```

CreateTask TaskProperties

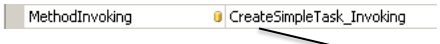
- Defines the details of an activity
  - Stored as SPWorkflowTaskProperties
  - Contains title, description, assigned to, etc…
  - Allows addition of extra data using ExtendedPropertes
  - Only needed while activity executes



Assigning TaskProperties Explicitly

- Storing task properties in workflow wasteful
  - Why not create them only when needed?
  - Can't we just assign the properties in constructor?

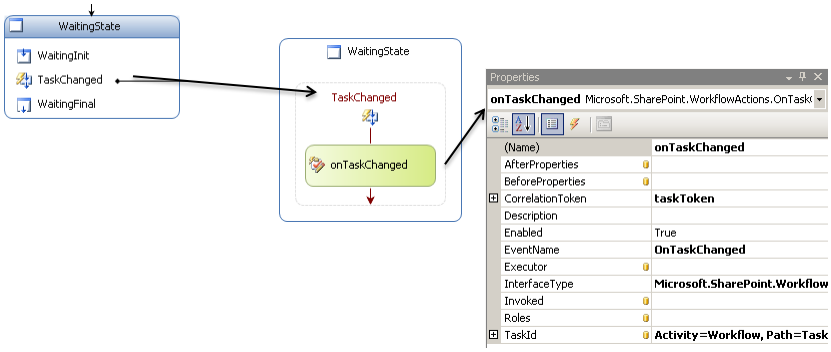
    Not always. Sometimes the activities are copied.
  - Solution; handle MethodInvoking event and use sender




```
private void CreateSimpleTask_Invoking(object sender, EventArgs e) {
  TaskId = Guid.NewGuid();

  CreateTask activity = sender as CreateTask;
  activity.TaskProperties = new SPWorkflowTaskProperties();
  activity.TaskProperties.AssignedTo = "Administrator";
  activity.TaskProperties.Title = "Simple Task";
  activity.TaskProperties.Description = "Complete this simple task.";
}
```

Waiting for task changes

- Done using OnTaskChanged event activity
  - Waits for ITaskService.OnTaskChanged event
  - Initialized using correlation token
  - Doesn't flag task as complete, that's up to the workflow



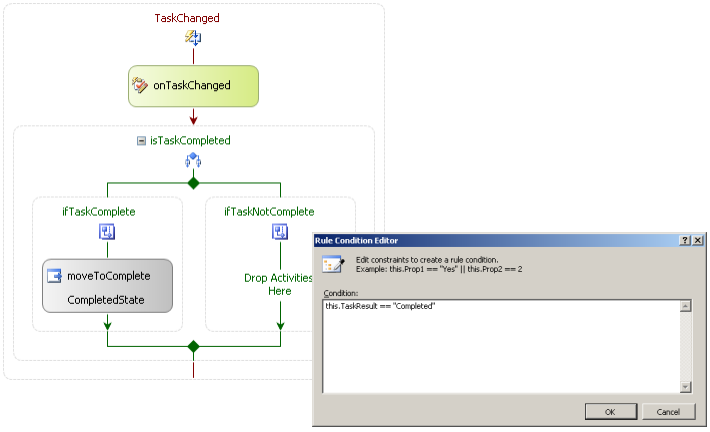
Accessing Task Properties

- Properties available in OnTaskChanged activity
  - BeforeProperties contain properties before changes
  - AfterProperties contain properties after changes
  - Both exposed as SPWorkflowTaskProperties
  - Can be bound or accessed directly using Invoked event


```
private void TaskChanged_Invoked(object sender,
  ExternalDataEventArgs e)
{
    SPTaskServiceEventArgs args = e as SPTaskServiceEventArgs;
    string state = args.afterProperties.ExtendedProperties["..."];

    /* Code operating on task state */
}
```

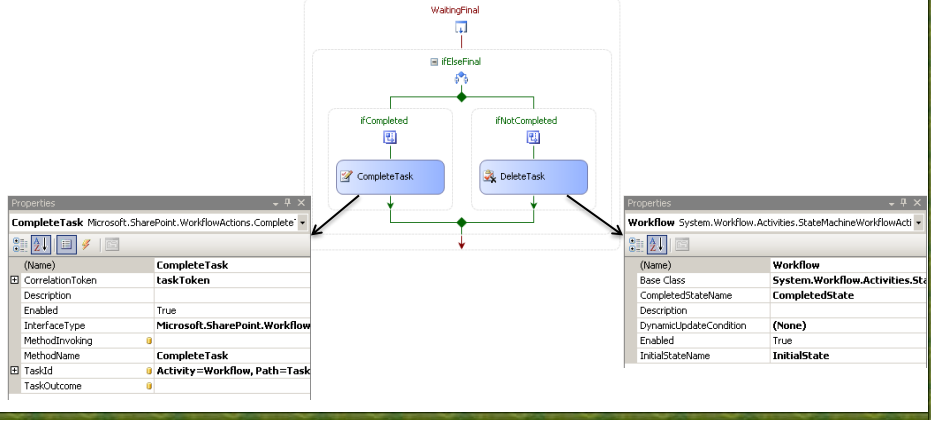
Using Task Properties

- Decisions made by workflow based on properties
  - "Status" property often used to determine next steps



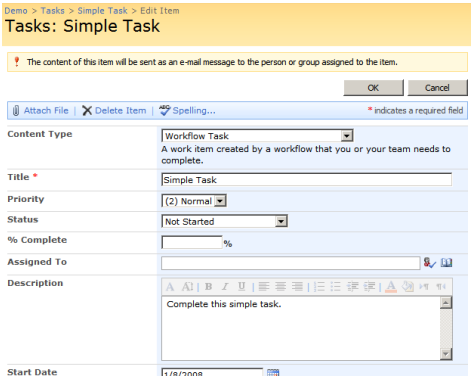
Completing/Deleting Tasks

- Done using CompleteTask or DeleteTask activity
  - Both require CorrelationToken and TaskId
  - Often used in state finalization activity



Custom Task Forms

- Default task forms aren't enough
  - Very generic, no task specific information
  - No way to use custom extended properties



Content Type Forms

- Content types used to differentiate task types
  - Each task created has a specific content type
  - Custom task content types have custom forms
  - Content type forms defined in feature
    `http://schemas.microsoft.com/sharepoint/v3/contenttype/forms/url`

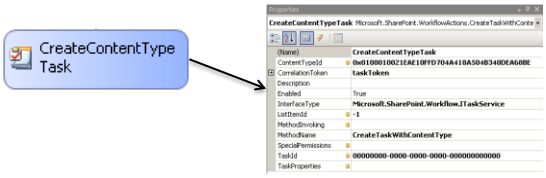
```
<ContentType ID="0x0108010021EAE10FFD704A418A504B340DEA68BE" Name="…"
             Group="…" Description="…" Version="0" Hidden="False">
  <FieldRefs />
  <XmlDocuments>
    <XmlDocument NamespaceURI ="http://...">
      <FormUrls xmlns="http://...">
        <Edit>_layouts/WssDemo/CustomFormsTaskForm.aspx</Edit>
        <Display>_layouts/WssDemo/CustomFormsTaskForm.aspx</Display>
      </FormUrls>
    </XmlDocument>
  </XmlDocuments>
</ContentType>
```

Using Task Content Types

- Two ways of defining task content type
  - Default content type id defined in workflow feature
  - Content type set in CreateTaskWithContentType

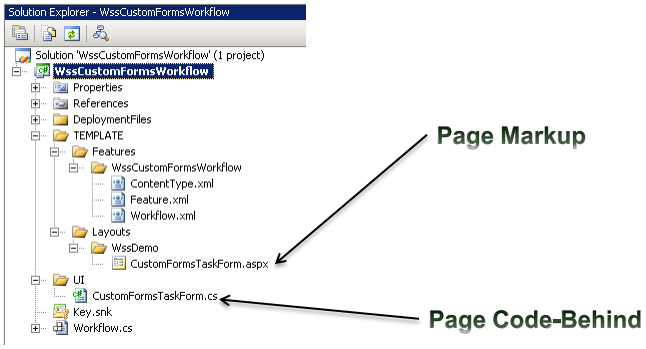
    Done with ContentTypeId property

```
<Workflow ...
    TaskListContentTypeId="0x0108010021EAE10FFD704A418A504B340DEA68BE">
  ...
</Workflow>
```



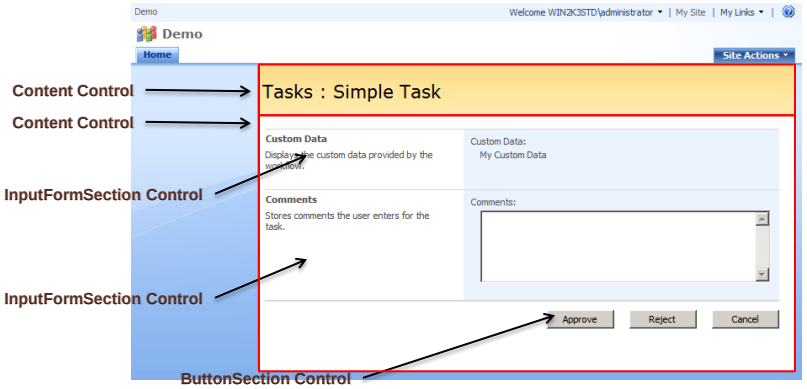
Building Custom Task Forms

- Task forms are aspx pages
  - Visual Studio doesn't support forms directly
  - Markup and Code-Behind connected manually



Page Visual Design

- WSS Provides support for common look and feel
  - Application.master used to provide standard “chrome”
  - Special UserControls used to provide layout



Master Page and Code Behind

- ASPX Page directive defines page properties
  - Defines master page
  - Defines code behind class
- Assembly directive references workflow assembly

```
<%@ Assembly Name="WssCustomFormsWorkflow, ..." %>
<%@ Page Language="C#" MasterPageFile="~/_layouts/application.master"
        Inherits="WssCustomFormsWorkflow.UI.CustomFormsTaskForm" %>
```

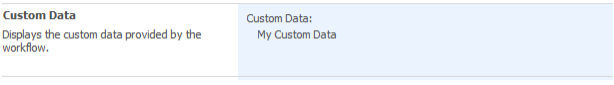
WSS Layout User Controls

- Controls used by WSS to define look and feel
  - Provides simple standard structure
  - Rendered HTML decorated with standard CSS classes
  - Registered on each page using Register directive

```
<%@ Register TagPrefix="wssuc" TagName="InputFormSection"
             Src="/_controltemplates/InputFormSection.ascx" %>
<%@ Register TagPrefix="wssuc" TagName="InputFormControl"
             Src="/_controltemplates/InputFormControl.ascx" %>
<%@ Register TagPrefix="wssuc" TagName="ButtonSection"
             Src="/_controltemplates/ButtonSection.ascx" %>
```

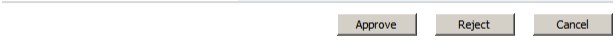
WSS Layout User Controls

```
<wssuc:InputFormSection Title="Custom Data" Description="..." runat="server">
  <template_inputformcontrols>
    <wssuc:InputFormControl LabelText="Custom Data:" runat="server">
      <Template_Control>
        <asp:Label ID="lblCustomData" runat="server" />
      </Template_Control>
    </wssuc:InputFormControl>
  </template_inputformcontrols>
</wssuc:InputFormSection>
```



WSS Layout User Controls

```
<wssuc:ButtonSection runat="server" ShowStandardCancelButton="false">
  <template_buttons>
    <asp:PlaceHolder runat="server">
      <asp:Button ID="btnApprove" Text="Approve“ UseSubmitBehavior="false"
                  class="ms-ButtonHeightWidth" OnClick="Approve_Click"
                  runat="server" />
       
      <asp:Button ID="btnReject" Text="Reject" UseSubmitBehavior="false"
                  class="ms-ButtonHeightWidth" OnClick="Reject_Click"
                  runat="server" />
       
      <asp:Button ID="btnCancel" Text="Cancel" UseSubmitBehavior="false"
                  class="ms-ButtonHeightWidth" OnClick="Cancel_Click"
                  runat="server" />
    </asp:PlaceHolder>
  </template_buttons>
</wssuc:ButtonSection>
```



Form Code-Behind

- Task forms are provided a set of URL parameters
  - Parameters determine the list and id of the task item
  - List and task item used to find the workflow
  - Lookup often done in OnLoad event

```
protected override void OnLoad(EventArgs e)
{
    _taskList = Web.Lists[new Guid(Request.Params["List"])];
    _taskItem = _taskList.GetItemById(int.Parse(Request.Params["ID"]));
    _workflow = new SPWorkflow(Web,
                  new Guid(_taskItem["WorkflowInstanceID"] as string));
    _task = _workflow.Tasks[0];
    _list = Web.Lists[_workflow.ListId];

    ...
}
```

Populating the Form

- The Task SPListItem used to populate data
  - SPListItem fields can be accessed direction
  - Extended properties accessed with SPWorkflowTask

```
protected override void OnLoad(EventArgs e)
{
  ...

  if (!IsPostBack)
  {
    Hashtable extendedProperties =
      SPWorkflowTask.GetExtendedPropertiesAsHashtable(_taskItem);
    lblCustomData.Text =
      extendedProperties["CustomData"] as string;
  }

  ...
}
```

Closing the Form

- User can close form in multiple ways
  - All ways of closing the form are task specific

    Ex. Approve, Reject, Delegate, Cancel
  - Code-behind sends properties back to the workflow
  - Page is closed by redirecting to another page

```
protected void Cancel_Click(object sender, EventArgs e)
{
    SPUtility.Redirect(_list.DefaultViewUrl,
        SPRedirectFlags.Default, this.Context);
}
```

Sending Properties to Workflow

- Response sent to workflow using ExtendedProps
  - SPWorkflowTask.AlterTask accepts a Hashtable of data
  - Data contains one or more properties used by workflow

```
protected void Approve_Click(object sender, EventArgs e)
{
    // build properties to send to workflow
    Hashtable data = new Hashtable();
    data.Add("Result", "Approved");
    data.Add("Comments", txtComments.Text);

    // update the task and redirect to list's default view
    SPWorkflowTask.AlterTask(_taskItem, data, true);
    SPUtility.Redirect(_list.DefaultViewUrl,
        SPRedirectFlags.Default, this.Context);
}
```

Using Task Results in Workflow

- Task extended properties accessible in workflow
  - Accessible via afterProperties
  - Contains a hashtable named ExtendedProperites
  - ExtendedProperties also contains task item columns
    Accessible by list item field IDs

```
private void TaskChanged_Invoked(object sender, ExternalDataEventArgs e)
{
    SPTaskServiceEventArgs args = e as SPTaskServiceEventArgs;

    // access a named extended property
    TaskResult =
        args.afterProperties.ExtendedProperties["Result"] as string;

    // access a list item field by id
    Guid commentsId = new Guid("{52578FC3-1F01-4f4d-B016-94CCBCF428CF}");
    TaskComments =
        args.afterProperties.ExtendedProperties[commentsId] as string;
}
```

Summary

- Discuss workflow/task interaction
- Create a simple workflow using tasks
- Add a custom form a task

Creating Workflow Association Forms

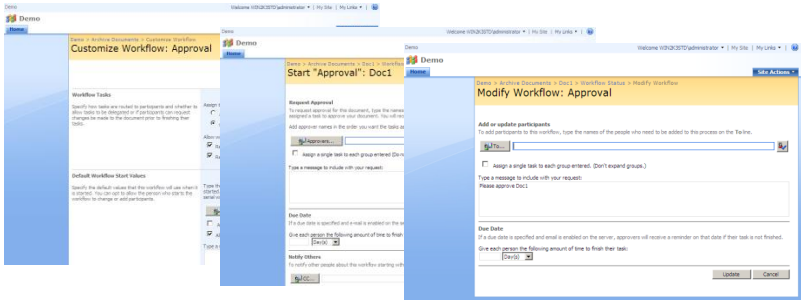
Developing SharePoint Workflow Templates with Visual Studio

Agenda

- Custom Workflow Forms in SharePoint
- Workflow Form Data Flow
- Building & Registering Custom Association Forms
- Creating/Updating Workflow Associations

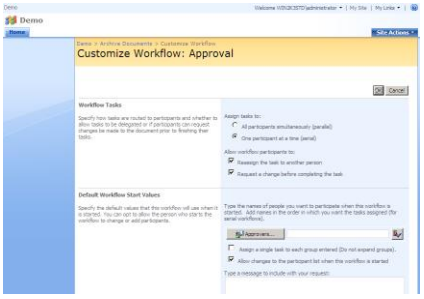
Workflow Forms

- WSS Workflows reference three types of forms
  - Association forms for attachment to list/content type
  - Instantiation forms for starting workflow on a list item
  - Modification forms for changing settings while running
  - All types of forms are optional



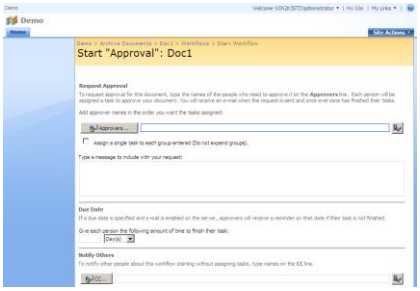
Association Forms

- Displayed when associating a workflow template
  - Displayed after default association page
  - Specific to each workflow template
  - Often used to setup “default” parameters
  - Responsible for creating the SPWorkflowAssociation



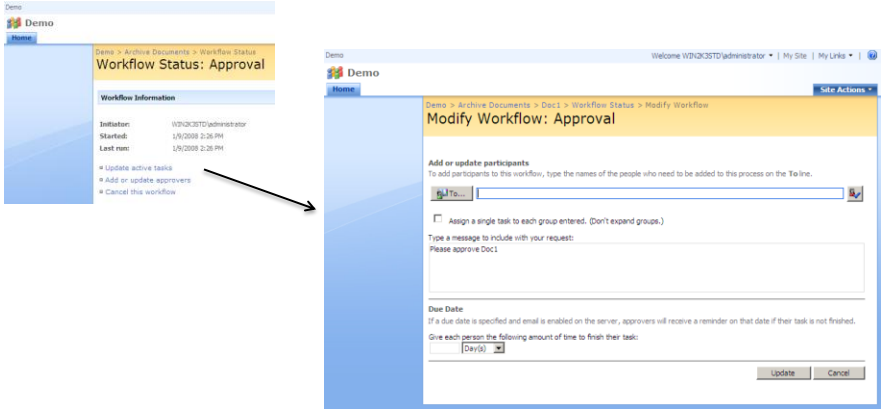
Instantiation Forms

- Displayed whenever a workflow instance started
  - Displayed immediately when workflow started
  - Specific to each workflow template
  - Often initialized using the association form's data
  - Responsible for starting the workflow using



Modification Forms

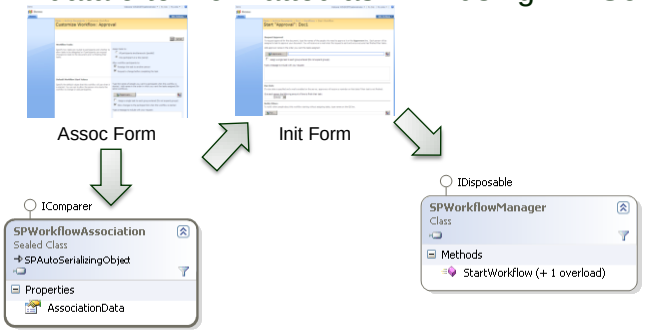
- Accessible from the workflow status page
  - Modification points registered by workflow
  - Links to forms displayed on status page



Form Data Flow

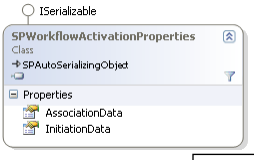
- Data flows from Assoc form to Instantiation form
  - Assoc form data stored in SPWorkflowAssociation
  - Init data passed from form to SPWorkflowManager

    `Data passed via parameter in StartWorkflow method`
  - Form data often formatted as XML using XmlSerializer



Using Form Data in Workflow

- Workflow accesses Assoc and Initiation Data
  - Provided to workflow in OnWorkflowActivated activity
  - Up to the workflow developer to use data appropriately




```
private void onWorkflowActivated_Invoked(
    object sender, ExternalDataEventArgs e)
{
    SPActivationEventArgs args = e as
        SPActivationEventArgs;

    AssociationData assocData = AssociationData.Deserialize(
        args.properties.AssociationData);
    this.CustomData = assocData.DefaultCustomData;
}
```

Registering Custom Forms

- Registration of custom forms done in feature
  - Workflow element provides form URL attributes
  - Form URL is used to load a custom .aspx page
  - If no URL is registered, no form is displayed

```
<Workflow
   Name="Wss Custom Forms Workflow"
   Description="…
   Id="3768e1f1-560e-4b95-8d60-dcbb53c37a56"
   CodeBesideClass="WssCustomFormsWorkflow.Workflow"
   CodeBesideAssembly="WssCustomFormsWorkflow, ..."
   TaskListContentTypeId="0x0108010021EAE10FFD704A418A504B340DEA68BE"
   AssociationUrl ="_layouts/WssDemo/CustomFormsAssocForm.aspx"
   InstantiationUrl="_layouts/WssDemo/CustomFormsInitForm.aspx"
   ModificationUrl="_layouts/WssDemo/CustomFormsModForm.aspx">
  ...
</Workflow>
```

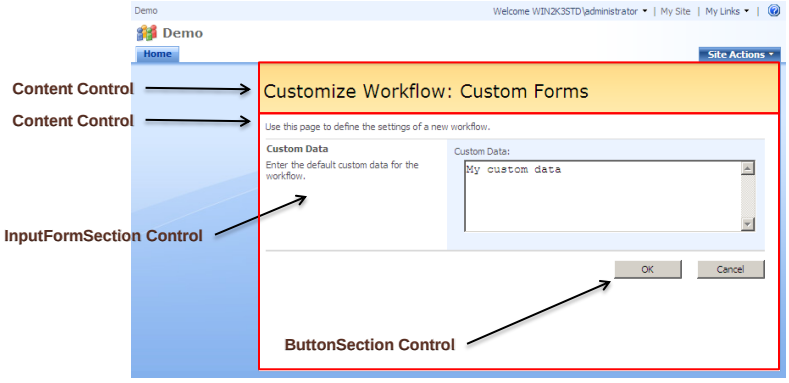
Building Custom Association Form

- Steps for building association forms:
  - Design aspx page
  - Create code-behind class

    ```
    Initialize the UI elements
    Load any existing association data
    Create or Update the Workflow Association
    ```
  - Register the new custom form

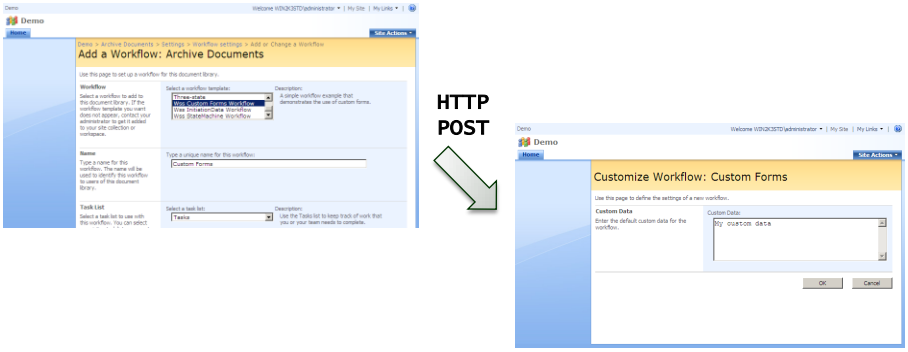
Designing Association Form ASPX

- Uses the same layout concepts as task form
  - Uses same master page
  - Uses same WSS layout UserControls



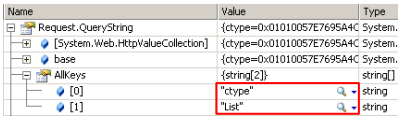
Association Forms are unique?

- Association form is accessed via post back
  - Previous page posts to the association form
  - Passes key workflow information via post form
  - Requires some special handling by form developer

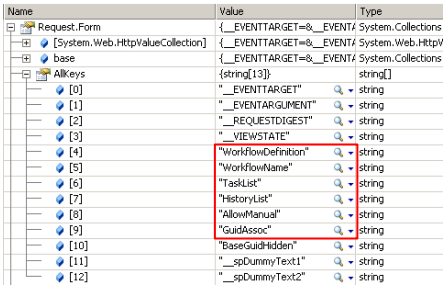


Association Form Post and URL data

- Data contained in URL and Post form defines
  - Is this a new association or an existing association?
  - Is this association attached to a list or a content type?
  - What are the names of the tasks and history lists?
  - What permissions and startup handlers are used?



Name	Value	Type
Request.QueryString	{ctype=0x01010057E7695A4C	System.
[System.Web.HttpValueCollection]	{ctype=0x01010057E7695A4C	System.
base	{ctype=0x01010057E7695A4C	System.
AllKeys	{string[2]}	string[]
[0]	"ctype"	string
[1]	"List"	string



Name	Value	Type
Request.Form	{__EVENTTARGET=&__EVENTA	System.Collections
[System.Web.HttpValueCollection]	{__EVENTTARGET=&__EVENTA	System.Web.HttpV
base	{__EVENTTARGET=&__EVENTA	System.Collections
AllKeys	{string[13]}	string[]
[0]	"__EVENTTARGET"	string
[1]	"__EVENTARGUMENT"	string
[2]	"__REQUESTDIGEST"	string
[3]	"__VIEWSTATE"	string
[4]	"WorkflowDefinition"	string
[5]	"WorkflowName"	string
[6]	"TaskList"	string
[7]	"HistoryList"	string
[8]	"AllowManual"	string
[9]	"GuidAssoc"	string
[10]	"BaseGuidHidden"	string
[11]	"__spDummyText1"	string
[12]	"__spDummyText2"	string

Why a POST?

- Why use a HTTP POST? Why not a GET?
  - First step of association shouldn't create association
    `All data collected on first steps must be sent`
  - Get doesn't allow data transmission beyond URL
  - Post allows more data to be transmitted easily

How is handling a POST different?

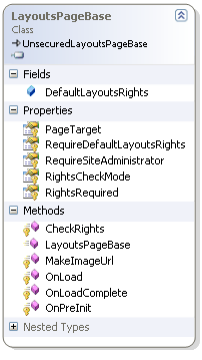
- ASP.NET uses a postback to perform processing
  - Data on the page's form posted back to current page
  - If not careful, the initial post's data is lost
  - Solution; put the data from initial post into page's form

```
<input type="hidden" name="WorkflowDefinition"
       value=<%= Request.Form["WorkflowDefinition"]; %>>
<input type="hidden" name="WorkflowName"
       value=<%= Request.Form["WorkflowName"]; %>>
<input type="hidden" name="AddToStatusMenu"
       value=<%= Request.Form["AddToStatusMenu"]; %>>
<input type="hidden" name="AllowManual" value=<%= Request.Form["AllowManual"]; %>>
<input type="hidden" name="RoleSelect" value=<%= Request.Form["RoleSelect"]; %>>
<input type="hidden" name="AutoStartCreate"
       value=<%= Request.Form["AutoStartCreate"]; %>>
<input type="hidden" name="AutoStartChange"
       value=<%= Request.Form["AutoStartChange"]; %>>
<input type="hidden" name="GuidAssoc" value=<%= Request.Form["GuidAssoc"]; %>>
```

Creating Code-Behind Class

- Code behind class derives from LayoutPageBase
  - Provides base functionality of pages in layouts folder

```
public class CustomFormsAssocForm :
  LayoutsPageBase
{
}
```



Processing Page Parameters

- Page parameters come from URL and form
  - Immediate focus finding association's parent
  - Can be list, content type, or both
  - Determined with List and ctype URL parameters

```
protected override void OnLoad(EventArgs e) {
    // read the form level parameters
    string listId = this.Request.Params["List"];
    string ctypeId = this.Request.Params["ctype"];

    // determine the type of association
    if (!string.IsNullOrEmpty(listId) && !string.IsNullOrEmpty(ctypeId))
        this._associationType = AssociationType.ListContentType;
    else if (!string.IsNullOrEmpty(listId) && string.IsNullOrEmpty(ctypeId))
        this._associationType = AssociationType.List;
    else if (string.IsNullOrEmpty(listId) && !string.IsNullOrEmpty(ctypeId))
        this._associationType = AssociationType.ContentType;

    ...
}
```

Create or Update Association

- GuidAssoc param determines create or update
  - If form contains GuidAssoc, association exists
  - GuidAssoc used to lookup the association object

```
// read the form level parameters
string guidAssocId = Request.Params["GuidAssoc"];

// validate the workflow association id
Guid? workflowAssociationId = new Guid?();
if (!string.IsNullOrEmpty(guidAssocId))
    workflowAssociationId = new Guid(guidAssocId);
```

Lookup existing Association

- Location of association based on parent

```
case AssociationType.ContentType:
    _contentType = Web.AvailableContentTypes[new SPContentTypeId(ctypeId)];
    _workflowAssociation =
        _contentType.WorkflowAssociations[workflowAssociationId.Value];
    break;

case AssociationType.List:
    _list = Web.Lists[new Guid(listId)];
    _workflowAssociation =
        _list.WorkflowAssociations[workflowAssociationId.Value];
    break;

case AssociationType.ListContentType:
    _list = Web.Lists[new Guid(listId)];
    _contentType = _list.ContentTypes[new SPContentTypeId(ctypeId)];
    _workflowAssociation =
        _contentType.WorkflowAssociations[workflowAssociationId.Value];
    break;
```

Storing Form parameters for later

- URL and Form parameters used to load data
  - Form parameters won't be available in postback
  - Form parameters written back into hidden fields

```
protected override void OnPreRender(EventArgs e) {
    // register parameters in the hidden field
    ClientScript.RegisterHiddenField("WorkflowName", Request.Params["WorkflowName"]);
    ClientScript.RegisterHiddenField("WorkflowDefinition",
        Request.Params["WorkflowDefinition"]);
    ClientScript.RegisterHiddenField("AddToStatusMenu",
        Request.Params["AddToStatusMenu"]);
    ClientScript.RegisterHiddenField("AllowManual", Request.Params["AllowManual"]);
    ClientScript.RegisterHiddenField("RoleSelect", Request.Params["RoleSelect"]);
    ClientScript.RegisterHiddenField("GuidAssoc", Request.Params["GuidAssoc"]);
    ClientScript.RegisterHiddenField("SetDefault", Request.Params["SetDefault"]);
    ClientScript.RegisterHiddenField("HistoryList", Request.Params["HistoryList"]);
    ClientScript.RegisterHiddenField("TaskList", Request.Params["TaskList"]);
    ClientScript.RegisterHiddenField("UpdateLists", Request.Params["UpdateLists"]);
    ClientScript.RegisterHiddenField("AutoStartCreate",
        Request.Params["AutoStartCreate"]);
    ClientScript.RegisterHiddenField("AutoStartChange",
        Request.Params["AutoStartChange"]);

    ...
```

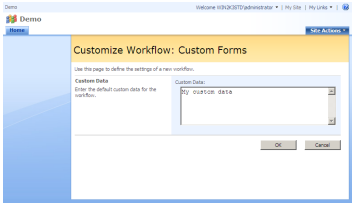
Displaying existing Association

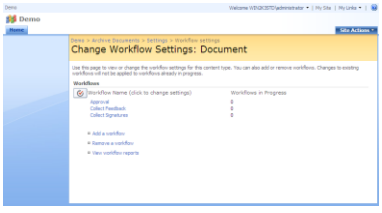
- If association exists the UI needs to be populated
  - UI population usually done in OnLoad if !IsPostBack
  - Initial assoc form load is a postback
  - Solution is to initialize in OnPreRender

```
protected override void OnPreRender(EventArgs e)
{
    // bind the association data (if any)
    if (_workflowAssociation != null)
    {
        // deserialize the association data
        AssociationData assocData = AssociationData.Deserialize(
            _workflowAssociation.AssociationData);

        // bind the controls to the association data
        txtCustomData.Text = assocData.DefaultCustomData;
    }
```

Handling Button Events

- The user can click OK or Cancel to submit form
  - OK causes the SPWorkflowAssociation object creation
  - Cancel causes no object creation
  - Both end up redirecting back to workflow settings page





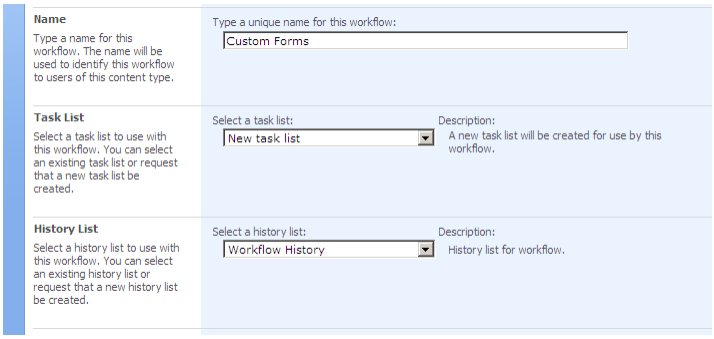
Handling OK Button Click Events

- Create and update actions require assoc data
  - First step of handler is to serialize assoc data to string
  - Often done using a custom class and XmlSerializer

```
AssociationData associationData = new AssociationData();
associationData.DefaultCustomData = txtCustomData.Text;
string serializedData = associationData.Serialize();


public class AssociationData
{
    public string DefaultCustomData { get; set; }
    public string Serialize()
    {
        XmlSerializer serializer = new XmlSerializer(typeof(AssociationData));
        using (StringWriter writer = new StringWriter())
        {
            serializer.Serialize(writer, this);
            return writer.ToString();
        }
    }
```

Finding/Creating Supporting Lists

- Workflow associations depend on special lists
  - Lists store tasks and history for workflow instances
  - Previous page allows choice of existing or new lists
  - Association form responsible for finding or creating lists



Finding/Creating Supporting Lists

- List name or ID encoded in Form parameters
  - If name starts with 'z', it's the name of a new list
  - Otherwise the name is the list name or ID

  `If parent is list, it's a name or ID`

```
SPList LookupOrCreateList(string paramList, SPListTemplateType listType) {
    if (paramList.StartsWith("z")) {
        Guid newListId =
            Web.Lists.Add(paramList.Substring(1), "Workflow Tasks",
            listType);
        return Web.Lists[newListId];
    }
    else {
        return Web.Lists[new Guid(paramList)];
    }
}
```

Creating or Updating Association?

- Create or update based on lookup in OnLoad
  - If a workflow association was found, it's an update

```
protected void Submit_Click(object sender, EventArgs e) {
    ...


    // create or find the lists
    SPList taskList = LookupOrCreateList(Request.Params["TaskList"],
        SPListTemplateType.Tasks);
    SPList historyList = LookupOrCreateList(Request.Params["HistoryList"],
        SPListTemplateType.WorkflowHistory);

    // create or update the workflow association
    if (_workflowAssociation != null)
        UpdateWorkflowAssociation(taskList, historyList,
            associationData.Serialize());
    else
        CreateWorkflowAssociation(taskList, historyList,
            associationData.Serialize());
```

Updating Existing Association

- Existing SPWorkflowAssociation values updated
  - Values from HTTP Form and URL updated first
  - AssociationData from custom form updated next

```
// assign base workflow association information
_workflowAssociation.Name = Request.Params["WorkflowName"];
_workflowAssociation.AutoStartCreate =
    (Request.Params["AutoStartCreate"] == "ON");
_workflowAssociation.AutoStartChange =
    (Request.Params["AutoStartChange"] == "ON");
_workflowAssociation.AllowManual = (Request.Params["AllowManual"] == "ON");
_workflowAssociation.AssociationData = associationData;

// assign the chosen task list to the association
if (_workflowAssociation.TaskListId != taskList.ID)
    _workflowAssociation.SetTaskList(taskList);

// assign the chosen history list to the association
if (_workflowAssociation.HistoryListId != historyList.ID)
    _workflowAssociation.SetHistoryList(historyList);
```

Commiting Association Updates

- Once changes made, they must be committed
  - Done with parent's UpdateWorkflowAssociation method

```
switch (_associationType)
{
    case AssociationType.ContentType:
        // commit the changes to the existing workflow association
        _contentType.UpdateWorkflowAssociation(_workflowAssociation);
        break;

    case AssociationType.List:
        // commit the changes to the existing workflow association
        _list.UpdateWorkflowAssociation(_workflowAssociation);
        break;

    case AssociationType.ListContentType:
        // commit the changes to the existing workflow association
        _contentType.UpdateWorkflowAssociation(_workflowAssociation);
        break;
}
```

Creating New Associations

- Done with SPWorkflowAssociation static methods
  - One method for each type of parent
  - All take similar parameters

```
case AssociationType.ContentType:
    _workflowAssociation =
        SPWorkflowAssociation.CreateSiteContentTypeAssociation(
            workflowTemplate, workflowName,
            taskList.Title, historyList.Title);
    break;

case AssociationType.List:
    _workflowAssociation = SPWorkflowAssociation.CreateListAssociation(
        workflowTemplate, workflowName, taskList, historyList);
    break;

case AssociationType.ListContentType:
    _workflowAssociation =
        SPWorkflowAssociation.CreateListContentTypeAssociation(
            workflowTemplate, workflowName, taskList, historyList);
break;
```

Adding the new Association to Parent

- Once new association is populated, it's added
  - Each parent type has AddWorkflowAssociation method

```
case AssociationType.ContentType:
    PopulateWorkflowAssociation(taskList, historyList, assocData);
    _contentType.AddWorkflowAssociation(_workflowAssociation);
    break;

case AssociationType.List:
    PopulateWorkflowAssociation(taskList, historyList, assocData);
    _list.AddWorkflowAssociation(_workflowAssociation);
    break;

case AssociationType.ListContentType:
    PopulateWorkflowAssociation(taskList, historyList, assocData);
    _contentType.AddWorkflowAssociation(_workflowAssociation);
    break;
```

Redirecting to Workflow Settings

- On OK or Cancel handler complete, form “closes”
  - Closes by redirecting to WorkflowSettings page
  - URL parameters differ based on association parent

```
string url = null;
string list = Request.Params["List"];
string ctype = Request.Params["ctype"];

switch (_associationType)
{
    case AssociationType.ContentType:
        url = "WrkSetng.aspx?ctype=" + paramCtype;
    case AssociationType.List:
        url = "WrkSetng.aspx?List=" + paramList;
    case AssociationType.ListContentType:
        url = "WrkSetng.aspx?List=" + paramList + "&ctype=" + paramCtype;
}

SPUtility.Redirect(url, SPRedirectFlags.RelativeToLayoutsPage, this.Context);
```

ContentTypes and WF Associations

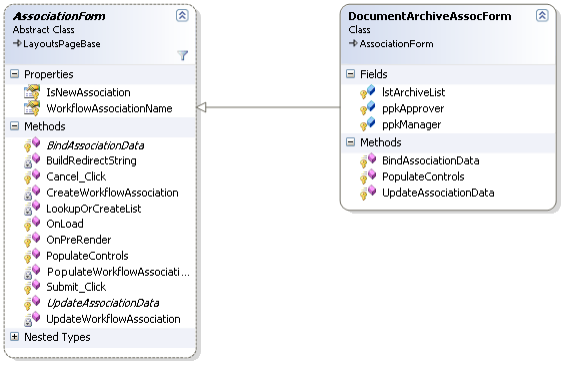
- Are content type association changes inherited?
  - When UpdateWorkflowAssociationsOnChildren called
  - Association form should call if UpdateLists is true

    Should be called after Add and Update call

```
// check whether updates to content types cascade to their lists
_updateContentTypeLists = (Request.Params["UpdateLists"] == "TRUE");

...
// update the association and update child associations
_contentType.UpdateWorkflowAssociation(_workflowAssociation);
if (_updateContentTypeLists)
    _contentType.UpdateWorkflowAssociationsOnChildren(true, true, true);

...
// add the association and update child associations
_contentType.AddWorkflowAssociation(_workflowAssociation);
if (_updateContentTypeLists)
    _contentType.UpdateWorkflowAssociationsOnChildren(true, true, true);
```

Making it Simpler

- Use a standard base class
  - Most of the code in an association form never changes
  - Code that varies displays and saves form data



Summary

- Custom Workflow Forms in SharePoint
- Workflow Form Data Flow
- Building & Registering Custom Association Forms
- Creating/Updating Workflow Associations

Creating Workflow Instantiation and Modification Forms

Developing SharePoint Workflow Templates with Visual Studio

Agenda

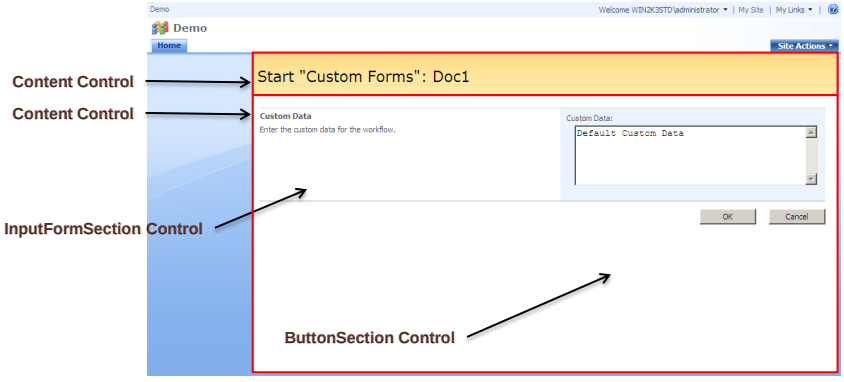
- Create and Register Initiation Forms
- Create Modification Forms
- Register and Enable Modification Forms

Building Custom Initiation Forms

- Steps for building initiation forms:
  - Design aspx page
  - Create code-behind class
    Initialize the UI elements
    Load any existing association data
    Start the workflow instance
  - Register the new custom form

Designing Initiation Form ASPX

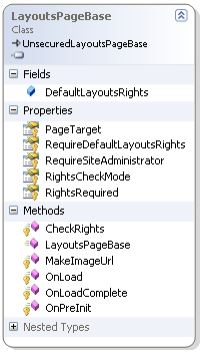
- Uses same layout concepts as association form
  - Uses same master page
  - Uses same WSS layout UserControls



Creating Code-Behind Class

- Code behind pages derive from LayoutPageBase
  - Provides base application page functionality

```
public class CustomFormsInitForm :
  LayoutsPageBase
{
}
```



Processing Page Parameters

- Page parameters come from URL
  - List and ctype define the parent of the association
  - ID identifies the list item the workflow instance is for
  - TemplateId identifies the workflow association

```
protected override void OnLoad(EventArgs e)
{
    // read the form level parameters
    string listId = Request.Params["List"];
    string listItemId = Request.Params["ID"];
    string ctypeId = Request.Params["ctype"];
    string templateId = Request.Params["TemplateId"];

    // find the list and list item
    _list = Web.Lists[new Guid(listId)];
    _listItem = _list.GetItemById(int.Parse(listItemId));
```

Locating the Workflow Association

- Workflow association may be in two locations
  - Could be associated with the list
  - Could be associated with the content type
  - Easiest way to find it is check both locations

```
// check for the workflow association in the list
_workflowAssociation = _list.WorkflowAssociations[
    new Guid(templateId)];

// if the association wasn't found, check the content type
if (_workflowAssociation == null)
{
    SPContentType _contentType = _list.ContentTypes[
        new SPContentTypeId(ctypeId)];
    _workflowAssociation = _contentType.WorkflowAssociations[
        new Guid(templateId)];
}
```

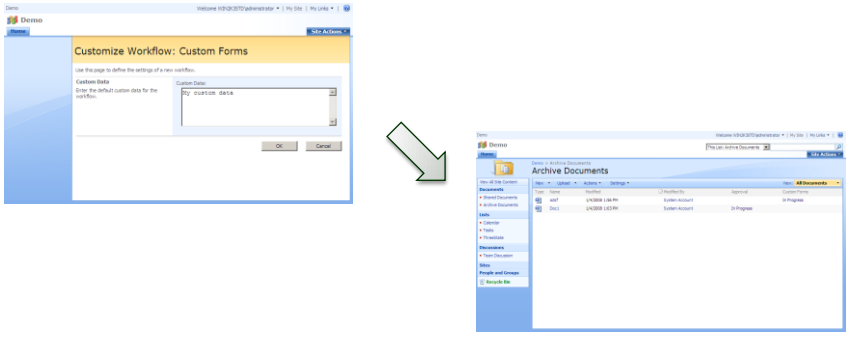
Initializing the UI

- The initial page load is not caused by a postback
  - Much simpler than Association form
  - Override OnLoad event and load when !IsPostBack

```
protected override void OnLoad(EventArgs e)
{
    ...

    // bind the association data (if any)
    if (!IsPostBack && (_workflowAssociation != null))
    {
        // bind the controls to the association data
        AssociationData assocData =
            AssociationData.Deserialize(_workflowAssociation.AssociationData);
        txtCustomData.Text = assocData.DefaultCustomData;
    }
```

Handling Button Events

- The user can click Start or Cancel to submit form
  - Start causes the workflow instance to start
  - Cancel causes no workflows to start
  - Both end up redirecting back to the list's default view



Handling Start Button Click

- Starting the workflow requires Initiation Data
  - Initiation data is gathered from the UI and serialized
  - Workflow is started using the site's workflow manager

    Requires the workflow association and list item

```
protected void Start_Click(object sender, EventArgs e)
{
    // serialize the initiation data
    InitiationData initiationData = new InitiationData();
    initiationData.CustomData = txtCustomData.Text;

    // start the new workflow instance
    Web.Site.WorkflowManager.StartWorkflow(
        _listItem, _workflowAssociation,
        initiationData.Serialize());
```

Redirecting to List Default View

- Start and Cancel buttons both redirect to list
  - Redirects to list's default view url

```
protected void Cancel_Click(object sender, EventArgs e)
{
    // redirect to the list default view
    SPUtility.Redirect(_list.DefaultViewUrl,
        SPRedirectFlags.Default, this.Context);
}
```

Registering Instantiation Forms

- Registration of custom forms done in feature
  - Workflow element provides form URL attributes
  - Form URL is used to load a custom .aspx page
  - If no URL is registered, no form is displayed

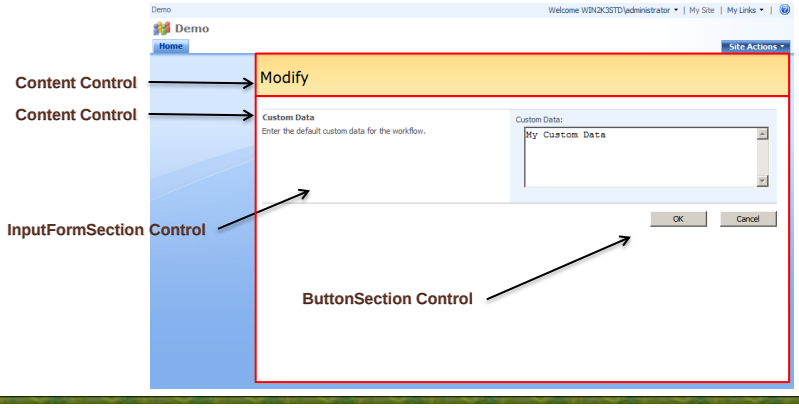
```
<Workflow
   Name="Wss Custom Forms Workflow"
   Description="…
   Id="3768e1f1-560e-4b95-8d60-dcbb53c37a56"
   CodeBesideClass="WssCustomFormsWorkflow.Workflow"
   CodeBesideAssembly="WssCustomFormsWorkflow, ..."
   TaskListContentTypeId="0x0108010021EAE10FFD704A418A504B340DEA68BE"
   AssociationUrl ="_layouts/WssDemo/CustomFormsAssocForm.aspx"
   InstantiationUrl="_layouts/WssDemo/CustomFormsInitForm.aspx"
   ModificationUrl="_layouts/WssDemo/CustomFormsModForm.aspx">
  ...
</Workflow>
```

Building Custom Modification Forms

- Steps for building modification forms:
- Design aspx page
- Create code-behind class
  - **Initialize the UI elements**
  - **Load any existing association data**
  - **Start the workflow instance**
- Register the new custom form
- Add metadata defining link
- Enable modification in workflow

Designing Modification Form ASPX

- Uses same layout concepts as association form
  - Uses same master page
  - Uses same WSS layout UserControls



Processing Page Parameters

- Page parameters come from URL
  - List and ID identify the list item related to the workflow
  - WorkflowInstanceID identifies the workflow instance
  - ModificationID identifies a SPWorkflowModification

```
protected override void OnLoad(EventArgs e)
{
    // read the form level parameters
    string listId = Request.Params["List"];
    string listItemId = Request.Params["ID"];
    string workflowId = Request.Params["WorkflowInstanceID"];
    string modificationId = Request.Params["ModificationID"];

    // find the list, list item, workflow, and modification
    _list = Web.Lists[new Guid(listId)];
    _listItem = _list.GetItemById(Convert.ToInt32(listItemId));
    _workflow = _listItem.Workflows[new Guid(workflowId)];
    _modification = _workflow.Modifications[new Guid(modificationId)];
```

Initializing the UI

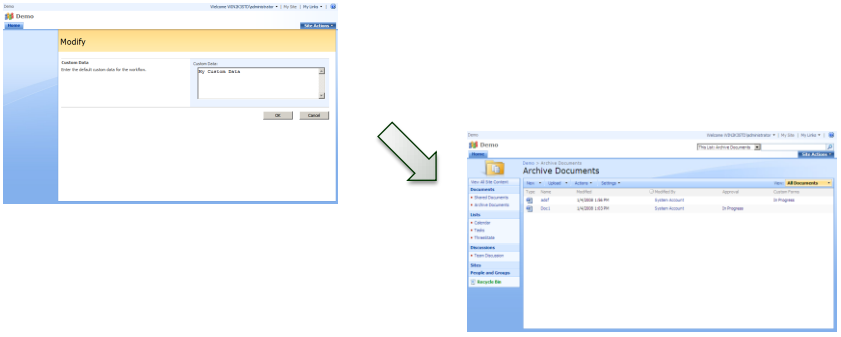
- The initial page load is not caused by a postback
  - Override OnLoad event and load when !IsPostBack
  - Use SPWorkflowModification.ContextData to load UI

    Often ContextData is XML created by XmlSerializer

```
protected override void OnLoad(EventArgs e)
{
    ...

    // populate the controls
    if (!IsPostBack)
    {
        // deserialize the association data
        ModificationData modData =
            ModificationData.Deserialize(_modification.ContextData);

        // bind the controls to the association data
        txtCustomData.Text = modData.CustomData;
    }
```

Handling Button Events

- The user can click OK or Cancel to submit form
  - OK applies modifications to workflow
  - Cancel causes no modifications to be made
  - Both end up redirecting back to the list's default view



Handling OK Button Click

- Changes applied using ModifyWorkflow method
  - Modification data gathered from UI and serialized
  - Workflow is modified using the site's workflow manager

  Requires SPWorkflow and SPWorkflowModification

```
protected void Start_Click(object sender, EventArgs e)
{
    // populate the modification data using the UI
    ModificationData modData = new ModificationData();
    modData.CustomData = txtCustomData.Text;

    // modify the workflow
    Web.Site.WorkflowManager.ModifyWorkflow(
        _workflow, _modification, modData.Serialize());
```

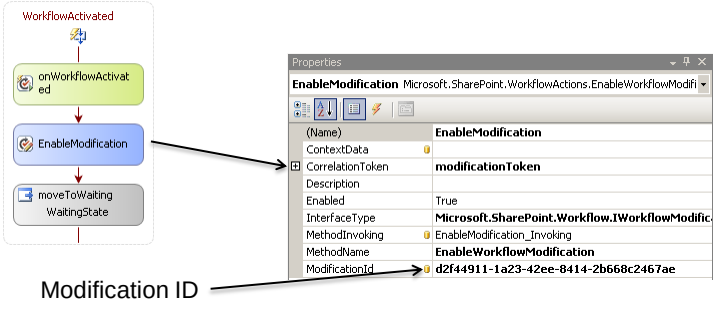
Redirecting to List Default View

- Just like instantiation form, close redirects to list
  - Redirects to list's default view url

```
protected void Cancel_Click(object sender, EventArgs e)
{
    // redirect to the list default view
    SPUtility.Redirect(_list.DefaultViewUrl,
        SPRedirectFlags.Default, this.Context);
}
```

Using the Modification Form

- Each type of modification identified by an ID
  - Used when modifications are enabled in the workflow
  - Specified in the EnableWorkflowModification activity
  - Related to a new modification correlation token



Enabling Workflow Modifications

- Modification form needs data to populate form
  - Association or Initiation data may be out dated
  - Better solution is to store data in modification object

    `Use SPWorkflowModification.ContextData property`

```
void EnableModification_Invoking(object sender, EventArgs e)
{
    ModificationData modData = new ModificationData();
    modData.CustomData = this.CustomData;

    EnableWorkflowModification enableModification =
        sender as EnableWorkflowModification;
    enableModification.ContextData = modData.Serialize();
}
```

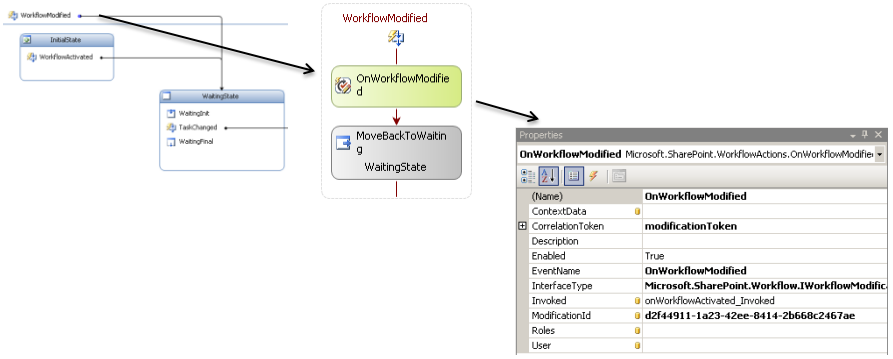
Registering Workflow Modifications

- Registration of custom forms done in feature
  - Form registration isn't only change required
  - The name of the modification is needed as well

    `Used when displaying the modification link`
  - Name identified by Modification ID

Responding to Modifications

- Modifications are received via event
  - Received using OnWorkflowModified activity
  - Uses modification correlation token and modification ID
  - Provides the context data returned from the form



Integrating Modified Data

- OnWorkflowModified event returns form data
  - Available in the SPModificationEventArgs.data property
  - Can be processed in the MethodInvoked handler

```
private void OnWorkflowModified_Invoked(
    object sender, ExternalDataEventArgs e)
{
    SPModificationEventArgs args = e as SPModificationEventArgs;

    ModificationData modData =
        ModificationData.Deserialize(args.data);
    this.CustomData = modData.CustomData;
}
```

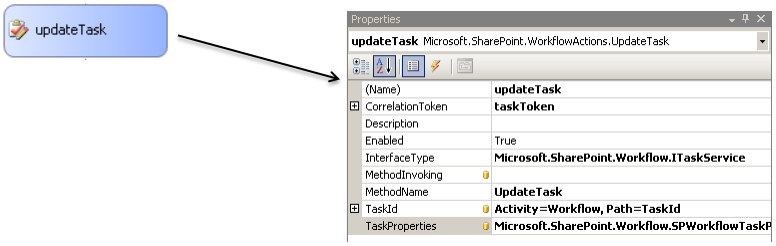
Resetting Workflow State

- Modifications often made as workflow is waiting
  - Tasks will most likely need to be updated
  - Two options

    `Update Task`
    `Delete and Create Task`

Updating Tasks

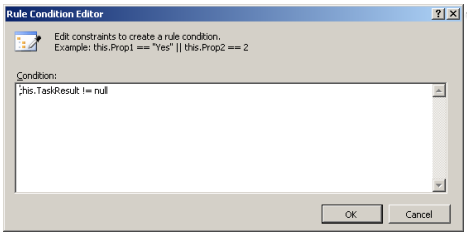
- Done using the UpdateTask activities
  - Requires task correlation token
  - Can't be done outside the token's scope
  - Task updated with SPWorkflowTaskProperties object

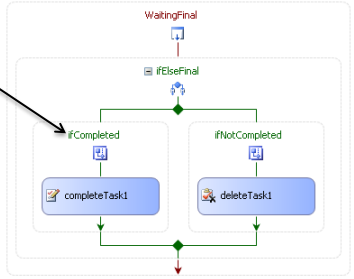


Delete and Re-create Task

- Can use state initializer and finalizer
  - Create task in initializer
  - Complete or delete task in finalizer based on result

    `Store result when task is complete`
  - State transition causes initializer and finalizer execution





Summary

- Create and Register Initiation Forms
- Create Modification Forms
- Register and Enable Modification Forms

Integrating InfoPath Forms into SharePoint Workflow

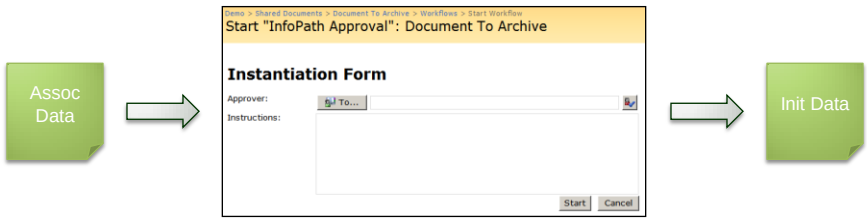
Developing SharePoint Workflow Templates with Visual Studio

Agenda

- Why use InfoPath?
- How it works
- Creating workflow forms with InfoPath
- Interacting with hosting environment
- Custom .NET code

Workflow Forms have commonalities

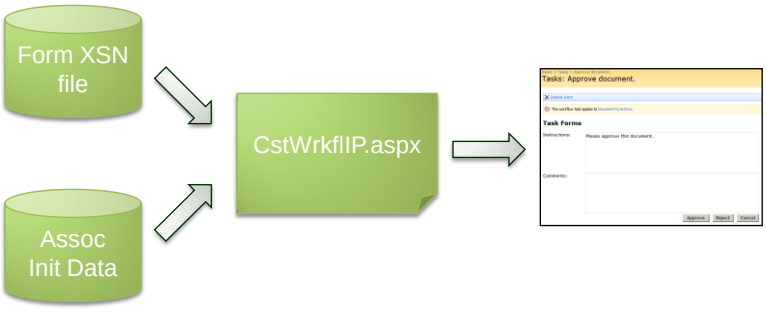
- Every workflow form type has many similarities
  - Essentially accept data, modify it, and pass it on
  - Accepting and passing on data is identical across forms
  - The modification of data is the unique part



Common Forms Framework

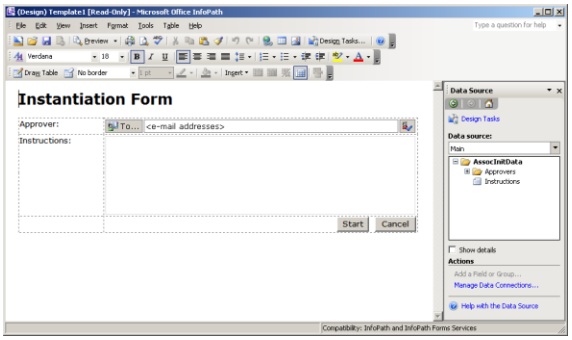
- Abstracting out the modification simplifies forms
  - One solution is to create a standard base class
  - Another solution is to define forms as metadata

    `Layout as well as behavior are needed`



What is InfoPath?

- InfoPath is a tool that stores forms as metadata
  - Designer stores layout and behavior in a XSN file
  - Viewer opens XSN file and loads/saves data as XML
  - Designer intended to be useable by non-developers

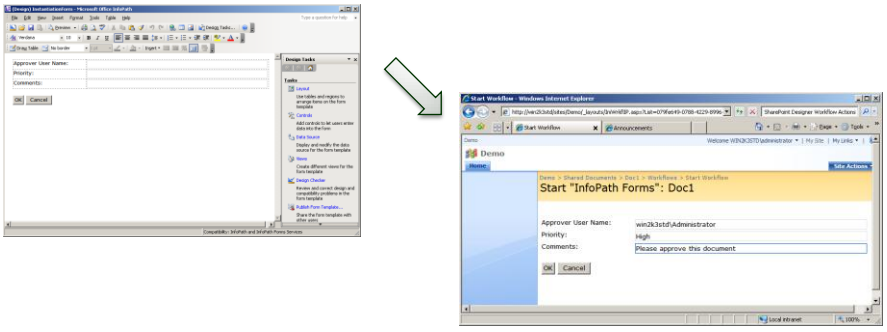


Difficulties Deploying InfoPath 2003

- InfoPath 2003 is a client side only application
  - Requires InfoPath on the end users machine
  - Trust issues required complex signing process
  - A better way was needed to deploy InfoPath forms

InfoPath Forms Services

- SharePoint 2007 introduced Forms Services
  - Ships as part of MOSS
  - Interprets InfoPath files and displays as web page
  - InfoPath forms now accessible to anyone via browser



Using InfoPath Forms in Workflow

- InfoPath data model imposes some restrictions
  - InfoPath forms have same input and output schema
  - Requires Association and Initiation data share schema

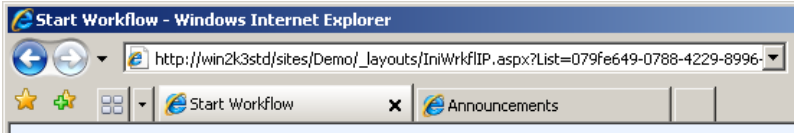
    `Rarely is this a problem`

Forms Services and Workflow

- MOSS defines standard workflow ASPX pages
  - One ASPX page for each type of form

Association Form	_layouts/CstWrkflIP.aspx
Instantiation Form	_layouts/IniWrkflIP.aspx
Modification Form	_layouts/ModWrkflIP.aspx
Task Form	_layouts/WrkTaskIP.aspx

  - Pages automatically load the correct InfoPath form



InfoPath Forms ASPX Pages

- Page markup contains a special web control
  - XmlFormView control renders the InfoPath form
  - Control requires two pieces of information

    The InfoPath form to host
    The XML data used to initialize the form

```
<%@ Register Tagprefix="InfoPath"
             Namespace="Microsoft.Office.InfoPath.Server.Controls"
             Assembly="Microsoft.Office.InfoPath.Server, ..." %>

<asp:Content ContentPlaceHolderId="PlaceHolderMain" runat="server">
    <InfoPath:XmlFormView id="XmlFormView" runat="server" />
    ...
</asp:Content>
```

InfoPath and Workflow Metadata

- WSS Workflow Features can have metadata
  - This metadata is accessible via an API
  - Forms Services pages access standard metadata tags
  - These tags define a URN for each type of form

```
<Workflow Id="66dd3439-b412-423e-8e15-cc972c9eb36a"
  ...
  AssociationUrl ="_layouts/CstWrkflIP.aspx"
  InstantiationUrl="_layouts/IniWrkflIP.aspx">
  <MetaData>
    <Association_FormURN>
      urn:...:AssociationForm:urn-WssInfoPathFormsWorkflow
    </Association_FormURN>
    <Instantiation_FormURN>
        urn:...:InstantiationForm:urn-WssInfoPathFormsWorkflow
    </Instantiation_FormURN>
  </MetaData>
</Workflow>
```

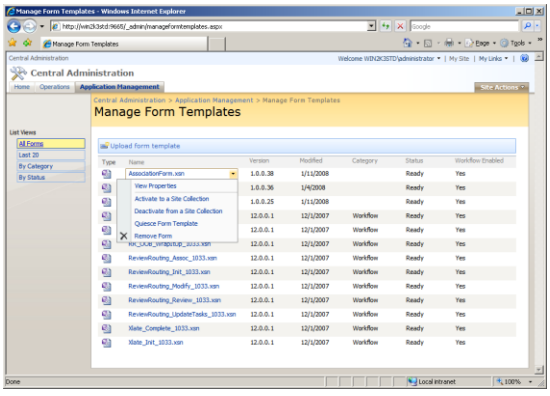
Registration of InfoPath Forms

- Locating forms by URN requires form registration
  - On feature activation event receiver registers forms
    `Microsoft.Office.Workflow.Feature.WorkflowFeatureReceiver`
  - Event receiver uses path in feature properties element

```
<Feature Id="695fb738-da73-438d-aa10-d50669dc3457"
  ReceiverAssembly="Microsoft.Office.Workflow.Feature, ..."
  ReceiverClass="Microsoft.Office.Workflow.Feature.WorkflowFeatureReceiver"
  xmlns="http://schemas.microsoft.com/sharepoint/">
  ...
  <Properties>
    ...
    <Property Key="RegisterForms" Value="Forms\*.xsn" />
  </Properties>
</Feature>
```

Form Registration with Central Admin

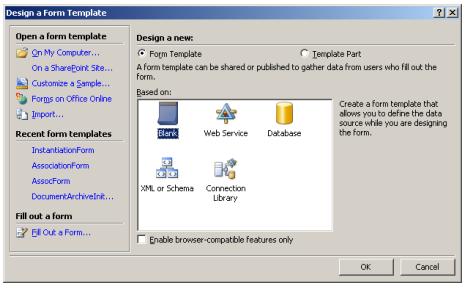
- Central Admin allows form management
  - Accessed from Application Management tab
  - Allows upload and installation of custom forms

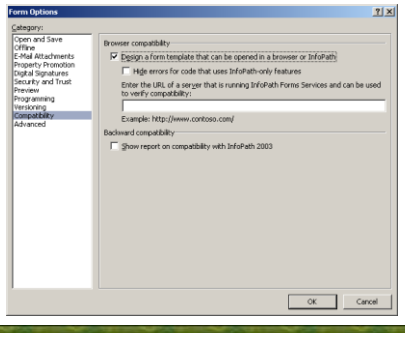


Creating custom InfoPath Forms

- Create a new form by designing a blank form
  - Some actions are not supported in Form Services
  - Use Tools -> Form Options to choose web form

  `Raises validation error if form not supported`

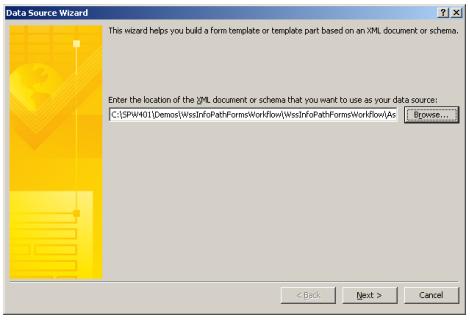


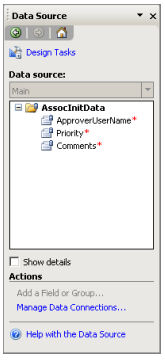


Importing Data Schema

- Form schema can be imported or created
  - Created XML schemas use generated namespace

    Complications when sharing schema between forms
  - Imported using Tools -> Convert Main Data Source

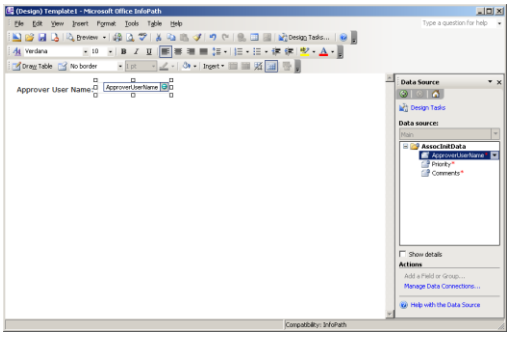




Adding Input Controls

- Data has a natural relationship to controls
  - Controls bound to data elements
  - Data can be “dragged” onto design service as controls

    Control type and label inferred from data

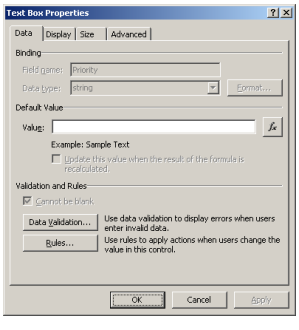


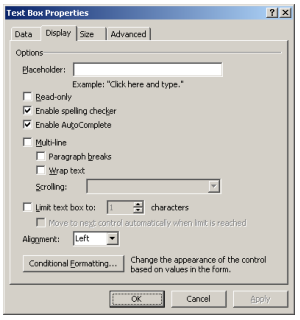
Layout Tables

- InfoPath provides formatting tools to designer
  - Layout tables provides structure
  - Text can have Word like layout and font applied
  - Experience intended to be similar to Word

Control Properties

- Each InfoPath control has specific properties
  - Allows restrictions such as read only
  - Allow sizing and formatting options
  - Provides custom data validation rules

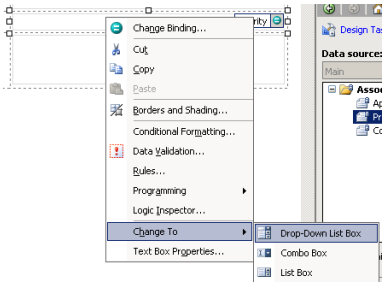


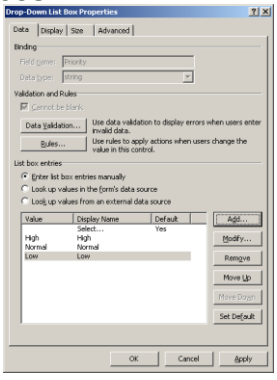


Using Drop Down Lists

- Drop Down Lists allow choosing from a list
  - TextBox converted using Change To context menu
  - Selected value bound to primary data source
  - Options can come from multiple sources

    `Fixed list, external data source`

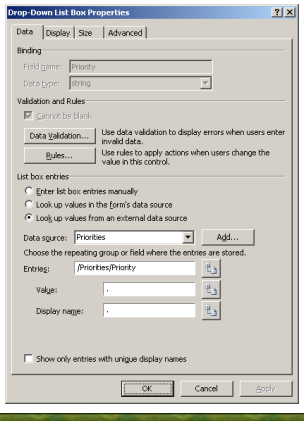


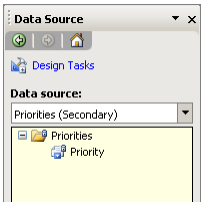


Binding to Secondary Data Sources

- External data can be accessed for list population
  - Data can come from multiple locations

    `Xml, WebService, SharePoint lists, etc…`

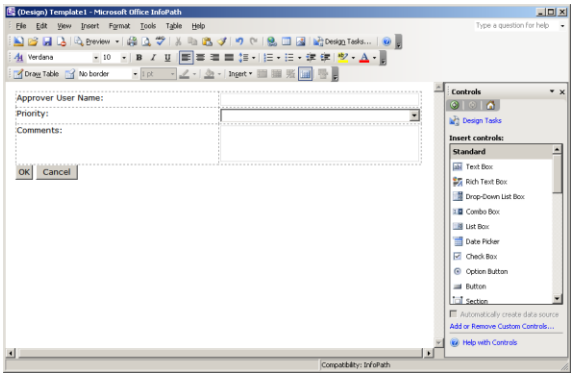




Submitting the Data

- Data submission initiated by button clicks
  - Controls can be added without data binding

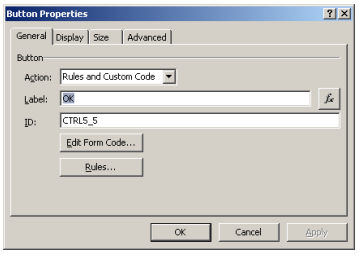
    `Common for controls with no display`
  - Button controls often used to submit or close

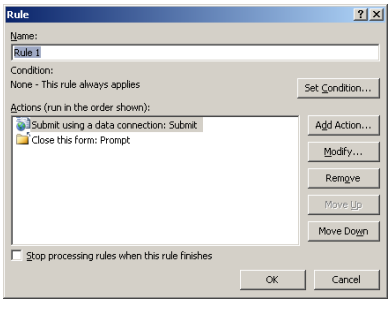


Applying Rules to Buttons

- Data submission done using InfoPath rules
  - Rules are executed when button is clicked
  - Forms submitted using submit to host

    `Special external data source`
  - Forms closed using Close Form action

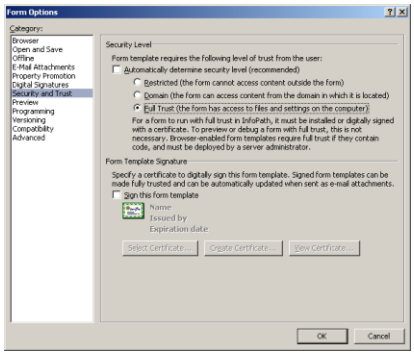




Choosing Trust Level

- InfoPath forms request a level of trust
  - InfoPath defaults to Restricted

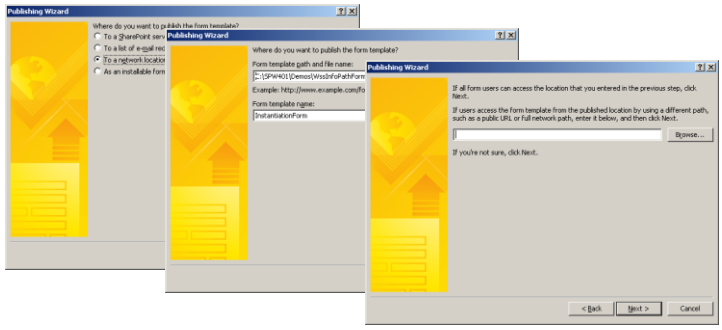
    `Causes InfoPath to fail while installing form`
  - Most WSS applications require Domain or Full Trust



Publishing InfoPath Forms

- Publishing is required to create a usable form
  - Signs form if a certificate is attached to the form
  - Attaches a publish location

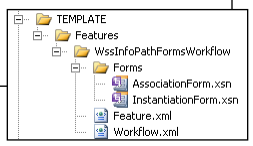
    `Publish location must be empty for SharePoint`



Deploying InfoPath Forms

- Published forms deployed with feature
  - Often located in Forms sub folder
  - Copied to the FEATURES folder along with feature xml
  - Registered by the feature activation event receiver

```
<Feature Id="695fb738-da73-438d-aa10-d50669dc3457"
    ...
    ReceiverAssembly="Microsoft.Office.Workflow.Feature, ..."
    ReceiverClass="Microsoft.Office.Workflow.Feature.WorkflowFeatureReceiver"
    xmlns="http://schemas.microsoft.com/sharepoint/">
  ...
  <Properties>
    <Property Key="RegisterForms" Value="Forms\*.xsn" />
  </Properties>
</Feature>
```



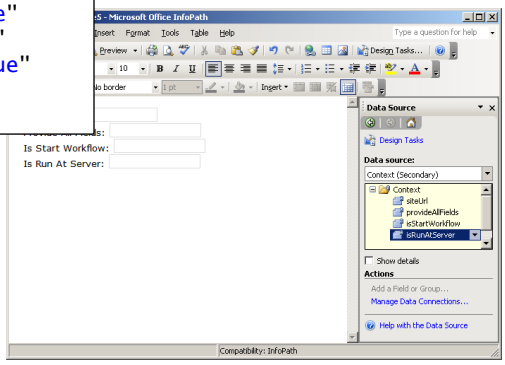
Forms Services Host Interaction

- InfoPath Forms have no connection to WSS
  - ASPX Forms can access WSS Object Model
  - InfoPath Forms are isolated from the Object Model
  - InfoPath does have access to Context data source
  - Task Forms can request task fields via ItemMetadata

Context Data Source

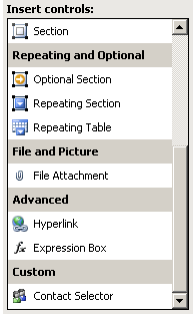
- Context data source based on XML file
  - Data automatically populated by ASPX host page
  - Allows InfoPath form access to critical context data

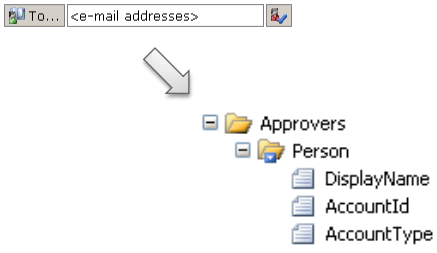
```
<Context
  isStartWorkflow="true"
  isRunAtServer="false"
  provideAllFields="true"
  siteUrl="" />
```



Using People Picker

- External ActiveX control used to pick users
  - Added to the Controls list in InfoPath
  - Bound to a special data structure

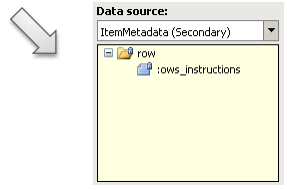




Accesing Task Extended Properties

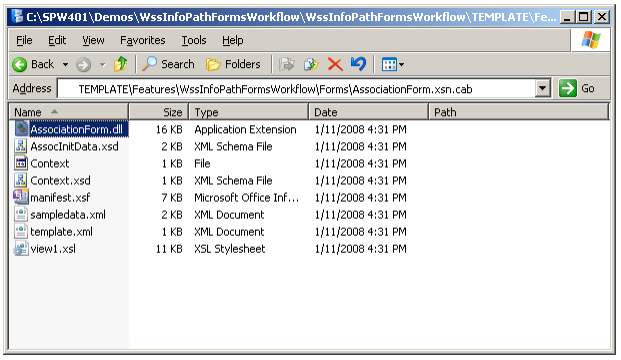
- Task properties provided via ItemMetadata
  - ItemMetadata.xml provided by task form designer
  - Defines which properties should be provided
    ows_ prefix required
  - Accessed as secondary data source by form designer

```
<z:row
xmlns:z="#RowsetSchema"
  ows_Instructions="" />
```



InfoPath Code Behind

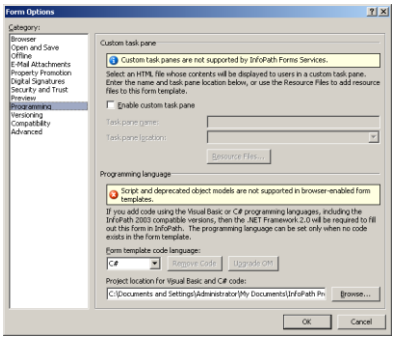
- InfoPath 2007 Supports .NET Code Behind
  - Key events can be handled using C# or VB Code
  - Generates an assembly that is embedded in template
    Template is just a CAB file



Editing Code Behind

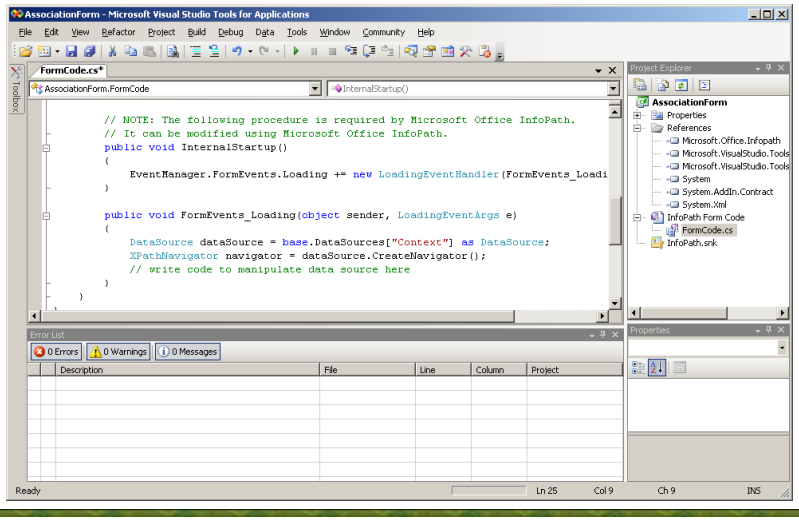
- Visual Studio Tools for Applications (VSTA) used
  - Deployed as part of InfoPath 2007
  - Allows full editing and compilation of C# and VB Code
  - Project created automatically when handling events

`Language chosen in Form Options`



Building Simple Initialization Code

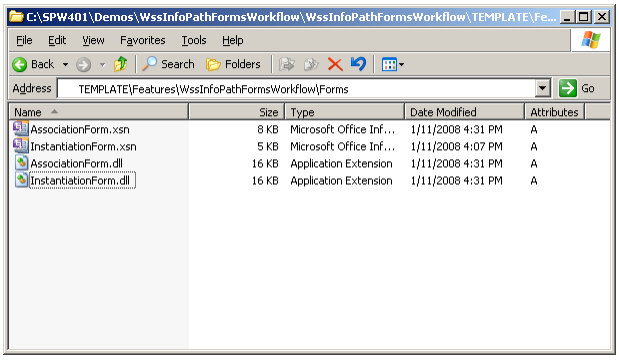
- Code Behind has direct access to form data



Forms Services and Code Behind

- Forms Services has unique requirements
  - Won't access assembly inside of the XSN
  - Requires assembly be placed in same folder as XSN

`Currently no automated way to perform`

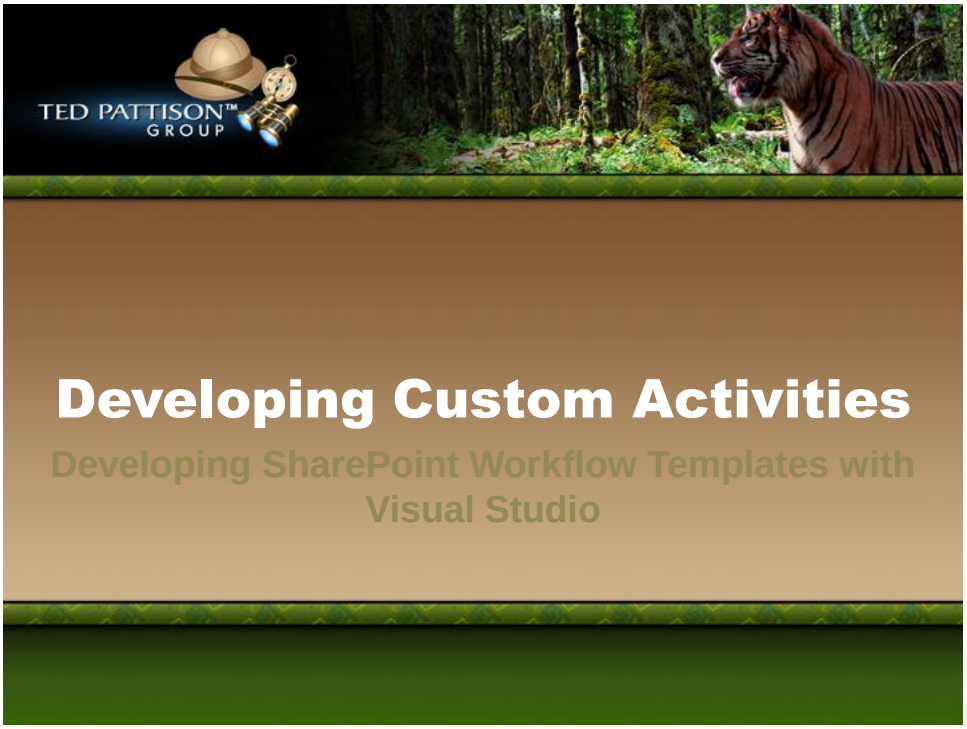


Common Code Behind Problems

- Deploying code behind can be difficult
  - Make sure the form's security level is correct
  - Make sure the assembly is outside the XSN
  - Don't forget to check LOGS folder

Summary

- Why use InfoPath?
- How it works
- Creating workflow forms with InfoPath
- Interacting with hosting environment
- Custom .NET code



Developing Custom Activities

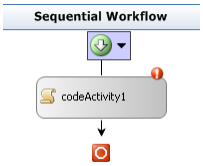
Developing SharePoint Workflow Templates with Visual Studio

Agenda

- Create activities
  - Events and Dependency Properties
  - Custom Activity Validators
- Understanding Activity Lifecycle
- Creating Composite Activities
- Custom Activity Designers

Why Custom Activities?

- Activities provide functionality encapsulation
  - Similar in concept to methods in C# or VB
  - Activity developers provide libraries to workflow builder
  - Custom designers communicate intent
  - Custom validators prevent simple mistakes



Hello World Activity

- All activities derive from Activity class
  - Key activity method is Execute
  - Execute is where the activity starts
  - Returning status of closed indicates the activity is done

```
public class HelloWorldActivity : Activity
{
    protected override ActivityExecutionStatus Execute(
                       ActivityExecutionContext executionContext)
    {
        Console.WriteLine("Hello World");
        return ActivityExecutionStatus.Closed;
    }
}
```

Adding Parameters to Activities

- Parameters allow activities to be reused
  - Activity properties are just .NET properties
  - Should wrap DependencyProperty to allow binding

```
public static DependencyProperty MessageProperty =
   DependencyProperty.RegisterAttached("Message",
      typeof(string), typeof(WriteConsoleActivity));

public string Message
{
    get { return base.GetValue(MessageProperty) as string; }
    set { base.SetValue(MessageProperty, value); }
}
```

Adding Events to Activities

- Events provide callback capabilities
  - Activity events are just .NET events
  - Should wrap DependencyProperty to allow binding
  - Event implementation delegates to Activity members

```
public static DependencyProperty InvokingEvent =
    DependencyProperty.RegisterAttached("Invoking",
      typeof(EventHandler), typeof(WriteConsoleActivity));

public event EventHandler Invoking
{
    add { base.AddHandler(InvokingEvent, value); }
    remove { base.RemoveHandler(InvokingEvent, value); }
}
```

Raising Events in Activities

- Events raised using Activity.RaiseEvent method
  - Raises the event registered using AddHandler
  - Automatically deals with any bindings

```
protected override ActivityExecutionStatus Execute(
    ActivityExecutionContext executionContext)
{
    // raise the bound event
    base.RaiseEvent(InvokingEvent, this, EventArgs.Empty);

    // write the value stored in the dependency property
    Console.WriteLine(this.Message);

    // tell the framework that this activity is done
    return ActivityExecutionStatus.Closed;
}
```

Validating Activities

- Activities can have an associated validator
  - Must derive from ActivityValidator
  - Validated at design time and at compile time
  - Validate method performs the validation

    Receives activity, returns an error collection

```
internal class WriteConsoleActivityValidator : ActivityValidator
{
    public override ValidationErrorCollection Validate(
        ValidationManager manager, object obj)
    {
        ...

        // return the errors
        return errors;
    }
}
```

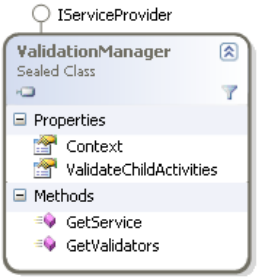
Validating Activities

- Validation fails when contained by an activity type

```
public override ValidationErrorCollection Validate(
    ValidationManager manager, object obj) {
    WriteConsoleActivity activity = obj as WriteConsoleActivity;
    if (activity == null)
        throw new InvalidOperationException();

    // check if any parent actions are a transaction scope
    ValidationErrorCollection errors = new ValidationErrorCollection();
    Activity parent = activity.Parent;
    while (parent != null) {
        if (parent is TransactionScopeActivity)
            errors.Add(new ValidationError(
                "Error Message", 100));
        parent = parent.Parent;
    }

    // call the base validation method
    errors.AddRange(base.Validate(manager, obj));
    return errors;
}
```

ValidationManager

- Allows communication between validators
  - Context allows data to be passed to child validators
  - ValidateChildActivities determines scope of validation
  - GetValidators and GetService allow context access



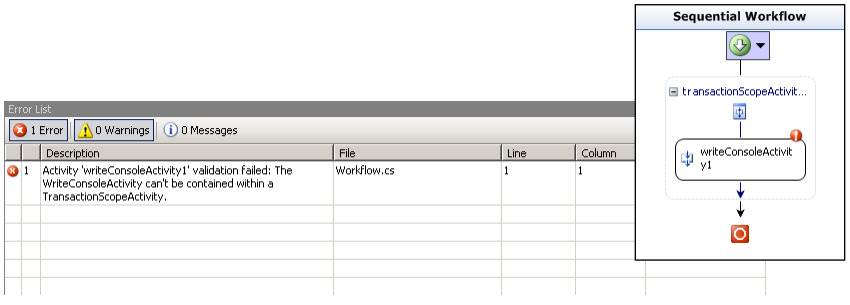
Attaching the Validator

- Validator attached to activity by .NET attribute
  - ActivityValidatorAttribute defines the validator type
  - Workflow designer and compiler reference attribute

```
[ActivityValidator(typeof(WriteConsoleActivityValidator))]
public class WriteConsoleActivity : Activity
{
    ...
}
```

Validators In Action

- Validators fire at design time and compile time
  - Design time validator errors are displayed in designer
  - Compile time validator errors displayed in error list
  - Multiple validator errors can be displayed at once

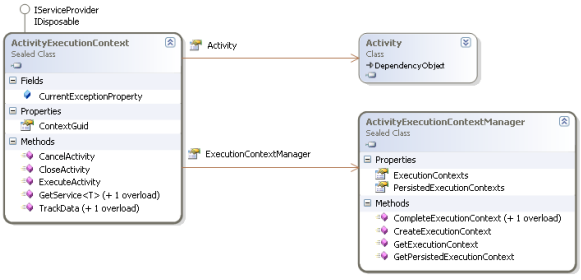


Activity Execution Context

- Created automatically for each activity
  - Manages the activities lifecycle

    `Allows activities to close or cancel themselves`
  - Can create new AECs using ExecutionContextManager

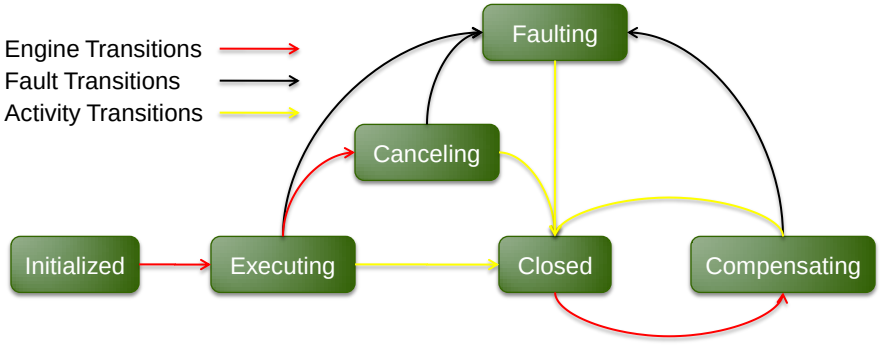
    `Used for looping activities; While or Replicator`



Activity States

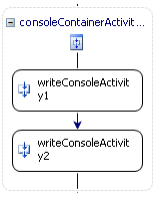
- Activity developers must be aware of states
  - State transitions are often influenced by custom code

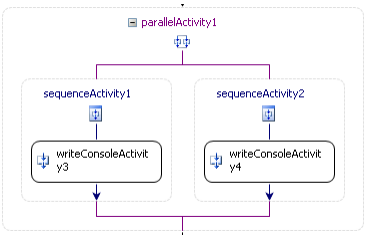
    `Ex. If execute completes but activity not done`
  - Valid state transitions defined by Workflow Runtime



Composite Activities

- Some activities contain multiple child elements
  - These activities execute child activities in different ways
  - Sequence executes on child at a time
  - Parallel executes all simultaneously
  - If/Else executes based on condition





Sequence Activity

- Execute method isn't always synchronous
  - Execute method returns the state of the activity
  - Activity can wait for events from the runtime

    Often used to initiate activity state transition

```
public class ConsoleContainerActivity : SequenceActivity,
{
    protected override ActivityExecutionStatus Execute(
        ActivityExecutionContext executionContext)
    {
        ...

        // start the first child activity and register for closed event
        base.EnabledActivities[0].RegisterForStatusChange(
            Activity.ClosedEvent, this);
        executionContext.ExecuteActivity(base.EnabledActivities[0]);

        // tell the runtime this activity is still executing
        return ActivityExecutionStatus.Executing;
    }
```

IActivityEventListener Interface

- Activities send events when their state changes
  - Event received via IActivityEventListener
  - Parent activities monitor state of children using events

    Register for events using RegisterForStatusChange
- Receive events in IActivityEventListener.OnEvent

```
base.EnabledActivities[0].RegisterForStatusChange(
    Activity.ClosedEvent, this);

void OnEvent(object sender,
    ActivityExecutionStatusChangedEventArgs e) {
    ...

    // unregister the handler for status events
    e.Activity.UnregisterForStatusChange(
        Activity.ClosedEvent, this);
}
```

Handling Child Closed Event

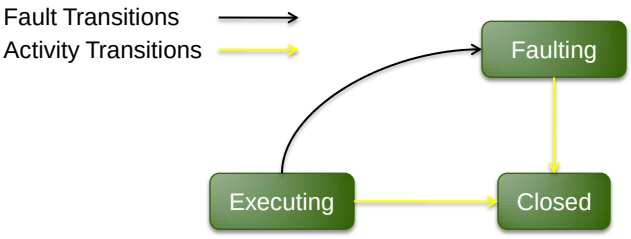
- Next activity executed when previous closed
  - Execute method registers for child activity closed event
  - When event received, next activity started
  - When no activities remain, sequence activity closes

```
public void OnEvent(object sender,
     ActivityExecutionStatusChangedEventArgs e)
{
    ActivityExecutionContext executionContext =
        sender as ActivityExecutionContext;
    ...
    if (activity.ExecutionStatus == ActivityExecutionStatus.Executing)
    {
        // if no more children exist
        if (!this.TryScheduleNextChild(executionContext))
            executionContext.CloseActivity();
    }
}
```

Handling Faults

- Exceptions during execution lead to faulting state
  - Runtime transitions activity to faulting state
  - Executes Activity.HandleFault and enters closed state

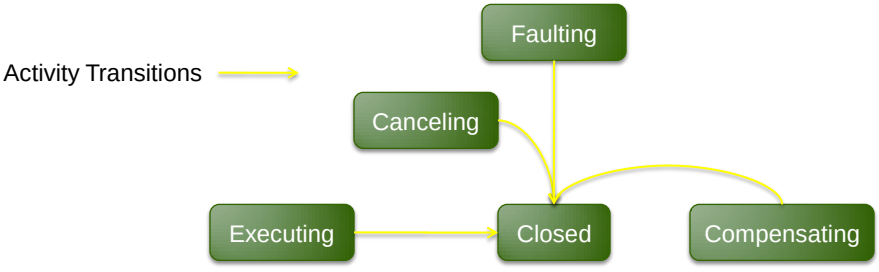
    ```
    This is where activities can clean up
    ```
  - Faults in HandleFault cause transition to faulting state



Activity Completion

- When the activity completes it is closed
  - Completed by ActivityExecutionContext.CloseActivity
  - Activities can override OnClosed to perform cleanup

    ```
    Called any time the closed state is entered
    Can be called multiple times
    ```



Composite Activity Validators

- Derives from CompositeActivityValidators
  - Adds support for fault and cancel views
  - Often used to restrict types of activities

    Ex. Don't allow any EventDriven activities
    Can check immediate children and descendents

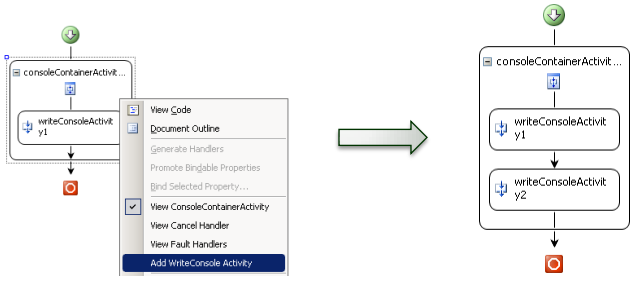
```
internal class ConsoleContainerActivityValidator : CompositeActivityValidator {
    public override ValidationErrorCollection Validate(
        ValidationManager manager, object obj) {
        ...

        ValidationErrorCollection errors = new ValidationErrorCollection();
        foreach (Activity childActivity in activity.EnabledActivities)
            if (!(childActivity is WriteConsoleActivity))
                errors.Add(new ValidationError("...", 100));

        ...
    }
}
```

Activity Designers

- Designers control Activity usage in designer
  - Designers can restrict an activity's parent and children

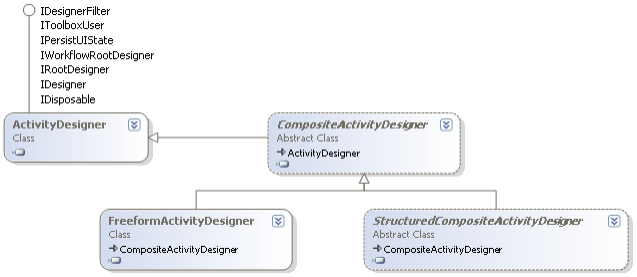
    Insert, Delete events can be controled
  - Designers can control the look and feel of an activity

    Colors, border styles, etc…



Creating Designers

- Custom designers derive from ActivityDesigner
  - More advanced designers can derive from others
  - Methods are overridden to control designer behavior



Restricting Parent Activities

- Designers can restrict an activity's parent activity
  - Done by override CanBeParentedTo
  - Decision made based on the parent designer
  - Often used to keep activity in specific parent activities

```
internal class WriteConsoleActivityDesigner : ActivityDesigner
{
    public override bool CanBeParentedTo(
        CompositeActivityDesigner parentActivityDesigner)
    {
        return (parentActivityDesigner.Activity is
                  ConsoleContainerActivity);
    }
}
```

Restricting Child Activities

- Designers can restrict an activities children
  - Done by overriding several methods

    CanInsertActivities
    CanMoveActivities
    CanDeleteActivities

```
public class ConsoleContainerActivityDesigner :
    SequentialActivityDesigner {
    public override bool CanInsertActivities(
        HitTestInfo insertLocation,
        ReadOnlyCollection<Activity> activitiesToInsert) {
        foreach (Activity activity in activitiesToInsert)
            if (!(activity is WriteConsoleActivity))
                return false;
        return true;
    }
}
```

Custom Designer Verbs

- Activity designers support custom verbs
  - If/Else Activity has Add Branch verb
  - Verbs perform a variety of actions

    Add new branches
    Create new child activities
    Etc…

Adding Custom Verbs

- Verbs defined by the activity's designer
  - Verbs represented by ActivityDesignerVerb
  - List of verbs provided by designer's Verbs property

```
protected override ActivityDesignerVerbCollection Verbs {
    get {
        // create the list containing the verbs
        ActivityDesignerVerbCollection verbs =
                  new ActivityDesignerVerbCollection(base.Verbs);

        // add the new verb
        verbs.Add(
            new ActivityDesignerVerb(this, DesignerVerbGroup.Actions,
                "Add WriteConsole Activity",
                new EventHandler(AddWriteConsole_Click)));

        // return the list of verbs
        return verbs;
    }
}
```

Handling Verb Click Events

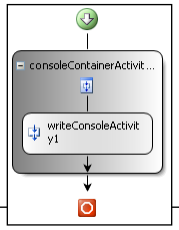
- Verbs have event handler code attached
  - EventHandler delegate attached to verb object
  - When the verb is clicked, the event handler executed

```
private void AddWriteConsole_Click(object sender, EventArgs e)
{
    CompositeActivity activity = base.Activity as CompositeActivity;

    // create the list of new activites
    List<Activity> activities = new List<Activity>();
    activities.Add(new WriteConsoleActivity());

    // determine where to add the activities
    ConnectorHitTestInfo location =
        new ConnectorHitTestInfo(this,
            HitTestLocations.Designer, activity.Activities.Count);

    // add the activities to the designer
    this.InsertActivities(location, activities.AsReadOnly());
}
```

Activity Designer Themes

- Defines visual display characteristics of activity
  - Theme class derives from ActivityDesignerTheme
  - Added to designer with ActivityDesignerThemeAttribute
  - Defines fore and back colors, activity connectors, etc…

```
[ActivityDesignerTheme(typeof(ConsoleContainerActivityTheme))]
public class ConsoleContainerActivityDesigner

internal class ConsoleContainerActivityTheme : CompositeDesignerTheme
{
    public ConsoleContainerActivityTheme(WorkflowTheme theme)
      : base(theme)
    {
        this.ShowDropShadow = true;
        this.BackColorStart = Color.Gray;
        this.BackColorEnd = Color.White;
    }
}
```



Activity Toolbox Items

- Allows definition of activity “package” in toolbox
  - Class derives from ActivityToolboxItem
  - Attached to activity using ToolboxItemAttribute
  - Override CreateComponentsCore to Create activities

    Default implementation creates one activity

```
[ToolboxItem(typeof(ConsoleContainerActivityToolboxItem))]
public class ConsoleContainerActivity : SequenceActivity

public class ConsoleContainerActivityToolboxItem : ActivityToolboxItem
{
    protected override IComponent[] CreateComponentsCore(
        IDesignerHost host) {
        ConsoleContainerActivity container =
            new ConsoleContainerActivity();
        container.Activities.Add(new WriteConsoleActivity());
        return new IComponent[] { container };
    }
}
```

Summary

- Create activities
  - Events and Dependency Properties
  - Custom Activity Validators
  - Custom Activity Designers
- Understanding Activity Lifecycle
- Creating Composite Activities

Extending SharePoint Designer with Custom Activities

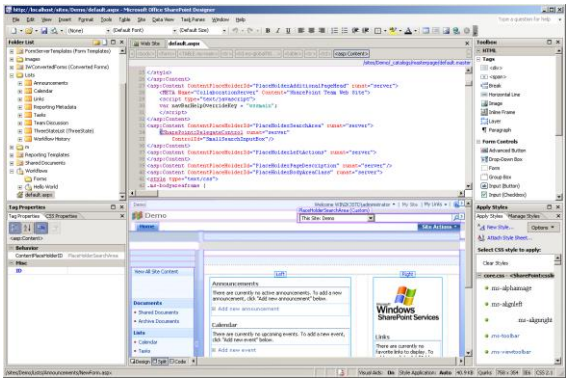
Developing SharePoint Workflow Templates with Visual Studio

Agenda

- SharePoint Designer vs. VS 2008
- No Code Workflows
- Building Workflows in SharePoint Designer
- Extending SharePoint Designer with Custom Activities

What is the SharePoint Designer?

- SPD is designed for non developers
  - Allows WYSIWYG page management
  - Allows rule based workflow development

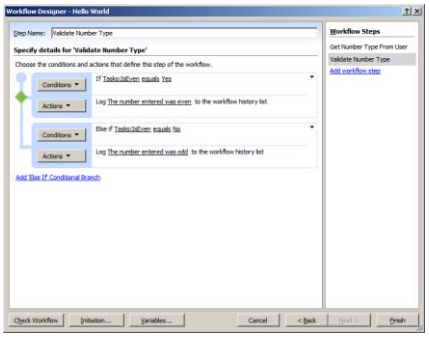


Which is better? VS 2008 or SPD

- SPD is primarily a non-developer tool
  - Allows tasks to be done easily, not efficiently
  - Modifications to content not easily redeployed
  - Useful for per-site modifications
- Rule of thumb
  - SPD for end user and non deployable solutions
  - VS 2008 for deployable solution packages

No Code Workflows

- Declarative definition made up of multiple files
  - Wfconfig.xml referencing all needed iles
  - XOML file containing the workflow
  - ASPX files defining the workflow forms



Wfconfig.xml file

- Defines the structure of the workflow
  - Defines workflow template and association
  - Defines the location and data layout of all forms
  - Always named <workflow file>>.wfconfig.xml

```
<WorkflowConfig>
  <Template
      BaseID="{8F5D9FAF-6F2A-4EF9-9A70-5E7CD49C294D}"
      DocLibID="{1891A50F-275C-4180-B338-6B589BDD04F6}"
      XomlHref="Workflows/Hello World/Hello World.xoml“
      RulesHref="Workflows/Hello World/Hello World.xoml.rules"
      ... >
  </Template>
  <Association
      ListID="{079FE649-0788-4229-8996-8863DBA5DCD7}"
      TaskListID="{0B84F02E-031D-48ED-BAAC-D8A878CFADBC}"
      StartManually="true”>
  </Association>
```

Wfconfig.xml file

```
<ContentTypes>
  <ContentType Name="Get Number Type" ContentTypeID="0x01080100…"
    Form="/Workflows/Hello World/Get Number Type.aspx" ...>
    <Fields>
      <Field DisplayName="IsEven" ...>
        <Default>1</Default>
      </Field>
    </Fields>
  </ContentType>
</ContentTypes>
<Initiation URL="Workflows/Hello World/Hello World.aspx">
  <Fields>
    <Field Name="Value" DisplayName="Value" Type="Number" ...>
      <Default>0</Default>
    </Field>
  </Fields>
  <Parameters>
    <Parameter Name="Value" Type="System.Double" />
  </Parameters>
</Initiation>
</WorkflowConfig>
```

Deploying No Code Workflows

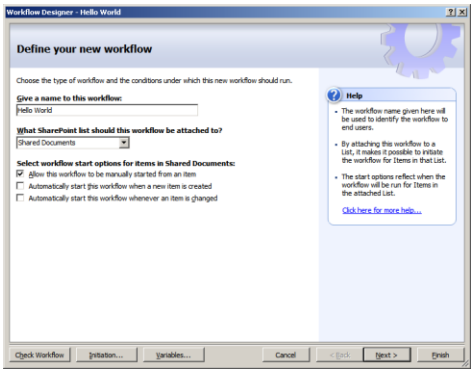
- Deployment handled via web service
  - websvcWebPartPages exposes workflow methods

    `FetchLevelWorkflowActions`
    `ValidateWorkflowMarkupAndCreateSupportObjects`
    `AssociationWorkflowMarkup`

- Used by SharePoint Designer to create workflow

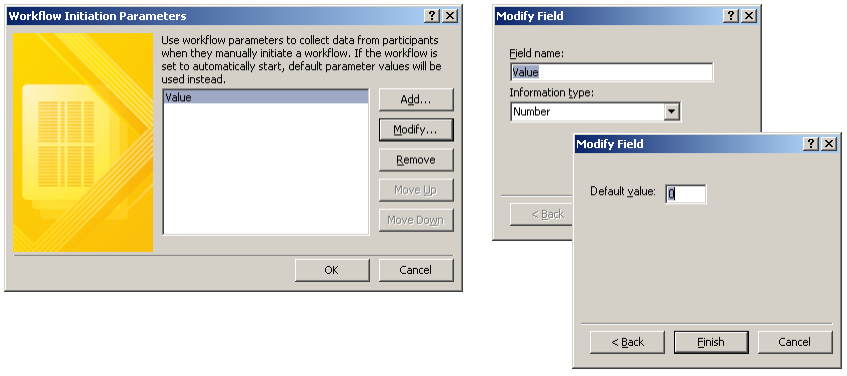
Creating SPD Workflow

- Association created automatically
  - List and startup parameters chosen at design time



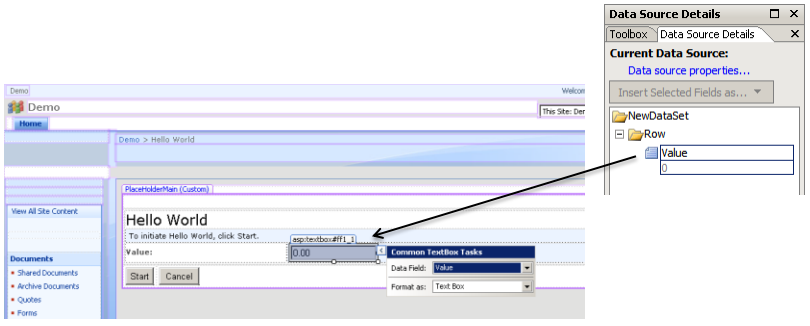
Workflow InstantiationForm Definition

- Instantiation form generated automatically
  - Initiation form made up of a set of fields
  - Each field has a name and type



Customizing Instantiation Form

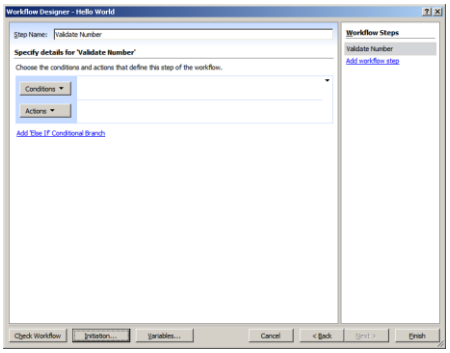
- Generated form designed as an .ASPX page.
  - Initiation form fields available in Data Source pane
  - Controls displaying fields can be formatted
  - Page is designed just like any other .ASPX page



Building Workflow in SPD

- SharePoint Designer provides workflow builder
  - Build workflow based on basic rules
  - Rules made up of conditions and actions

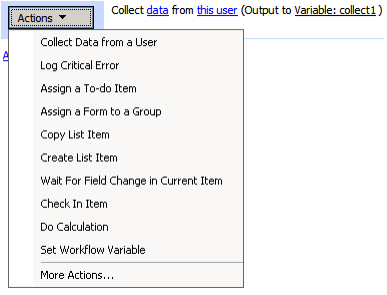
    `Similar to Outlook rules`

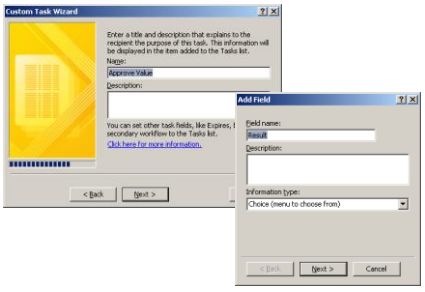


Workflow Task Form Definitions

- Tasks handled automatically by designer
  - Single action creates task and waits for completion

    `Entire task process in one activity`
  - Form is generated automatically based on fields

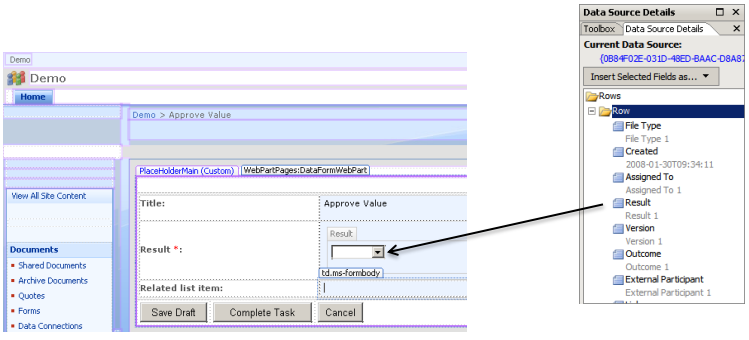




Customizing Task Form

- Generated by workflow wizard as .ASPX page
  - Same process as editing Instantiation form

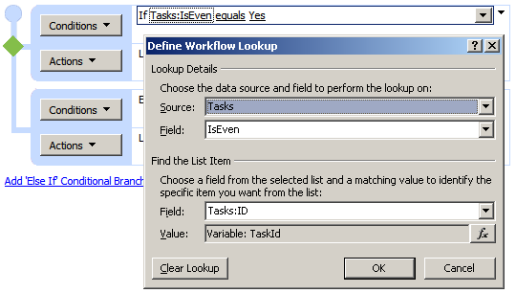
    `Different data source`
  - Task form fields available in Data Source pane



Adding Conditions

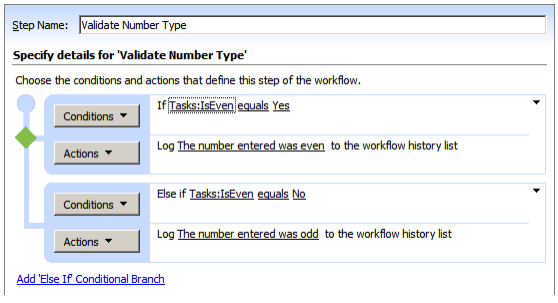
- Conditions determine if related actions execute
  - Conditions can compare fields or constant values
  - Fields defined as fields from forms or current item
  - Other lists can be accessed using simple “queries”

```
Task data is stored in Tasks list
```



Multiple Conditions

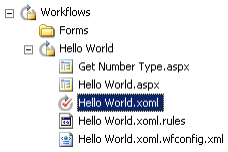
- Steps can contain multiple condition/actions
  - Each condition executed separately
  - Allows if/else structure in workflow rules

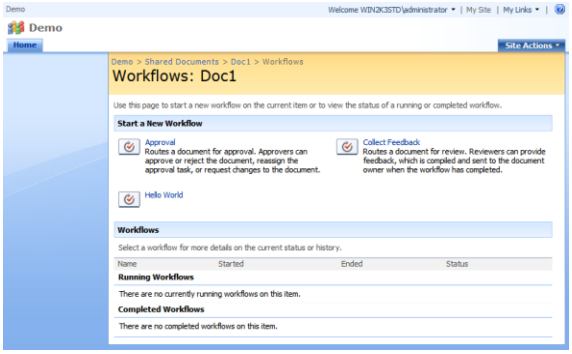


Completed Workflow

- Workflow stored in SharePoint site
  - All files needed are generated automatically

    `Hard coded lists make deployment difficult`
  - Workflow initiated same as any other workflow





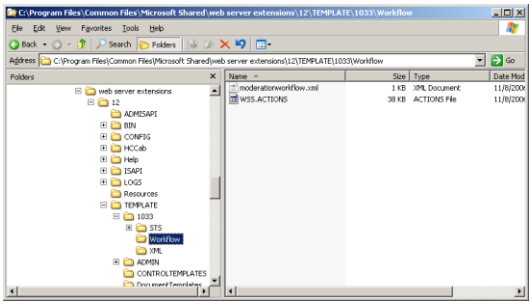
What are Conditions and Actions?

- Conditions and Actions are shortcuts
  - Groups smaller actions into larger functional pieces
  - Allows non-developers to avoid fine grained activities
  - Developers build actions for non-developers to use

Developers	Non-Developers
Build Custom Actions Build Custom Conditions	Build Workflows • Use Custom Actions • Use Custom Conditions

WSS.ACTIONS

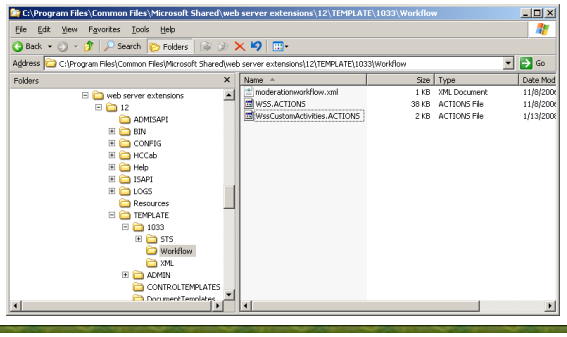
- Contains definitions for conditions and actions
  - Maps conditions to methods
  - Maps actions to custom activities
  - Parameters are mapped to user friendly options
  - Resides in TEMPLATE\1033\Workflow folder



Custom Conditions and Actions

- Add custom items by creating second .actions file
  - Must have the .actions extension
  - Made up of a conditions and actions section
  - Must exist in the TEMPLATE\1033\Workflow folder

    `1033 is US English, for others use appropriate ID`



Defining Custom Conditions

- Custom conditions are static methods
  - Method must have specific signature

    Must return bool

    Must accept context parameters

```
public class WssWorkflowConditions
{
    public static bool IsEven(WorkflowContext context,
        string listId, int itemId, double data)
    {
        return ((int)data % 2) == 0;
    }
```

Custom Conditions in .ACTIONS

- .ACTIONS maps methods to sentences
  - Sentences define placeholders for parameters
  - Placeholders are mapped to method parameters

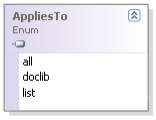
```
<Condition Name="Is even" FunctionName="IsEven" AppliesTo="all"
    ClassName="WssCustomActivities.WssWorkflowConditions"
    Assembly="WssCustomActivities, ...">
      <RuleDesigner Sentence="%1 is even">
        <FieldBind Id="1" Field="_1_" Text="value" />
      </RuleDesigner>
      <Parameters>
        <Parameter Name="_1_" Direction="In"
                   Type="System.Double, mscorlib" />
      </Parameters>
    </Condition>
```

Scoping Conditions

- Most conditions and activities apply globally
  - What if a they only work on a list item?
  - .ACTIONS files allow scoping to specific types
  - Defined in the AppliesTo attribute

    Applies to conditions and activities

```
<Action ... AppliesTo="all" >
```



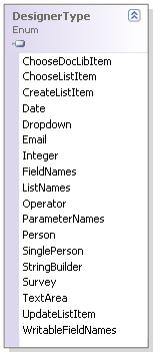
Rule Designers

- Designer made up of sentence and fields
  - Sentence defines what the user sees
  - Placeholders replaced by parameters in designer

    Click placeholder to select parameter
  - Dialog displayed depends on DesignerType

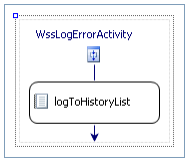
    Defaults to field chooser

```
<RuleDesigner Sentence="Log '%1' as critical error">
  <FieldBind Field="ErrorMessage" Id="1"
             DesignerType="TextArea" />
</RuleDesigner>
```



Defining Custom Actions

- Custom Activities are defined as WinWF Activities
  - Activities are often designer based Sequential activities
    `Although any activity will work`
  - Exposes parameters as dependency properties




```
public static DependencyProperty ErrorMessageProperty =
    DependencyProperty.Register("ErrorMessage",
    typeof(System.String),
    typeof(WssLogErrorActivity));

public String ErrorMessage
{
    get { ... }
    set { ... }
}
```

Custom Activities in .ACTIONS

- .ACTIONS maps Activities to sentences
  - Sentences define placeholders for parameters
  - Placeholders are mapped to method parameters
  - All parameters must be mapped to dependency properties

```
<Action Name="Log Critical Error" AppliesTo="all" Category="Core Actions"
        ClassName="WssCustomActivities.WssLogErrorActivity"
        Assembly="WssCustomActivities, ..." >
  <RuleDesigner Sentence="Log '%1' as critical error">
    <FieldBind Field="MyErrorMessage" DesignerType="TextArea" Id="1" />
  </RuleDesigner>
  <Parameters>
    <Parameter Name="MyErrorMessage" Direction="In"
               Type="System.String, mscorlib" />
  </Parameters>
</Action>
```

Special Activity Parameters

- Special parameters are available to activities
  - Defines the workflow context, current list or item
  - Special names required in the .ACTIONS file

```
<Parameter Name="__Context"
  Type="Microsoft.SharePoint.WorkflowActions.WorkflowContext, ..."
  Direction="In"/>
<Parameter Name="__ListId" Type="System.String, ..."
  Direction="In" />
<Parameter Name="__ListItem" Type="System.Int32, ..."
  Direction="In" />
```

Working with .ACTIONS files

- SP Designers accessed data via web services
  - Changes to .ACTIONS requires AppPool recycle
  - Changes do not require SP Designer restart

- Common Problems
  - Can't load custom conditions and activities

    `Make sure Method, Class, Assembly Names correct`
  - Can see custom items, but can't use them

    `Make sure parameter names and types match`

Summary

- SharePoint Designer vs. VS 2008
- No Code Workflows
- Building Workflows in SharePoint Designer
- Extending SharePoint Designer with Custom Activities

Lab 02: Windows Workflow Foundation Integration with SharePoint 2007

Lab Overview: Litware Inc. has been using SharePoint 2003 to store documents for several years and has recently upgraded to SharePoint 2007. They have read about the integrated workflow support in SharePoint 2007. Based on what they have read, they would like to evaluate it using a simple Approval process.

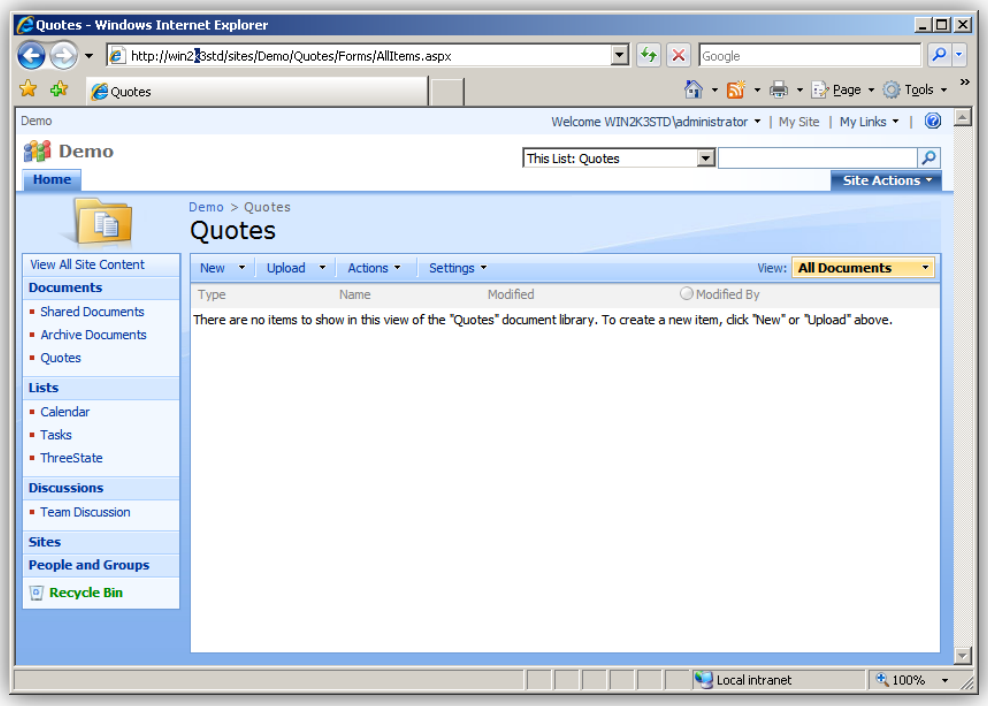
First, you will need to create a document library containing quotes submitted by others. Next, attach an Approval workflow to the Quotes document library. Configure the SharePoint to start a new Approval process for each new quote added to the list.

Exercise 0: Setting up the project

1) Open SharePoint and browse to the **Demo** site collection
   - The url is **http://litwareinc.com/sites/Demo**.
   - If the site collection does not exist, create it using the **CreateDemo.bat** file in the **C:\Labs\Files** folder.

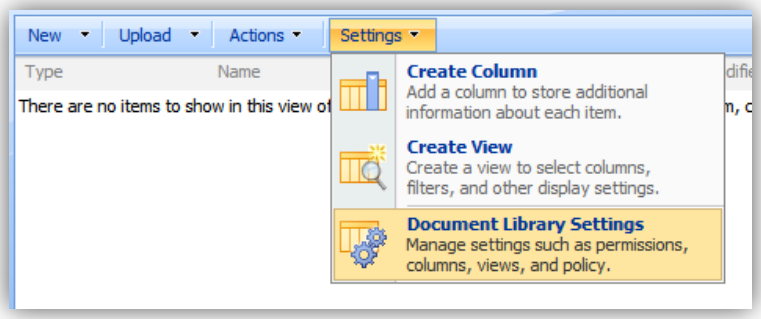
Exercise 1: Attaching an Approval Workflow

1) Create a new **Quotes** document library.
   - Click the **Site Actions -> Create** menu option.
   - On the **Create** page, click the **Document Library** link under the **Libraries** section header.
   - Enter a name of **Quotes** and click the **Create** button to create the new document library.

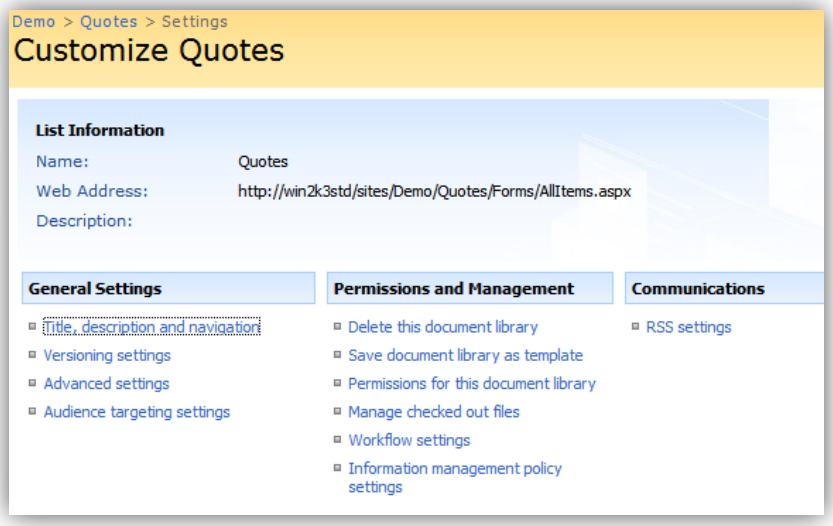


2) Associate an **Approval** workflow named **Quote approval** to the new **Quotes** document library.

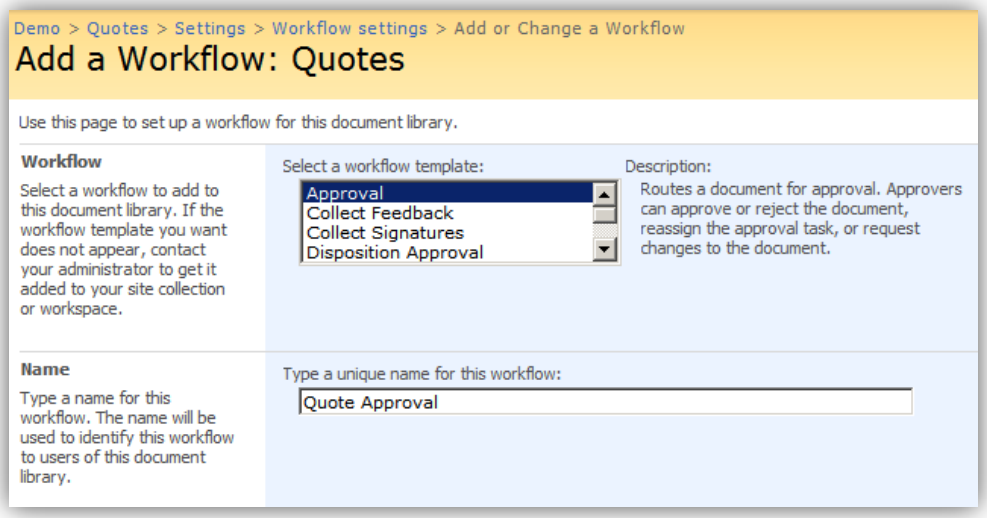
- Starting from the **Quotes** document library, click the **Settings -> Document Library Settings** link.



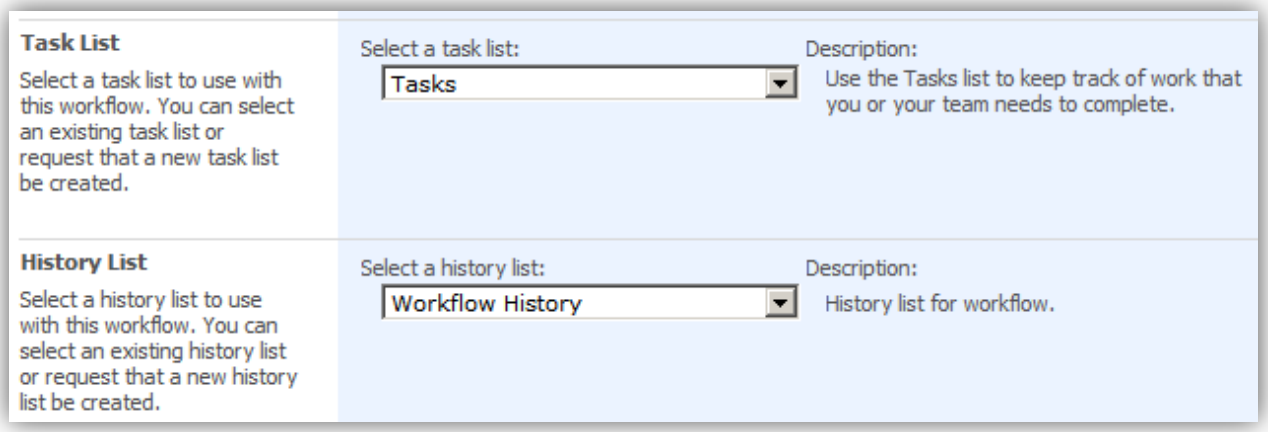
- On the **Customize** page, click the **Workflow settings** link in the **Permissions and Management** section header.



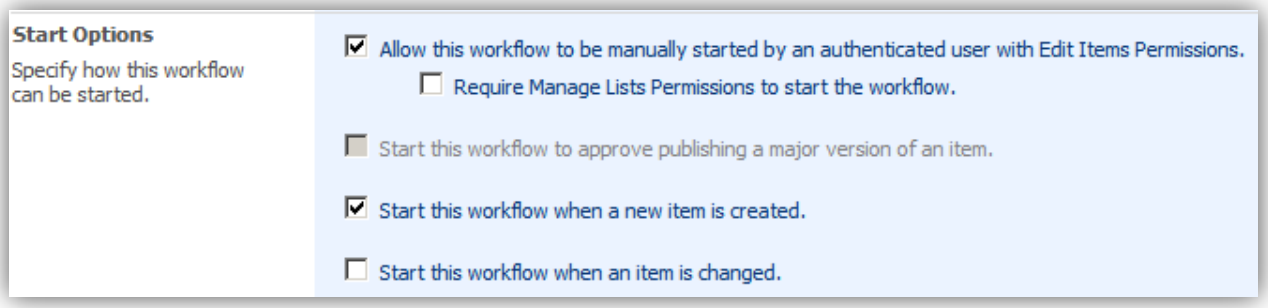
- Choose the **Approval** workflow template and enter a name of **Quote Approval** in the name text box.



- Choose the existing **Tasks** list for the task list and **Workflow History** list for the history list.

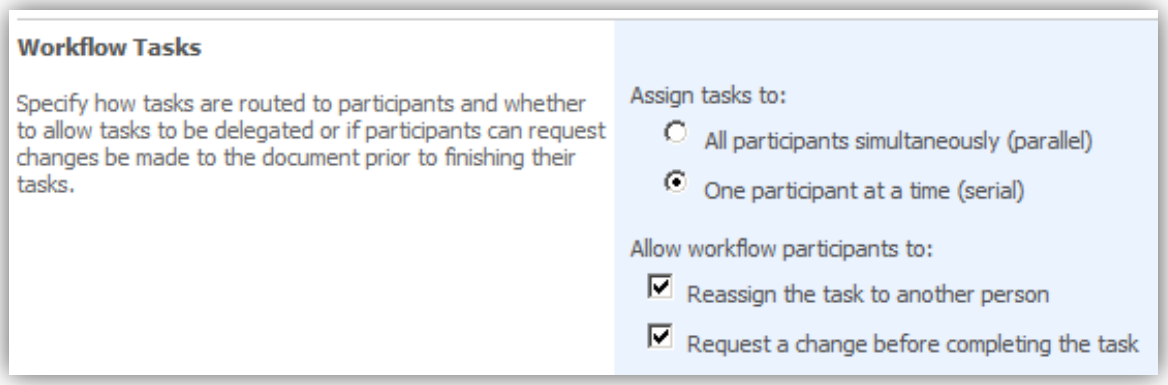


- Choose the **Allow this workflow to be manually started** checkbox to allow manual workflow starting. Also choose the **Start this workflow when a new item is created** option to automatically start the workflow.

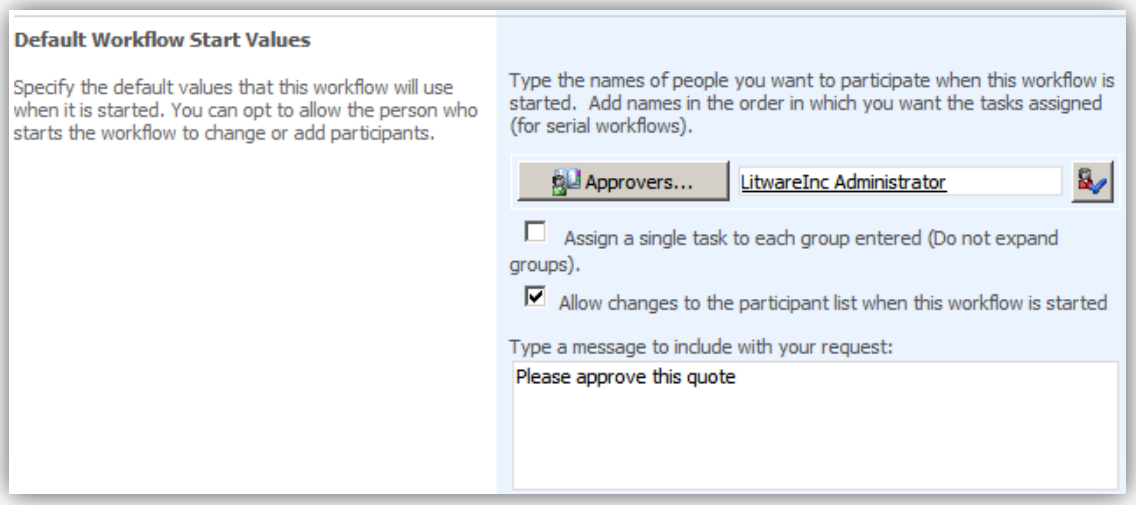


- Click **Next** to move to the next step.

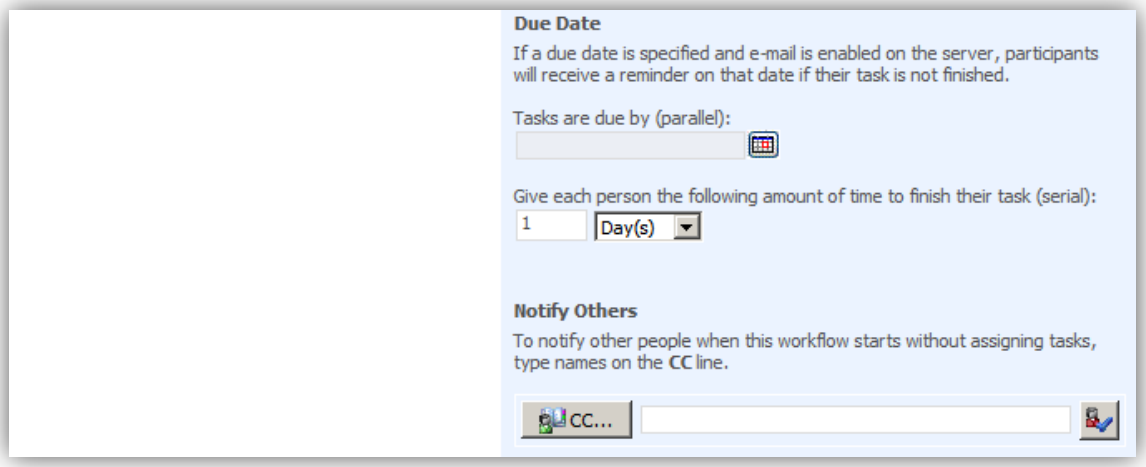
3) Define the **Approval** workflow specific parameters needed to finish the association of the **Quote Approval** workflow.
   - In the **Workflow Tasks** section, leave the options as they are allowing only one approver to approve at a time and to give the approver the option of reassigning approval or resubmitting to the original submitter.



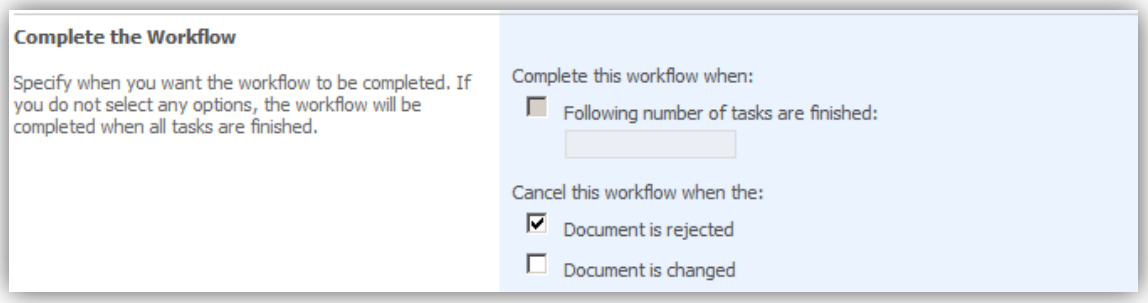
- In the **Default Workflow Start Values** section, add the **Administrator** to the approver list. If you'd like, enter a default message as well.



- At the bottom of the **Default Workflow Start Values** section give each approver 1 day to complete their task and don't notify anyone else of their task.



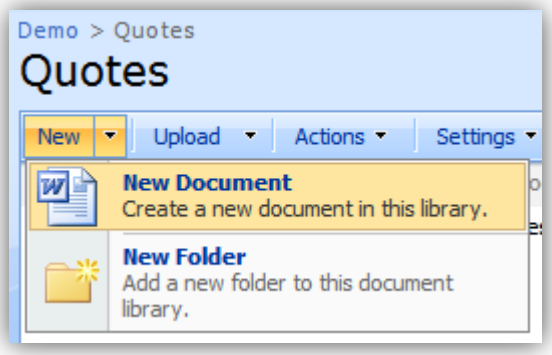
- In the **Complete the Workflow** section, check the **Document is rejected** option to cancel the workflow if any approver rejects the quote.



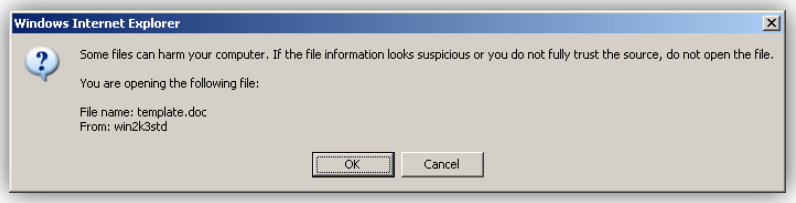
- Click **OK** to complete the workflow association.

4) Add a document to the **Quotes** document library.

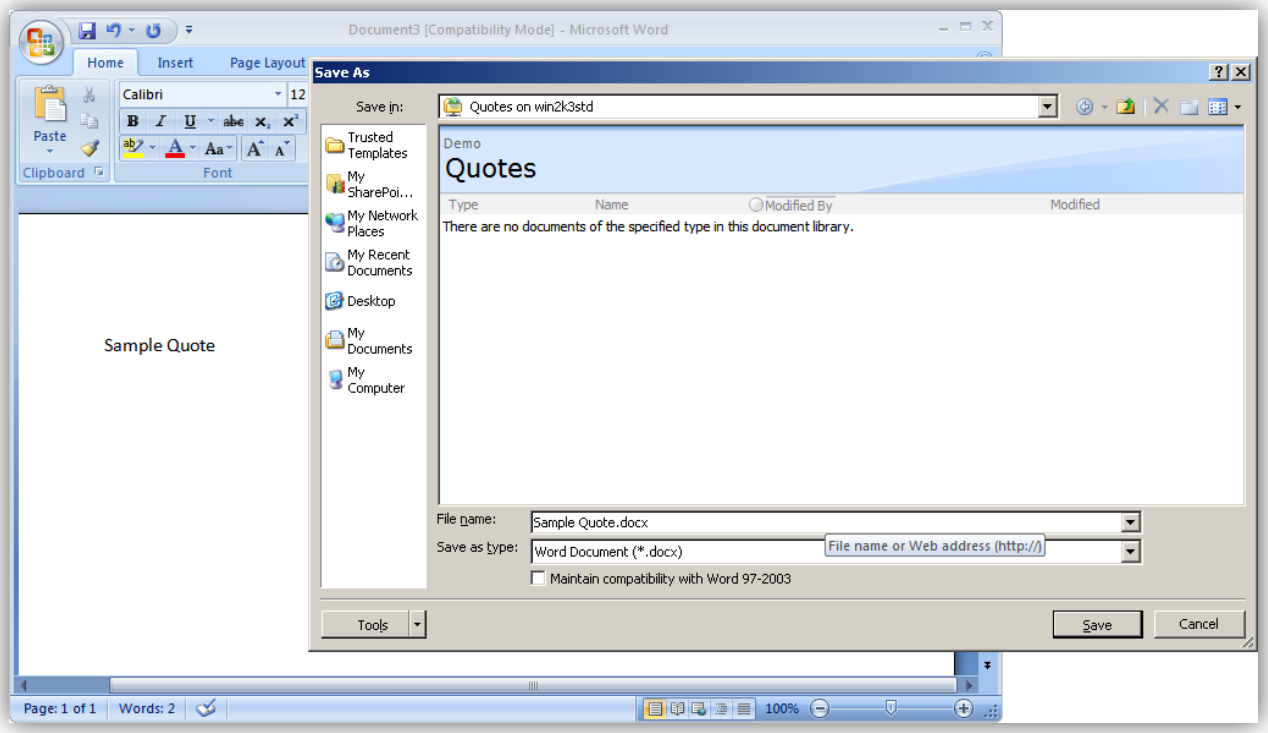
- Navigate to the **Quote** document library using the navigation links at the top of the page.
- Add a new **Quote** to the document library by clicking the **New -> New Document** button.



- If the warning dialog appears, click **OK** to open the new document.

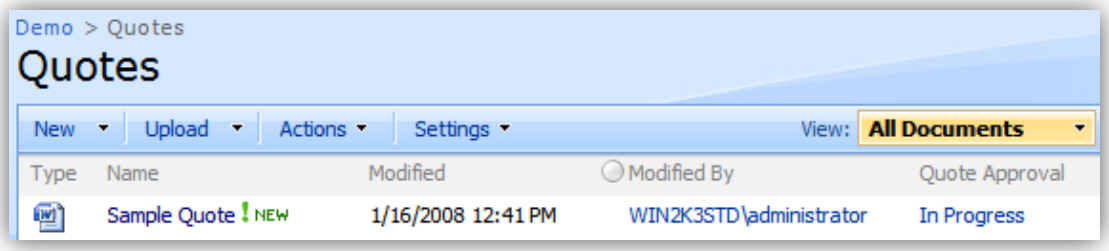


- In Word 2007, enter some text into the document and click **Save**. Save the document using the name **Sample Quote.docx** to the **Quotes** library.



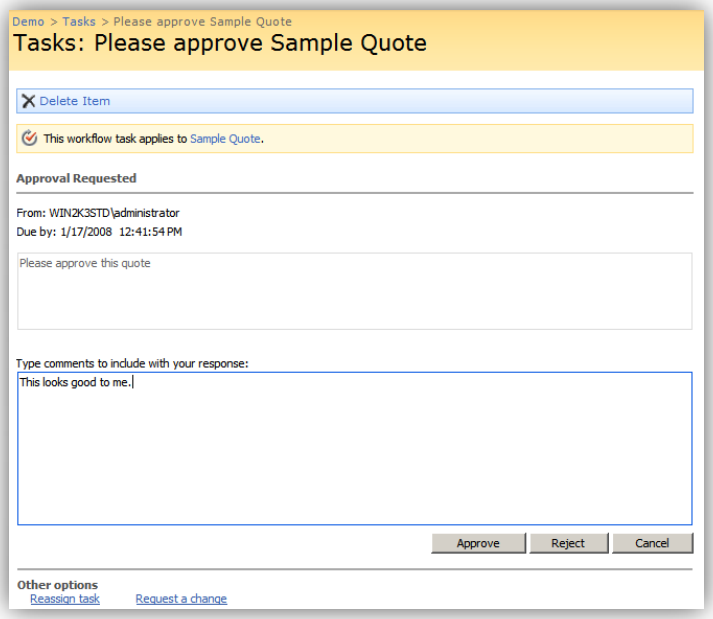
5) Approve the new quote.

- Switch back to the **Quotes** document library in **Internet Explorer** and refresh the page. You should see the new document with the **Quote Approval** column set to **In Progress**.



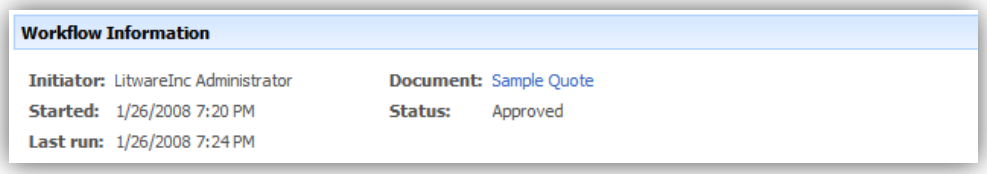
- Click the **In Progress** link to view the workflow instance's statistics.
- In the **Tasks** section, locate the task item in the list and click its **Title**.

- In the task form, enter some comments in the text box and click the **Approve** button to approve the workflow.

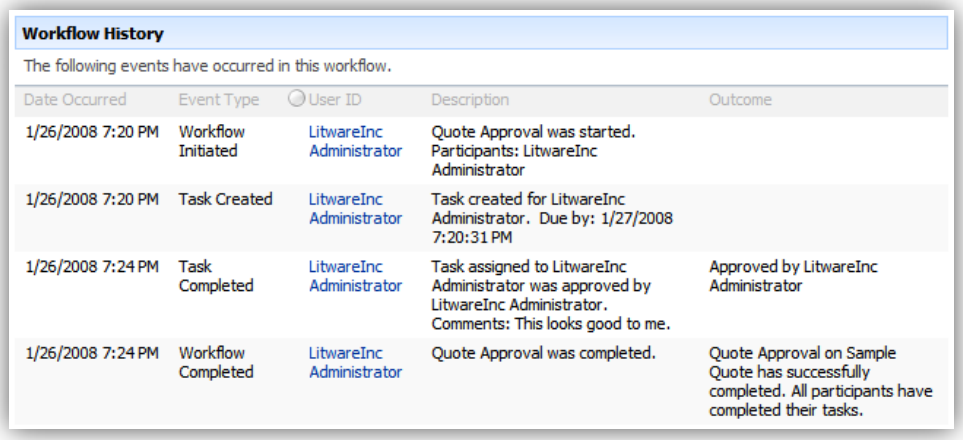


6) View the quote's approval status.

- In the **Workflow Status** form, verify the **Status** in the **Workflow Information** section shows **Approved**.

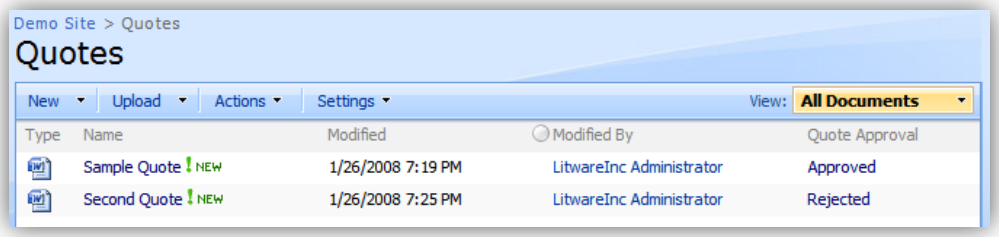


- In the **Workflow History** section, you should see a list of four history list items detailing the steps taken by the **Approval** workflow.



7) Create another document and repeat the process with the exception of **Rejecting** the task instead of approving it.

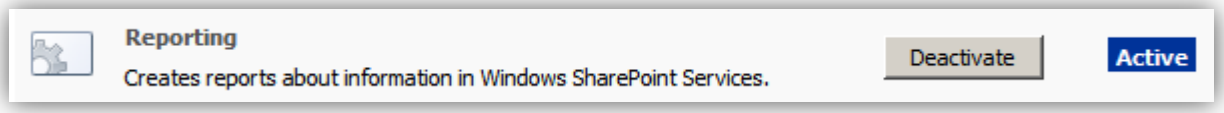
- Name the document **Second Quote.docx**.



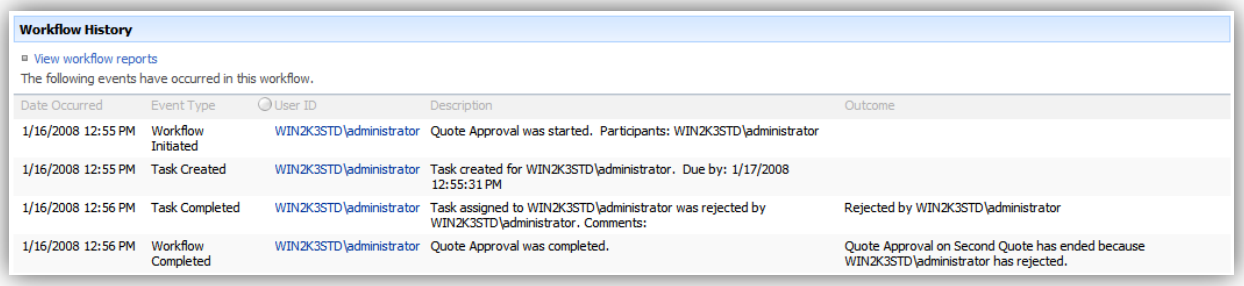
8) View statistics on all workflow instances

- Enable the **Reporting** Site collection feature.

  *Navigate to **Site Actions -> Site Settings**.*
  *Click the **Site collection features** link in the **Site Collection Administration** section.*
  *Verify that the **Reporting** feature is active.*



- Navigate to the **Quotes** document library and click the status link on one of the quotes.



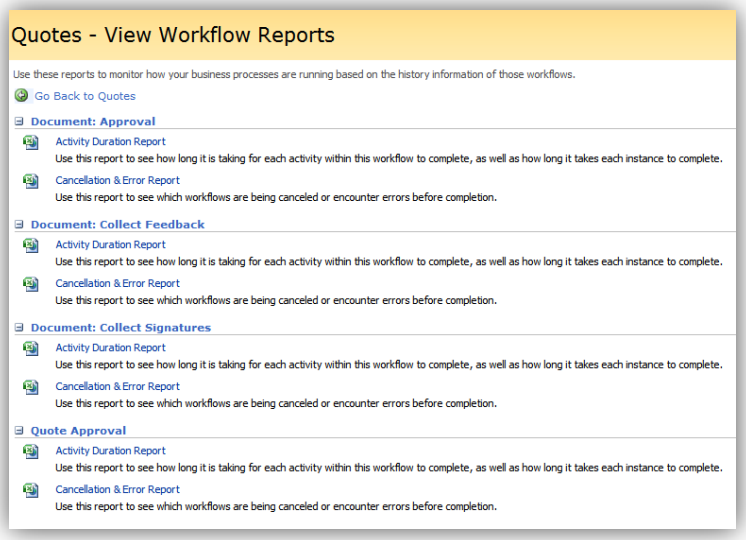
Workflow History

View workflow reports

The following events have occurred in this workflow.

Date Occurred	Event Type	User ID	Description	Outcome
1/16/2008 12:55 PM	Workflow Initiated	WIN2K3STD\administrator	Quote Approval was started. Participants: WIN2K3STD\administrator	
1/16/2008 12:55 PM	Task Created	WIN2K3STD\administrator	Task created for WIN2K3STD\administrator. Due by: 1/17/2008 12:55:31 PM	
1/16/2008 12:56 PM	Task Completed	WIN2K3STD\administrator	Task assigned to WIN2K3STD\administrator was rejected by WIN2K3STD\administrator. Comments:	Rejected by WIN2K3STD\administrator
1/16/2008 12:56 PM	Workflow Completed	WIN2K3STD\administrator	Quote Approval was completed.	Quote Approval on Second Quote has ended because WIN2K3STD\administrator has rejected.

- Click the **View workflow reports** in the **Workflow History** section.



Quotes - View Workflow Reports

Use these reports to monitor how your business processes are running based on the history information of those workflows.

Go Back to Quotes

Document: Approval

Activity Duration Report
Use this report to see how long it is taking for each activity within this workflow to complete, as well as how long it takes each instance to complete.

Cancellation & Error Report
Use this report to see which workflows are being canceled or encounter errors before completion.

Document: Collect Feedback

Activity Duration Report
Use this report to see how long it is taking for each activity within this workflow to complete, as well as how long it takes each instance to complete.

Cancellation & Error Report
Use this report to see which workflows are being canceled or encounter errors before completion.

Document: Collect Signatures

Activity Duration Report
Use this report to see how long it is taking for each activity within this workflow to complete, as well as how long it takes each instance to complete.

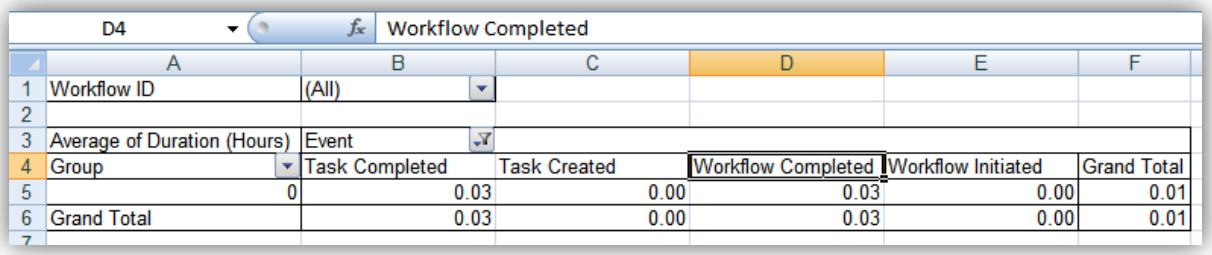
Cancellation & Error Report
Use this report to see which workflows are being canceled or encounter errors before completion.

Quote Approval

Activity Duration Report
Use this report to see how long it is taking for each activity within this workflow to complete, as well as how long it takes each instance to complete.

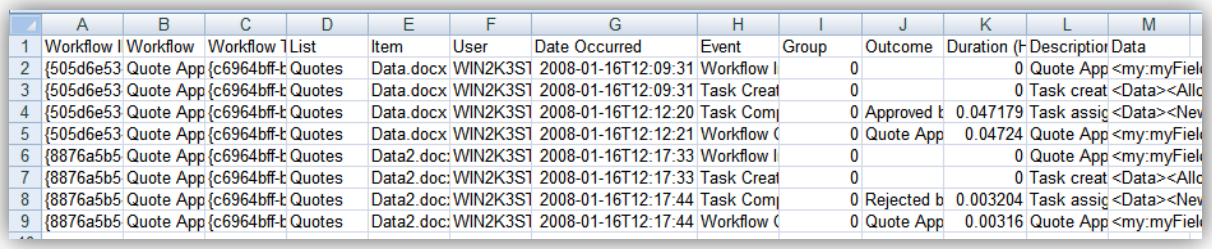
Cancellation & Error Report
Use this report to see which workflows are being canceled or encounter errors before completion.

- In the **Quote Approval** section, locate and click the **Activity Duration Report** link to generate an Excel report. The first page contains a pivot chart summarizing the raw data. Switch to the second page to see the raw data the pivot table is based on.



D4 — Workflow Completed

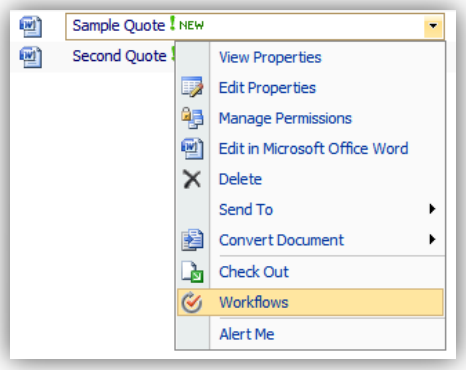
	A	B	C	D	E	F
1	Workflow ID	(All)				
2						
3	Average of Duration (Hours)	Event				
4	Group	Task Completed	Task Created	Workflow Completed	Workflow Initiated	Grand Total
5	0	0.03	0.00	0.03	0.00	0.01
6	Grand Total	0.03	0.00	0.03	0.00	0.01



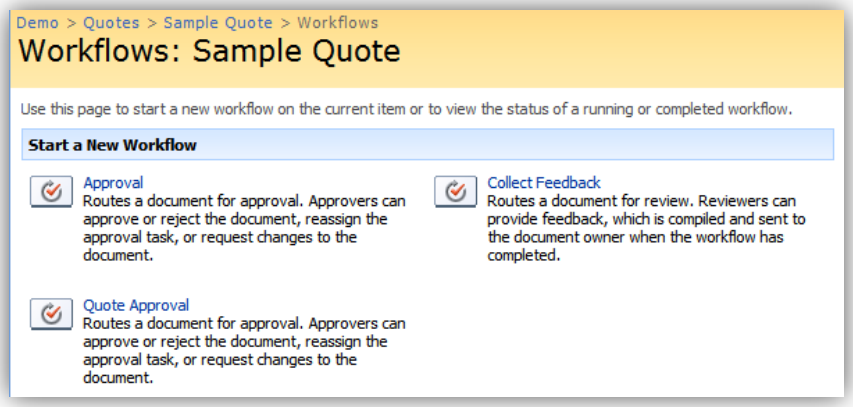
	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Workflow I	Workflow	Workflow T	List	Item	User	Date Occurred	Event	Group	Outcome	Duration (H	Descriptior	Data
2	{505d6e53	Quote App	{c6964bff-b	Quotes	Data.docx	WIN2K3ST	2008-01-16T12:09:31	Workflow I	0		0	Quote App	<my:myFiel
3	{505d6e53	Quote App	{c6964bff-b	Quotes	Data.docx	WIN2K3ST	2008-01-16T12:09:31	Task Creat	0		0	Task creat	<Data><Allc
4	{505d6e53	Quote App	{c6964bff-b	Quotes	Data.docx	WIN2K3ST	2008-01-16T12:12:20	Task Comp	0	Approved b	0.047179	Task assig	<Data><Nev
5	{505d6e53	Quote App	{c6964bff-b	Quotes	Data.docx	WIN2K3ST	2008-01-16T12:12:21	Workflow (	0	Quote App	0.04724	Quote App	<my:myFiel
6	{8876a5b5	Quote App	{c6964bff-b	Quotes	Data2.doc:	WIN2K3ST	2008-01-16T12:17:33	Workflow I	0		0	Quote App	<my:myFiel
7	{8876a5b5	Quote App	{c6964bff-b	Quotes	Data2.doc:	WIN2K3ST	2008-01-16T12:17:33	Task Creat	0		0	Task creat	<Data><Allc
8	{8876a5b5	Quote App	{c6964bff-b	Quotes	Data2.doc:	WIN2K3ST	2008-01-16T12:17:44	Task Comp	0	Rejected b	0.003204	Task assig	<Data><Nev
9	{8876a5b5	Quote App	{c6964bff-b	Quotes	Data2.doc:	WIN2K3ST	2008-01-16T12:17:44	Workflow (	0	Quote App	0.00316	Quote App	<my:myFiel

Exercise 2: Starting the Workflow Manually

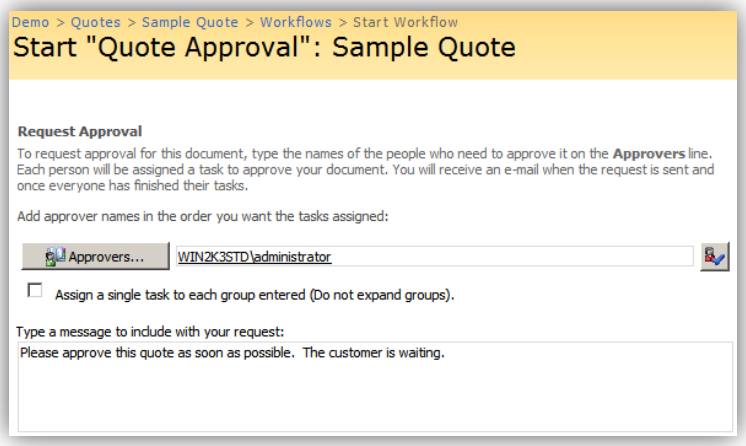
1) Start the **Quote Approval** workflow manually.
   - Navigate to the **Quotes** document library.
   - Hover over the **Sample Quote** document name and click the **Workflows** option in the drop down.



   - On the **Workflows** page click the **Quote Approval** workflow to start a new instance of it.



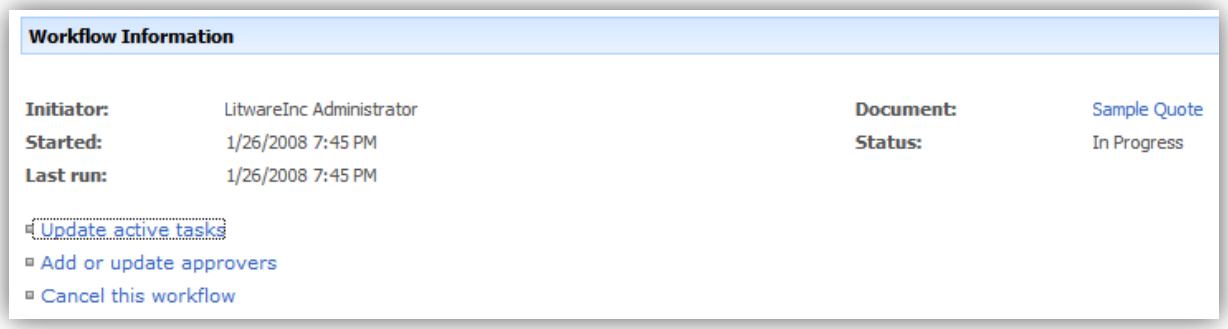
   - On the **Start Workflow** page, leave the approvers list the same, but change the message to something other than the default.



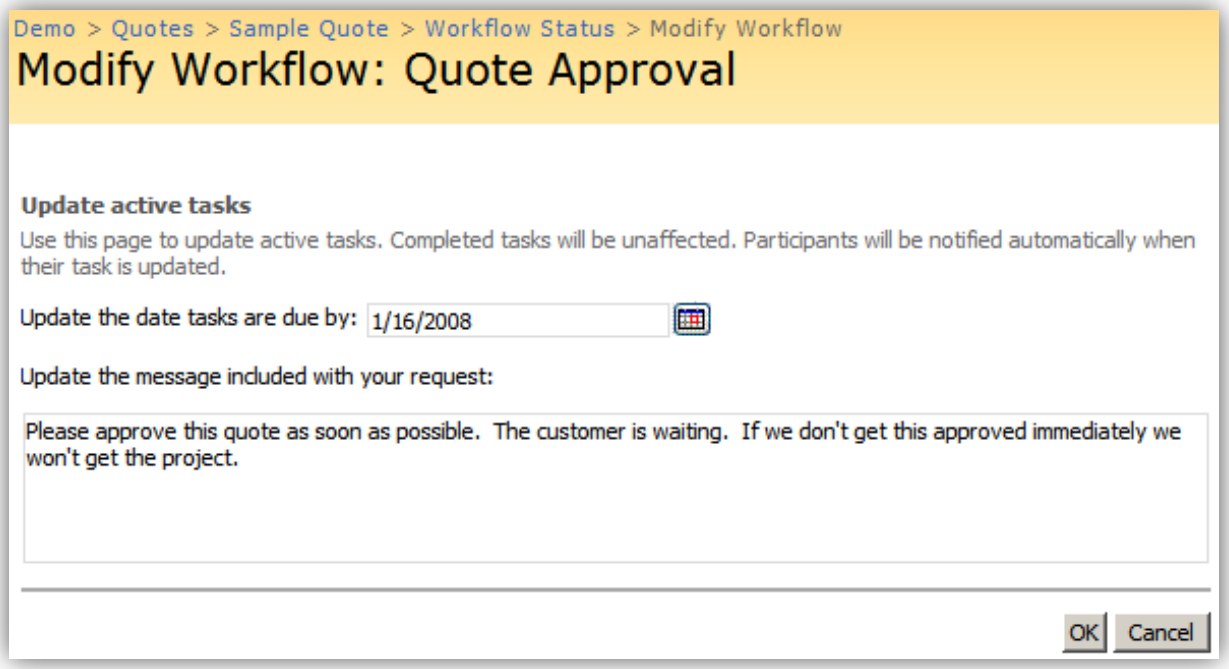
   - Click the **Start** button to start the workflow.

2) Modify the running workflow instance.

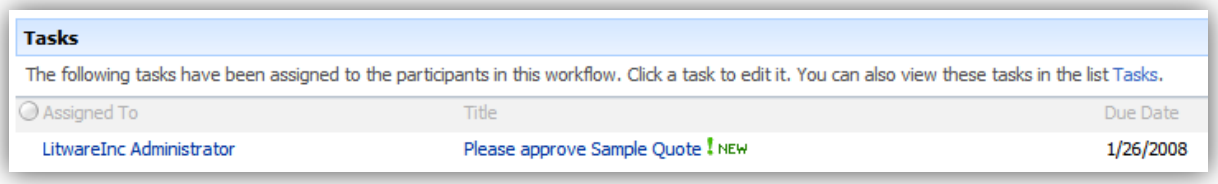
- Click the **In Progress** link on the **Sample Quote** document to view the workflow instance's status.
- In the **Workflow Information** section, click the **Update active tasks** list to update all current tasks for the workflow instance.



- Change the due date of the tasks to today's date and update the message to indicate the change.

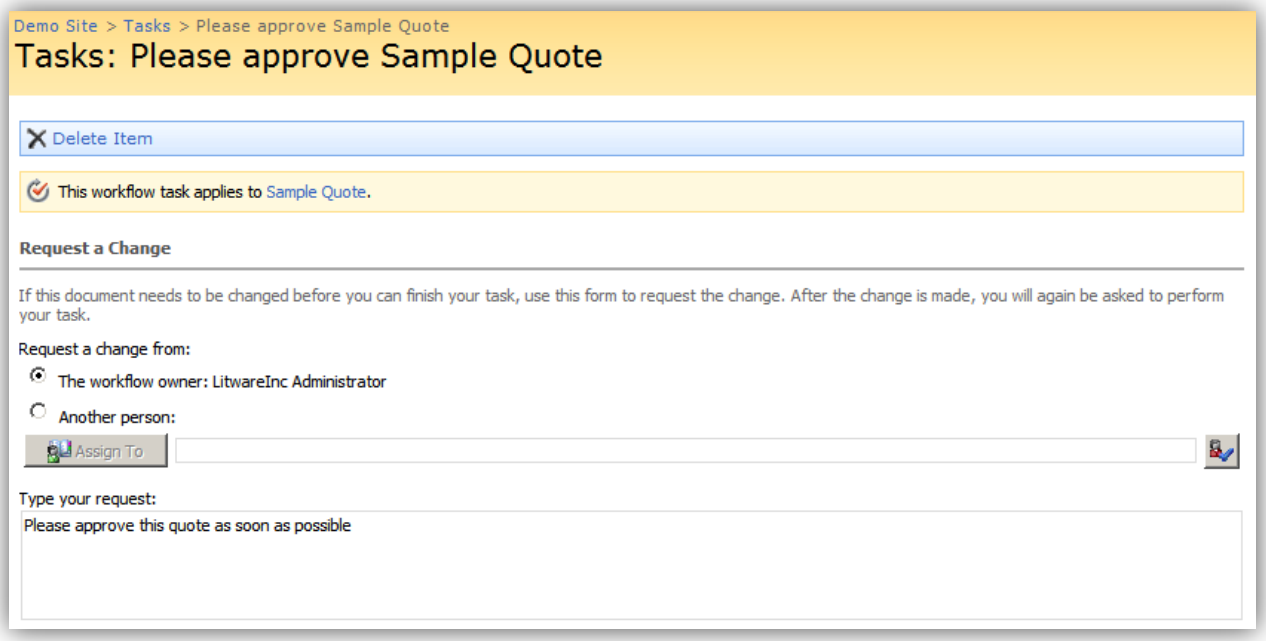


- Click **OK** to commit the changes.
- Verify the due date of the task has changed to today's date.

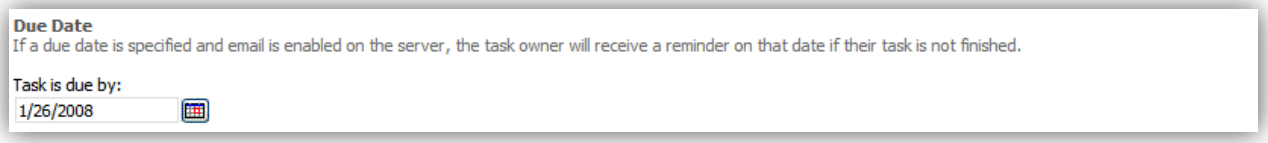


3) Request a change to the workflow.

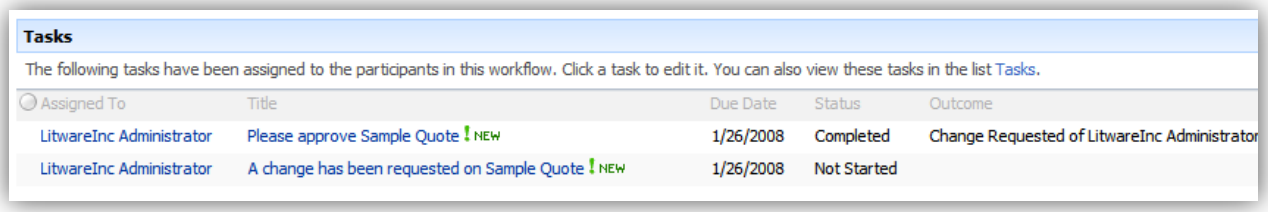
- On the **Workflow Status** page In the **Tasks** section, click the title of the approval task.
- On the tasks form, click the **Request** a change link at the bottom of the form.
- On the new **Tasks** page that is loaded, leave the user to request the change from as the workflow's owner.



- Set the due date to today's date.

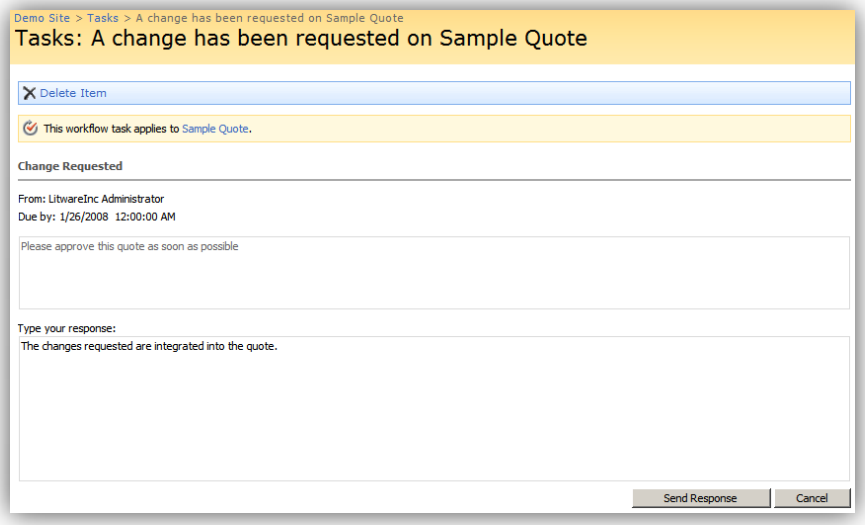


- Click **Send** to submit the new task.
- A new task will exist in the **Tasks** list requesting the change.

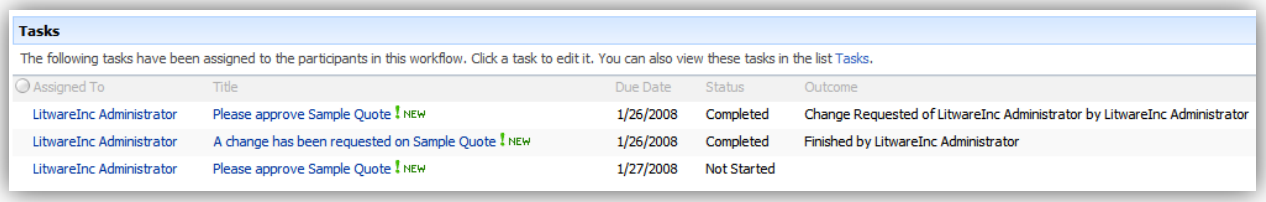


4) Complete the request change task.

- In the **Tasks** section, click the **Title** of new task.
- Enter the response as if you'd made the changes and click **Send Response**.

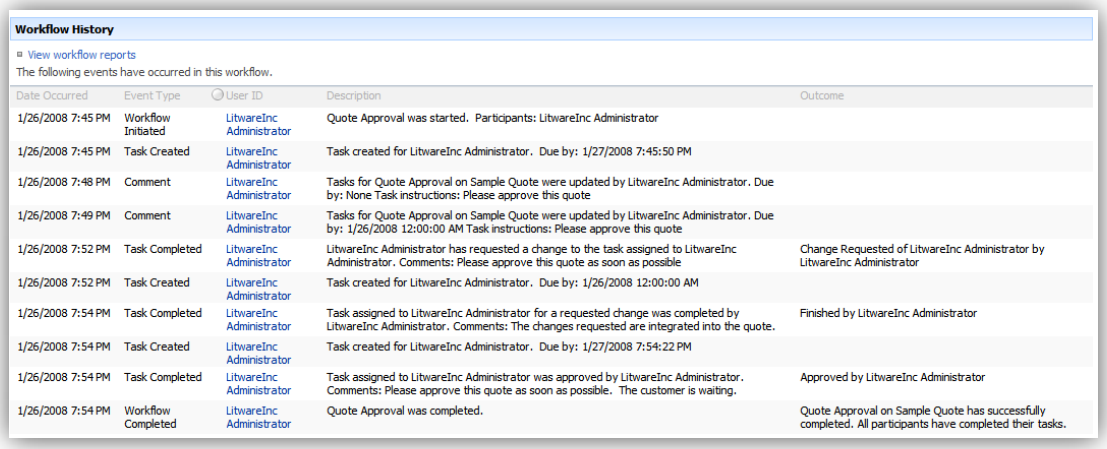


- In the **Tasks** section, verify the request change task is completed and a new approval task exists.



5) Complete the approval task.

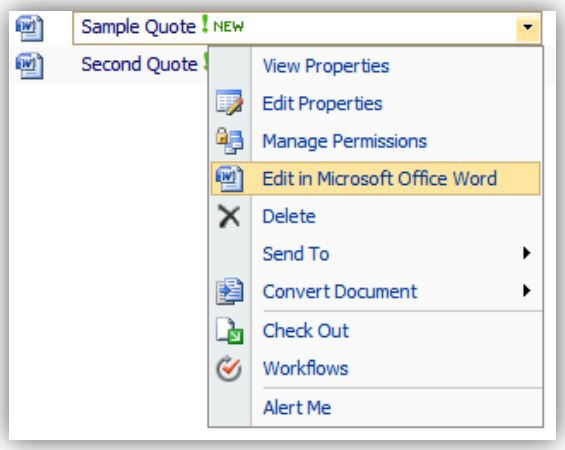
- Complete the new approval task by approving the tasks, just like in the previous exercise.
- The workflow's status in the **Workflow Information** section should be **Approved** and the **Workflow History** section should show the entire process.



Exercise 3: Executing Workflow using Office 2007

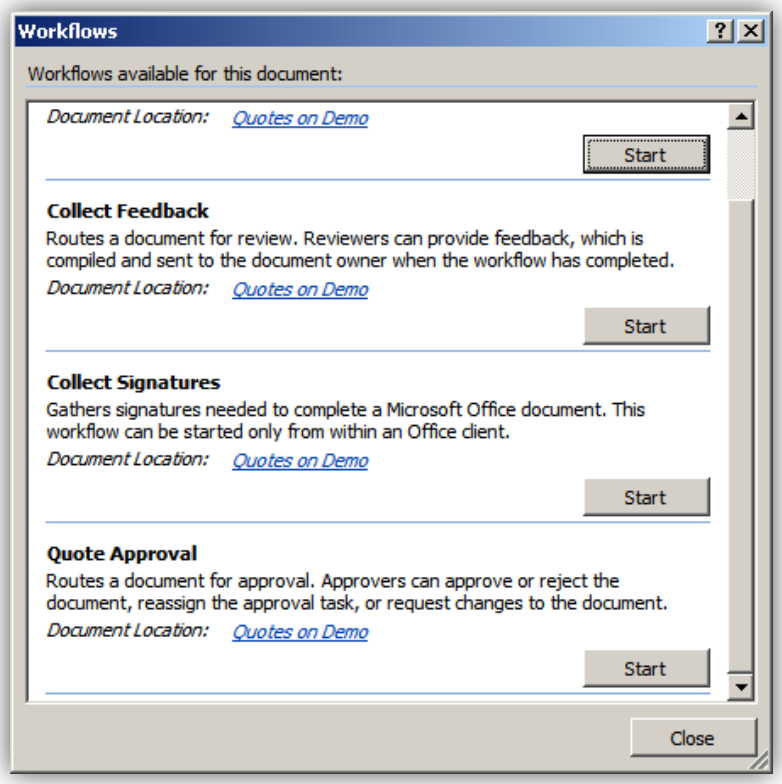
1) Open an existing quote document

- Navigate to the **Quotes** document library.
- Hover over the **Sample Quote** document name and click the **Edit in Microsoft Office Word** option in the drop down.

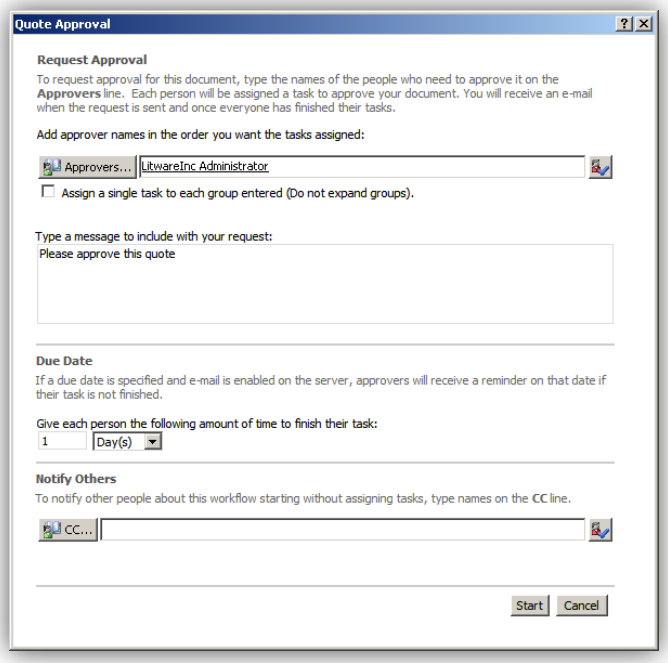


2) Start the **Quote Approval** in Word 2007

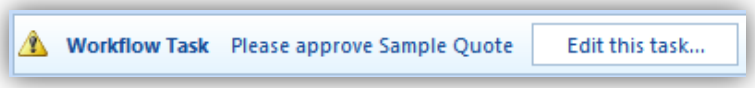
- Click the **Office Button** and select **Workflows** option from the menu.
- In the **Workflows** window, click the **Start** button under the **Quote Approval** workflow.



- Update the comments to something other than the default and click **Start** to start the workflow.

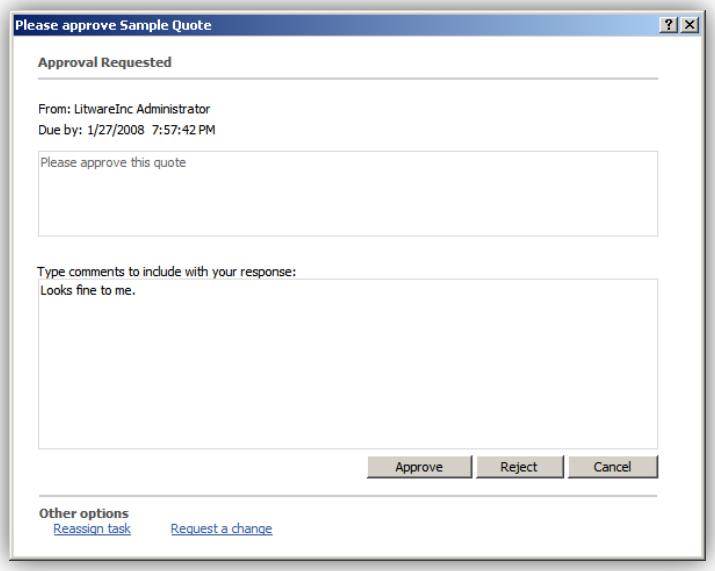


- Notice the **Workflow Task** bar at the top of the document. This notifies the editor of the document that they have a task that relates to this document.



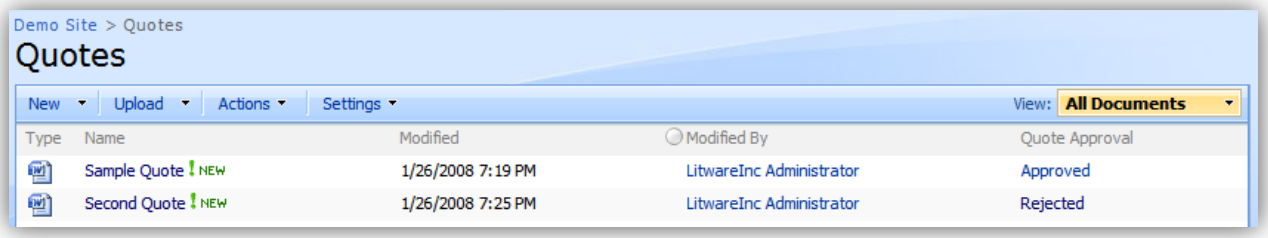
3) Approving the task in Word 2007

- Click the **Edit this task** button at the top of the document in Word 2007.
- In the **Task** dialog, enter some comments and click the **Approve** button to approve the document.



4) Verify the results in **Internet Explorer**.

- Navigate to the **Quotes** document library
- Verify that the **Sample Quote's Quote Approval** column contains **Approved**.



Challenge: Quote Content Type

Often specific document types are grouped by content type and not by document library. Therefore, it is more realistic to create a **Quotes** content type and attach the workflow to the content type.

Attach the same quote approval workflow to a new **Quote** content type and add the content type to any document library. Now as long as the document is a quote content type the workflow will be available no matter which document library it is in.

Do not forget to remove the workflow association from the **Quotes** document library or you may see that workflow association instead of the content type association.

Lab 03: Programming with SharePoint Workflow API

Lab Overview: Litware Inc. needs to extend the capabilities of SharePoint's workflow system. To do this you will need to create your own workflows with custom forms. To do this you will need to be familiar with the SharePoint APIs, specifically those that represent the workflow objects.

To do this you will write a simple Windows Forms application that will use the SharePoint APIs to create workflow associations, start workflow instances, and monitor running and completed workflow instances.

Exercise 0: Setup

1) If you didn't complete lab 4, you'll need to setup the **Quotes** list in the **Demo** site collection.
   - Open SharePoint and browse to the **Demo** site collection

     *The url is **http://litwareinc.com/sites/Demo**.*

     *If the site collection does not exist, create it using the **CreateDemo.bat** file in the **C:\Labs\Files folder**.*
   - Click the **Site Actions -> Create** menu option.
   - On the **Create** page, click the **Document Library** link under the **Libraries** section header.
   - Enter a name of **Quotes** and click the **Create** button to create the new document library.
   - Add a new **Quote** to the document library by clicking the **New -> New Document** button.
   - In the warning dialog, click **OK** to open the new document.
   - In Word 2007, enter some text into the document and click **Save**. Save the document using the name **Sample Quote.docx** to the **Quotes** library.

Exercise 1: Creating a Workflow Association

1) Connect the **Association** user control to the **MainForm** object.
   - Open the code for the **MainForm.cs** class by right clicking it in the **Solution Explorer** and choosing **View Code**.
   - In the **MainForm** class, add two private fields to store the **SPWeb** and **SPSite** objects.

```
private SPSite m_site;
private SPWeb m_web;
```

   - In the **MainForm_Load** method, add the code to connect to the WSS Site.

```
m_site = new SPSite("http://litwareinc.com/sites/Demo");
m_web = m_site.RootWeb;
```

   - Next, add the code that will provide the **ctlAssociation** object with the **SPWeb** object just created.

```
ctlAssociation.Initialize(m_web);
```

   - Cleanup by adding code to dispose of the **SPSite** and **SPWeb** objects in the **MainForm_FormClosed** method.

```
m_web.Dispose();
m_site.Dispose();
```

2) Use the **SPWeb** object to initialize the UI of the association user control

- Open the code for the **Association.cs** class by right clicking it in the **Solution Explorer** and choosing **View Code**.
- In the **Association** class, add two private fields to store the **SPWeb** and **SPWorkflowAssociation** objects needed to store the state of the user interface.

```
private SPWeb m_web;
private SPWorkflowAssociation m_current;
```

- Locate the **Initialize** method and store the **SPWeb** parameter in the **m_web** private field.

```
m_web = web;
```

- Populate the **Lists** combo box using the **SPWeb.Lists** collection and the **PopulateComboBox** method.

  *Look at the **PopulateComboBox** method to understand how it works.*

```
PopulateComboBox(m_web.Lists, lstWebLists);
```

- Populate the **WorkflowTemplates** combo box using the **SPWeb.WorkflowTemplates** collection and the **PopulateComboBox** helper method.

```
PopulateComboBox(m_web.WorkflowTemplates, lstWorkflowTemplates);
```

- Populate the **Tasks Lists** combo box using a filtered **SPWeb.Lists** collection and a filtering version of the **PopulateComboBox** method.

  *Only add **SPList** objects that have a **BaseTemplate** of **SPListTemplateType.Tasks**.*

  *Look at the **PopulateComboBox** method to understand how filtering is performed with a lambda function.*

```
PopulateComboBox(m_web.Lists.Cast<SPList>(),
    lstTaskList, n => n.BaseTemplate == SPListTemplateType.Tasks);
```

- Populate the **History Lists** combo box using a filtered **SPWeb.Lists** collection and a filtering version of the **PopulateComboBox** method.

  *Only add **SPList** objects that have a **BaseTemplate** of **SPListTemplateType.WorkflowHistory**.*

```
PopulateComboBox(m_web.Lists.Cast<SPList>(),
    lstHistoryList, n => n.BaseTemplate == SPListTemplateType.WorkflowHistory);
```

3) Implement the **DisplayAssociation** method that will populate the UI based on an existing **SPWorkflowAssociation** object.

- Locate the **DisplayAssociation** method and add the code to set the **txtName** control's text to the name of the workflow association.

```
// initialize the name and workflow template
txtName.Text = association.Name;
```

- Add code to set the selected item of **lstWbrkflowTemplates** to the workflow template defined in the **SPWorkflowAssociation's BaseTemplate.Id** property.

  LINQ is used here to perform a query using the items in the combo box.

```
lstWorkflowTemplates.SelectedItem =
    lstWorkflowTemplates.Items.Cast<SPWorkflowTemplate>().First(
        n => n.Id == association.BaseTemplate.Id);
```

- Using the same technique used to select the workflow template, select the task and history lists from their respective combo boxes. Compare the **ID** of the **SPList** to the **SPWorkflowAssociatoins TaskListId** and **HistoryListId**.

```
// find the task and history list in the drop down lists
lstTaskList.SelectedItem =
    lstTaskList.Items.Cast<SPList>().First(
        n => n.ID == association.TaskListId);
lstHistoryList.SelectedItem =
    lstHistoryList.Items.Cast<SPList>().First(
        n => n.ID == association.HistoryListId);
```

- Update the state of the UI to allow the user to change some information, but not change the selected workflow template.

```
// enable editing controls
lstWorkflowTemplates.Enabled = false; // not enabled since it is already chosen
grpAssociation.Enabled = true;
btnDelete.Enabled = false;
```

4) Implement the **DisplayNewAssociation** method that will populate the UI to allow creation of a new **SPWorkflowAssociation**

- Locate the **DisplayNewAssociation** method and add code to set the **txtName** control's text to an empty string and select the first item in the **lstWorkflowTemplates** control.

```
// choose a default name and workflow template
txtName.Text = string.Empty;
lstWorkflowTemplates.SelectedIndex = 0;
```

- Add code to set the selected item of **lstTaskList** and **lstHistoryList** to the first item in the list.

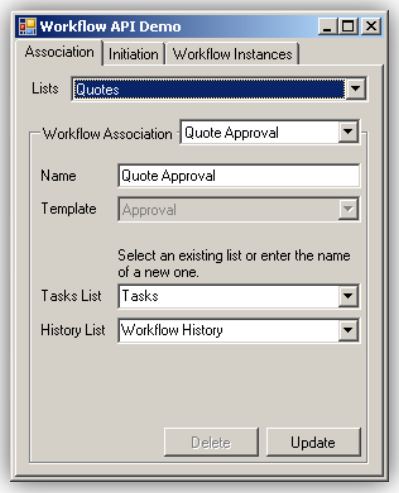
```
// select the first task or history list available
lstTaskList.SelectedItem = 0;
lstHistoryList.SelectedItem = 0;
```

- Update the status of the UI to allow the user to enter all the information they need.

```
// enable editing controls
lstWorkflowTemplates.Enabled = true;
grpAssociation.Enabled = true;
btnDelete.Enabled = false;
```

5) Test the user interface.

- Start the application using the debugger by selecting **Debug -> Start Debugging** or pressing **F5**.
- Switch to the **Association** tab and explore the lists and their workflow associations.



6) Implement the **CreateWorkflowAssociation** method to allow creation of new **SPWorkflowAssociation** objects.

- Locate the **CreateWorkflowAssociation** method and replace the existing code with the code to lookup the selected **SPList** and **SPWorkflowTemplate** objects from their combo box controls.

```
SPList list = lstWebLists.SelectedItem as SPList;
SPWorkflowTemplate template =
    lstWorkflowTemplates.SelectedItem as SPWorkflowTemplate;
```

- Check if the selected history list already exists by comparing **lstHistoryList.SelectedItem** to **null**. If not, create it using the **CreateList** method. If so, cast the **SelectedItem** property to **SPList** store it for later.

```
// create the history list if it doesn't exist
SPList historyList = (lstHistoryList.SelectedItem == null) ?
    CreateList(lstHistoryList.Text, SPListTemplateType.WorkflowHistory) :
    lstHistoryList.SelectedItem as SPList;
```

- Find the selected tasks list, creating it if necessary and store it for later.

```
// create the task list if it doesn't exist
SPList taskList = (lstTaskList.SelectedItem == null) ?
    CreateList(lstTaskList.Text, SPListTemplateType.Tasks) :
    lstTaskList.SelectedItem as SPList;
```

- Create the new workflow association using the **SPWorkflowAssociation.CreateListAssociation** static method. Provide the **SPWokflowTemplate**, name, task list, and history list retrieved previously and store the new **SPWorkflowAssociation** for later.

```
// create the association
SPWorkflowAssociation result =
    SPWorkflowAssociation.CreateListAssociation(template, txtName.Text,
        taskList, historyList);
```

- Using the **AssociationData** property of the **SPWorkflowTemplate**, assign the **AssociationData** property of the new **SPWorkflowAssociation**.

```
result.AssociationData = template.AssociationData;
```

- Attach the new **SPWorkflowAssociation** object to the selected list using the **SPList.AddWorkflowAssociation** method. Do not forget to call the list's **Update** method.

```
// add the association to the list
list.AddWorkflowAssociation(result);
list.Update();
```

- Return the new **SPWorkflowAssociation** object.

```
return result;
```

7) Implement the **UpdateWorkflowAssocition** method to allow updating existing **SPWorkflowAssociation** objects.

- Update the name of the **SPWorkflowAssociation** object to the value in the **txtName** control.

```
// update the association name
association.Name = txtName.Text;
```

- Find the selected history and tasks lists, creating them if necessary and storing them for later.

```
// create the history list if it doesn't exist
SPList historyList = (lstHistoryList.SelectedItem == null) ?
    CreateList(lstHistoryList.Text, SPListTemplateType.WorkflowHistory) :
    lstHistoryList.SelectedItem as SPList;

// create the task list if it doesn't exist
SPList taskList = (lstTaskList.SelectedItem == null) ?
    CreateList(lstTaskList.Text, SPListTemplateType.Tasks) :
    lstTaskList.SelectedItem as SPList;
```

- If the selected history list's **ID** does not match the **SPWorkflowAssociation's** current **HistoryListId's** value, update the list using the **SPWorkflowAssociation.SetHistoryList** method.

```
// update the history list if it's changed
if (association.HistoryListId != historyList.ID)
    association.SetHistoryList(historyList);
```

- Check and update the tasks list if necessary using the **SPWorkflowAssociation.SetTasksList** method.

```
// update the task list if it's changed
if (association.TaskListId != taskList.ID)
    association.SetTaskList(taskList);
```

- Update the **SPWorkflowAssociation** object using its **ParentLists's UpdateWorkflowAssociaiton** method.

```
// update the association
association.ParentList.UpdateWorkflowAssociation(association);
```

8) Implement the **DeleteWorkflowAssociation** method to allow deletion of existing **SPWorkflowAssociation** objects.

- Find the **SPWorkflowAssociation's** parent list using the **ParentList** property and store it for later.

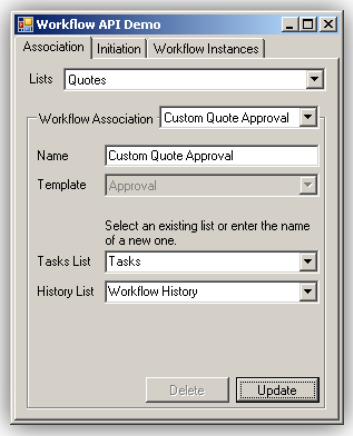
```
SPList parentList = association.ParentList;
```

- Remove the workflow association from the parent list using the **SPList.RemoveWorkflowAssociation** method. Do not forget to call the list's **Update** method.

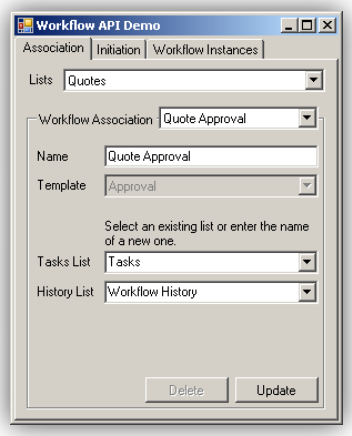
```
// remove the workflow association
parentList.RemoveWorkflowAssociation(association);
parentList.Update();
```

9) Test the application by adding, updating, and deleting a workflow association.

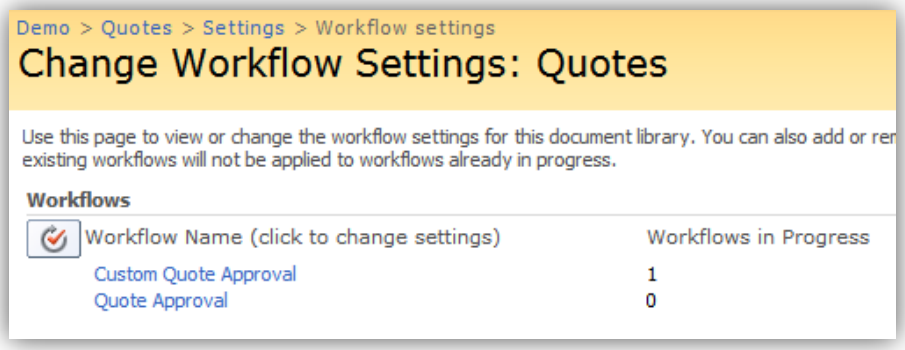
- Start the application using the debugger by selecting **Debug -> Start Debugging** or pressing **F5**.
- Switch to the **Association** tab and select the **Quotes** list in the **Lists** dropdown.
- In the **Workflow Association** dropdown, verify selection of the **Quote Approval** workflow association and verify the selected tasks and history lists that.
- Change the name of the **Quote Approval** workflow association to **Custom Quote Approval** and click **Update** to change its name.



- Select **New Item…** in the **Workflow Association** drop down and enter a name of **Quote Approval**. Also, select a template of type **Approval**. When done, click **Update** to commit the changes.



- In **Internet Explorer**, browse to the **Quotes** list and click **Settings -> Document Library Settings**. Next click the **Workflow settings** link in the **Permissions and Management** section and verify the listing of **Custom Quote Approval** and **Quote Approval**.



Exercise 2: Starting a Workflow

1) Connect the **Initiation** user control to the **MainForm** object.

- Open the code for the **MainForm.cs** class by right clicking it in the **Solution Explorer** and choosing **View Code**.
- In the **MainForm_Load** method, add the code that will provide the **ctlInitiation** object with the **SPWeb** object just created.

```
ctlInitiation.Initialize(m_web);
```

2) Use the **SPWeb** object to initialize the UI of the initiation user control.

- Open the code for the **Initiation.cs** class by right clicking it in the **Solution Explorer** and choosing **View Code**.
- In the **Initiaiton** class, add a private field to store the **SPWeb** object to store the state of the user interface.

```
private SPWeb m_web;
```

- Locate the **Initialize** method and store the **SPWeb** parameter in the **m_web** private field.

```
m_web = web;
```

- Populate the **Lists** combo box using the **SPWeb.Lists** collection and the **PopulateComboBox** method.

```
PopulateComboBox(m_web.Lists, lstWebLists);
```

3) Implement the **PopulateListDetails** method that will populate the list items and workflow associations drop down lists based on the currently selected list.

- Locate the **PopulateListDetails** method and add the code to populate the list items drop down list with the list items in the currently selected list. Use the **SPList.Items** collection to access a list's list items.

```
// populate the combo box containing the list items
// and workflow associations for the list
PopulateComboBox(list.Items, lstListItems);
```

- Add the code to populate the list of workflow associations using the currently selected list's **WorkflowAssociations** collection.

```
PopulateComboBox(list.WorkflowAssociations, lstAssociations);
```

4) Implement the **ContainsRunningWorkflow** method that will determine if a specific list item has a running instance of a specific workflow association. This disallows the starting of a workflow if one is already running.

- Locate the **ContainsRunningWorkflow** method and add convert the **SPListItem.Workflows** collection to an **IEnumerable<SPWorkflow>** object by using the **Cast<SPWorkflow>** method.

  This will be used later to check if any workflow's of a specific type are currently running.

```
IEnumerable<SPWorkflow> workflows = listItem.Workflows.Cast<SPWorkflow>();
```

- Add code that will search through the list of workflows and find any workflows of the selected type. If any exist, find those with an **InternalState** of **Running**.

  *The method **FirstOrDefault** will return the first **SPWorkflow** object that matches the criteria in the lambda expression. Return **null** if no **SPWorkflow** object exists.*

```
// find the first item in the list that matches the condition
// FirstOrDefault returns null if no item is found
SPWorkflow workflow = workflows.FirstOrDefault(
    n => n.AssociationId == association.Id &&
    n.InternalState == SPWorkflowState.Running);
```

- Return **true** if a **SPWorkflow** was found, **false** otherwise.

```
// return true if a workflow was found
return (workflow != null);
```

5) Implement the **StartWorkflow** method that will use the current UI information to start a new instance of a workflow.

- Locate the **StartWorkflow** method and add code that locates the currently selected list item and workflow association. Cast them to a **SPListItem** and **SPWorkflowAssociation** and store them for later.

```
SPListItem listItem = lstListItems.SelectedItem as SPListItem;
SPWorkflowAssociation association =
    lstAssociations.SelectedItem as SPWorkflowAssociation;
```

- Use the **SPWorkflowManager.StartWorkflow** method to start the workflow. Use the text in **txtInitiationData.Text** as the **eventData** to use when starting the workflow.

  *The last parameter of **isAutoStart** tells SharePoint to try to restart the workflow if a problem occurs. This is important, as it will always fail on the first attempt to start since it is running in our process. Failure schedules a retry, which will work since it will run in SharePoint's process.*

```
// start the workflow
m_web.Site.WorkflowManager.StartWorkflow(
    listItem, association, txtInitiationData.Text, true);
```

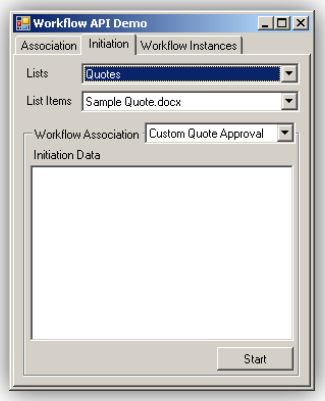
- Display a message box indicating the workflow is starting and clear the **initiation data** text box.

```
// update the UI to indicate the workflow has started
MessageBox.Show(this, "Workflow started");
txtInitiationData.Text = string.Empty;
```

6) Test the application by starting a new instance of the **Quote Approval** workflow created in the previous exercise.

- Start the application using the debugger by selecting **Debug -> Start Debugging** or pressing **F5**.

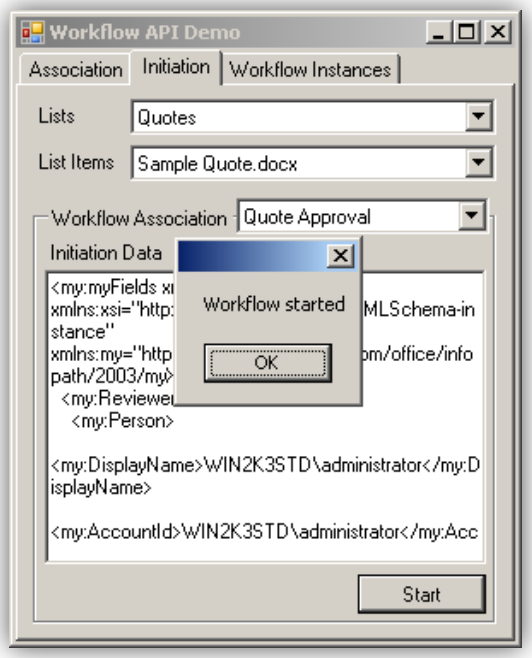
- Switch to the **Initiation** tab, select the Quote list in the Lists dropdown, and select **Sample Quote.docx** in the list items drop down list.



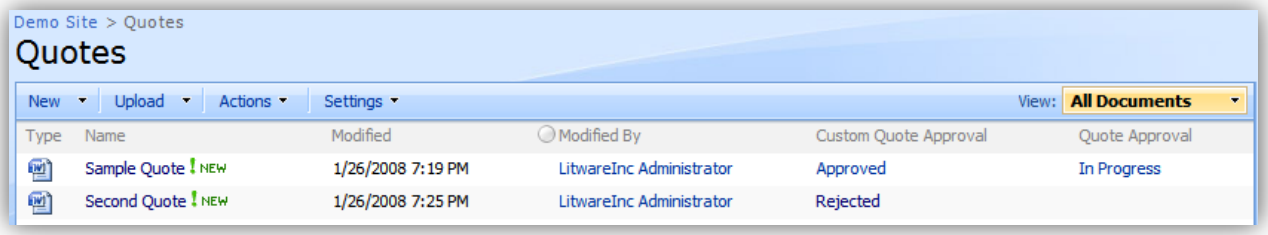
- In the **Workflow Association** drop down list, select **Quote Approval**.
- Enter the following xml into the initiation data page. This is the initiation data to start an **Approval** workflow.

```
<my:myFields xml:lang="en-us" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:my="http://schemas.microsoft.com/office/infopath/2003/myXSD">
  <my:Reviewers>
    <my:Person>
      <my:DisplayName> LITWAREINC \administrator</my:DisplayName>
      <my:AccountId>LITWAREINC\administrator</my:AccountId>
      <my:AccountType>User</my:AccountType>
    </my:Person>
  </my:Reviewers>
  <my:CC></my:CC>
  <my:DueDate xsi:nil="true"></my:DueDate>
  <my:Description></my:Description>
  <my:Title></my:Title>
  <my:DefaultTaskType>1</my:DefaultTaskType>
  <my:CreateTasksInSerial>true</my:CreateTasksInSerial>
  <my:AllowDelegation>true</my:AllowDelegation>
  <my:AllowChangeRequests>true</my:AllowChangeRequests>
  <my:StopOnAnyReject xsi:nil="true"></my:StopOnAnyReject>
  <my:WantedTasks xsi:nil="true"></my:WantedTasks>
  <my:SetMetadataOnSuccess>false</my:SetMetadataOnSuccess>
  <my:MetadataSuccessField></my:MetadataSuccessField>
  <my:MetadataSuccessValue></my:MetadataSuccessValue>
  <my:ApproveWhenComplete>false</my:ApproveWhenComplete>
  <my:TimePerTaskVal xsi:nil="true"></my:TimePerTaskVal>
  <my:TimePerTaskType xsi:nil="true"></my:TimePerTaskType>
  <my:Voting>false</my:Voting>
  <my:MetadataTriggerField></my:MetadataTriggerField>
  <my:MetadataTriggerValue></my:MetadataTriggerValue>
  <my:InitLock>false</my:InitLock>
  <my:MetadataStop>false</my:MetadataStop>
  <my:ItemChangeStop>false</my:ItemChangeStop>
  <my:GroupTasks>false</my:GroupTasks>
</my:myFields>
```

- Click the **Start** button to start the workflow. The starting of the workflow will silently fail and SharePoint will automatically retry. This process may take up to 5 minutes.



- In **Internet Explorer**, browse to the **Quotes** list and check the status of the workflow on the **Sample Quote** document. Most likely, the value will be **Starting**, when it changes to **In Progress**, the workflow is running.



Exercise 3: Viewing Current Workflow Instances

1) Connect the **Workflow** user control to the **MainForm** object.

- Open the code for the **MainForm.cs** class by right clicking it in the **Solution Explorer** and choosing **View Code**.
- In the **MainForm_Load** method, add the code that will provide the **ctlWorkflow** object with the **SPWeb** object just created.

```
ctlWorkflow.Initialize(m_web);
```

2) Use the **SPWeb** object to initialize the UI of the workflow user control.

- Open the code for the **Workflow.cs** class by right clicking it in the **Solution Explorer** and choosing **View Code**.
- In the **Workflow** class, add a private field to store the **SPWeb** object needed to store the state of the user interface.

```
private SPWeb m_web;
```

- Locate the **Initialize** method and store the **SPWeb** parameter in the **m_web** private field.

```
m_web = web;
```

- Populate the **Lists** combo box using the **SPWeb.Lists** collection and the **PopulateComboBox** helper method.

```
PopulateComboBox(m_web.Lists, lstWebLists);
```

3) Implement the **PopulateListItems** method that will populate the list items drop down list based on the currently selected list.

- Locate the **PopulateListItems** method and populate the **lstListItems** drop down list using the **PopulateComboBox** method and the Items collection of the current list.

```
PopulateComboBox(list.Items, lstListItems);
```

4) Implement the **PopulateWorkflows** method that will populate the workflows drop down list based on the currently selected list item.

- Locate the **PopulateWorkflows** method and populate the **lstAssociations** drop down list using the **PopulateComboBox** method and the **Workflows** collection of the current list item.

```
PopulateComboBox(listItem.Workflows, lstAssociations);
```

5) Implement the **DisplayWorkflow** method that will use the currently selected workflow to populate the workflow details UI.

- Locate the **DisplayWorkflow** method and use current **SPWorkflow** object to initialize the initiator, started, and workflow state UI controls.

```
// setup the workflow state UI
txtInitiator.Text = workflow.AuthorUser.Name;
txtStarted.Text = workflow.Created.ToString();
txtWorkflowState.Text = workflow.InternalState.ToString();
```

- Add the code to populate the **lstModifications** drop down list using the **PopulateListBox** and the **Modifications** collection of the current workflow.

```
// populate the list of modifications
PopulateListBox(workflow.Modifications, lstModifications);
```

- Use the **SPWorkflowManager.GetItemTasks** method to retrieve the tasks for the current workflow. You will need to provide the **SPListItem** object the workflow attached to and the **SPWorkflowFilter** object associated with the current **SPWorkflow**.

```
// populate the list of tasks
SPWorkflowTaskCollection tasks =
    m_web.Site.WorkflowManager.GetItemTasks(
        workflow.ParentItem, workflow.TaskFilter);
```

- Using the retrieved **SPWorkflowTaskCollection**, populate the **lstTasks** drop down list using the **PopulateListBox** method.

```
PopulateListBox(tasks, lstTasks);
```

6) Implement the **LookupModificationName** method that will convert a **SPWorkflowModification** object into a user-friendly string using the **SPWorkflowTemplate's** metadata collection.

- Locate the **LookupModificationname** method and add the code that formats the **SPWorkflowModification's Id** property into the metadata string used to lookup the modification's friendly name.

```
string metadataName = string.Format("Modification_{0}_Name", modification.Id);
```

- Using the metadata string, retrieve the modification name from the **SPWorkflowTemplate** object the current **SPWorkflow** is based on.

```
return workflow.ParentAssociation.BaseTemplate[metadataName] as string;
```

7) Implement the **LookupModificationContextData** method, which uses the **SPWorkflowModification** class's **ContextData** property to return the context data associated with the modification.

- Locate the **LookupModificationContextData** method and add the code to return the current **SPWorkflowModification's** context data.

```
return modification.ContextData;
```

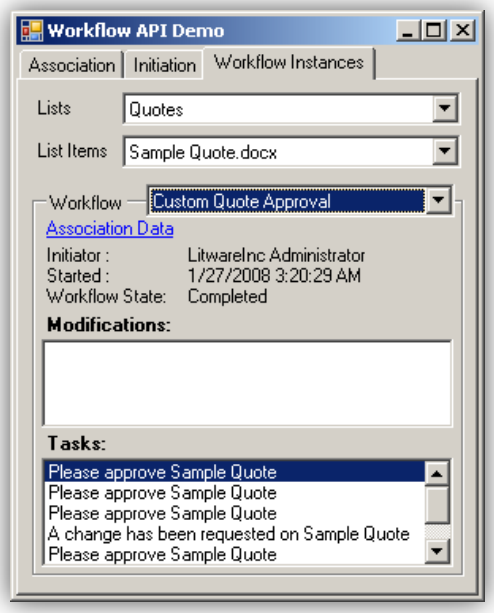
8) Implement the **LookupAssociationData** method to return the association data stored in the workflow's association to the list. As the workflow attached itself to the list, this data is set.

9) Locate the **LookupAssociationData** method and add the code to return the current **SPWorkflow's** association data by using its parent **SPWorkflowAssociation** object.

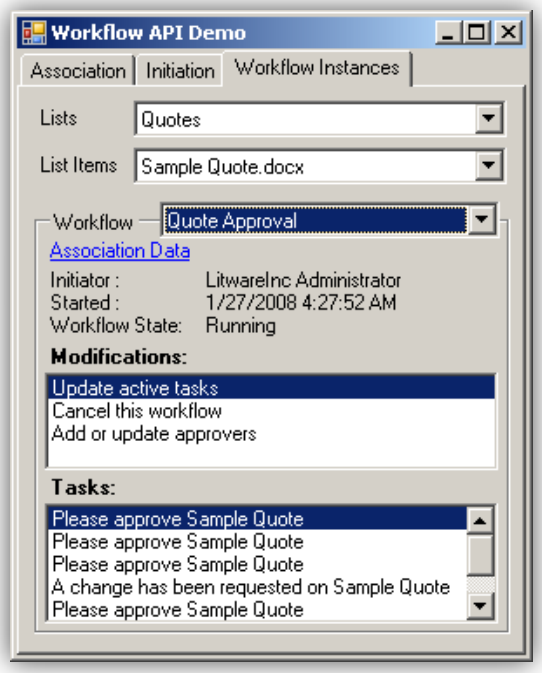
```
return workflow.ParentAssociation.AssociationData;
```

10) Test the application by looking at the workflow instance started in the previous exercise.

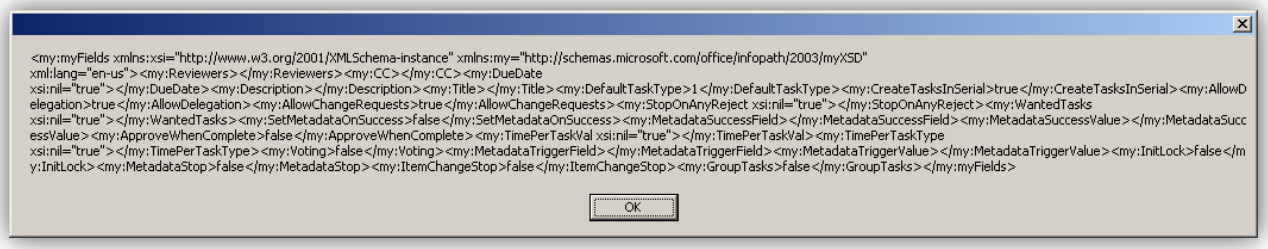
- Start the application using the debugger by selecting **Debug -> Start Debugging** or pressing **F5**.
- Switch to the **Workflow Instances** tab, select the Quote list in the Lists dropdown, and select **Sample Quote.docx** in the list items drop down list.



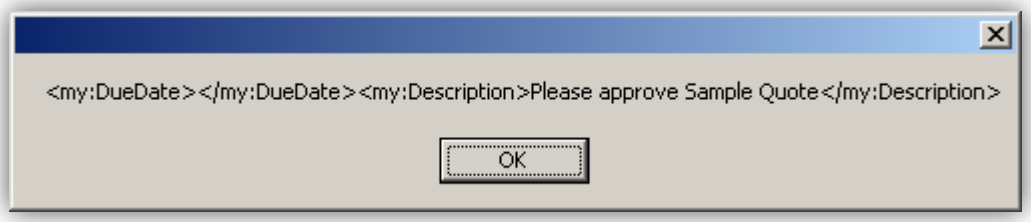
- Work your way through the **Workflow** drop down to look at all workflows that are running or have run for the selected list/list item pair. Locate the workflow that has a workflow state or **Running**.



- Click the **Association Data** link to display the workflow's association data.



- Double click one of the items in the **Modifications** list. The data displayed is the context data associated with the modification.



Lab 04: Developing SharePoint Workflow Templates with Visual Studio 2008

Lab Overview: Now that the management of Litware Inc. has become comfortable with the built in functionality of SharePoint workflows, they have started to request extra features. They have requested a workflow system that archives documents in another document library. When the document library manager chooses to archive a document, they manually start a workflow, which creates an approval task.

To build this workflow, you will need to deal with many different aspects of SharePoint workflow, so you will start simple. In the first part or implementing the **DocumentArchive** project, you will focus on creating the workflow that will simply log a message to the Workflow History list.

Exercise 0: Setup

1) If you did not complete lab 4, you will need to create the **Demo** site collection.

   Open SharePoint and browse to the Demo site collection

   *The url is **http://litwareinc.com/sites/Demo**.*

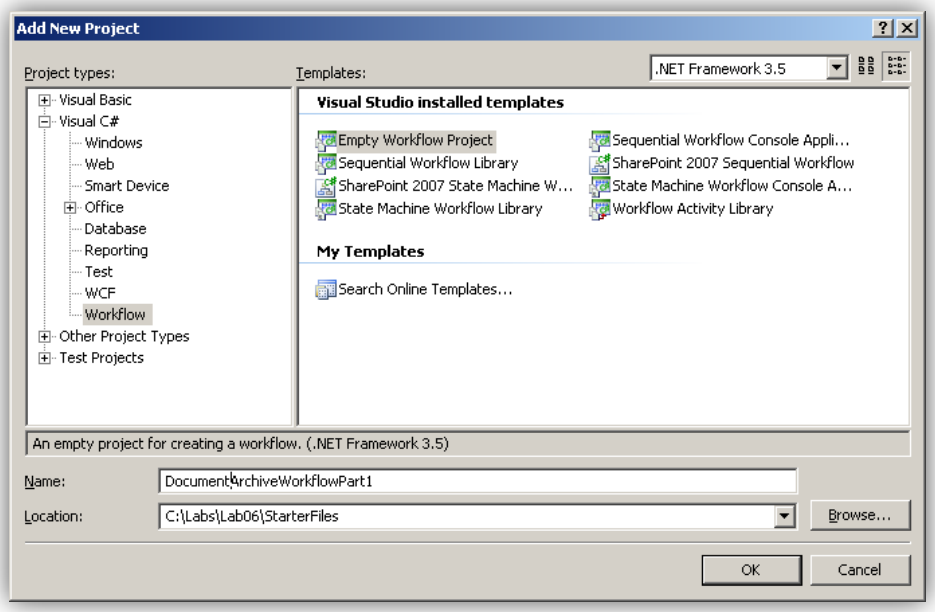
   *If the site collection does not exist, create it using the **CreateDemo.bat** file in the **C:\Labs\Files folder**.*

2) Open the starter **VS 2008** solution at **\Labs\Lab04\StarterFiles\DocumentArchive Part 1.sln**.

Exercise 1: Build Hello World Workflow

1) Add a new **Empty Workflow Project** named **DocumentArchiveWorkflowPart1**.

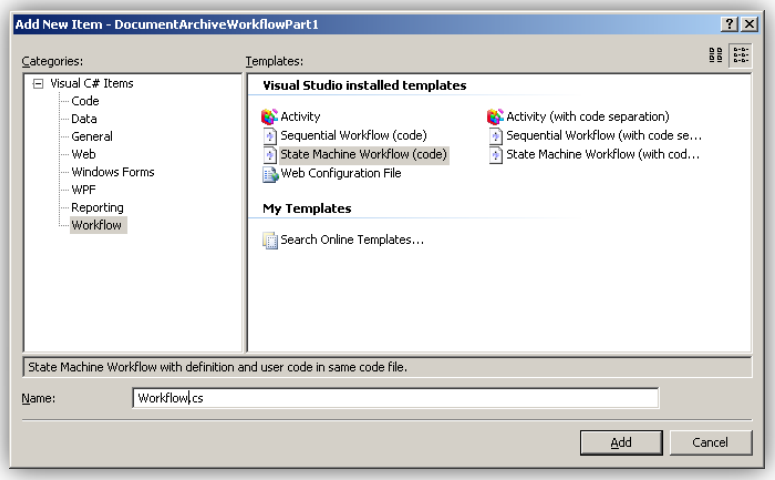
   - Right click on the solution in the **Solution Explorer** and select **Add -> New Project**.
   - In the **Add New Project** dialog, select the project type of **Workflow** on the left and a template of **Empty Workflow Project** on the right.



   - Name the project **DocumentArchiveWorkflowPart1** and click **OK**.

2) Add a new state machine workflow named **Workflow** to the project.

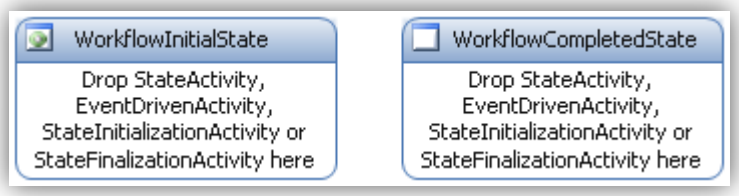
- Right click on the **DocumentArchiveWorkflowPart1** project in the solution explorer and select **Add -> New Item…**
- In the **Add New Item** dialog, select the **Workflow** category on the left and a template of **State Machine Workflow (code)** on the right.
- Name the new item **Workflow.cs** and click **Add**.



3) Add a new state to the workflow named **WorkflowCompletedState**.

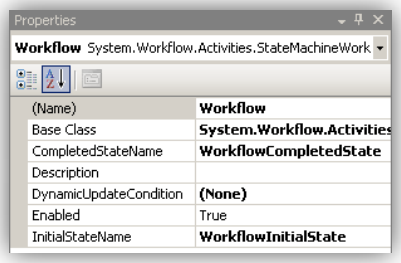
- Right click on the **State Machine** designer canvas and click **Add State**.
- Set the name of the new state to **WorkflowCompletedState**.

  *In the properties pane, set the **Name** property to **WorkflowCompletedState**.*



4) Set the initial and completed states of the workflow

- Right click on the **State Machine** designer canvas and click **Properties**.
- Verify the **InitalStateName** property in the Properties pane is set to **WorkflowInitialState**.
- Set the **CompletedStateName** property to **WorkflowCompletedState**.

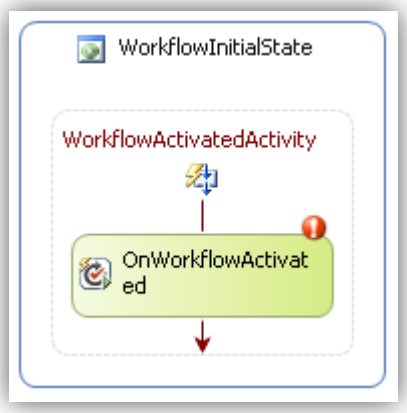


5) Add an **OnWorkflowActivated** event driven activity to signal the start of the workflow.

- Right click on the **WorkflowInitialState** state and select **Add EventDriven**.
- Rename the new event driven activity **WorkflowActivatedActivity**.

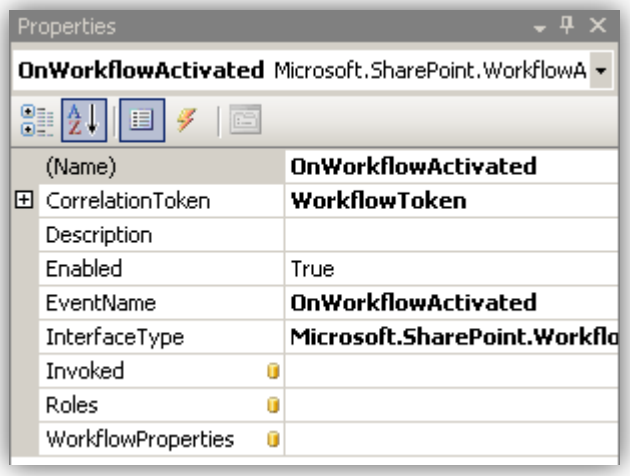
  *Right click the new event driven activity and click **Properties**.*
  *In the properties pane, set the **Name** property to **WorkflowActivatedActivity**.*
- Drag an **OnWorkflowActivated** activity from the toolbox into the **WorklowActivatedActivity** activity and name it **OnWorkflowActivated**.

  *In the properties pane, set the **Name** property to **OnWorkflowActivated**.*



- Create a new correlation token for the **OnWorkflowActivated** activity. Scope it to the workflow.

  *In the properties pane, set the **CorrelationToken** property to **WorkflowToken**.*
  *Expand the **CorrelationToken** property in the property pane.*
  *Set the **OwnerActivityName** property to **Workflow**.*

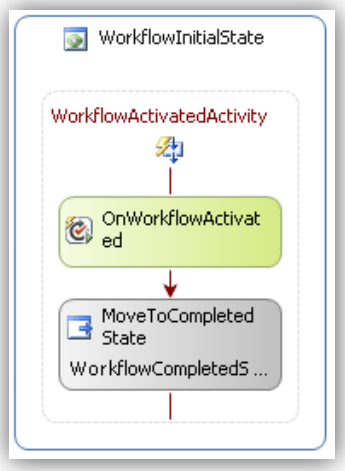


6) Transition to the **WorkflowCompletedState** state after the **OnWorkflowActivated** activity executes.

- Drag a **SetState** activity from the toolbox into the **WorkflowActivatedActivity** immediately following the **OnWorkflowActivated** activity.

  *Make sure you use the **SetState** activity in the **Windows Workflow v3.0** group in the toolbox.*
- Change the new activity's name to **MoveToCompletedState** and set the target state to **WorkflowCompletedState**.

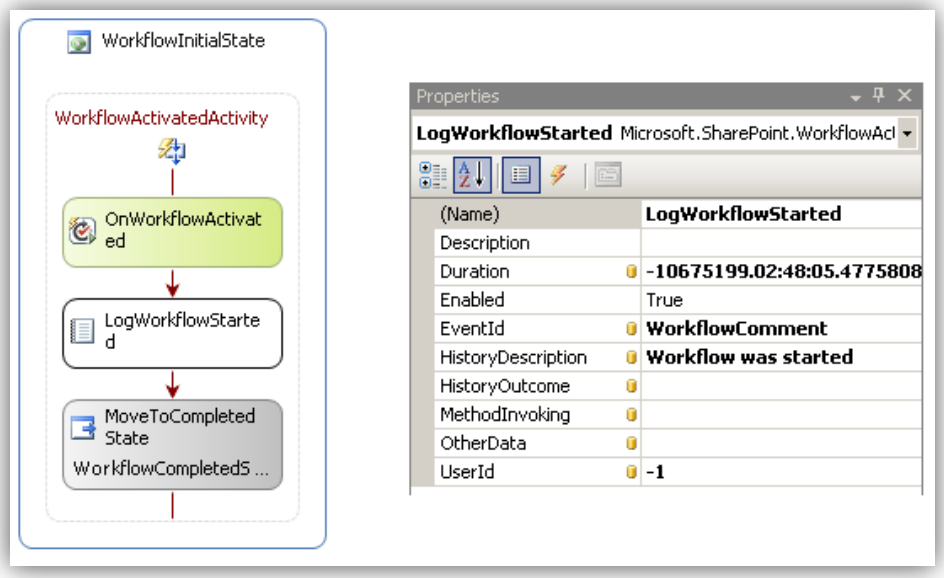
*Set the **TargetStateName** property to **WorkflowCompletedState**.*



7) Log a started message to the workflow's history to indicate that the workflow ran.

- Drag a **LogToHistoryListActivity** from the toolbox into the **WorkflowActivatedActivity** between the **OnWorkflowActivated** and **MoveToCompletedState** activities.
- Change the new activity's name to **LogWorkflowStarted** and set its **HistoryDescription** to **'Workflow was started'**.

*In the properties pane, set the **Name** property to **LogWorkflowStarted**.*
*Set the **HistoryDescription** property to **Workflow was started**.*

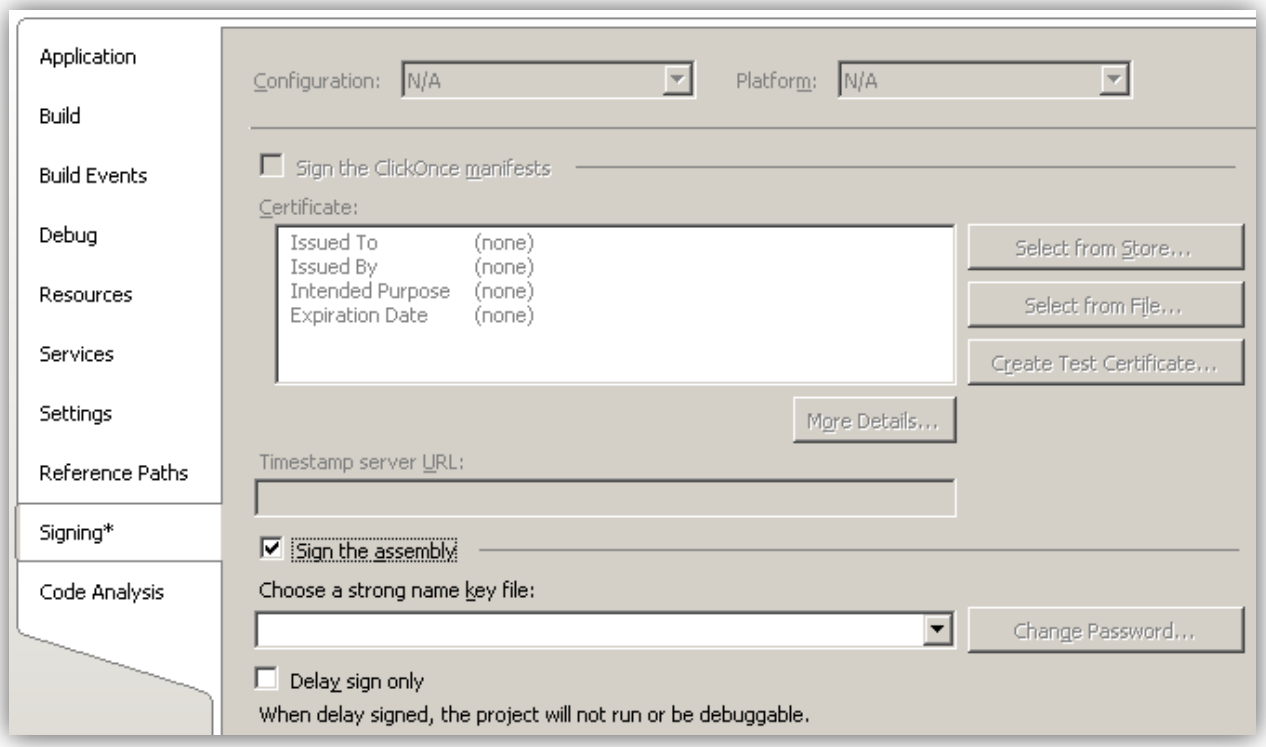


8) Build the project.

- Right click the project in the **Solution Explorer** and click **Build**.

Exercise 2: Install the workflow as a Feature

1) Sign the assembly to allow installation in the GAC.
   - Right click the project in the solution explorer and click **Properties**.
   - Click the **Signing** tab on the left hand side of the project properties window.
   - Check the **Sign the assembly** checkbox.



   - Choose **<New …>** new from the drop down list box.
   - In the **Create Strong Name Key** dialog box, enter a **Key file name** of **Key**.
   - Uncheck the **Protect my key file with a password** and click **OK**.

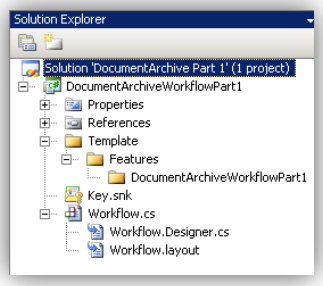
   - Close the project properties window.

2) Create the folder that will contain the feature files.

- Create a folder under the project named **Template**.

  *Right click on the project in the **Solution Explorer** and click **Add -> New Folder**.*
  *Right click the folder and click **Rename** to rename the folder to **Template**.*

- Create another folder named under **Template** named **Features**.
- Create a third folder under **Features** named **DocumentArchiveWorkflowPart1**.



3) Create the **feature.xml** in the **DocumentArchiveWorkflowPart1** folder.

- Create a new file named **Feature.xml** in the **DocumentArchiveWorkflowPart1** folder.

  *Right click the **DocumentArchiveWorkflowPart1** folder and click **Add -> New Item…**.*
  *Select the **Data** category on the left and select a template of **XML file**.*
  *Name the new file **Feature.xml** and click **Add**.*

- Insert the **Windows SharePoint Service Workflow Feature.xml** snippet.

  *Right click in the editor and select **Insert Snippet…**.*
  *In the popup window, select the **Windows SharePoint Service Workflow** option then select **Feature.xml Code**.*

- Replace **GUID** with a new Guid generated using **GuidGen**.

  *Click **Tools -> Create Guid** then copy and paste the registry formatted guid into the snippet's GUID placeholder.*

- Replace **Title** with **Document Archive – Part 1**.
- Replace **Description** with **"Contains workflows and custom forms designed to aid in document archiving"**.

```
<Feature  Id="29ac22d1-7d8c-4513-9566-11109ab25983"
          Title="Document Archive - Part 1"
          Description="Request approval to archive this document in another
document library."
          Version="12.0.0.0"
          Scope="Site"
          xmlns="http://schemas.microsoft.com/sharepoint/">
  <ElementManifests>
    <ElementManifest Location="workflow.xml" />
  </ElementManifests>
  <Properties>
    <Property Key="GloballyAvailable" Value="true" />
  </Properties>
</Feature>
```

- Save and close the file.

4) Create the **workflow.xml** file needed to define the details of the workflow.

- Create a new file named Workflow.xml in the **DocumentArchiveWorkflowPart1** folder.
- Insert the **Windows SharePoint Service Workflow Workflow.xml** snippet.

  *Right click in the editor and select **Insert Snippet…**.*
  *In the popup window, select the **Windows SharePoint Service Workflow** option.*
  *In the second popup window, select **Workflow.xml Code**.*
- Replace **GUID** with a new Guid generated using GuidGen.

  *Click **Tools -> Create Guid***
  Copy and paste the registry formatted guid into the snippet's GUID placeholder.
- Replace **Title** with **Document Archive – Part 1**.
- Replace **Description** with **"Requests approval to archive this document in another document library"**.
- Replace **CodeBesideClass** with **DocumentArchiveWorkflowPart1.Workflow**.
- Replace the **PublicKeyToken** of the **CodeBesideAssembly** with the public key token of the assembly.

  *Open a new instance of the **Visual Studio 2008 Command Prompt** located in **Start -> All Programs -> Microsoft Visual Studio 2008 -> Visual Studio Tools**.*
  Navigate to C:\Labs\Lab04\StarterFiles\DocumentArchiveWorkflowPart1\bin\Debug
  *Run **sn.exe –T DocumentArchiveWorkflowPart1.dll** to display the public key token.*
- Remove the **TaskListContentTypeId**, **AssociationUrl**, **InstantiationUrl**, **ModificationUrl** attributes.

  These will be used later.
- Remove the content of the **MetaData** element.

  These will also be used later.

```
<Elements xmlns="http://schemas.microsoft.com/sharepoint/">
  <Workflow
       Name="Document Archive - Part 1"
       Description="Requests approval to archive this document in another
document library."
       Id="78c8fd78-0b7f-4893-9361-ff005c32b12c"
       CodeBesideClass="DocumentArchiveWorkflowPart1.Workflow"
       CodeBesideAssembly="DocumentArchiveWorkflowPart1, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=15812f954569663f"
       StatusUrl="_layouts/WrkStat.aspx">
    <Categories/>
    <MetaData>
    </MetaData>
  </Workflow>
</Elements>
```

5) Add post build steps to aid in developing and deploying the workflow.

- Right click the project in the **Solution Explorer** and click **Properties**.
- Click the **Build Events** tab in the project properties window.
- Enter the following commands into the **Post-build event command line** window.

  *Copy the contents of the **Template** folder to SharePoint's **Template** folder.*

  *Install the workflow assembly into the GAC using **gacutil.exe**.*

  Reset the app domain that may have loaded the workflow assembly.

  *Forcibly install the SharePoint feature using **stsadm.exe**.*

```
xcopy "$(ProjectDir)\TEMPLATE" "C:\Program Files\Common Files\Microsoft
Shared\web server extensions\12\TEMPLATE" /E /Y
"$(DevEnvDir)\..\..\SDK\v2.0\bin\gacutil.exe" /i "$(TargetPath)" /f
%windir%\system32\cscript.exe c:\windows\system32\iisapp.vbs /a
"SharePointDefaultAppPool" /r
"C:\Program Files\Common Files\Microsoft Shared\web server
extensions\12\bin\stsadm.exe" -o installfeature -name
DocumentArchiveWorkflowPart1 –force
```

- Save and close the project properties window.

6) Rebuild and deploy the new workflow feature.

- Right click the project in the solution explorer and click **Rebuild**.

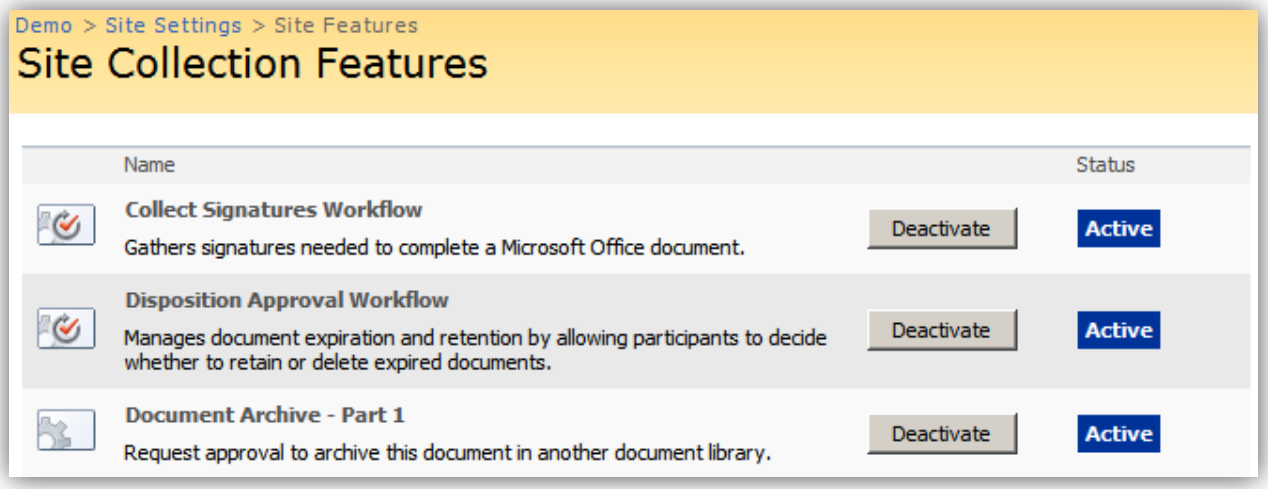
  *If no code changes are made, **Visual Studio 2008** does not re-execute the post build events. If you need to force a redeploy do a rebuild.*

- Look at the **Output** window to determine the post build steps executed successfully.

  *If the **Output** window is not visible, display it using **View -> Output**.*

7) Activate the new **DocumentArchiveWorkflow** feature.

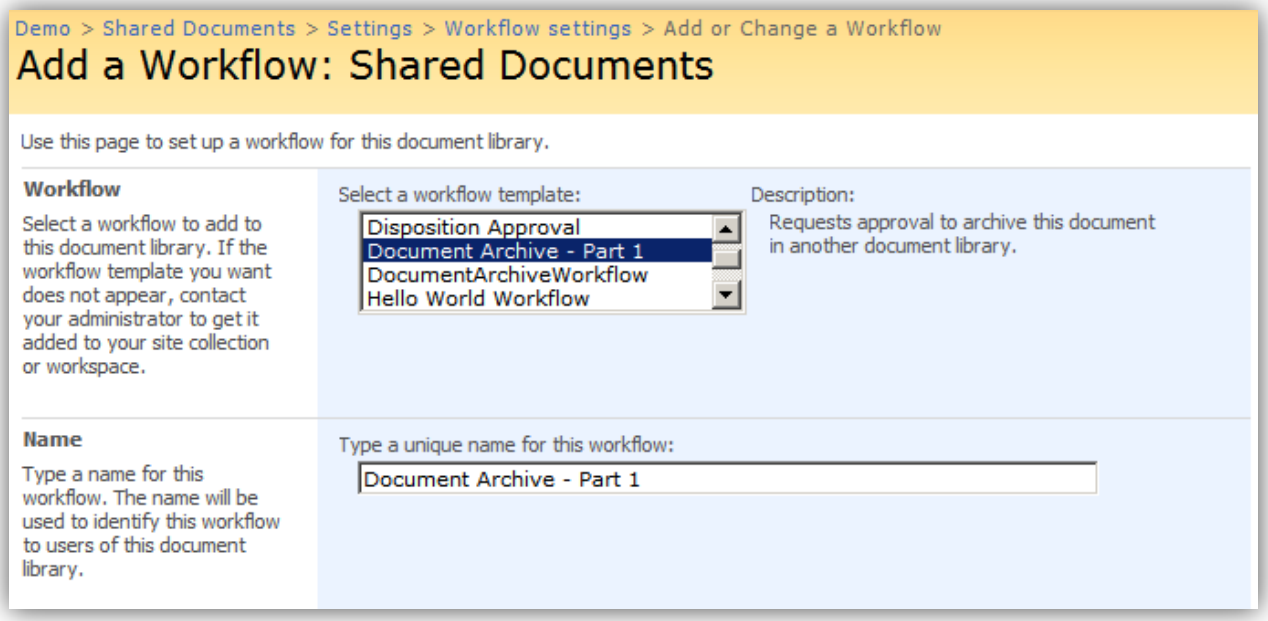
- Using **Internet Explorer** navigate to the **Demo** site collection at **http://litwareinc.com/sites/Demo**.
- Open the features list by clicking **Site Actions -> Site Settings**.
- On the **Site Settings** page, click **Site collection features** in the **Site Collection Administration** section.
- Click the **Activate** button next to the **Document Archive – Part 1** feature.



8) Create an association between the **Shared Documents** document library and the new **Document Archive – Part 1** workflow.

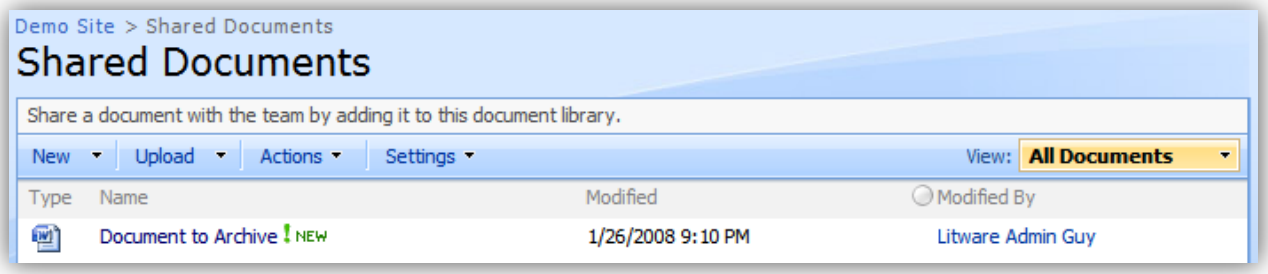
- Navigate to the **Shared Documents** document library in the **Demos** site.
- Click **Settings -> Document Library Settings** to load the settings page.
- Click the **Workflow settings** link in the **Permissions and Management** section.
- Create a new workflow using the **Document Archive – Part 1** workflow template and a name of **Document Archive - Part 1**.

  Use the default values for both list and startup options.



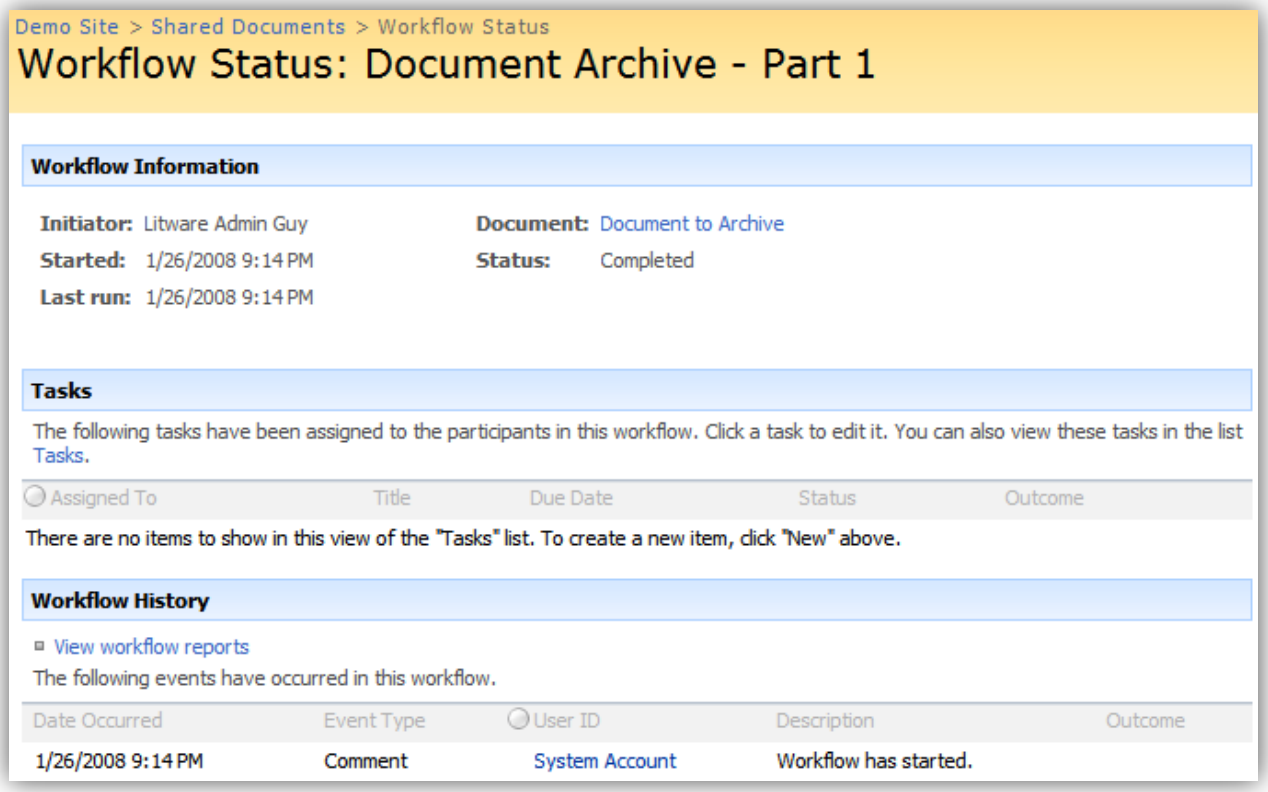
9) Add a new document to the **Shared Documents** folder.

- Navigate to the **Shared Documents** document library in the **Demo** site.
- Click **New -> New document** to create a new document.
- Click **OK** to allow Word 2007 to open the file.
- Enter some data and save the document as **Document To Archive.docx**.



10) Run the **Document Archive – Part 1** workflow on the newly created document.

- Hover over the new document and select **Workflows** from the drop down menu.
- In the workflows page, click the **Document Archive – Part 1** to start the workflow.
- In the **Shared Documents** document library, verify the workflow has completed.
- Click the **Completed** link to view the workflow status and verify the started message was logged to the workflow's history.



Challenge: Deploy the workflow as a Solution Package

1) Create the folder to store the **Solution Package** definition files.

- Create a folder under the project named **Solution**.

  *Right click on the project in the **Solution Explorer** and click **Add -> New Folder**.*
  *Right click the folder and click **Rename**.*
  *Rename the folder **Solution**.*

2) Create the **Manifest.xml** in the **Solution** folder.

- Right click the **Solution** folder and click **Add -> New Item…**.
- Select the **Data** category on the left and select a template of **XML file**.
- Name the new file **Manifest.xml** and click **Add**.

3) Paste the following XML into your manifest file.

- The **FeatureManifest** element defines the features that are part of this solution.
- The **Assembly** elements define the target location for any assemblies that are part of the solution. The GAC is the target location in this case.

```
<?xml version="1.0" encoding="utf-8" ?>
<Solution SolutionId="78c8fd78-0b7f-4893-9361-ff005c32b12c"
xmlns="http://schemas.microsoft.com/sharepoint/">
  <FeatureManifests>
    <FeatureManifest Location="DocumentArchiveWorkflowPart1\Feature.xml" />
  </FeatureManifests>
  <Assemblies>
    <Assembly Location="DocumentArchiveWorkflowPart1.dll"
DeploymentTarget="GlobalAssemblyCache"/>
  </Assemblies>
</Solution>
```

4) Create the **Package.ddf** in the **Solution** folder. Its purpose is to define the generated package.

- Right click the **Solution** folder and click **Add -> New Item…**.
- Select the **General** category on the left and select a template of **Text file**.
- Name the new file **Package.ddf** and click **Add**.

5) Paste the following text into your DDF file to define the files to be packaged.

```
.OPTION EXPLICIT     ; Generate errors
.Set CabinetNameTemplate=DocumentArchiveWorkflowPart1.wsp
.set DiskDirectoryTemplate=CDROM ; All cabinets go in a single directory
.Set CompressionType=MSZIP;** All files are compressed in cabinet files
.Set UniqueFiles="ON"
.Set Cabinet=on
.Set DiskDirectory1=Package

..\..\Solution\Manifest.xml manifest.xml

..\..\Template\Features\DocumentArchiveWorkflowPart1\Feature.xml
DocumentArchiveWorkflowPart1\Feature.xml
..\..\Template\Features\DocumentArchiveWorkflowPart1\Workflow.xml
DocumentArchiveWorkflowPart1\Workflow.xml

DocumentArchiveWorkflowPart1.dll DocumentArchiveWorkflowPart1.dll
```

6) Add another step to the post build process that creates the solution package.

- Right click the project in the **Solution Explorer** and click **Properties**.
- Click the **Build Events** tab in the project properties window.
- Enter the following commands into the **Post-build event command line** window at the top of the list.

```
makecab /f "$(ProjectDir)\Solution\Package.ddf"
```

7) Rebuild the project.

- Right click on the project in the **Solution Explorer** and click **Rebuild**.
- Verify the rebuild was successful.

8) Manually uninstall the previously deployed **Document Archive** feature.

- Open a command window in the SharePoint **bin** directory.

  *Click **Start -> Run** and enter **cmd**.*

  *Navigate to C**:\Program Files\Common Files\Microsoft Shared\web server extensions\12\BIN**.*

- Execute **stsadm.exe** to uninstall the **DocuentArchiveWorkflowPart1** feature.

```
stsadm –o uninstallfeature –name DocumentArchiveWorkflowPart1 –force
```

- Delete the folder at **C:\Program Files\Common Files\Microsoft Shared\web server extensions\12\Template\Features\DocumentarchiveWorkflowPart1**.

9) Install and deploy the new solution package.

- In the same command window as the previous step, execute **stsadm** to add the feature to SharePoint.

```
stsadm –o addsolution –filename
"C:\Labs\Lab04\StarterFiles\DocumentArchiveWorkflowPart1\bin\Debug\Package\Docum
entArchiveWorkflowPart1.wsp"
```

- Deploy the solution to the farm using the **deploysolution** command in **stsadm**.

```
stsadm –o deploysolution –name DocumentArchiveWorkflowPart1.wsp –
allowgacdeployment –local
```

10) Verify the feature is still available and activated.

- Using **Internet Explorer** navigate to the **Demo** site collection at **http://litwareinc.com/sites/Demo**.
- Open the features list by clicking **Site Actions -> Site Settings**.
- On the **Site Settings** page, click **Site collection features** in the **Site Collection Administration** section.
- Verify the **Document Archive – Part 1** feature is active.

Challenge: Using the workflow activation properties.

The **OnWorkflowActivated** activity provides a **WorkflowProperties** property. This property contains information about the **SPListItem** that the workflow is attached to. Use the **DisplayName** property in this object to populate the **HistoryOutcome** of the **LogToHistoryList** activity.

Lab 05: Creating and Waiting on SharePoint Tasks

Lab Overview: Now that the management of Litware Inc. has become comfortable with the built in functionality of SharePoint workflows, they have started to request extra features. They have requested a workflow system that archives documents in another document library. When the document library manager chooses to archive a document, they manually start a workflow, which creates an approval task.

In this lab, you will be extending what you have done in the previous lab. Previously you built the first part of the workflow that simply executed and made an entry into the history list. In this lab, you will be extending that example to include tasks. When the workflow starts, it will create a new task for a user. The workflow will then wait for the task to complete before continuing. Once the tasks are integrated you will create a custom task form that will change the user experience when editing a task from a standard task edit form to a custom form you create.

Exercise 0: Setup

1) If you did not complete lab 4, you'll need to create the **Demo** site collection.

   Open **SharePoint** and browse to the **Demo** site collection

   *The url is **http://litwareinc.com/sites/Demo**.*
   *If the site collection does not exist, create it using the **CreateDemo.bat** file in the **C:\Labs\Files folder**.*

2) Open the starter **VS 2008** solution at **\Labs\Lab05\StarterFiles\DocumentArchive Part 2.sln**.

Exercise 1: Creating SharePoint Tasks in a Workflow

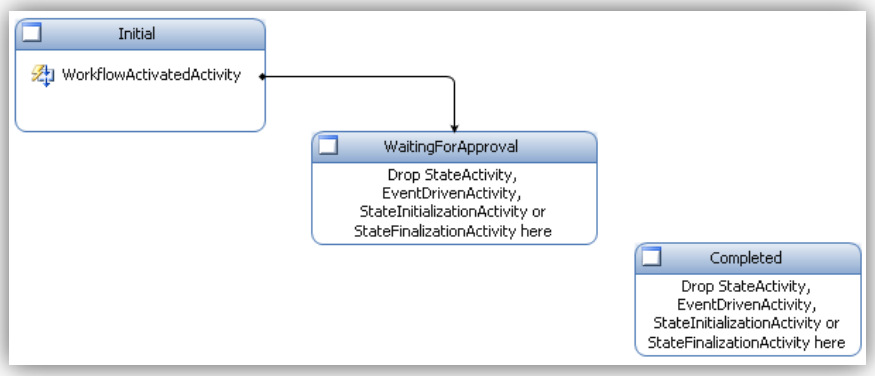
1) Add a new state named **WaitingForApproval** to the workflow and change **WorkflowActivatedActivity** so it transitions to the **WaitingForApproval** state instead of the **Completed** state.

   - Open **Workflow.cs** in design mode by right clicking it in the **Solution Explorer** and clicking **View Designer**.
   - Right click on the **State Machine** designer canvas and click **Add State**.
   - Set the name of the new state to **WaitingForApproval**.

     *Right click the new state and click **Properties**.*
     *In the properties pane, set the **Name** property to **WaitingForApproval**.*
   - Double click the **WorkflowActivatedActivity** to open its designer.
   - Change the name of **MoveToCompleted** state to **MovingToWaitingForApproval**.

     *Right click the **MoveToCompleted** activity and click **Properties**.*
     *In the properties pane, set the **Name** property to **MoveToWaitingForApproval**.*
   - Change the new **MoveToWaitingForApproval** activity's **TargetStateName** to **WaitingForApproval**.

     *Right click the **MoveToWaitingForApproval** activity and click **Properties**.*
     *In the properties pane, set the **TargetStateName** property to **WaitingForApproval**.*
   - Switch back to the **Workflow** view by clicking the **Workflow** link in the upper left hand corner of the State Machine designer canvas.



2) Add a new **StateInitializationActivity** to the **WaitingForApproval** state and name if **WaitingForApprovalInit**.

- Right click the **WaitingForApproval** state and click **Add StateInitialization**.
- Set the name of the new **StateInitializationActivity** to **WaitingForApprovalInit**.

  *In the properties pane, set the **Name** property to **WaitingForApprovalInit**.*

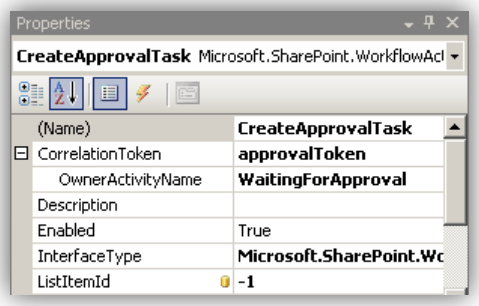
3) Create a new task in the **WaitingForApprovalInit** activity.

- Drag a new **CreateTask** activity from the Toolbox into the **WaitingForAprovalInit** activity and name it **CreateApprovalTask**

  *In the properties pane, set the **Name** property to **CreateApprovalTask**.*

- In the **CreateApprovalTask's** properties pane, create a new correlation token scoped to the **WaitingFormApproval** state named **ApprovalToken**.

  *In the properties pane, set the **CorrelationToken** property to **ApprovalToken**.*
  *Expand the **CorrelationToken** property in the property pane.*
  *Set the **OwnerActivityName** property to **WaitingForApproval***



- Add a **TaskId** property to the workflow class to store the Id of the new task.

  *View the code for the **Workflow** class by right clicking it in the **Solution Explorer** and selecting **View Code**.*

  *Add a public property named **ApprovalTaskId** to the class with a type of **Guid**.*

```
public Guid ApprovalTaskId { get; set; }
```

- Bind the **TaskId** property of the **CreateApprovalTask** activity to the **ApprovalTaskId** property of the workflow.

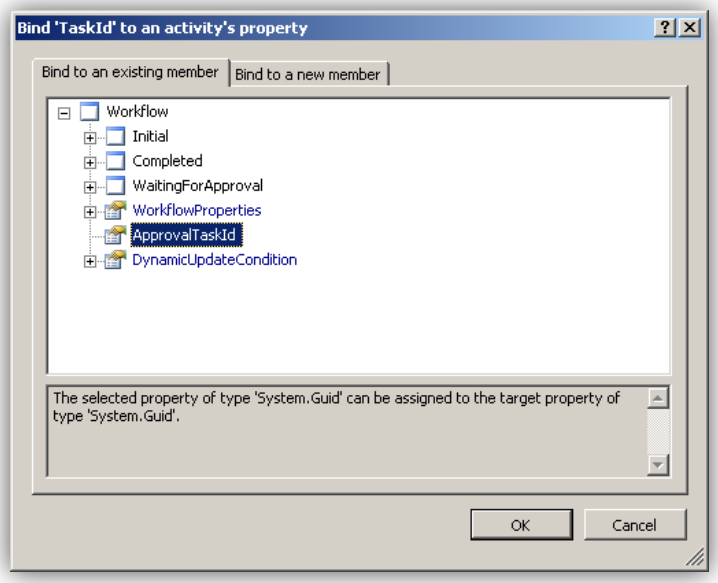
  *View the workflow designer by clicking **Workflow.cs** in the **Solution Explorer** and selecting **View Designer**.*

  *Open the **WaitForApprovalInit** activity by double clicking it.*

  *Right click the **CreateApprovaltask** activity and click **Properties**.*

  *Open the **Binding** dialog by selecting the **TaskId** property and clicking the **…** button.*

  *In the **Binding** dialog select the **ApprovalTaskId** property and click **OK**.*



- Create a handler for the **MethodInvoking** event that will set the details of the new task to create.

  *In the properties pane, set the **MethodInvoking** property to **CreateApprovalTask_Invoking** and press **Enter**.*

- Initialize **ApprovalTaskId** and set the details of the **TaskProperties** property of the **CreateApprovalTask** activity.

  *In the **CreateApprovalTask_Invoking** method, initialize the value of **ApprovalTaskId** to the results of the **Guid.NewGuid** method.*

  *Create a new **SPWorkflowTaskProperties** object named **taskProperties**.*

  *Set its **AssignedTo** property to **Administrator**.*

  *Set its **Title** to "**Archive Approval**".*

  *Set its **Description** to "**This document was flagged to be archived. Please approve or reject this action.**"*

```
ApprovalTaskId = Guid.NewGuid();

SPWorkflowTaskProperties taskProperties = new SPWorkflowTaskProperties();
taskProperties.AssignedTo = "LITWAREINC\\Administrator";
taskProperties.Title = "Archival Approval";
taskProperties.Description = "This document was flagged to be archived.  Please
approve or reject this action.";
```

- Assign the **taskProperties** object to the **CreateTask** activity that is currently executing.

*Cast the **sender** property to a **CreateTask** object and store the result in a **createApprovalTask** variable.*
*Set **createApprovalTask's TaskProperties** property using the **taskProperties** variable.*

```
CreateTask createApprovalTask = sender as CreateTask;
createApprovalTask.TaskProperties = taskProperties;
```

4) Add a new **EventDrivenActivity** named **WaitingForApprovalTaskChanged** to the **WaitingForApproval** state.

- Right click **Workflow.cs** in the **Solution Explorer** and click **View Designer**.
- Right click the **WaitingForApproval** state and click **Add EventDriven**.
- Set the name of the new **EventDrivenActivity** to **WaitingForApprovalTaskChanged**.

*Right click the new **EventDrivenActivity** and click **Properties**.*
*In the properties pane, set the **Name** property to **WaitingForApprovalTaskChanged**.*

5) Create a new **ApprovalTaskResult** property to the workflow to use when storing the result of the activity.

- To store task results efficiently, create a new **enum** named **TaskResult** to identify the task result.

*Right click on the project in the **Solution Explorer** and click **Add -> Class**.*
*Name the class **TaskResults.cs** and click **Add**.*
*Replace the class definition with a public **enum** definition named **TaskResults** with options for **None** and **Approved**.*

```
public enum TaskResults
{
    None,
    Approved,
    Rejected
}
```

- Open the **Workflow.cs** code and add a new public property named **ApprovalTaskResult** of type **TaskResult**.

*Right click on the **Workflow.cs** workflow in the **Solution Explorer** and click **View Code**.*
*Add a new property named **ApprovalTaskResult** of type **TaskResult** to the class.*

```
public Guid ApprovalTaskId { get; set; }
public TaskResults ApprovalTaskResult { get; set; }
```

6) Add an **OnTaskChanged** event to the **WaitingForApproval** state that waits for the previously created task to change.

- Drag a new **OnTaskChanged** activity from the **Toolbox** into the **WaitingForApprovalTaskChanged** activity and name it **OnApprovalTaskChanged.**

  *In the properties pane, set the **Name** property to **OnApprovalTaskChanged**.*

- In the **OnApprovalTaskChanged** properties pane, select the existing **ApprovalToken** correlation token.

  *Right click the **OnApprovalTaskChanged** activity and click **Properties**.*
  *In the properties pane, set the **CorrelationToken** property to **ApprovalToken**.*

- In the **OnApprovalTaskChanged** properties pane, bind **TaskId** to **ApprovalTaskId**.

  *Open the **Binding** dialog by selecting the **TaskId** property and clicking the **…** button.*
  *In the **Binding** dialog select the **ApprovalTaskId** property and click **OK**.*

7) When the task changes, check if the status is **Completed**.

- Add an **Invoked** handler to the **OnApprovalTaskChanged** activity.

  *Right click the **OnApprovalTaskChanged** activity and click **Properties**.*
  *In the properties pane, set the **Invoked** property to **OnApprovalTaskChanged_Invoked** and press **Enter**.*

- In the **OnApprovalTaskChanged_Invoked** method cast the **e** parameter to **SPTaskServiceEventArgs** to access the activity specific event parameters and store it for later in the **args** variable.

```
SPTaskServiceEventArgs args = e as SPTaskServiceEventArgs;
```

- Access the **Status** column of the task by accessing it in the **args's afterProperties.ExtendedProperties** property. Use the Guid ID of the column to retrieve its value.

```
Guid statusFieldId = new Guid("{c15b34c3-ce7d-490a-b133-3f4de8801b76}");
string result = args.afterProperties.ExtendedProperties[statusFieldId] as
string;
```

- If the result of the task change is **Completed**, set the **ApprovalTaskResult** property to **TaskResults.Approved**.

```
if (result == "Completed")
    this.ApprovalTaskResult = TaskResults.Approved;
```

8) If the task change indicated that the task was completed, log the result and transition to the completed state.

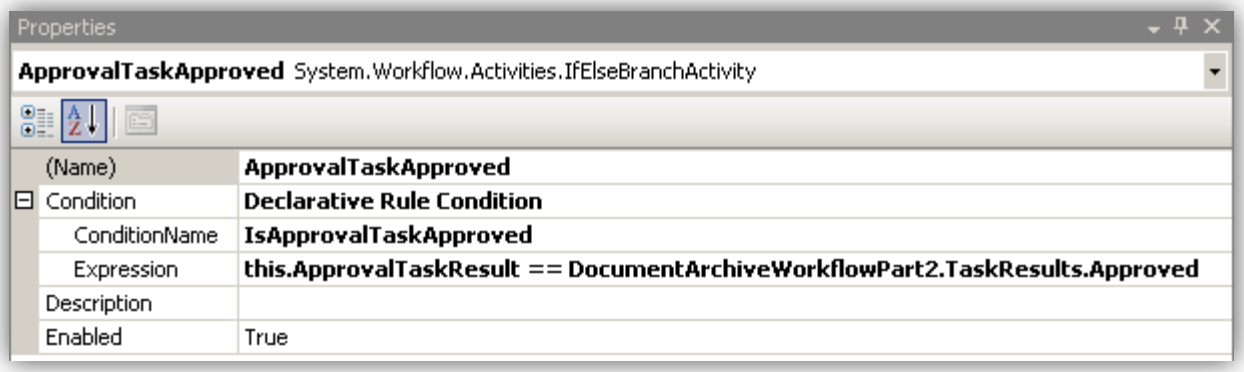
- Add a new **IfElse** activity to the **WaitingForApprovalTaskChanged** activity with the name of **CheckApprovalTaskResult**.

  *Drag a new **IfElse** activity from the **Toolbox** into the **WaitingForApprovalTaskChanged** following the **OnApprovalTaskChanged** activity.*
  *In the properties pane, set the **Name** property to **CheckApprovalTaskResult**.*

- Set the left branch of the **CheckApprovalTaskResult** activity's name to **ApprovalTaskApproved** and setup its condition.

  *In the properties pane, set the **Name** property to **ApprovalTaskApproved**.*
  *Set the **Condition** property to **Declarative Rule Condition** and expand the property.*
  *Set the **ConditionName** property to **IsApprovalTaskApproved**.*
  *Set the **Expression** property to:*

```
this.ApprovalTaskResult == TaskResults.Approved
```



- Set the right branch of the **CheckApprovalTaskResult** activity's name to **ApprovalTaskInProgress**.

  *Right click the right branch of **CheckApprovalTaskResult** and click **Properties**.*
  *In the properties pane, set the **Name** property to **ApprovalTaskInProgess**.*

- Add a new **LogToHistoryListActivity** to the **ApprovalTaskApproved** activity and name it **LogApprovalTaskApproved**.

  *In the properties pane, set the **Name** property to **LogApprovalTaskApproved**.*
  *Set the **HistoryDescription** property to **"Approval task approved."***

- Add a new **SetState** activity to the **ApprovalTaskApproved** activity following the **LogApprovalTaskApproved** and name it **MoveOnApprovalTaskApproved**.

  *In the properties pane, set the **Name** property to **MoveOnApprovalTaskApproved**.*
  *Set the **TargetStateName** to **Completed**.*

9) Add a new **StateFinalizationActivity** to the **WaitingForApproval** state and name it **WaitingForApprovalFinal**.

- Add a **StateFinalization** activity to the **WaitingForApproval** state and name it **WaitingForApprovalFinal**

  *Right click the **WaitingForApproval** state and select **Add StateFinalization***
  *Right click the new **StateFinalization** activity and set its name to **WaitingForApprovalFinal**.*

10) In the **WaitingForApprovalFinal** activity, if the activity was completed, use the **CompleteTask** activity to mark the task as completed.

- Add a new **IfElse** activity to the **WaitingForApprovalTaskFinal** activity with the name of **CheckApprovalTaskFinalAction**.

  *Drag a new **IfElse** activity from the **Toolbox** into the **WaitingForApprovalFinal**.*
  *In the properties pane, set the **Name** property to **CheckApprovalTaskFinalAction**.*

- Set the left branch of the **CheckApprovalTaskFinalAction** activity's name to **ApprovalTaskComplete** and setup its condition.

  *Right click the left branch of **CheckApprovalTaskFinalAction** and click **Properties**.*
  *In the properties pane, set the **Name** property to **ApprovalTaskComplete**.*
  *Set the **Condition** property to **Declarative Rule Condition** and expand the property.*
  *Set the **ConditionName** property to **IsApprovalTaskComplete**.*
  *Set the **Expression** property to*

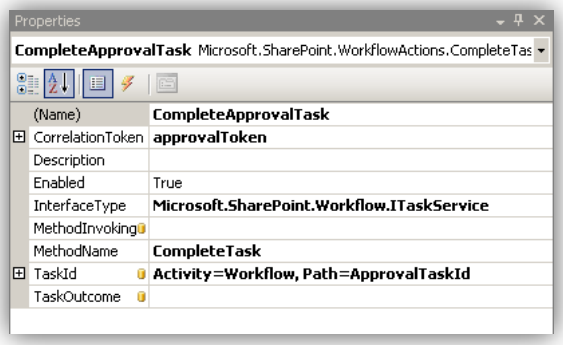
```
this.ApprovalTaskResult != TaskResults.None
```

- Set the right branch of the **CheckApprovalTaskFinalAction** activity's name to **ApprovalTaskIncomplete**.

  *Right click the right branch of **CheckApprovalTaskFinalAction** and click **Properties**.*
  *In the properties pane, set the **Name** property to **ApprovalTaskIncomplete**.*

- Mark the task as complete using the **CompleteTask** activity if the task was determined to be complete.

  *Drag a new **CompleteTask** activity into the **ApprovalTaskComplete** activity.*
  *In the properties pane, set the **Name** property to **CompleteApprovalTask**.*
  *Set the **CorrelationToken** property to **ApprovalToken**.*
  *Bind the **TaskId** to the **ApprovalTaskId** property in the Workflow class.*

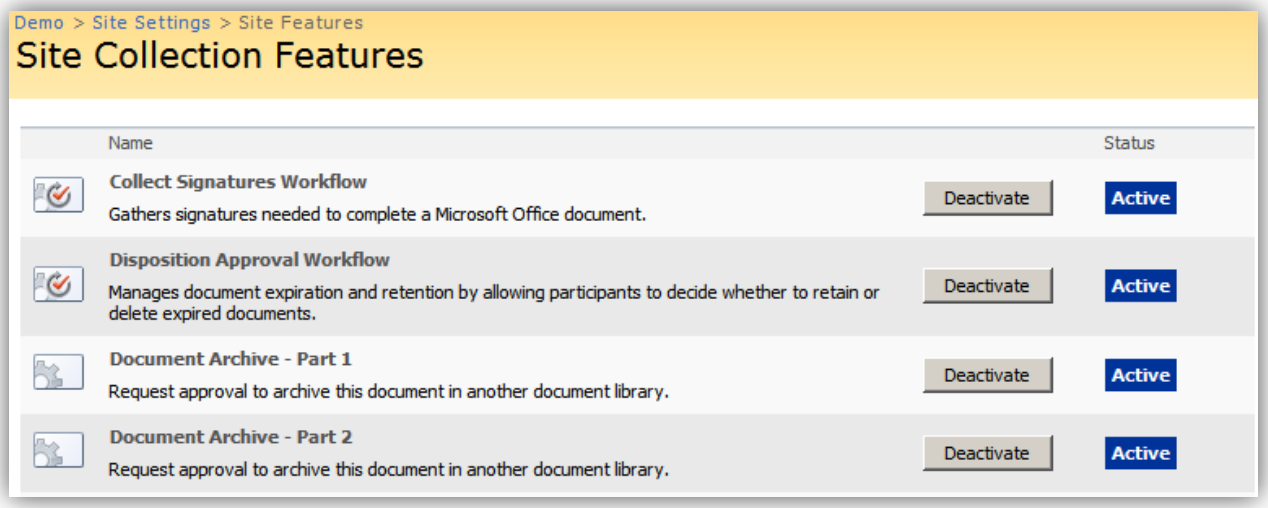


11) Rebuild the workflow.

- Right click the project in the **Solution Explorer** and click **Rebuild**.
- In the **Output** window, verify the post build actions completed successfully.

12) Activate the new **DocumentArchiveWorkflow** feature.

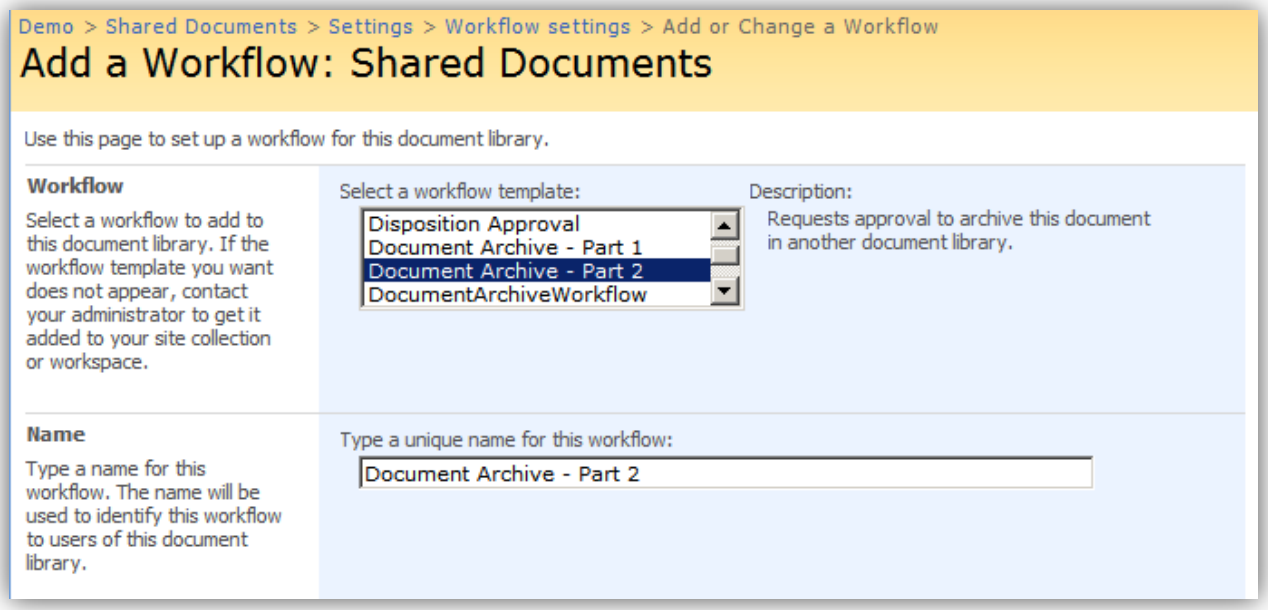
- Using **Internet Explorer** navigate to the **Demo** site collection at **http://litwareinc.com/sites/Demo**.
- Open the features list by clicking **Site Actions -> Site Settings**.
- On the **Site Settings** page, click **Site collection features** in the **Site Collection Administration** section.
- Click the **Activate** button next to the **Document Archive – Part 2** feature.



13) Create an association between the **Shared Documents** document library and the new **Document Archive – Part 2** workflow.

- Navigate to the **Shared Documents** document library in the **Demos** site.
- Click **Settings -> Document Library Settings** to load the settings page.
- Click the **Workflow settings** link in the **Permissions and Management section**.
- Create a new workflow using the **Document Archive – Part 2** workflow template and a name of **Document Archive - Part 2**.

  Use the default values for both list and startup options.

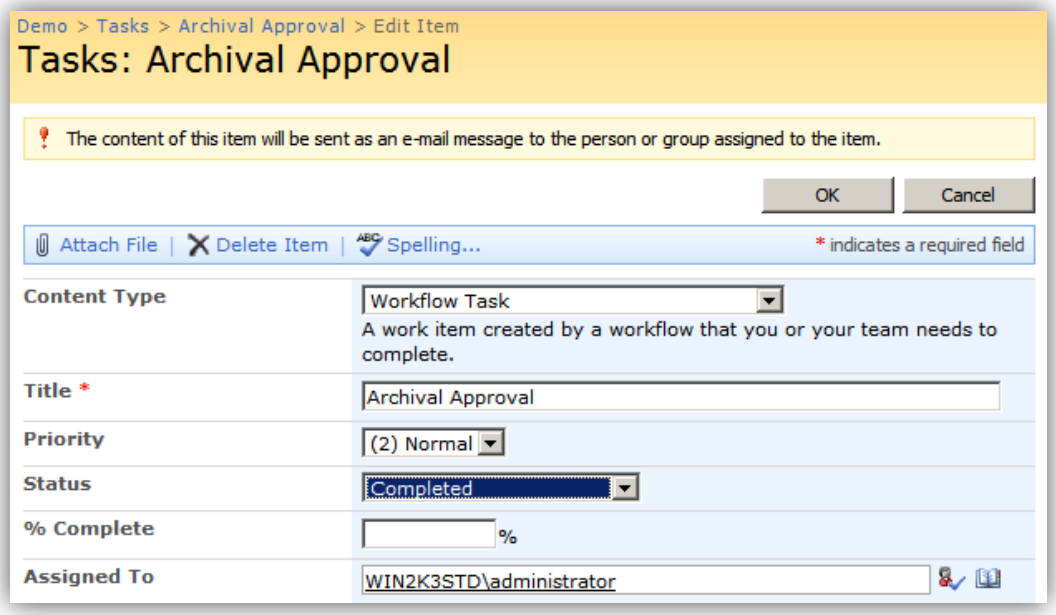


14) Run the **Document Archive – Part 2** workflow on a document.

- Hover over the new document and select **Workflows** from the drop down menu.
- In the workflows page, click the **Document Archive – Part 2** to start the workflow.
- In the **Shared Documents** document library, verify the workflow is running.
- Click the **In Progress** link to view the workflow status and verify the started message was logged to the workflow's history.
- Edit the task and set its **Status** to **Completed**.

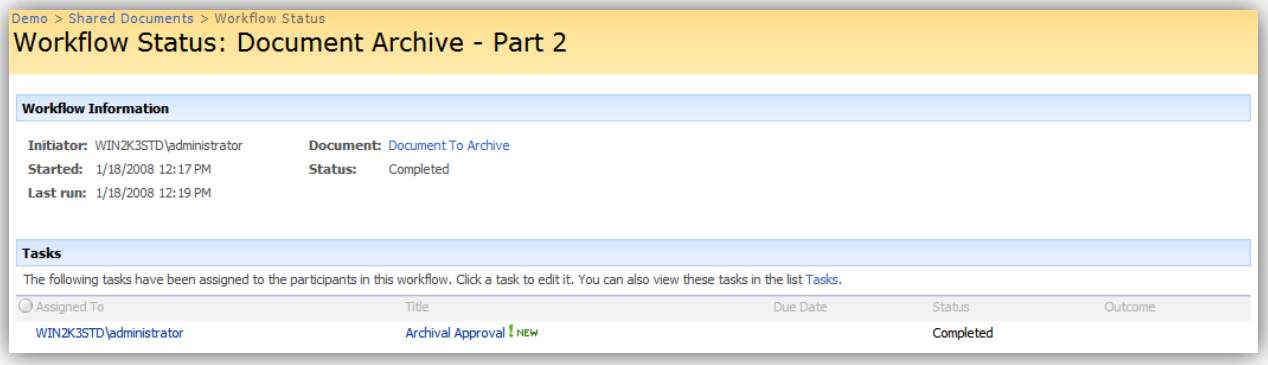
  *In the **Tasks** section, hover over the task's title and click **Edit Item** in the drop down menu.*

  *In the **Edit Task** page set its **Status** to **Completed** and click **Save**.*



- Verify that the workflow is now completed and the task is marked as completed.

  You may have to refresh the status page to see the change.



Exercise 2: Creating custom Task Forms

1) Add a new base class named **DocArchivePart2TaskForm** that will be uses as the code behind for an ASPX page.

   - Create a new folder named **UI** under the project.

     *Right click on the project in the **Solution Explorer** and click **Add -> New Folder**.*
     *Right click the folder and click **Rename**.*
     *Rename the folder **UI**.*

   - Create a new class named **DocArchivePart2TaskForm** in the new UI folder.

     *Right click the **UI** folder and click **Add -> Class…**.*
     *Name the new file **DocArchivePart2TaskForm.cs** and click **Add**.*

   - Change the **DocArchivePart2TaskForm** class into a public class that derives from **Microsoft.SharePoint.WebControls.LayoutsPageBase**.

```
public abstract class DocArchivePart2TaskForm : LayoutsPageBase
{ }
```

   - Add private fields to the class to store information about the task item, the workflow, and the list the workflow is attached to.

```
private SPWorkflow _workflow;
private SPWorkflowTask _task;
private SPList _taskList;
private SPListItem _taskItem;
```

2) Add a protected field and property that will allow interaction between the ASPX page and the code behind.

   - Add a protected field named **lblListName** of type **Label** to the class.
   - Add a protected field named **lnkItem** of type **Hyperlink** to the class.
   - Add a protected field named **lstArchiveList** of type **DropDownList** to the class.
   - Add a protected property named **TaskItemName** of type string to the class. Have it return the **DisplayName** property of the **_taskItem** field.

```
protected Label lblListName;
protected HyperLink lnkItem;
protected DropDownList lstArchiveList;

protected string TaskItemName
{
    get { return _taskItem.DisplayName; }
}
```

3) Override the **OnLoad** method in the new **DocArchivePart2TaskForm** class so it reads the **Url** parameters and creates the necessary SharePoint API objects and binds that data to the UI.

   - Override the **OnLoad** method of the base class.

```
protected override void OnLoad(EventArgs e)
{
}
```

- Read the **List** and **ID** url parameters and use them to find the task list and task list item to edit.

```
// find the tasks list and task list item
_taskList = Web.Lists[new Guid(Request.Params["List"])];
_taskItem = _taskList.GetItemById(int.Parse(Request.Params["ID"]));
```

- Using the **_taskItem** object's **WorkflowInstanceID** field, find the **SPWokflow** object representing the running workflow.

```
// use the task list item to find the workflow object
_workflow = new SPWorkflow(Web,
  new Guid(_taskItem["WorkflowInstanceID"] as string));
```

- Use the **SPWorkflow** object to lookup the **SPWorkflowTask** object using the **Tasks** collection.

```
// using the workflow object, lookup the task
_task = _workflow.Tasks[0];
```

- If this request is not a postback, call the **PopulateControls** and **BindTaskData** methods to give the derived class a chance to populate the form.

```
// bind the task data
if (!this.IsPostBack)
{
    // filter the list of lists for all document libraries that aren't hidden
    IEnumerable<SPList> documentLibraries =
        Web.Lists.Cast<SPList>().Where(
            n => n is SPDocumentLibrary && !n.Hidden);

    // add each visible document library to the list
    foreach (SPList documentLibrary in documentLibraries)
        lstArchiveList.Items.Add(
            new ListItem(documentLibrary.Title, documentLibrary.ID.ToString()));

    // initialize the controls
    lnkItem.Text = _workflow.ParentItem.DisplayName;
    lnkItem.NavigateUrl = _workflow.ParentItem.Url;
    lblListName.Text = _workflow.ParentList.Title;
}
```

- Call the base implementation of the **OnLoad** method

```
// call the base implementation
base.OnLoad(e);
```

4) Implement the event handler called after a click of the **Cancel** button.

- Create a **protected** event handler method named **Cancel_Click**.
- Use the **SPUtility.Redirect** method to handle the automatic redirection to either the source url parameter or the task list's.

```
protected void Cancel_Click(object sender, EventArgs e) {
    // redirect to the page in the source url parameter or the default page
    SPUtility.Redirect(_taskList.DefaultViewUrl,
        SPRedirectFlags.UseSource, this.Context);
}
```

5) Implement the event handler called after a click of the **Reject** button.

- Create a **protected** event handler method named **Reject_Click**.
- Create a new **Hashtable** object with a single entry named Result. Set its value to **TaskResults.Rejected**.
- Use the **SPWorkflowTask.AlterTask** to make the changes defined in the **Hashtable** to the task object.
- Use the **SPUtility.Redirect** method to handle the automatic redirection to either the source url parameter or the task list's.

```
protected void Reject_Click(object sender, EventArgs e) {
    // setup the task's new data
    Hashtable data = new Hashtable();
    data.Add("Result", TaskResults.Rejected);

    // submit the changes to the task
    SPWorkflowTask.AlterTask(_taskItem, data, true);

    // redirect to the page in the source url parameter or the default page
    SPUtility.Redirect(_taskList.DefaultViewUrl,
        SPRedirectFlags.UseSource, this.Context);
}
```

6) Implement the event handler called after a click of the **Approve** button.

- Create a **protected** event handler method named **Approve_Click**.
- Copy the code from **Reject_Click** into **Approve_Click** and change the result to **TaskResult.Approved**.
- Add one more parameter to the **Hashtable** named **ArchiveListId** with a value of **lstArchiveList.SelectedValue**.

```
protected void Approve_Click(object sender, EventArgs e) {
    // setup the task's new data
    Hashtable data = new Hashtable();
    data.Add("Result", TaskResults.Approved);
    data.Add("ArchiveListId", lstArchiveList.SelectedValue);

    // submit the changes to the task
    SPWorkflowTask.AlterTask(_taskItem, data, true);

    // redirect to the page in the source url parameter or the default page
    SPUtility.Redirect(_taskList.DefaultViewUrl,
        SPRedirectFlags.UseSource, this.Context);
}
```

7) Create a new .ASPX page named **DocArchivePart2TaskForm.aspx** that represents the ASP.NET markup for the task page.

- Create a new **Layouts** folder under the existing **Template** folder.

  *Right click on the **Template** folder in the solution explorer and click **Add -> New Folder**.*

  *Right click the folder and click **Rename**.*

  *Rename the folder **Layouts**.*

- Create a new text file named **DocArchivePart2TaskForm.aspx** in the new **Layouts** folder.

  *Right click the **Layouts** folder and click **Add -> New Item…**.*

  *Select the **Generate** category on the left and select a template of **Text file**.*

  *Name the new file **DocArchivePart2TaskForm.aspx** and click **Add**.*

- Enter the page directives that define the code behind class and the master page for the task form.

```
<%@ Assembly Name="DocumentArchiveWorkflowPart2, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=15812f954569663f" %>

<%@ Page Language="C#" MasterPageFile="~/_layouts/application.master"
    EnableSessionState="true" ValidateRequest="False"
    Inherits="DocumentArchiveWorkflowPart2.UI.DocArchivePart2TaskForm" %>
```

- Register the tag prefix and locations of the SharePoint web and user controls used to format the page.

```
<%@ Register TagPrefix="SharePoint" Namespace="Microsoft.SharePoint.WebControls"
    Assembly="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce111e9429c" %>
<%@ Register TagPrefix="Utilities" Namespace="Microsoft.SharePoint.Utilities"
    Assembly="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce111e9429c" %>
<%@ Register TagPrefix="wssuc" TagName="InputFormSection"
Src="/_controltemplates/InputFormSection.ascx" %>
<%@ Register TagPrefix="wssuc" TagName="InputFormControl"
Src="/_controltemplates/InputFormControl.ascx" %>
<%@ Register TagPrefix="wssuc" TagName="ButtonSection"
Src="/_controltemplates/ButtonSection.ascx" %>
```

- Define the content for the page title and the form specific headers.

```
<asp:Content ID="PageTitle" ContentPlaceHolderID="PlaceHolderPageTitle"
runat="server">Workflow Task</asp:Content>
<asp:Content ID="PageTitleInTitleArea"
ContentPlaceHolderID="PlaceHolderPageTitleInTitleArea"
    runat="server">
    <%= "Tasks : " + this.TaskItemName %>
</asp:Content>
```

8) Define the ASPX markup that will be used to create the data entry section of the custom task form.

- Define the **ContentPlaceHolder** control for **PlaceHolderMain** and place a table within it.

```
<asp:Content ID="Main" ContentPlaceHolderID="PlaceHolderMain" runat="server">
    <table cellspacing="0" cellpadding="0" style="border: none; width: 100%"
class="ms-propertysheet">
    </table>
</asp:Content>
```

- Implement the first **InputFormSection** inside the **table** element that will display summary information about the task. It will contain two **Label** controls with the names **lblListName** and **lnkItem**.

```
<%-- Hyperlink to item/document --%>
<wssuc:InputFormSection Title="Item Requiring Approval" Description="Please
review this item/document."
    runat="server">
    <template_inputformcontrols>
        <wssuc:InputFormControl LabelText="Click on item hyperlink below to
review item/document" runat="server">
          <Template_Control>
            <table border="0" cellpadding="3" cellspacing="0" style="font-
size:8pt;">
              <tr>
                <td>List:</td>
                <td><asp:Label runat="server" ID="lblListName" /></td>
              </tr>
              <tr>
                <td>Item:</td>
                <td><asp:HyperLink runat="Server" ID="lnkItem" Target="_blank"
/></td>
              </tr>
            </table>
          </Template_Control>
        </wssuc:InputFormControl>
    </template_inputformcontrols>
</wssuc:InputFormSection>
```

- Implement the second **InputFormSection** that will allow the approver to override the list where the document will be archived.

```
<%-- Archive List Input --%>
<wssuc:InputFormSection Title="Archive Document Library" Description="Specify
the document library used to store archived documents."
    runat="server">
    <template_inputformcontrols>
        <wssuc:InputFormControl LabelText="Archive Document Library:"
runat="server">
                <Template_Control>
                    <asp:DropDownList ID="lstArchiveList" runat="server" />
                </Template_Control>
        </wssuc:InputFormControl>
    </template_inputformcontrols>
</wssuc:InputFormSection>
```

- Implement the last section containing the **Approve**, **Reject**, and **Cancel** buttons.

```
<%-- Approve, Reject and Cancel Buttons --%>
<wssuc:ButtonSection runat="server" ShowStandardCancelButton="false">
    <template_buttons>
   <asp:PlaceHolder runat="server">
        <asp:Button UseSubmitBehavior="false" runat="server" class="ms-
ButtonHeightWidth" OnClick="Approve_Click" Text="Approve" id="btnApprove" />
 
            <asp:Button UseSubmitBehavior="false" runat="server" class="ms-
ButtonHeightWidth" OnClick="Reject_Click" Text="Reject" id="btnReject"  />
 
            <asp:Button UseSubmitBehavior="false" runat="server" class="ms-
ButtonHeightWidth" OnClick="Cancel_Click" Text="Cancel" id="btnCancel"  />
    </asp:PlaceHolder>
</template_buttons>
</wssuc:ButtonSection>
```

Exercise 3: Integrating custom Task Forms into the workflow

1) Update the **Workflow** code that will execute when a task changes. Use the new **Result** extended property to determine the result of the workflow instead of the **Status** field.

   - View the code for **Workflow.cs** by right clicking it in the **Solution Explorer** and selecting **View Code**.
   - Locate the **OnApprovaltaskChanged_Invoked** method and replace the code processing the extended properties with code that looks from the **Result** property and converts it into a **TaskResult** enumeration.

```
//Guid statusFieldId = new Guid("{c15b34c3-ce7d-490a-b133-3f4de8801b76}");
//string result = args.afterProperties.ExtendedProperties[statusFieldId] as
string;
//if (result == "Completed")
//    this.ApprovalTaskResult = TaskResults.Approved;

string result = args.afterProperties.ExtendedProperties["Result"] as string;
this.ApprovalTaskResult = (TaskResults)Enum.Parse(typeof(TaskResults), result);
```

2) Add another branch in the **WaitingForApprovalTaskChanged** to handle the Rejected result from the custom task form.

   - Add a new branch named **ApprovalTaskRejected**.to the **CheckApprovalTaskResult** activity.

     *Right click on the **CheckApprovalTaskResult** and click **Add Branch**.*
     Drag the new branch into the middle of the existing branches.
     *In the properties pane, set the **Name** to **ApprovalTaskRejected**.*
     *Set the **Condition** property to **Declarative Rule Condition** and expand the property.*
     *Set the **ConditionName** property to **IsApprovalTaskRejected**.*
     *Set the **Expression** property to*

```
this.ApprovalTaskResult == TaskResults.Rejected
```

   - Add a new **LogToHistoryListActivity** to the **ApprovalTaskRejected** activity and name it **LogApprovalTaskRejected**.

     *Right click the new **LogToHistoryListActivity** and click **Properties**.*
     *In the properties pane, set the **Name** property to **LogApprovalTaskRejected**.*
     *Set the **HistoryDescription** property to **"Approval task rejected."***

- Add a new **SetState** activity to the **ApprovalTaskRejected** activity following the **LogApprovalTaskRejected** and name it **MoveOnApprovalTaskRejected**.

  *Right click the new **SetState** activity and click **Properties**.*

  *In the properties pane, set the **Name** property to **MoveOnApprovalTaskRejected**.*

  *Set the **TargetStateName** to **Completed**.*

Exercise 4: Deploying custom Task Forms

1) Create a new content type for the workflow's tasks.

   - Open the **Workflow.xml** file in the **Template\Features\DocumentArchiveWorkflowPart2** folder.
   - Following the **Workflow** element, add the following XML fragment defining the new content type.

     *The definition of the **Edit** and **Display** forms for the content type determine the form shown when the user views or edits the task.*

```
<ContentType ID="0x01080100F16BC2E95B0E4F1D862D3773F914E20A"
Name="DocumentArchiveWokflowPart2 Task" Group="DocArchiveWf Content Types"
Description="Task used by the Document Archive workflow." Version="0"
Hidden="False">
  <FieldRefs>
  </FieldRefs>
  <XmlDocuments>
    <XmlDocument NamespaceURI
="http://schemas.microsoft.com/sharepoint/v3/contenttype/forms/url">
      <FormUrls
xmlns="http://schemas.microsoft.com/sharepoint/v3/contenttype/forms/url">
        <Edit>_layouts/DocArchivePart2TaskForm.aspx</Edit>
        <Display>_layouts/DocArchivePart2TaskForm.aspx</Display>
      </FormUrls>
    </XmlDocument>
  </XmlDocuments>
</ContentType>
```

2) Register the new content type as the default type for the workflow by adding a **TaskListContentTypeId** attribute to the feature's **Workflow** element.

```
<Workflow ...
     TaskListContentTypeId="0x01080100F16BC2E95B0E4F1D862D3773F914E20A"
     StatusUrl="_layouts/WrkStat.aspx">
  <Categories/>
  <MetaData>
  </MetaData>
</Workflow>
```

3) Rebuild the workflow.

   - Right click the project in the **Solution Explorer** and click **Rebuild**.
   - In the **Output** window, verify the post build actions completed successfully.

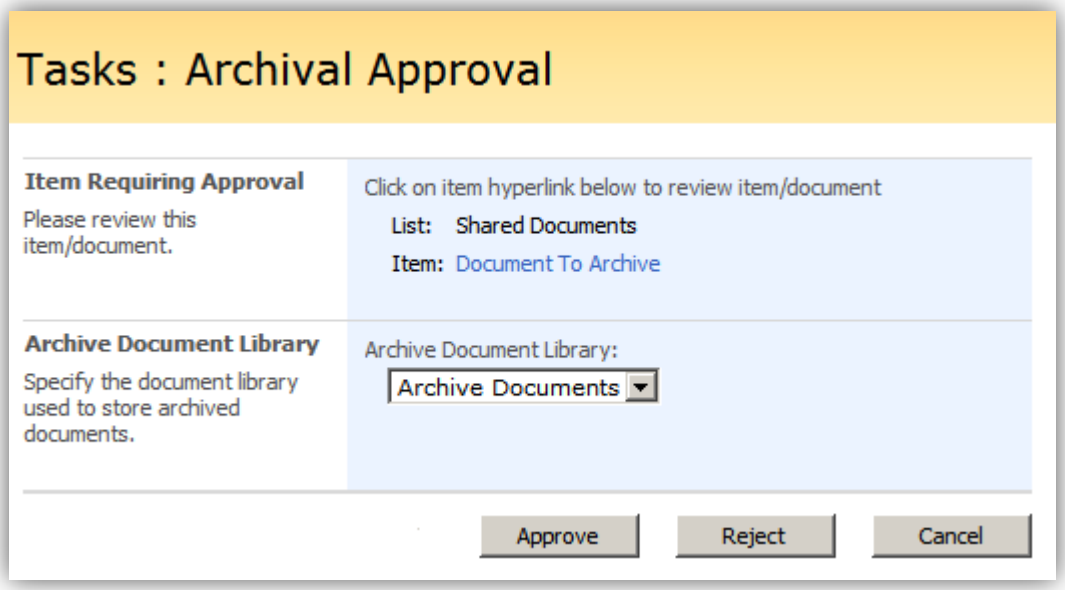
4) Run the **Document Archive – Part 2** workflow on a document.

   - Hover over the new document and select **Workflows** from the drop down menu.
   - In the workflows page, click the **Document Archive – Part 2** to start the workflow.
   - In the **Shared Documents** document library, verify the workflow is running.

- Click the **In Progress** link to view the workflow status and verify the started message was logged to the workflow's history.
- Edit the task and reject the archive process.

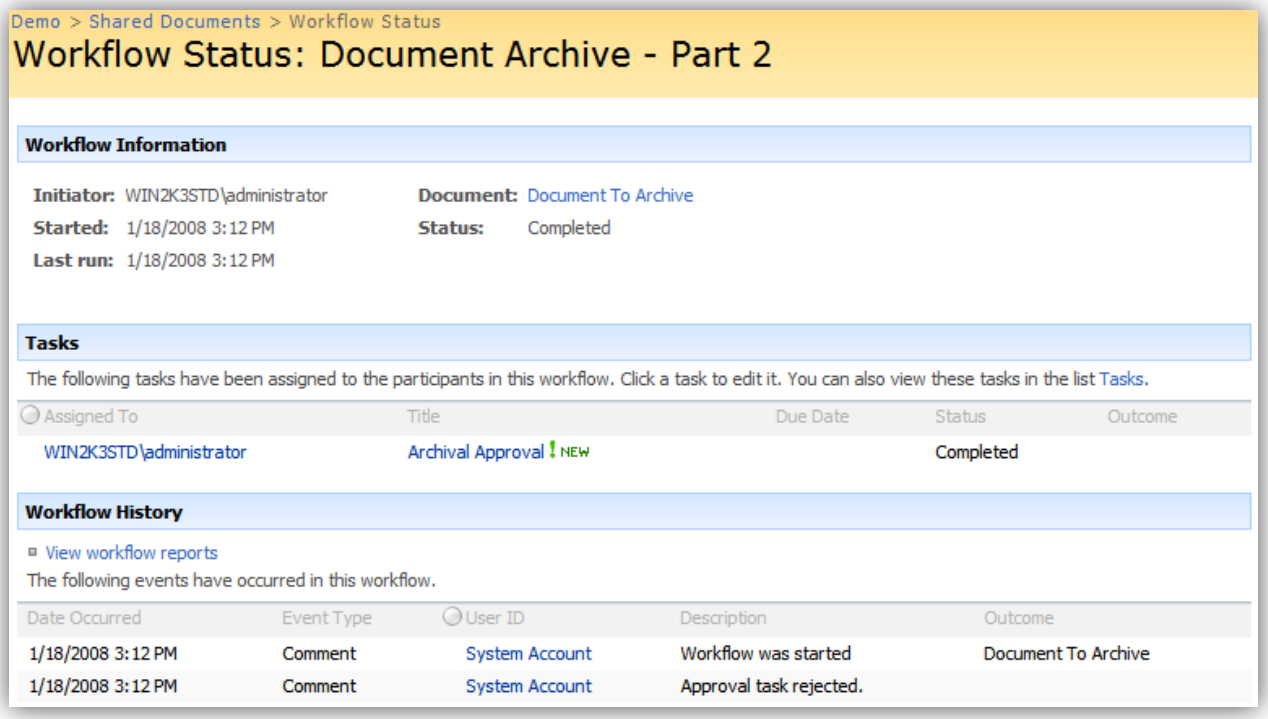
  *In the **Tasks** section, click the link to the task.*

  *In the custom task form, click the **Reject** button to reject the task.*



- Verify that the workflow is now completed and the task is marked as completed.

  You may have to refresh the status page to see the change.



Challenge: Integrating Task Forms into Solution Packages

1) Add the task form's .ASPX file to the **Manifest.xml** file.
   - Open the **Manifest.xml** file in the **Solution** folder.
   - Add a **TemplateFiles** and **TemplateFile** element immediately following the **Assemblies** element. This tells the Solution installer to install the task form when the solution is deployed.

```
<TemplateFiles>
  <TemplateFile Location="LAYOUTS\DocArchivePart2TaskForm.aspx"/>
</TemplateFiles>
```

2) Add the custom task form to the **Package.ddf** file.
   - Open the **Package.ddf** file in the **Solution** folder.
   - Add a line that packages the **DocArchivePart2TaskForm.aspx** in the **LAYOUTS** folder in the .cab file.

```
..\..\Template\Layouts\DocArchivePart2TaskForm.aspx
LAYOUTS\DocArchivePart2TaskForm.aspx
```

3) Manually uninstall the previously deployed **Document Archive** feature.
   - Open a command window in the SharePoint **bin** directory.

     *Click **Start -> Run** and enter **cmd**.*

     *Navigate to C**:\Program Files\Common Files\Microsoft Shared\web server extensions\12\BIN**.*
   - Execute **stsadm.exe** to uninstall the **DocuentArchiveWorkflowPart2** feature.

```
stsadm –o uninstallfeature –name DocumentArchiveWorkflowPart2 –force
```

   - Delete the folder at **C:\Program Files\Common Files\Microsoft Shared\web server extensions\12\Template\Features\DocumentarchiveWorkflowPart2**.
   - Delete the **DocArchivePart2TaskForm.aspx** file from the **C:\Program Files\Common Files\Microsoft Shared\web server extensions\12\Template\Layouts** folder.

4) Install and deploy the new solution package.
   - In the same command window as the previous step, execute **stsadm** to add the feature to SharePoint.

```
stsadm –o addsolution –filename
"C:\Labs\Lab06\StarterFiles\DocumentArchiveWorkflowPart2\bin\Debug\Package\Docum
entArchiveWorkflowPart2.wsp"
```

   - Deploy the solution to the farm using the **deploysolution** command in **stsadm**.

```
stsadm –o deploysolution –name DocumentArchiveWorkflowPart2.wsp –
allowgacdeployment –local
```

5) Verify the feature is still available and activated.
   - Using **Internet Explorer** navigate to the **Demo** site collection at **http://litwareinc.com/sites/Demo**.
   - Open the features list by clicking **Site Actions -> Site Settings**.
   - On the **Site Settings** page, click **Site collection features** in the **Site Collection Administration** section.
   - Verify the **Document Archive – Part 2** feature is active.

Lab 06: Creating custom Workflow Association Forms

Lab Overview: Now that the management of Litware Inc. has become comfortable with the built in functionality of SharePoint workflows, they have started to request extra features. They have requested a workflow system that archives documents in another document library. When the document library manager chooses to archive a document, they manually start a workflow, which creates an approval task.

In this lab, you'll be adding an Association form to the workflow. This will allow the user associating the workflow with the document library to choose a default approver and archive list. Once this is done, every workflow instance will run with these settings automatically.

Exercise 0: Setup

1) If you did not complete lab 4, you'll need to create the **Demo** site collection.

   Open SharePoint and browse to the Demo site collection

   *The url is **http://litwareinc.com/sites/Demo**.*

   *If the site collection does not exist, create it using the **CreateDemo.bat** file in the **C:\Labs\Files folder**.*

2) Open the starter **VS 2008** solution at **\Labs\Lab06\StarterFiles\DocumentArchive Part 3.sln**.

Exercise 1: Creating an Association Form base class

1) Create the **AssociationForm** base class in the UI folder.

   - Create a new file named **AssociationForm.cs** in the UI folder.

     *Right click the **UI** folder and click **Add -> Class…**.*

     *Name the new file **AssociationForm.cs** and click **Add**.*

   - Modify the class definition to the **AssociationForm** class to make it **public abstract** and derive from **LayoutsPageBase**.

```
public abstract class AssociationForm : LayoutsPageBase
{
}
```

2) Add the base methods to be overridden in derived classes to perform the population and extraction of values from the UI. These methods will need to be virtual or abstract so they can be overridden in derived classes.

   - Create a **protected virtual** method named **PopulateControls** to populate the controls not directly tied to the association data.
   - Create a **protected abstract** method named **BindAssociationData** that takes a single string parameter. This method populates the UI with the association data from a previously created association.
   - Create a **protected abstract** method named **UpdateAssociationData** that returns a string. This method extracts the association data from the form on the submit event.

```
protected virtual void PopulateControls() { }
protected abstract void BindAssociationData(string value);
protected abstract string UpdateAssociationData();
```

3) Since a workflow can be associated to a list, content type, or a list's content type, create an enumeration to track the type of association being managed.

- Create a private **AssociationType** enumeration inside the **AssociationForm** class with three values: **List**, **ContentType**, or **ListContentType**.

```
private enum AssociationType
{
    List,
    ContentType,
    ListContentType
}
```

- Add a private field to the class named **_associtionType** of type **AssociationType**.

```
private AssociationType _associationType;
```

4) When the page is loaded, determine the association type of the association to manage. Using the **Url** and **Form** parameters, determine if the workflow is related to a list, content type, or both and whether it's already been created.

- In the **AssociationForm** class, override the **OnLoad** method.

```
protected override void OnLoad(EventArgs e)
{
}
```

- Use the **HttpRequest** parameters to find the list id, content type id, and the existing association id.

  The existence of these values determines the type of association and its existence.

```
// read the form level parameters
string listId = Request.Params["List"];
string ctypeId = Request.Params["ctype"];
string guidAssocId = Request.Params["GuidAssoc"];
```

- Using the existence of the request parameters, determine the association type

```
// determine the type of association
if (!string.IsNullOrEmpty(listId) && !string.IsNullOrEmpty(ctypeId))
    _associationType = AssociationType.ListContentType;
else if (!string.IsNullOrEmpty(listId) && string.IsNullOrEmpty(ctypeId))
    _associationType = AssociationType.List;
else if (string.IsNullOrEmpty(listId) && !string.IsNullOrEmpty(ctypeId))
    _associationType = AssociationType.ContentType;
```

- If an existing workflow association id exists, convert it into a **Guid**.

```
// validate the workflow association id
Guid? workflowAssociationId = new Guid?();
if (!string.IsNullOrEmpty(guidAssocId))
    workflowAssociationId = new Guid(guidAssocId);
```

5) Based on the association type and the existence of the workflow association, lookup the **SPList**, **SPContentType**, and **SPWorkflowAssociation** objects.

- Add **private** fields to the class to store the content type, list and workflow association SharePoint APIs.

```
private SPContentType _contentType;
private SPList _list;
private SPWorkflowAssociation _workflowAssociation;
```

- At the end of the **OnLoad** method, add a switch statement to differentiate between a **ContentType**, **List**, or **ListContentType** association.

```
// lookup the list, content type, and existing workflow associations
switch (_associationType)
{
    case AssociationType.ContentType:
        break;
    case AssociationType.List:
        break;
    case AssociationType.ListContentType:
        break;
}
```

- If the association type is **ContentType**, find the **SPContentType** object using the **PageLayoutBase.Web** property. If a workflow association already exists, find it as well.

```
case AssociationType.ContentType:
    _contentType = Web.AvailableContentTypes[new  SPContentTypeId(ctypeId)];
    if (workflowAssociationId.HasValue)
        _workflowAssociation =
            _contentType.WorkflowAssociations[workflowAssociationId.Value];
```

- If the association type is **List**, find the **SPList** object using the **PageLayoutBase.Web** property. If a workflow association already exists, find it as well.

```
case AssociationType.List:
    _list = Web.Lists[new Guid(listId)];
    if (workflowAssociationId.HasValue)
        _workflowAssociation =
            _list.WorkflowAssociations[workflowAssociationId.Value];
```

- If the association type is **ListContentType**, find the **SPList** object and then use it to find the **SPContentType**. If a workflow association already exists, find it as well.

```
case AssociationType.ListContentType:
    _list = Web.Lists[new Guid(listId)];
    _contentType = _list.ContentTypes[new SPContentTypeId(ctypeId)];
    if (workflowAssociationId.HasValue)
        _workflowAssociation =
            _contentType.WorkflowAssociations[workflowAssociationId.Value];
```

- Call the base implementation of **OnLoad**.

```
// call the base implementation
base.OnLoad(e);
```

6) Since many of the page's parameters are in the posted form, write them back into the page so they will be available in subsequent postbacks.

- In the **AssociationForm** class, override the **OnPreRender** method.

```
protected override void OnPreRender(EventArgs e)
{ }
```

- Using the **Page.ClientScript.RegisterHiddenField** method, write the values of the form parameters into hidden fields in the page.

```
// register parameters in the hidden field
ClientScript.RegisterHiddenField("WorkflowName",
    Request.Params["WorkflowName"]);
ClientScript.RegisterHiddenField("WorkflowDefinition",
    Request.Params["WorkflowDefinition"]);
ClientScript.RegisterHiddenField("AddToStatusMenu",
    Request.Params["AddToStatusMenu"]);
ClientScript.RegisterHiddenField("AllowManual", Request.Params["AllowManual"]);
ClientScript.RegisterHiddenField("RoleSelect", Request.Params["RoleSelect"]);
ClientScript.RegisterHiddenField("GuidAssoc", Request.Params["GuidAssoc"]);
ClientScript.RegisterHiddenField("SetDefault", Request.Params["SetDefault"]);
ClientScript.RegisterHiddenField("HistoryList", Request.Params["HistoryList"]);
ClientScript.RegisterHiddenField("TaskList", Request.Params["TaskList"]);
ClientScript.RegisterHiddenField("UpdateLists", Request.Params["UpdateLists"]);
ClientScript.RegisterHiddenField("AutoStartCreate",
    Request.Params["AutoStartCreate"]);
ClientScript.RegisterHiddenField("AutoStartChange",
    Request.Params["AutoStartChange"]);
```

7) Since the **IsPostBack** property will always return true, population of the UI will be done in the **OnPreRender** event

- Call the virtual **PopulateControls** method.

```
// populate controls unrelated to the association data
this.PopulateControls();
```

- If an existing workflow association was found, call the **BindAssociationData** method and use the **AssociationData** property of the association as the parameter.

```
// bind the association data (if any)
if (_workflowAssociation != null)
    this.BindAssociationData(_workflowAssociation.AssociationData);
```

- Call the base implementation of **OnPreRender**.

```
// call the base implementation
base.OnPreRender(e);
```

8) Create a handler for the **Cancel** button that will redirect to the workflow settings page for the current list or content type.

- Create a **private** method named **BuildRedirectString**. It will return a string that is build based on the association type and the form parameters for the list and content type.

```
private static string BuildRedirectString(AssociationType associationType,
string paramList, string paramCtype)
{ }
```

- Use a **switch** statement to generate a different string for each association type.

  *If the type is **ContentType**, add a **ctype** url query string.*
  *If the type is **List**, add a list url query string.*
  *If the type is **ListContentType**, add both*

```
switch (associationType)
{
    case AssociationType.ContentType:
        return "WrkSetng.aspx?ctype=" + paramCtype;
    case AssociationType.List:
        return "WrkSetng.aspx?List=" + paramList;
    case AssociationType.ListContentType:
        return "WrkSetng.aspx?List=" + paramList + "&ctype=" + paramCtype;
    default:
        throw new InvalidOperationException();
}
```

- Create a **protected** event handler named **Cancel_Click**.

```
protected void Cancel_Click(object sender, EventArgs e)
{
}
```

- Use **SPUtility.Redirect** to redirect to the string generated by the **BuildRedirectString** method.

```
// redirect back to the workflow settings page
SPUtility.Redirect(
    BuildRedirectString(_associationType, Request.Params["List"],
        Request.Params["ctype"]),
        SPRedirectFlags.RelativeToLayoutsPage, this.Context);
```

9) Create a handler for the **Submit** button that will create or update the workflow association.

- Create a **protected** event handler named **Submit_Click**.

```
protected void Submit_Click(object sender, EventArgs e)
{
}
```

- Call the abstract **UpdateAssociationData** method to extract the association data from the form.

```
// get the association data
string associationData = UpdateAssociationData();
```

10) Verify the task and history lists exist. If they do not exist, create them.

- Create a new method named **LookupOrCreateList**. It will return a **SPList** object and accept the name of the list and a **SPListTemplateType** to determine the type of list to create if it does not already exist.

```
private SPList LookupOrCreateList(string paramList, SPListTemplateType listType)
{
}
```

- Check if the first character of the list is a lower case **z**. This indicates that the name following the **z** is the name of the list to create.

```
// lookup the history list
if (paramList.StartsWith("z"))
{
}
else
{ }
```

- If the list name started with **z**, create a new list and return the new **SPList** object.

```
Guid newListId =
    Web.Lists.Add(paramList.Substring(1), "Workflow Tasks", listType);
return Web.Lists[newListId];
```

- If the name did not start with **z**, find the existing list. Normally the list would be looked for by guid and by name then created if it was not found. In this example we're going to assume it's an existing.

```
return Web.Lists[new Guid(paramList)];
```

- In the **Submit_Click** method, add two lines of code to get the **SPList** objects for the **Tasks** and **History** lists. Use the **LookupOrCreateList** method to help.

```
// create or find the lists
SPList taskList = LookupOrCreateList(
    Request.Params["TaskList"], SPListTemplateType.Tasks);
SPList historyList = LookupOrCreateList(
    Request.Params["HistoryList"], SPListTemplateType.WorkflowHistory);
```

11) Check if the association already exists. If so call the update method, if not call create.

- Back in the **Submit_Click** method; check if derived content types should be updated by checking the **UpdateLists** form parameter.

```
// check whether updates to content types cascade to their lists
bool updateContentTypeLists = (Request.Params["UpdateLists"] == "TRUE");
```

- Check if the workflow association already exists. If it does, call the **UpdateWorkflowAssociation** method created in a following step. If it does not exist, call **CreateWorkflowAssociation** created in another following step.

```
// create or update the workflow association
if (_workflowAssociation != null)
    UpdateWorkflowAssociation(
        Request.Params["TaskList"], Request.Params["HistoryList"],
        associationData, updateContentTypeLists);
else
    CreateWorkflowAssociation(
        Request.Params["TaskList"], Request.Params["HistoryList"],
        associationData, updateContentTypeLists);
```

12) If the workflow association already exists, update it.

- Create a new method named **PopulateWorkflowAssociation** that accepts the task list, history list, and association data. This method will set the properties of the **_workflowAssociation** field based on the form parameters.

```
private void PopulateWorkflowAssociation(string taskListName,
  string historyListName, string associationData)
{ }
```

- Use the form parameters to initialize several of the startup and association data parameters in the **_workflowAssociation** field.

```
// assign the form data to the new workflow association
_workflowAssociation.Name = Request.Params["WorkflowName"];
_workflowAssociation.AutoStartCreate =
    (Request.Params["AutoStartCreate"] == "ON");
_workflowAssociation.AutoStartChange =
    (Request.Params["AutoStartChange"] == "ON");
_workflowAssociation.AllowManual = (Request.Params["AllowManual"] == "ON");
_workflowAssociation.AssociationData = associationData;
```

- Check if the task list and history lists have changed. If they have changed, update them in the **_workflowAssociation** field. In the case of a non content type, lookup and assign the lists directly. In the case of a content type, assign the list names.

```
// assign/update the tasks and history lists
if (this._associationType != AssociationType.ContentType) {
    SPList taskList = LookupOrCreateList(
        taskListName, SPListTemplateType.Tasks);
    SPList historyList = LookupOrCreateList(
        historyListName, SPListTemplateType.WorkflowHistory);

    if (_workflowAssociation.TaskListId != taskList.ID)
        _workflowAssociation.SetTaskList(taskList);

    if (_workflowAssociation.HistoryListId != historyList.ID)
        _workflowAssociation.SetHistoryList(historyList);
}
else {
    if (_workflowAssociation.TaskListTitle != taskListName)
        _workflowAssociation.TaskListTitle = taskListName;
    if (_workflowAssociation.HistoryListTitle != historyListName)
        _workflowAssociation.HistoryListTitle = historyListName;
}
```

- Create a new **UpdateWorkflowAssociation** method that accepts the task list, history list, association data and a flag indicating whether updates to a content type are applied to derived content types.

```
private void UpdateWorkflowAssociation(string taskListName,
    string historyListName, string associationData, bool updateContentTypeLists)
{ }
```

- Call the new **PopulateWorkflowAssociation** method to update the form parameters in the workflow association.

```
PopulateWorkflowAssociation(taskListName, historyListName, associationData);
```

- Create a **switch** statement that will use a different update method for each association type.

```
switch (_associationType)
{
    case AssociationType.ContentType:
        break;
    case AssociationType.List:
        break;
    case AssociationType.ListContentType:
        break;
}
```

- If the association type is **ContentType**, call the content type's **UpdateWorkflowAssociation** method. If the **updateContentTypeLists** flag is set, also call the content type's **UpdateWorkflowAssociationsOnChildren** method.

```
case AssociationType.ContentType:
    _contentType.UpdateWorkflowAssociation(_workflowAssociation);
    if (updateContentTypeLists)
        _contentType.UpdateWorkflowAssociationsOnChildren(true, true, true);
```

- If the association type is **List**, call the list's **UpdateWorkflowAssociation** method.

```
case AssociationType.List:
    _list.UpdateWorkflowAssociation(_workflowAssociation);
```

- If the association type is **ListContentType**, call the content type's **UpdateWorkflowAssociation** method.

```
case AssociationType.ListContentType:
    _contentType.UpdateWorkflowAssociation(_workflowAssociation);
```

13) If the workflow association does not exist, create a new one.

- Create a new **CreateWorkflowAssociation** method that accepts the task list, history list, association data and a flag indicating whether updates to a content type apply to derived content types.

```
private void CreateWorkflowAssociation(string taskListName,
  string historyListName, string associationData, bool updateContentTypeLists)
{
}
```

- Lookup the workflow name and template using the **WorkflowName** and **WorkflowDefinition** form parameters and use them to locate the **SPWorkflowTemplate** object.

```
// lookup the workflow template and the workflow name
string workflowName = Request.Params["WorkflowName"];
Guid workflowTemplateId = new Guid(Request.Params["WorkflowDefinition"]);
SPWorkflowTemplate workflowTemplate = Web.WorkflowTemplates[workflowTemplateId];
```

- Create a **switch** statement that will use a different create method for each association type.

```
switch (_associationType)
{
    case AssociationType.ContentType:
        break;
```

```
    case AssociationType.List:
        break;
    case AssociationType.ListContentType:
        break;
}
```

- If the association type is **ContentType**, create the new workflow association, populate it using **PopulateWorkflowAssociation**, add it to the content type, and finally update any child content types.

  *Call **SPWorkflowAssociation.CreateSiteContentTypeAssociation** to create the workflow association.*

  *Call the previously created method **PopulateWorkflowAssociation** to populate the new workflow association.*

  *Call **SPContentType.AddWorkflowAssociation** to attach the new workflow association to the content type.*

  *Optionally call **SPContentType.UpdateWorkflowAssociationsOnChildren** to update child content types.*

```
case AssociationType.ContentType:
    _workflowAssociation =
        SPWorkflowAssociation.CreateSiteContentTypeAssociation(
            workflowTemplate, workflowName, taskListName, historyListName);
    PopulateWorkflowAssociation(taskListName, historyListName, associationData);
    _contentType.AddWorkflowAssociation(_workflowAssociation);
    if (updateContentTypeLists)
        _contentType.UpdateWorkflowAssociationsOnChildren(true, true, true);
```

- If the association type is **List**, create the new workflow association, populate it using **PopulateWorkflowAssociation**, and add it to the list.

  *Call **SPWorkflowAssociation.CreateListAssociation** to create the workflow association.*

  *Call the previously created method **PopulateWorkflowAssociation** to populate the new workflow association.*

  *Call **SPList.AddWorkflowAssociation** to attach the new workflow association to the content type.*

```
case AssociationType.List:
    _workflowAssociation =
        SPWorkflowAssociation.CreateListAssociation(
            workflowTemplate, workflowName,
            LookupOrCreateList(taskListName, SPListTemplateType.Tasks),
            LookupOrCreateList(historyListName,
                SPListTemplateType.WorkflowHistory));
    PopulateWorkflowAssociation(taskListName, historyListName, associationData);
    _list.AddWorkflowAssociation(_workflowAssociation);
```

- If the association type is **ListContentType**, create the new workflow association, populate it using **PopulateWorkflowAssociation**, and add it to the list.

  *Call **SPWorkflowAssociation.CreateListContentTypeAssociation** to create the workflow association.*

  *Call the previously created method **PopulateWorkflowAssociation** to populate the new workflow association.*

  *Call **SPContentType.AddWorkflowAssociation** to attach the new workflow association to the content type.*

```
case AssociationType.ListContentType:
    _workflowAssociation =
        SPWorkflowAssociation.CreateListContentTypeAssociation(
            workflowTemplate, workflowName,
            LookupOrCreateList(taskListName, SPListTemplateType.Tasks),
            LookupOrCreateList(historyListName,
                SPListTemplateType.WorkflowHistory));
    PopulateWorkflowAssociation(taskListName, historyListName, associationData);
    _contentType.AddWorkflowAssociation(_workflowAssociation);
```

14) Finish the **Submit_Click** button event handler by redirecting to the settings page for the current list or content type.

- Use the **SPUtility.Redirect** and the **BuildRedirectString** methods to redirect to the workflow settings page.

```
// redirect back to the workflow settings page
SPUtility.Redirect(
    BuildRedirectString(
        _associationType, Request.Params["List"], Request.Params["ctype"]),
        SPRedirectFlags.RelativeToLayoutsPage, this.Context);
```

15) Add two properties that will expose information about the current workflow association to derived association forms.

- Create a property named **IsNewAssocition** of type **bool**. Return whether the current association is new.

```
protected bool IsNewAssociation
{
    get { return string.IsNullOrEmpty(Request.Params["GuidAssoc"]); }
}
```

- Create a property named **WorkflowAssociationName** that returns the name of the current association.

```
protected string WorkflowAssociationName
{
    get { return Request.Params["WorkflowName"]; }
}
```

Exercise 2: Creating a custom Association Form

1) Add a new base class named **DocArchivePart3AssocForm** that will be uses as the code behind for an ASPX page.

- Create a new class named **DocArchivePart3AssocForm** in the UI folder.

  *Right click the **UI** folder and click **Add -> Class…**.*

  *Name the new file **DocArchivePart3AssocForm.cs** and click **Add**.*

- Change the **DocArchivePart3AssocForm** class into a public class that derives from the **AssociationForm** base class.

```
public class DocArchivePart3AssocForm : AssociationForm
{ }
```

- Add two protected fields to the class to access the controls in the ASPX form.

```
protected PeopleEditor ppkApprover;
protected DropDownList lstArchiveList;
```

2) Populate the form by overriding the **PopulateControls** and **BindAssociationData** methods.

- Override the **PopulateControls** method.

```
protected override void PopulateControls()
{ }
```

- Filter the list of webs and populate the **lstArchiveLists** with all of the document libraries and are not hidden.

```
// filter the list of lists for all document libraries that aren't hidden
```

```
IEnumerable<SPList> documentLibraries =
    Web.Lists.Cast<SPList>().Where(n => n is SPDocumentLibrary && !n.Hidden);

// add each visible document library to the list
foreach (SPList documentLibrary in documentLibraries)
    lstArchiveList.Items.Add(
        new ListItem(documentLibrary.Title, documentLibrary.ID.ToString()));
```

- Override the **BindAssociationData** method.

```
protected override void BindAssociationData(string value)
{ }
```

- Deserialize the association data using the **AssocInitData.Deserialize** method.

  *Uses the **XmlSerializer** to convert xml to a typed class.*

```
// deserialize the association data
AssocInitData assocInitData = AssocInitData.Deserialize(value);
```

- Populate the controls using the new **AssocInitData** class.

```
// bind the controls to the association data
ppkApprover.CommaSeparatedAccounts = assocInitData.ApproverUserName;
lstArchiveList.SelectedValue = assocInitData.ArchiveListId.ToString();
```

3) Extract the data from the page when the form is submitted.

- Override the **UpdateAssociationData** class.

```
protected override string UpdateAssociationData()
{ }
```

- Extract the data from the controls into a new instance of the **AssocInitData** class.

```
// populate the association data using the UI
AssocInitData assocInitData = new AssocInitData();
assocInitData.ApproverUserName = (ppkApprover.Entities[0] as PickerEntity).Key;
assocInitData.ArchiveListId = lstArchiveList.SelectedValue;
```

- Return the xml representing the **AssocInitData** using the **AssocInitData.Serialize** method.

  *Uses the **XmlSerializer** to convert xml to a typed class.*

```
// serialize the return the association data
return assocInitData.Serialize();
```

4) Create a new .ASPX page named **DocArchivePart3AssocForm.aspx** that represents the ASP.NET markup for the task page.

- Create a new text file named **DocArchivePart3AssocForm.aspx** in the new **Layouts** folder.

  *Right click the **Layouts** folder and click **Add -> New Item…**.*
  *Select the **General** category on the left and select a template of **Text file**.*
  *Name the new file **DocArchivePart3AssocForm.aspx** and click **Add**.*

- Enter the page directives that define the code behind class and the master page for the task form.

```
<%@ Assembly Name="DocumentArchiveWorkflowPart3, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=15812f954569663f" %>

<%@ Page Language="C#" MasterPageFile="~/_layouts/application.master"
    EnableSessionState="true"   ValidateRequest="False"
    Inherits="DocumentArchiveWorkflowPart3.UI.DocArchivePart3AssocForm" %>
```

- Register the tag prefix and locations of the SharePoint WebControls and UserControls that will be used to format the page.

```
<%@ Register TagPrefix="SharePoint" Namespace="Microsoft.SharePoint.WebControls"
    Assembly="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce111e9429c" %>
<%@ Register TagPrefix="Utilities" Namespace="Microsoft.SharePoint.Utilities"
Assembly="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce111e9429c" %>
<%@ Register TagPrefix="wssuc" TagName="InputFormSection"
Src="/_controltemplates/InputFormSection.ascx" %>
<%@ Register TagPrefix="wssuc" TagName="InputFormControl"
Src="/_controltemplates/InputFormControl.ascx" %>
<%@ Register TagPrefix="wssuc" TagName="ButtonSection"
Src="/_controltemplates/ButtonSection.ascx" %>
```

- Define the content for the page title and the form specific headers.

```
<asp:Content ID="PageTitle" ContentPlaceHolderID="PlaceHolderPageTitle"
runat="server">
    <%= this.IsNewAssociation?"Customize Workflow":"Change a Workflow"
%></asp:Content>
<asp:Content ID="PageTitleInTitleArea"
ContentPlaceHolderID="PlaceHolderPageTitleInTitleArea"
    runat="server">
    <%= this.IsNewAssociation?"Customize Workflow: " +
this.WorkflowAssociationName:"Change a Workflow: " +
this.WorkflowAssociationName %></asp:Content>
<asp:Content ID="PageDescription"
ContentPlaceHolderID="PlaceHolderPageDescription"
    runat="server">
    <%= this.IsNewAssociation?"Use this page to define the settings of a new
workflow.":"Use this page to change the settings of an existing workflow."
%></asp:Content>
```

5) Define the ASPX markup to create the data entry section of the custom task form.

- Define the **ContentPlaceHolder** control for **PlaceHolderMain** and place a table within it.

```
<asp:Content ID="Main" ContentPlaceHolderID="PlaceHolderMain" runat="server">
    <table cellspacing="0" cellpadding="0" style="border: none; width: 100%"
class="ms-propertysheet">
    </table>
</asp:Content>
```

- Implement the first **InputFormSection** that allow entry of the approver.

  *The special SharePoint web control called **PeopleEditor** retrieves this information.*

```
<%-- Deafault Approver Input --%>
<wssuc:InputFormSection Title="Default Approver" Description="Specify the user
who will be the default approver."
    runat="server">
    <template_inputformcontrols>
        <wssuc:InputFormControl runat="server" LabelText="Approver:">
            <Template_Control>
                <SharePoint:PeopleEditor id="ppkApprover"
                    AllowEmpty="false"
                    ValidatorEnabled="true"
                    MultiSelect="false"
                    SelectionSet="User"
                    width='300px'
                    runat="server" />
            </Template_Control>
        </wssuc:InputFormControl>
    </template_inputformcontrols>
</wssuc:InputFormSection>
```

- Implement the second **InputFormSection** that will allow the approver to choose the default archive list.

```
<%-- Archive List Input --%>
<wssuc:InputFormSection Title="Archive Document Library"
    Description="Specify the document library used to store archived documents."
runat="server">
    <template_inputformcontrols>
        <wssuc:InputFormControl LabelText="Archive Document Library:"
runat="server">
                <Template_Control>
                    <asp:DropDownList ID="lstArchiveList" runat="server" />
                </Template_Control>
        </wssuc:InputFormControl>
    </template_inputformcontrols>
</wssuc:InputFormSection>
```

- Implement the last section containing the **Submit** and **Cancel** buttons.

```
<%-- Submit and Cancel Buttons --%>
<wssuc:ButtonSection runat="server" ShowStandardCancelButton="false">
    <template_buttons>
        <asp:PlaceHolder runat="server">
            <asp:Button UseSubmitBehavior="false" runat="server" class="ms-
ButtonHeightWidth" OnClick="Submit_Click" Text="OK" id="BtnSubmit" />  
              <asp:Button UseSubmitBehavior="false" runat="server" class="ms-
ButtonHeightWidth" OnClick="Cancel_Click" Text="Cancel" id="cmdCancel"
causesvalidation=false />
        </asp:PlaceHolder>
    </template_buttons>
</wssuc:ButtonSection>
```

Exercise 3: Integrating custom Association Forms into the workflow

1) Handle the **Invoked** event of the **OnWorkflowActivated** and deserialize the association data.

- Open the Workfl**o**w code by right clicking on the **Workflow** class in the **Solution Explorer** and clicking **View Code**.
- Add a private field named **AssocInitData** of type **AssocInitData** to the **Workflow** class.

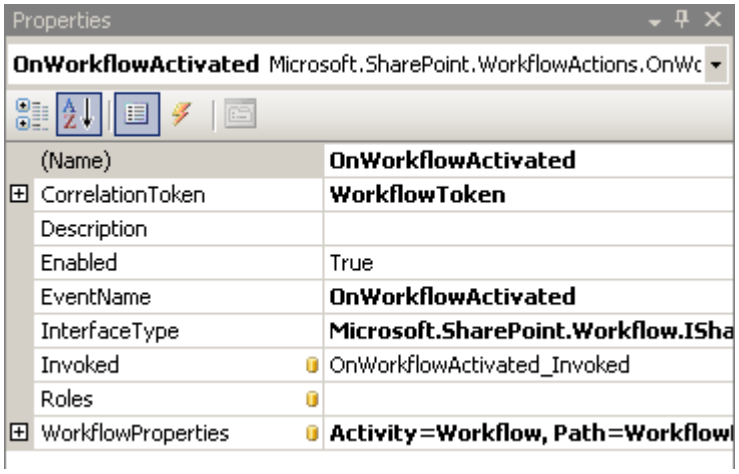
```
private AssocInitData AssocInitData { get; set; }
```

- Open the workflow in the designer by right clicking on the Workflow class in the **Solution Explorer** and clicking **View Designer**.
- Add an Invoked handler named **OnWorkflowActivated_Invoked** to the **OnWorkflowActivated** activity.

*Double click **WorkflowActivatedActivity** to open the sequential designer.*

*Right click the **OnWorkflowActivated** activity and select **Properties**.*

*In the properties pane, set Invoked to **OnWorkflowActivated_Invoked** and press **Enter**.*



- In the **OnWorkflowActivated_Invoked** method, deserialize the association data.

*Deserialize the **AssocInitData** using its **Deserialize** method.*

```
// deserialize init data, if none use defaults
this.AssocInitData = AssocInitData.Deserialize(
    WorkflowProperties.AssociationData);
```

2) Update the task creation to use the association data when initializing the task.

- Locate the **CreateApprovalTask_Invoking** method.
- Change the line that sets the **taskProperties.AssignedTo** property and get its value from **AssocInitData.ApproverUserName**.

```
//taskProperties.AssignedTo = "LITWAREINC\\Administrator";
taskProperties.AssignedTo = AssocInitData.ApproverUserName;
```

- Add an extended property to the **taskProperties** object. Use the **AssocInitData.ArchiveListId** to initialize the list selected in the task form.

```
taskProperties.ExtendedProperties.Add("ArchiveListId",
this.AssocInitData.ArchiveListId);
```

3) Update the task form to load the default archive list in the **OnLoad** method.

- Open the **DocArchivePart3TaskForm.cs** file in the UI folder.
- Locate the **OnLoad** Method.
- In the section where controls are initialized, set the **SelectedValue** of the **lstArchiveList** control using the extended properties of the task.

*Access the task's extended properties using the **SPWorkflowTask.GetExtendedPropertiesAsHashtable** method.*

```
// initialize the archive list selection
Hashtable extendedProperties =
    SPWorkflowTask.GetExtendedPropertiesAsHashtable(_taskItem);
lstArchiveList.SelectedValue = extendedProperties["ArchiveListId"] as string;
```

Exercise 4: Deploying custom Association Forms

1) Register the new custom Association form by assigning the **AssociationUrl** attribute in the feature's **Workflow** element.

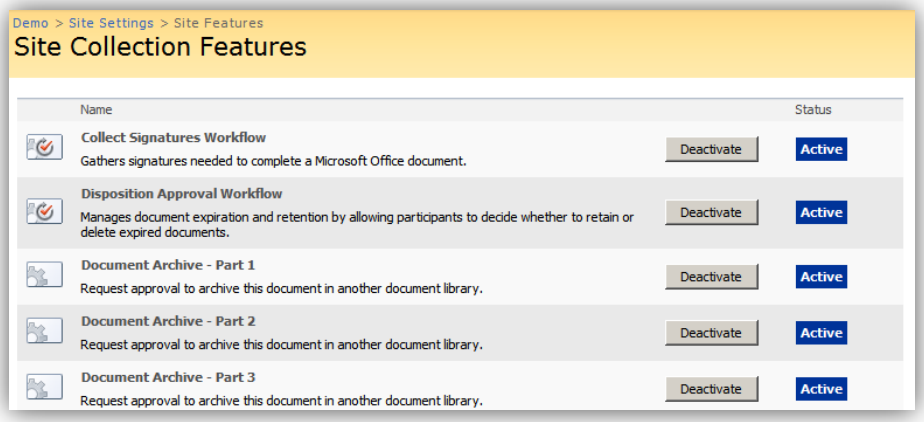
```
<Workflow ...
     AssociationUrl ="_layouts/DocArchivePart3AssocForm.aspx"
     StatusUrl="_layouts/WrkStat.aspx">
  <Categories/>
  <MetaData>
  </MetaData>
</Workflow>
```

2) Rebuild the workflow.

- Right click the project in the **Solution Explorer** and click **Rebuild**.
- In the **Output** window, verify the post build actions completed successfully.

3) Activate the new **DocumentArchiveWorkflowPart3** feature.

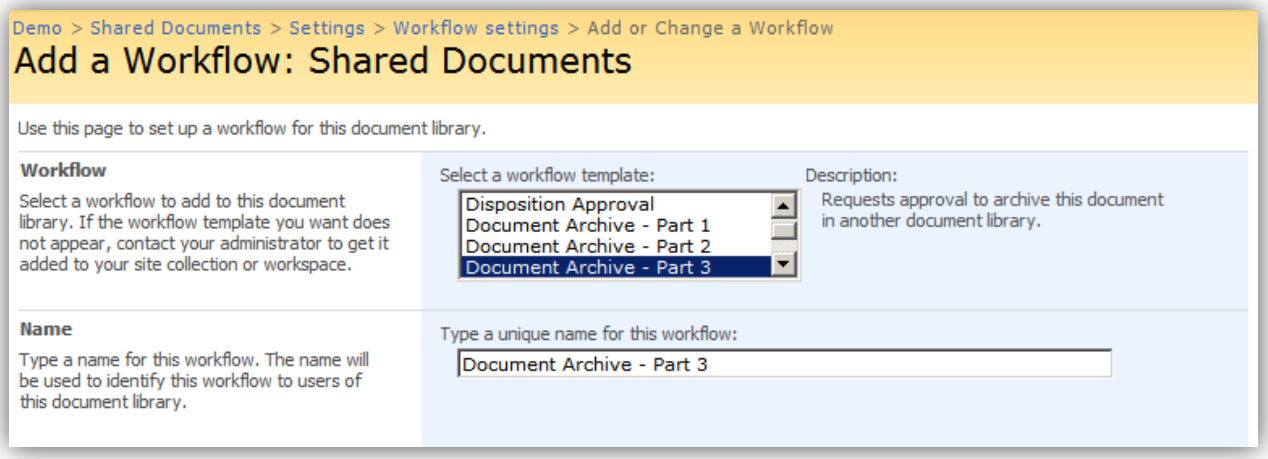
- Using **Internet Explorer** navigate to the **Demo** site collection at **http://litwareinc.com/sites/Demo**.
- Open the features list by clicking **Site Actions -> Site Settings**.
- On the **Site Settings** page, click **Site collection features** in the **Site Collection Administration** section.
- Click the **Activate** button next to the **Document Archive – Part 3** feature.



4) Create an association between the **Shared Documents** document library and the new **Document Archive – Part 3** workflow.

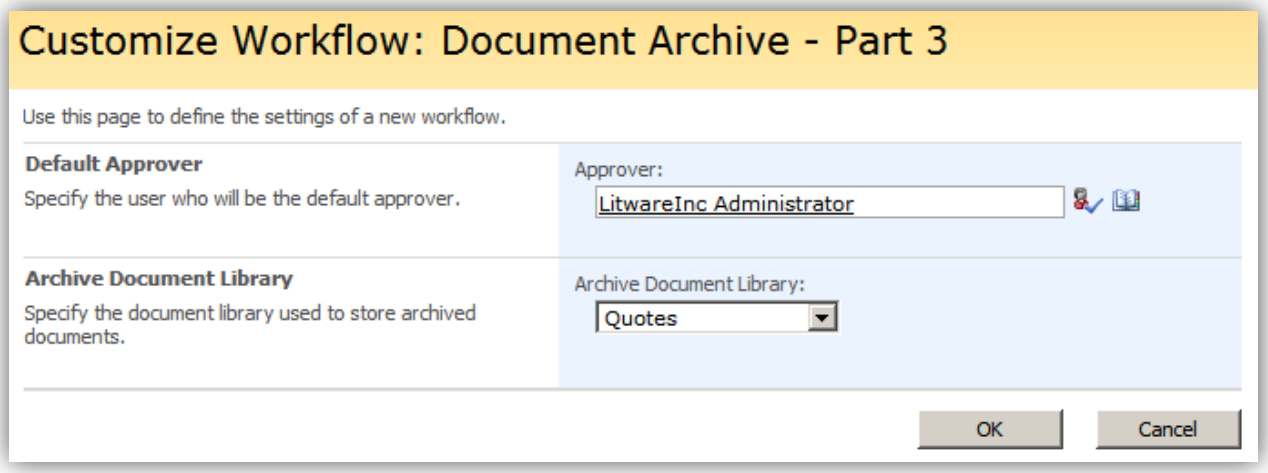
- Navigate to the **Shared Documents** document library in the **Demos** site.
- Click **Settings -> Document Library Settings** to load the settings page.
- Click the **Workflow settings** link in the **Permissions and Management section**.
- Create a new workflow using the **Document Archive – Part 3** workflow template and a name of **Document Archive - Part 3**.

  Use the default values for both list and startup options.



- In the custom association form, enter the Administrator as both the default approver and the manager. Select the **Quotes** document library as the archive location.

  *If you'd like, create an **Archive** document library to use instead.*



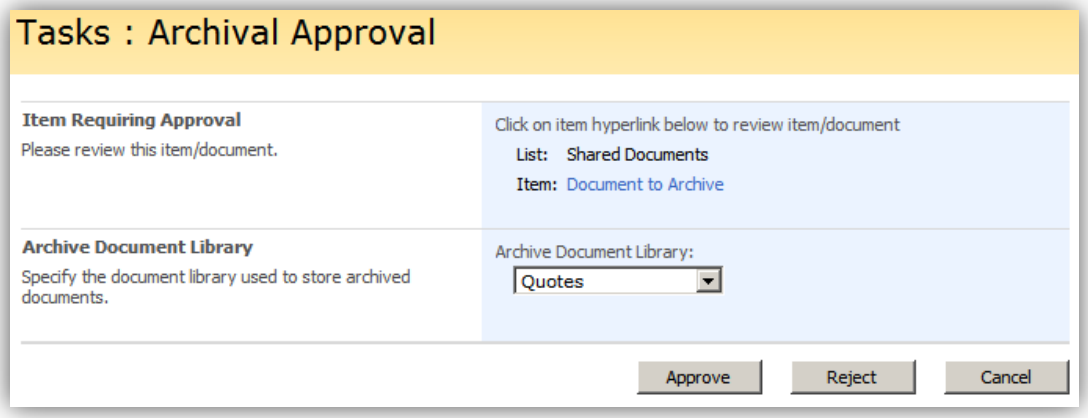
5) Run the **Document Archive – Part 3** workflow on a document.

- Navigate to the Shared Documents document library, hover over the new document, and select **Workflows** from the drop down menu.
- In the workflows page, click the **Document Archive – Part 3** to start the workflow.
- In the **Shared Documents** document library, verify the workflow is running.

- Click the **In Progress** link to view the workflow status and verify the started message was logged to the workflow's history.
- Edit the task and set its **Status** to **Completed**.

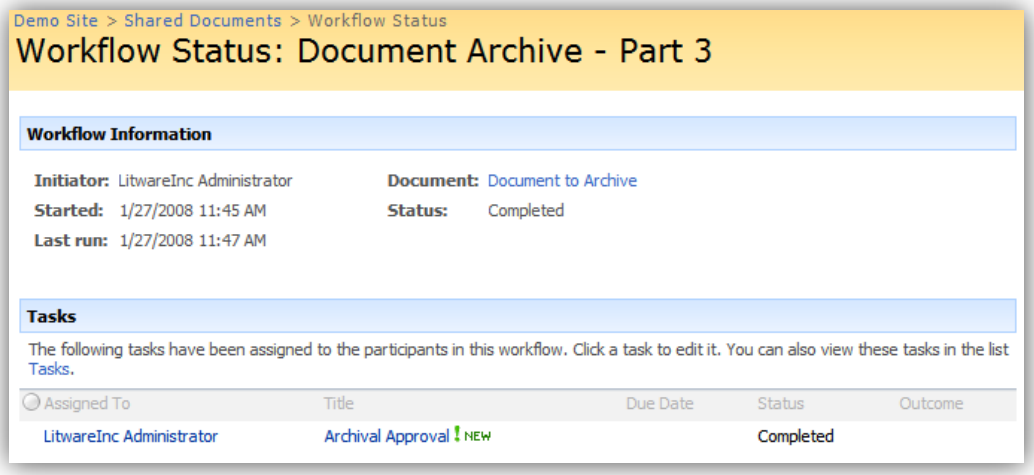
  *Notice that the **Archive Document List** defaults to the list set in the **Association Form**.*

  *Click the **Approve** button to approve the task.*



- Verify that the workflow is now completed and the task is marked as completed.

  You may have to refresh the status page to see the change.



Challenge: Integrating Association Forms into Solution Packages

1) Add the task form's .ASPX file to the **Manifest.xml** file.

- Open the **Manifest.xml** file in the **Solution** folder.
- Add a **TemplateFile** element telling the solution installer to install the association form when the solution is deployed.

```
<TemplateFiles>
  <TemplateFile Location="LAYOUTS\DocArchivePart3TaskForm.aspx"/>
  <TemplateFile Location="LAYOUTS\DocArchivePart3AssocForm.aspx"/>
</TemplateFiles>
```

2) Add the custom task form to the **Package.ddf** file.

- Open the **Package.ddf** file in the **Solution** folder.
- Add a line that packages the **DocArchivePart3AssocForm.aspx** and puts it in the **LAYOUTS** folder in the .cab file.

```
..\..\Template\Layouts\DocArchivePart3AssocForm.aspx
LAYOUTS\DocArchivePart3AssocForm.aspx
```

- Build the project by right clicking the project in the solution explorer and clicking **Rebuild**.

3) Manually uninstall the previously deployed **Document Archive** feature.

- Open a command window in the SharePoint **bin** directory.

  *Click **Start -> Run** and enter **cmd**.*

  *Navigate to C**:\Program Files\Common Files\Microsoft Shared\web server extensions\12\BIN**.*

- Execute **stsadm.exe** to uninstall the **DocuentArchiveWorkflowPart3** feature.

```
stsadm –o uninstallfeature –name DocumentArchiveWorkflowPart3 –force
```

- Delete the folder at **C:\Program Files\Common Files\Microsoft Shared\web server extensions\12\Template\Features\DocumentarchiveWorkflowPart3**.
- Delete the **DocArchivePart3AssocForm.aspx** and **DocArchivePart3TaskForm.aspx** files from the **C:\Program Files\Common Files\Microsoft Shared\web server extensions\12\Template\Layouts** folder.

4) Install and deploy the new solution package.

- In the same command window as the previous step, execute **stsadm** to add the feature to SharePoint.

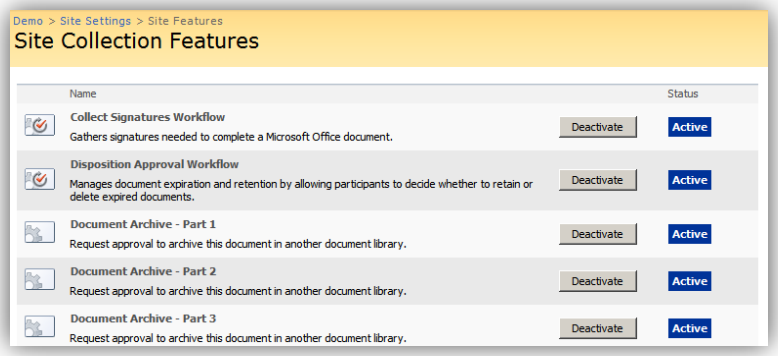
```
stsadm –o addsolution –filename
“C:\Labs\Lab06\StarterFiles\DocumentArchiveWorkflowPart3\bin\Debug\Package\Docum
entArchiveWorkflowPart3.wsp”
```

- Deploy the solution to the farm using the **deploysolution** command in **stsadm**.

```
stsadm –o deploysolution –name DocumentArchiveWorkflowPart3.wsp –
allowgacdeployment –local
```

5) Verify the feature is still available and activated.

- Using **Internet Explorer** navigate to the **Demo** site collection at **http://litwareinc.com/sites/Demo**.
- Open the features list by clicking **Site Actions -> Site Settings**.
- On the **Site Settings** page, click **Site collection features** in the **Site Collection Administration** section.
- Verify the **Document Archive – Part 3** feature is active.



Lab 07: Creating custom Instantiation and Modification Forms

Lab Overview: Now that the management of Litware Inc. has become comfortable with the built in functionality of SharePoint workflows, they have started to request extra features. They have requested a workflow system that archives documents in another document library. When the document library manager chooses to archive a document, they manually start a workflow, which creates an approval task.

In this lab, you will be adding an Instantiation form and a Modification form to the workflow. This will allow the user to override the association settings when starting the workflow and even change them while the workflow is running. With these two forms added, the archive workflow will be complete.

Exercise 0: Setup

1) If you did not complete lab 4, you'll need to create the **Demo** site collection.
   - Open SharePoint and browse to the **Demo** site collection

     *The url is **http://litwareinc.com/sites/Demo**.*
     *If the site collection does not exist, create it using the **CreateDemo.bat** file in the **C:\Labs\Files folder**.*
2) Open the starter **VS 2008** solution at **\Labs\Lab07\StarterFiles\DocumentArchive Part 4.sln**.

Exercise 1: Creating a Custom Instantiation Form

1) Add a new base class named **DocArchivePart4InitForm** that will be uses as the code behind for an ASPX page.
   - Create a new class named **DocArchivePart4InitForm** in the **UI** folder.

     *Right click the **UI** folder and click **Add -> Class…**.*
     *Name the new file **DocArchivePart4InitForm.cs** and click **Add**.*
   - Change the **DocArchivePart4InitForm** class into a public class that derives from **Microsoft.SharePoint.WebControls.LayoutsPageBase**.

```
public abstract class DocArchivePart4InitForm : LayoutsPageBase
{
}
```

   - Add private fields to the class to store information about the workflow association to be started as well as the list item the workflow instance will be attached to.

```
private SPWorkflowAssociation _workflowAssociation;
private SPList _list;
private SPListItem _listItem;
```

2) Add a **protected** field and property that will allow interaction between the ASPX page and the code behind.

- Add a **protected** field named **ppkApprover** of type **PeopleEditor** to the class.
- Add a **protected** field named **lstArchiveList** of type **DropDownList** to the class.
- Add a **protected** property named **WorkflowName** of type string to the class. Have it return the **Name** property of the **_workflowAssociation** field.
- Add a **protected** property named **ItemName** of type string to the class. Have it return the **DisplayName** property of the **_listItem** field.

```
protected PeopleEditor ppkApprover;
protected DropDownList lstArchiveList;

protected string WorkflowName
{
    get { return _workflowAssociation.Name; }
}

protected string ItemName
{
    get { return _listItem.DisplayName; }
}
```

3) Override the **OnLoad** method in the new **DocArchivePart4InitForm** class so it reads the **Url** parameters and creates the necessary SharePoint API objects and binds that data to the UI.

- Override the **OnLoad** method of the base class.

```
protected override void OnLoad(EventArgs e)
{ }
```

- Retrieve the list id, list item id, content type id, and the workflow template id from the URL parameters list.

```
// read the form level parameters
string listId = Request.Params["List"];
string listItemId = Request.Params["ID"];
string ctypeId = Request.Params["ctype"];
string templateId = Request.Params["TemplateId"];
```

- Read the **List** and **ID** url parameters and use them to find the list and list item to use when attaching the new workflow instance.

```
// find the list and list item
_list = Web.Lists[new Guid(listId)];
_listItem = _list.GetItemById(int.Parse(listItemId));
```

- Check if the workflow association can be found in the list's collection of workflow associations accessed using the **WorkflowAssociations** property.

```
// check for the workflow association in the list
_workflowAssociation = _list.WorkflowAssociations[new Guid(templateId)];
```

- If the workflow association was not found in the list's **WokflowAssociations** collection, find the content type the workflow association is connected to and check its **WorkflowAssociations** collection.

  *Access the content types using the **ctype** URL parameter and the **ContentTypes** property of **SPList**.*

  *Once the **SPContentType** object is found, use its **WorkflowAssociations** collection to lookup the workflow association.*

```
// if the association wasn't found, check the content type
if (_workflowAssociation == null)
{
    SPContentType _contentType =
        _list.ContentTypes[new SPContentTypeId(ctypeId)];
    _workflowAssociation =
        _contentType.WorkflowAssociations[new Guid(templateId)];
}
```

- In the event that a workflow association could not be found in either of the locations, throw a new **SPException** notifying the user of the problem.

```
// make sure the workflow association could be found
if (_workflowAssociation == null)
    throw new SPException("The requested workflow could not be found.");
```

4) If this request is not a postback, populate the **lstArchiveList** controls with all non-hidden document libraries in the current web.

- Check if the request is a postback using the **Page.IsPostBack** property. Inside this **if** statement is where all of the UI initialization will be done.

```
// populate the UI
if (!this.IsPostBack)
{
}
```

- Using the current site's **Lists** collection and some filtering, populate the **lstArchiveList** drop down list with every non-hidden document library on the web.

```
// filter the list of lists for all document libraries that aren't hidden
IEnumerable<SPList> documentLibraries =
    Web.Lists.Cast<SPList>().Where(n => n is SPDocumentLibrary && !n.Hidden);

// add each visible document library to the list
foreach (SPList documentLibrary in documentLibraries)
    lstArchiveList.Items.Add(new ListItem(documentLibrary.Title,
documentLibrary.ID.ToString()));
```

- Deserialize the association data related to workflow association using the **AssocInitData.Deserialize** method. Now attach the values in the association data to the controls to provide defaults for the form controls.

```
// populate the form using the association data
AssocInitData assocInitData =
AssocInitData.Deserialize(_workflowAssociation.AssociationData);
ppkApprover.CommaSeparatedAccounts = assocInitData.ApproverUserName;
lstArchiveList.SelectedValue = assocInitData.ArchiveListId.ToString();
```

- Finish up the **OnLoad** method by calling the base implementation of the **OnLoad** method

```
// call the base implementation
base.OnLoad(e);
```

5) Implement the event handler called after a click of the **Cancel** button.
   - Create a **protected** event handler named **Cancel_Click**.
   - Use the **SPUtility.Redirect** method to handle the automatic redirection to either the source url parameter or the current list's default view url.

```
protected void Cancel_Click(object sender, EventArgs e)
{
    // redirect to the page in the source url parameter or the default page
    SPUtility.Redirect(_list.DefaultViewUrl,
        SPRedirectFlags.UseSource, this.Context);
}
```

6) Implement the event handler called after a click of the **Start** button.
   - Create a **protected** event handler named **Start_Click**.

```
protected void Start_Click(object sender, EventArgs e)
{
}
```

   - Create a new **AssocInitData** object and populate it using the values currently stored in the form controls and serialize it using the **Serialize** method.

```
// populate the association data using the UI
AssocInitData assocInitData = new AssocInitData();
assocInitData.ApproverUserName = (ppkApprover.Entities[0] as PickerEntity).Key;
assocInitData.ArchiveListId = lstArchiveList.SelectedValue;

// get the initiation data
string initiationData = assocInitData.Serialize();
```

   - Start a new workflow instance using the **SPWorkflowManager.StartMethod**.

     *The **SPWorkflowManager** is accessible through the current's **SPWeb** object's **SPSite** parent object.*

```
// start the new workflow instance
Web.Site.WorkflowManager.StartWorkflow(_listItem,
    _workflowAssociation, initiationData);
```

   - Use the **SPUtility.Redirect** method to handle the automatic redirection to either the source url parameter or the current list's default view url.

```
// redirect to the list default view
SPUtility.Redirect(_list.DefaultViewUrl,
    SPRedirectFlags.UseSource, this.Context);
```

7) Create a new .ASPX page named **DocArchivePart4InitForm.aspx** that represents the ASP.NET markup for the task page.

- Create a new text file named **DocArchivePart4InitForm.aspx** in the **Layouts** folder.

  *Right click the **Layouts** folder and click **Add -> New Item…**.*

  *Select the **Generate** category on the left and select a template of **Text file**.*

  *Name the new file **DocArchivePart4InitForm.aspx** and click **Add**.*

- Enter the page directives that define the code behind class and the master page for the task form.

```
<%@ Assembly Name="DocumentArchiveWorkflowPart4, Version=1.0.0.0,
    Culture=neutral, PublicKeyToken=15812f954569663f" %>

<%@ Page Language="C#" MasterPageFile="~/_layouts/application.master"
    EnableSessionState="true" ValidateRequest="False"
    Inherits="DocumentArchiveWorkflowPart4.UI.DocArchivePart4InitForm" %>
```

- Register the tag prefix and locations of the SharePoint WebControls and UserControls that will be used to format the page.

```
<%@ Register TagPrefix="SharePoint" Namespace="Microsoft.SharePoint.WebControls"
    Assembly="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce111e9429c" %>
<%@ Register TagPrefix="Utilities" Namespace="Microsoft.SharePoint.Utilities"
Assembly="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce111e9429c" %>
<%@ Register TagPrefix="wssuc" TagName="InputFormSection"
Src="/_controltemplates/InputFormSection.ascx" %>
<%@ Register TagPrefix="wssuc" TagName="InputFormControl"
Src="/_controltemplates/InputFormControl.ascx" %>
<%@ Register TagPrefix="wssuc" TagName="ButtonSection"
Src="/_controltemplates/ButtonSection.ascx" %>
```

- Define the content for the page title and the form specific headers.

```
<asp:Content ID="PageTitle" ContentPlaceHolderID="PlaceHolderPageTitle"
runat="server">
    Start Workflow</asp:Content>
<asp:Content ID="PageTitleInTitleArea"
ContentPlaceHolderID="PlaceHolderPageTitleInTitleArea"
    runat="server">
    <%= "Start \"" + this.WorkflowName + "\": " + this.ItemName %></asp:Content>
```

8) Define the ASPX markup that will be used to create the data entry section of the custom task form.

- Define the **ContentPlaceHolder** control for **PlaceHolderMain** and place a table within it.

```
<asp:Content ID="Main" ContentPlaceHolderID="PlaceHolderMain" runat="server">
    <table cellspacing="0" cellpadding="0" style="border: none; width: 100%"
class="ms-propertysheet">
    </table>
</asp:Content>
```

- Implement the first **InputFormSection** that allow entry of the primary approver.

  *The special SharePoint web control called **PeopleEditor** retrieves this information.*

```
<%-- Deafault Approver Input --%>
<wssuc:InputFormSection Title="Default Approver" Description="Specify the user
who will be the default approver."  runat="server">
    <template_inputformcontrols>
            <wssuc:InputFormControl runat="server" LabelText="Approver:">
            <Template_Control>
                <SharePoint:PeopleEditor  id="ppkApprover"
                  AllowEmpty="false"
                        ValidatorEnabled="true"
                        MultiSelect="false"
                        runat="server"
                        SelectionSet="User"
                  width='300px' />
            </Template_Control>
        </wssuc:InputFormControl>
    </template_inputformcontrols>
</wssuc:InputFormSection>
```

- Implement the second **InputFormSection** that will allow the approver to choose the default list where the document will be archived.

```
<%-- Archive List Input --%>
<wssuc:InputFormSection Title="Archive Document Library"
    Description="Specify the document library used to store archived documents."
runat="server">
    <template_inputformcontrols>
        <wssuc:InputFormControl LabelText="Archive Document Library:"
runat="server">
                <Template_Control>
                    <asp:DropDownList ID="lstArchiveList" runat="server" />
                </Template_Control>
        </wssuc:InputFormControl>
    </template_inputformcontrols>
</wssuc:InputFormSection>
```

- Implement the last section containing the **Start** and **Cancel** buttons.

```
<%-- Start and Cancel Buttons --%>
<wssuc:ButtonSection runat="server" ShowStandardCancelButton="false">
    <template_buttons>
        <asp:PlaceHolder runat="server">
            <asp:Button UseSubmitBehavior="false" runat="server" class="ms-
ButtonHeightWidth" OnClick="Start_Click" Text="Start" id="BtnSubmit" />  
            <asp:Button UseSubmitBehavior="false" runat="server" class="ms-
ButtonHeightWidth" OnClick="Cancel_Click" Text="Cancel" id="cmdCancel"
causesvalidation=false />
        </asp:PlaceHolder>
    </template_buttons>
</wssuc:ButtonSection>
```

Exercise 2: Integration Instantiation Forms into the Workflow

1) Update the workflow to use the initiation data instead of the association data.
   - Open **Workflow.cs** as code by right clicking it in the **Solution Explorer** and clicking **View Code**.
   - Locate the **OnWorkflowActivated_Invoked** method.
   - This method uses **WorkflowProperties.AssociationData** to initialize the workflow. Change the code to use the **InitiationData** property instead.

```
private void OnWorkflowActivated_Invoked(object sender, ExternalDataEventArgs e)
{
    // deserialize init data, if none use defaults
    this.AssocInitData = AssocInitData.Deserialize(
        WorkflowProperties.InitiationData);
}
```

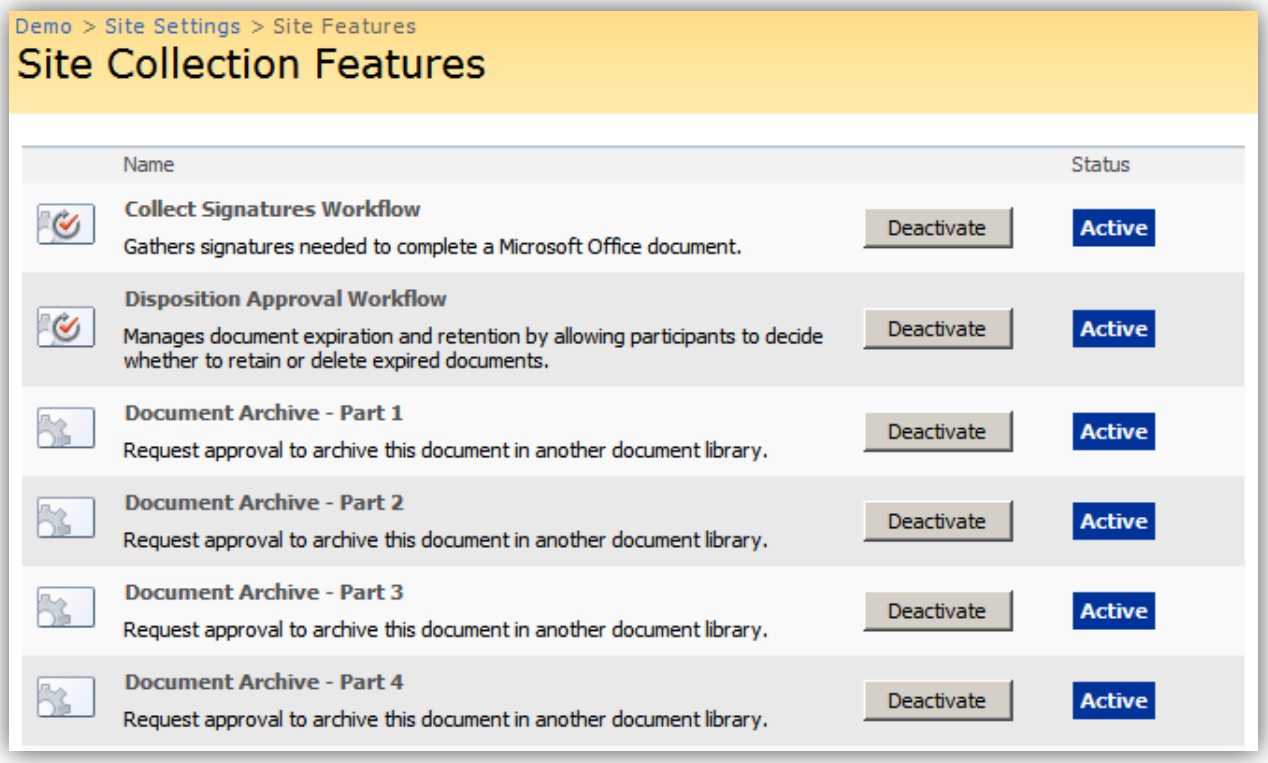
Exercise 3: Deploying custom Instantiation Forms

1) Register the new custom Instantiation form by assigning the **InstantiationUrl** attribute in the feature's **Workflow** element.

```
<Workflow ...
     InstantiationUrl="_layouts/DocArchivePart4InitForm.aspx"
     StatusUrl="_layouts/WrkStat.aspx">
  <Categories/>
  <MetaData>
  </MetaData>
</Workflow>
```

2) Rebuild the workflow.
   - Right click the project in the **Solution Explorer** and click **Rebuild**.
   - In the **Output** window, verify the post build actions completed successfully.

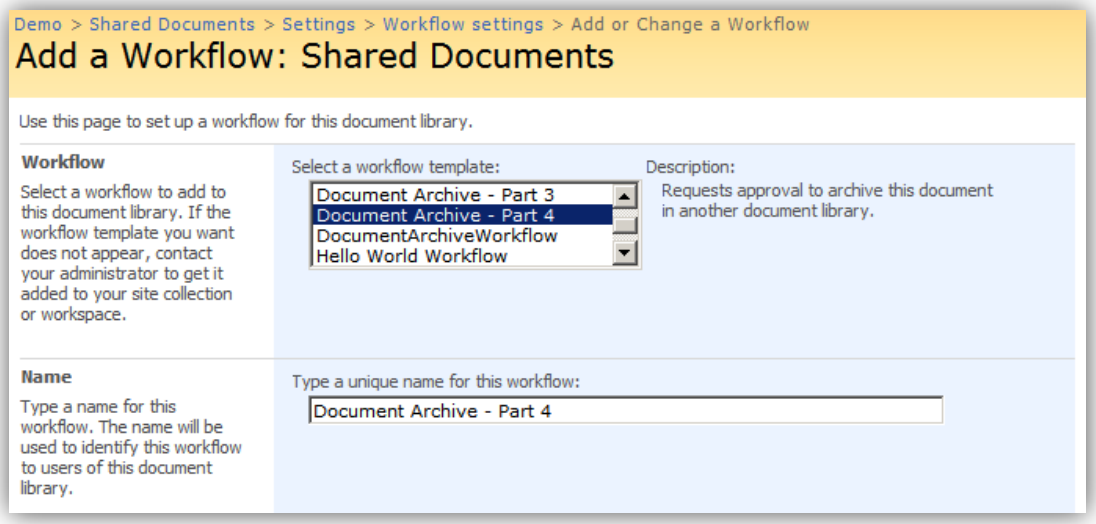
3) Activate the new **DocumentArchiveWorkflowPart4** feature.
   - Using **Internet Explorer** navigate to the **Demo** site collection at **http://litwareinc.com/sites/Demo**.
   - Open the features list by clicking **Site Actions -> Site Settings**.
   - On the **Site Settings** page, click **Site collection features** in the **Site Collection Administration** section.
   - Click the **Activate** button next to the **Document Archive – Part 4** feature.



4) Create an association between the **Shared Documents** document library and the new **Document Archive – Part 4** workflow.

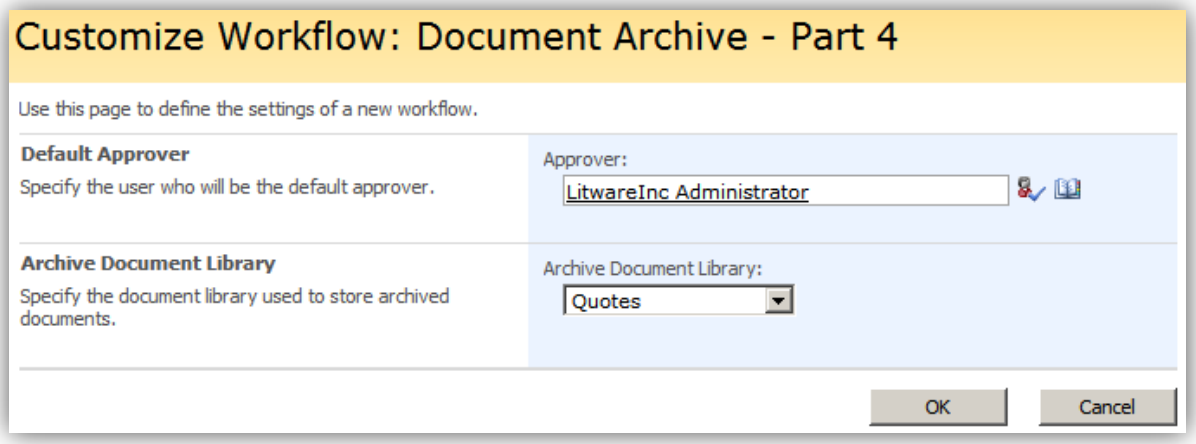
- Navigate to the **Shared Documents** document library in the **Demos** site.
- Click **Settings -> Document Library Settings** to load the settings page.
- Click the **Workflow settings** link in the **Permissions and Management section**.
- Create a new workflow using the **Document Archive – Part 4** workflow template and a name of **Document Archive - Part 4**.

  Use the default values for both list and startup options.



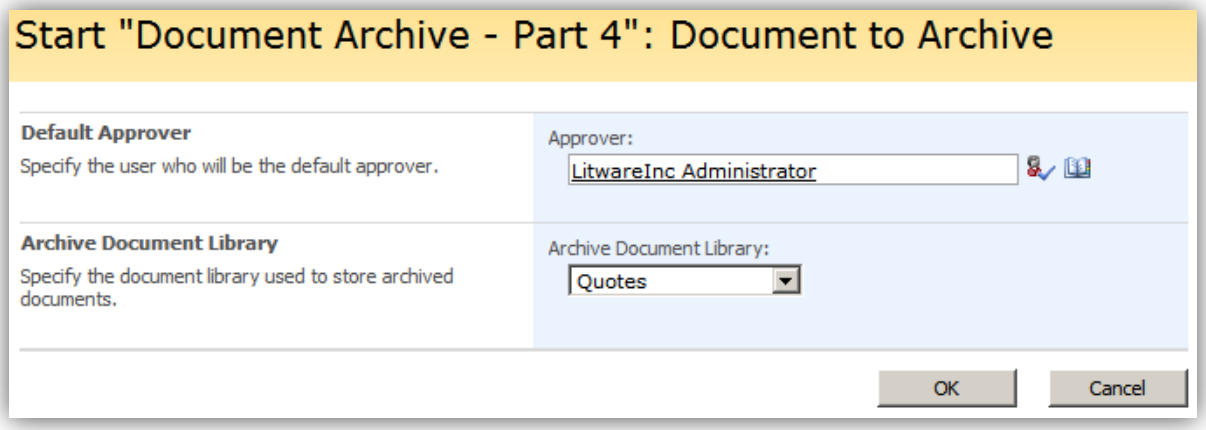
- In the custom association form, enter the **Administrator** as both the default approver and the manager. Select the **Quotes** document library as the archive location.

  *If you'd like, create an **Archive** document library to use instead.*



5) Run the **Document Archive – Part 4** workflow on a document.

- Navigate to the Shared Documents document library, hover over the new document, and select **Workflows** from the drop down menu.
- In the workflows page, click the **Document Archive – Part 4** to start the workflow.
- In the custom initiation page, verify the form initialized using the data provided in the custom association form.
- Change any values you would like and click **Start** to start a new instance of the workflow.



6) Complete the workflow by approving the task in the workflow status page.

Exercise 4: Creating a Custom Modification Form

1) Add a new base class named **DocArchivePart4ModForm** that will be uses as the code behind for an ASPX page.

- Create a new class named **DocArchivePart4ModForm** in the **UI** folder.

  *Right click the **UI** folder and click **Add -> Class…**.*
  *Name the new file **DocArchivePart4ModForm.cs** and click **Add**.*

- Change the **DocArchivePart4InitForm** class into a public class that derives from **Microsoft.SharePoint.WebControls.LayoutsPageBase**.

```
public abstract class DocArchivePart4ModForm : LayoutsPageBase
{ }
```

- Add private fields to the class to store information about the workflow modification as well as the list item the workflow instance being modified.

```
private SPList _list;
private SPListItem _listItem;
private SPWorkflow _workflow;
private SPWorkflowModification _modification;
```

2) Add a protected field and property that will allow interaction between the ASPX page and the code behind.

- Add a **protected** field named **pplApprover** of type **PeopleEditor** to the class.

```
protected PeopleEditor pplApprover;
protected PeopleEditor pplManager;
```

3) Override the **OnLoad** method in the new **DocArchivePart4InitForm** class so it reads the **Url** parameters and creates the necessary SharePoint API objects and binds that data to the UI.

- Override the **OnLoad** method of the base class.

```
protected override void OnLoad(EventArgs e)
{ }
```

- Retrieve the list id, list item id, workflow id, and the modification id from the URL parameters list.

```
// read the form level parameters
string listId = Request.Params["List"];
string listItemId = Request.Params["ID"];
string workflowId = Request.Params["WorkflowInstanceID"];
string modificationId = Request.Params["ModificationID"];
```

- Use the IDs retrieved from the URL to load the related SharePoint API objects.

```
// find the list and workflow modification objects
_list = Web.Lists[new Guid(listId)];
_listItem = _list.GetItemById(Convert.ToInt32(listItemId));
_workflow = _listItem.Workflows[new Guid(workflowId)];
_modification = _workflow.Modifications[new Guid(modificationId)];
```

- Check if the request is a postback using the **Page.IsPostBack** property. Inside this **if** statement is where all of the UI initialization will be done.

```
// populate the UI
if (!this.IsPostBack)
{
}
```

- Deserialize the modification data related to workflow association using the **ModificationData.Deserialize** method and attach the data to the form controls.

*Read the modification data from the **SPWorkflowModification.ContextData** property.*

```
// deserialize the association data
ModificationData modData =
ModificationData.Deserialize(_modification.ContextData);

// bind the controls to the association data
pplApprover.CommaSeparatedAccounts = modData.ApproverUserName;
```

- Finish up the **OnLoad** method by calling the base implementation of the **OnLoad** method

```
// call the base implementation
base.OnLoad(e);
```

4) Implement the event handler called after a click of the **Cancel** button.

- Create a **protected** event handler named **Cancel_Click**.
- Use the **SPUtility.Redirect** method to handle the automatic redirection to either the source url parameter or the current list's default view url.

```
protected void Cancel_Click(object sender, EventArgs e)
{
    // redirect to the page in the source url parameter or the default page
    SPUtility.Redirect(_list.DefaultViewUrl,
        SPRedirectFlags.UseSource, this.Context);
}
```

5) Implement the event handler that will be called when the **Start** button is clicked.

- Create a protected event handler named **Start_Click**.

```
protected void Submit_Click(object sender, EventArgs e)
{
}
```

- Create a new **AssocInitData** object and populate it using the values currently stored in the form controls and serialize it using the **Serialize** method.

```
// populate the modification data using the UI
ModificationData modData = new ModificationData();
modData.ApproverUserName = (pplApprover.Entities[0] as PickerEntity).Key;

// serialize the modification data
string modificationData = modData.Serialize();
```

- Modify the current workflow instance using the **SPWorkflowManager.ModifyWorkflow** method.

*The **SPWorkflowManager** is accessible through the current's **SPWeb** object's **SPSite** parent object.*

```
// apply the modification to the workflow
Web.Site.WorkflowManager.ModifyWorkflow(_workflow,
                                        _modification,
                                        modificationData);
```

- Use the **SPUtility.Redirect** method to handle the automatic redirection to either the source url parameter or the current list's default view url.

```
// redirect to the page defnined in the source url parameter or the default page
SPUtility.Redirect(_list.DefaultViewUrl,
    SPRedirectFlags.UseSource, this.Context);
```

6) Create a new .ASPX page named **DocArchivePart4ModForm.aspx** that represents the ASP.NET markup for the task page.

- Create a new text file named **DocArchivePart4ModForm.aspx** in the **Layouts** folder.

  *Right click the **Layouts** folder and click **Add -> New Item…**.*

  *Select the **Generate** category on the left and select a template of **Text file**.*

  *Name the new file **DocArchivePart4ModForm.aspx** and click **Add**.*

- Enter the page directives that define the code behind class and the master page for the task form.

```
<%@ Assembly Name="DocumentArchiveWorkflowPart4, Version=1.0.0.0,
    Culture=neutral, PublicKeyToken=15812f954569663f" %>

<%@ Page Language="C#" MasterPageFile="~/_layouts/application.master"
    EnableSessionState="true" ValidateRequest="False"
    Inherits="DocumentArchiveWorkflowPart4.UI.DocArchivePart4ModForm" %>
```

- Register the tag prefix and locations of the SharePoint web and user controls used to format the page.

```
<%@ Register TagPrefix="SharePoint" Namespace="Microsoft.SharePoint.WebControls"
    Assembly="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce111e9429c" %>
<%@ Register TagPrefix="Utilities" Namespace="Microsoft.SharePoint.Utilities"
Assembly="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce111e9429c" %>
<%@ Register TagPrefix="wssuc" TagName="InputFormSection"
Src="/_controltemplates/InputFormSection.ascx" %>
<%@ Register TagPrefix="wssuc" TagName="InputFormControl"
Src="/_controltemplates/InputFormControl.ascx" %>
<%@ Register TagPrefix="wssuc" TagName="ButtonSection"
Src="/_controltemplates/ButtonSection.ascx" %>
```

- Define the content for the page title and the form specific headers.

```
<asp:Content ID="PageTitle" ContentPlaceHolderID="PlaceHolderPageTitle"
runat="server">
    Modify Workflow</asp:Content>
<asp:Content ID="PageTitleInTitleArea"
    ContentPlaceHolderID="PlaceHolderPageTitleInTitleArea" runat="server">
    <%= "Modify" %></asp:Content>
```

7) Define the ASPX markup that will be used to create the data entry section of the custom task form.

- Define the **ContentPlaceHolder** control for **PlaceHolderMain** and place a table within it.

```
<asp:Content ID="Main" ContentPlaceHolderID="PlaceHolderMain" runat="server">
    <table cellspacing="0" cellpadding="0" style="border: none; width: 100%"
class="ms-propertysheet">
    </table>
</asp:Content>
```

- Implement the first **InputFormSection** that allows modification of the approver.

```
<%-- Deafault Approver Input --%>
<wssuc:InputFormSection Title="Default Approver" Description="Specify the user
who will be the default approver." runat="server">
    <template_inputformcontrols>
            <wssuc:InputFormControl runat="server" LabelText="Approver:">
            <Template_Control>
                <SharePoint:PeopleEditor id="pplApprover"
                  AllowEmpty="false"
                        ValidatorEnabled="true"
                        MultiSelect="false"
                        runat="server"
                        SelectionSet="User"
                  width='300px' />
            </Template_Control>
        </wssuc:InputFormControl>
    </template_inputformcontrols>
</wssuc:InputFormSection>
```

- Implement the last section containing the **OK** and **Cancel** buttons.

```
<%-- Submit and Cancel Buttons --%>
<wssuc:ButtonSection runat="server" ShowStandardCancelButton="false">
    <template_buttons>
        <asp:PlaceHolder runat="server">
            <asp:Button UseSubmitBehavior="false" runat="server" class="ms-
ButtonHeightWidth" OnClick="Submit_Click" Text="OK" id="BtnSubmit" />  
            <asp:Button UseSubmitBehavior="false" runat="server" class="ms-
ButtonHeightWidth" OnClick="Cancel_Click" Text="Cancel" id="cmdCancel"
causesvalidation=false />
        </asp:PlaceHolder>
    </template_buttons>
</wssuc:ButtonSection>
```

Exercise 5: Enabling Modifications in the Workflow

1) Communicate with the SharePoint Workflow hosting environment to allow modification of this workflow.

- Open **Workflow.cs** in the designer by right clicking it in the **Solution Explorer** and selecting **View Designer**.
- Double click the **WorkflowActivatedActivity** activity to open its designer.
- Drag an **EnableWorkflowModification** activity from the toolbox into the **WorkflowActivatedActivity** immediately following the **OnWorkflowActivated** activity and name it **EnableModification**..

  *In the properties pane, set the **Name** property to **EnableModification**.*

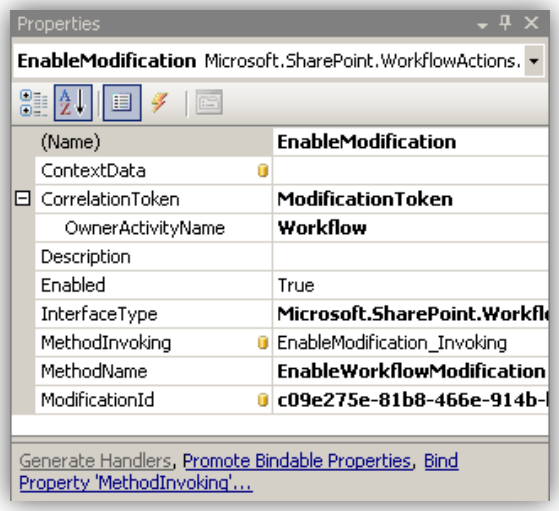
- Assign a correlation token to the modification and assign the id of the modification to enable.

  *In the properties pane, set the **CorrelationToken** property to **ModificationToken**.*

  *Expand the **CorrelationToken** property in the property pane.*

  *Set the **OwnerActivityName** property to **Workflow***

  *Set the **ModificationId** property to **c09e275e-81b8-466e-914b-bae95eb27dfa**.*



- Implement the **Invoking** method and use it to serialize a **ModificationData** class into the **ContextData** of the current activity.

  *In the properties pane, set the **Invoking** property to **EnableModification_Invoking** and press **Enter**.*

```
private void EnableModification_Invoking(object sender, EventArgs e)
{
    // populate the modification data object
    ModificationData modData = new ModificationData();
    modData.ApproverUserName = AssocInitData.ApproverUserName;

    // serialize and assign the context data to the modification
    EnableWorkflowModification activity = sender as EnableWorkflowModification;
    activity.ContextData = modData.Serialize();
}
```

2) Add an **EventDrivenActivity** to the workflow that will receive the notification the workflow has been modified. When that happens, it will update the **AssocInitData** object and reset any tasks already created.

- On the **State Machine** designer canvas, right click and select **Add EventDriven**.
- Rename the new event driven activity to **WorkflowModifiedActivity**.

  *In the properties pane, set the **Name** property to **WorkflowModifiedActivity**.*

- Drag an **OnWorkflowModified** activity from the toolbox into the **WorkflowModifiedActivity activity** and name it **OnWorkflowModified**..

  *In the properties pane, set the **Name** property to **OnWorkflowModified**.*

- Update the new activity to respond to the modification previously enabled.

  *Set the **CorrelationToken** property to **ModificationToken**.*
  *Set the **ModificationId** property to **c09e275e-81b8-466e-914b-bae95eb27dfa**.*

- Implement the **Invoked** event and retrieve the updated modification data for integration back into the running workflow.

  *Set the **Invoked** property to **OnWorkflowModified_Invoked** and press **Enter**.*
  *Cast the **e** parameter to **SPModificationEventArgs** to retrieve the modification data.*
  *Deserialize using **ModificationData.Deserialize** and copy the new data into the **AssocInitData** object.*

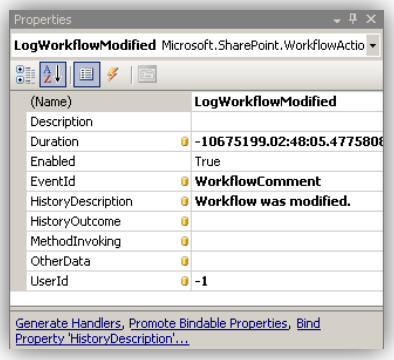
```
private void OnWorkflowModified_Invoked(object sender, ExternalDataEventArgs e)
{
    SPModificationEventArgs args = e as SPModificationEventArgs;

    // deserialize the modification data and apply the changes
    ModificationData modData = ModificationData.Deserialize(args.data);
    AssocInitData.ApproverUserName = modData.ApproverUserName;
}
```

3) Log the fact that the modification has taken place and transition back to the **WaitingForApproval** state.

- Drag a **LogToHistoryListActivity** from the toolbox into the **WorkflowModifiedActivity** activity and name it **LogWorkflowModified**.

  *In the properties pane, set the **Name** property to **LogWorkflowModified.***

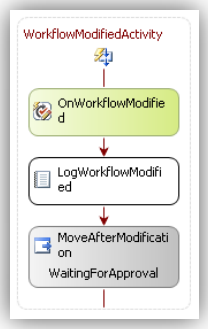
  *Set the **HistoryDescription** property to **"Workflow was modified."***



- Drag a **SetState** activity from the toolbox into the **WorkflowModifiedActivity** at the end of the sequence and name it **MoveAfterModification**.

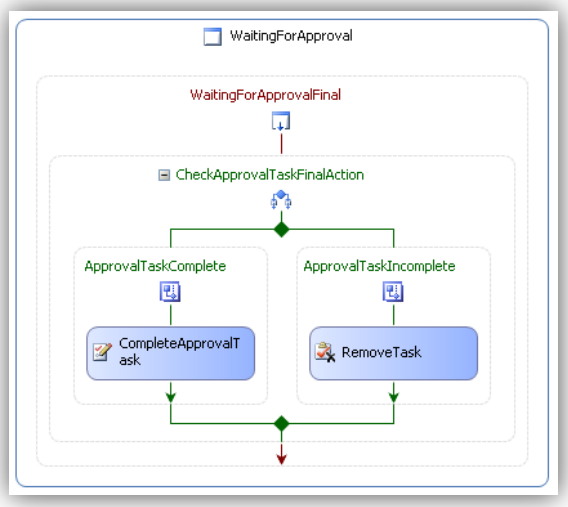
  *In the properties pane, set the Name property to **MoveAfterModification**.*

  *Set the **TargetStateName** property to **WaitingForApproval**.*



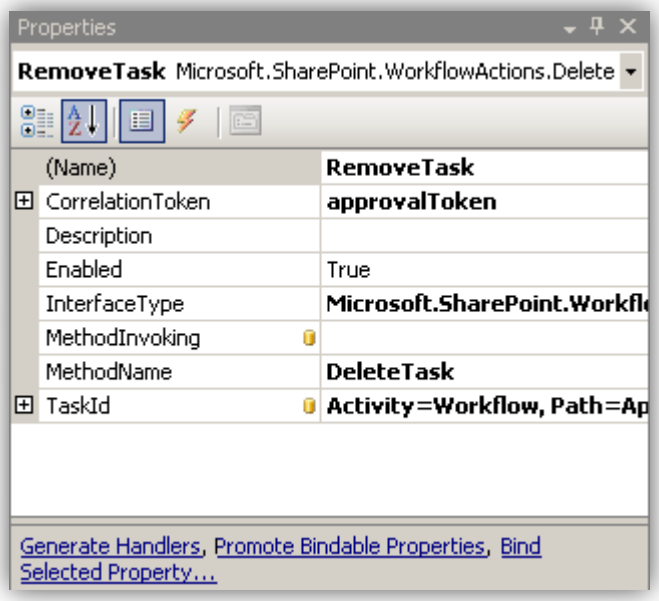
4) Since modifications will happen when tasks already exist, they will need to be recreated after the modification. To do this, the finalizer and initializer for the state come into play.

- Open the **WaitingForApprovalFinal** activity in the State Machine designer canvas.
- Drag a **DeleteTask** activity from the toolbox into the **ApprovalTaskIncomplete** branch and name it **RemoveTask**



- Assign the correlation token and **TaskId** necessary for the activity to delete the task.

  *In the properties pane, set the **CorrelationToken** property to **approvalToken**.*
  *Bind the **TaskId** property to the **ApprovalTaskId** property in the workflow class.*



Exercise 6: Deploying custom Modification Forms

1) Register the new custom modification form by assigning the **ModificationUrl** attribute in the feature's **Workflow** element and registering the modification metadata.

   - Open the **workflow.xml** file in the **Template/Features/DocumentArchiveWorkflowPart4** folder.
   - Add the **ModificationUrl** attribute to the **Workflow** element. Point it toward **_layouts/DocArchivePart4ModForm.aspx**.

```
<Workflow ...
     ModificationUrl="_layouts/DocArchivePart4ModForm.aspx"
     StatusUrl="_layouts/WrkStat.aspx">
  <Categories/>
  <MetaData>
  </MetaData>
</Workflow>
```

- Add an element in the Metadata that defines the display name for the modification enabled in the workflow.

  *Use the format **Modification_{0}_Name** where the **{0}** token is replaced by the modification's id.*

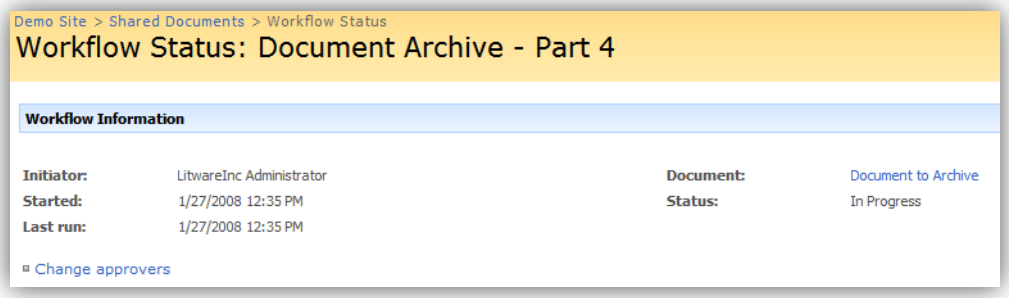
```
<MetaData>
  <Modification_c09e275e-81b8-466e-914b-bae95eb27dfa_Name>Change
approvers</Modification_c09e275e-81b8-466e-914b-bae95eb27dfa_Name>
</MetaData>
```

2) Rebuild the workflow.

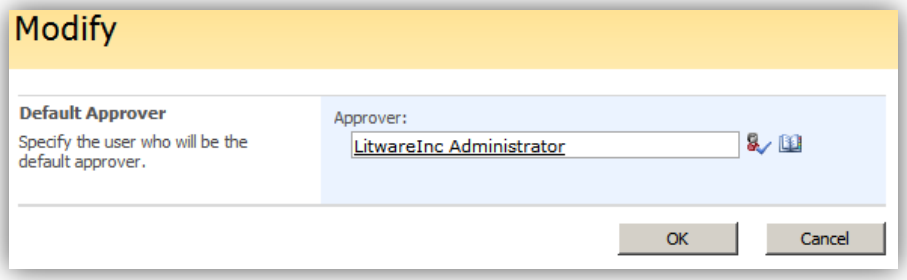
- Right click the project in the **Solution Explorer** and click **Rebuild**.
- In the **Output** window, verify the post build actions completed successfully.

3) Run the **Document Archive – Part 4** workflow on a document.

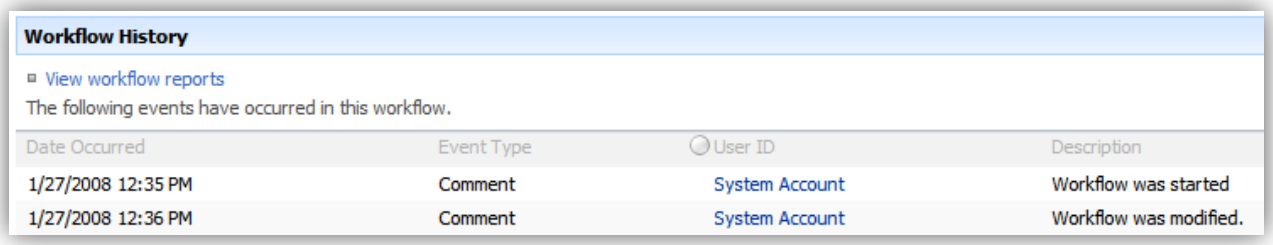
- Navigate to the **Shared Documents** document library, hover over the new document, and select **Workflows** from the drop down menu.
- In the workflows page, click the **Document Archive – Part 4** to start the workflow.
- When the custom instantiation page opens, click **Start** to create the workflow instance with the default values.
- In the Shared Documents UI, click the In Progress link to view the status of the workflow. Click the **Change approvers** link in the Workflow Information section to modify the workflow.



- In the modification page, do not change anything, but click **Ok** to apply the modification.



- In the workflow status page, verify that there is a modification message in the workflow history.



4) Complete the workflow by approving the task.

Challenge: Integrate Instantiation and Modification Forms into Solution Packages

1) Add the task form's .ASPX file to the **Manifest.xml** file.
   - Open the **Manifest.xml** file in the **Solution** folder.
   - Add a **TemplateFile** element telling the solution installer to install the association form when the solution is deployed.

```
<TemplateFiles>
  <TemplateFile Location="LAYOUTS\DocArchivePart4TaskForm.aspx"/>
  <TemplateFile Location="LAYOUTS\DocArchivePart4AssocForm.aspx"/>
  <TemplateFile Location="LAYOUTS\DocArchivePart4InitForm.aspx"/>
  <TemplateFile Location="LAYOUTS\DocArchivePart4ModForm.aspx"/>
</TemplateFiles>
```

2) Add the custom task form to the **Package.ddf** file.
   - Open the **Package.ddf** file in the **Solution** folder.
   - Add lines that package the **DocArchivePart4InitForm.aspx** and **DocArchivePart4ModForm.aspx** and puts it in the **LAYOUTS** folder in the .cab file.

```
..\..\Template\Layouts\DocArchivePart4InitForm.aspx
LAYOUTS\DocArchivePart4InitForm.aspx
..\..\Template\Layouts\DocArchivePart4ModForm.aspx
LAYOUTS\DocArchivePart4ModForm.aspx
```

3) Manually uninstall the previously deployed **Document Archive** feature.
   - Open a command window in the SharePoint **bin** directory.

     *Click **Start -> Run** and enter **cmd**.*

     *Navigate to C:**\Program Files\Common Files\Microsoft Shared\web server extensions\12\BIN**.*
   - Execute **stsadm.exe** to uninstall the **DocuentArchiveWorkflowPart4** feature.

```
stsadm –o uninstallfeature –name DocumentArchiveWorkflowPart4 –force
```

   - Delete the folder at **C:\Program Files\Common Files\Microsoft Shared\web server extensions\12\Template\Features\DocumentarchiveWorkflowPart4**.
   - Delete the **DocArchivePart4AssocForm.aspx, DocArchivePart4InitForm.aspx, DocArchivePart4TaskForm.aspx,** and **DocArchivePart4ModForm.aspx** files from the **C:\Program Files\Common Files\Microsoft Shared\web server extensions\12\Template\Layouts** folder.

4) Install and deploy the new solution package.

- In the same command window as the previous step, execute **stsadm** to add the feature to SharePoint.

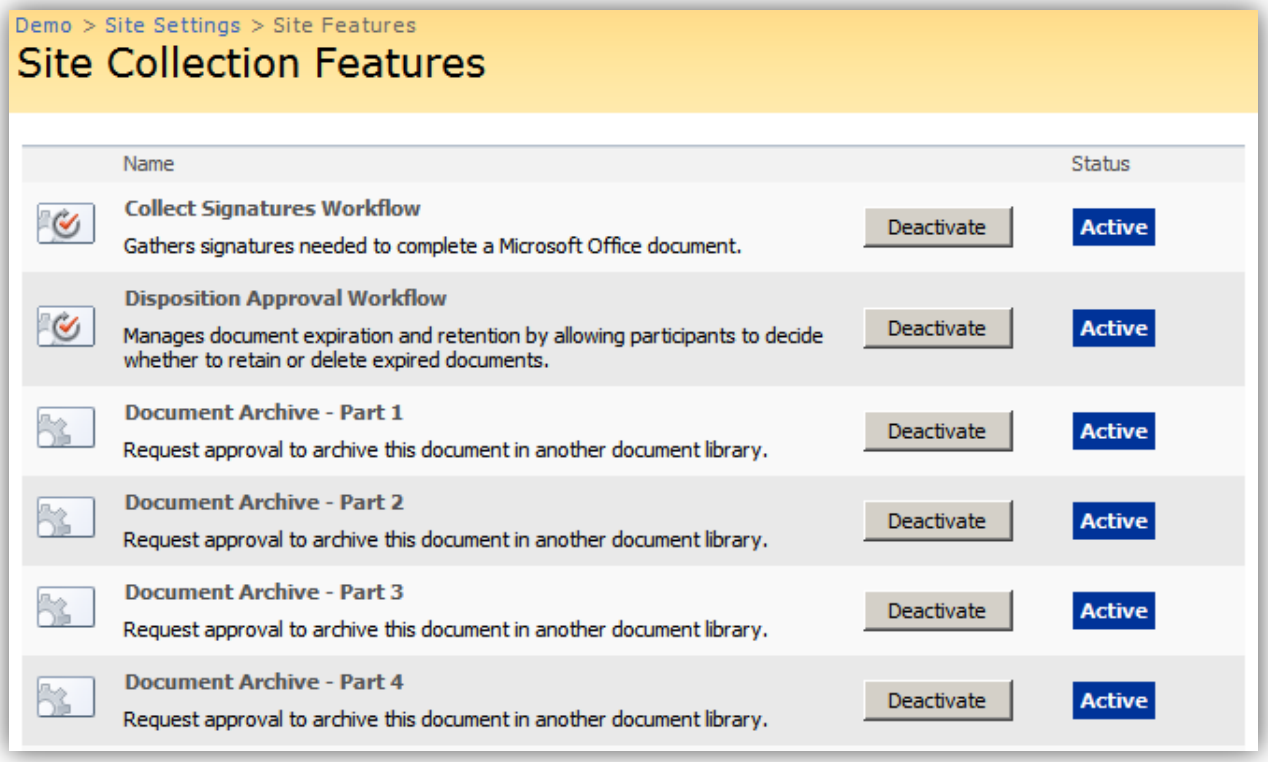
```
stsadm -o addsolution -filename
"C:\Labs\Lab07\StarterFiles\DocumentArchiveWorkflowPart4\bin\Debug\Package\Docum
entArchiveWorkflowPart4.wsp"
```

- Deploy the solution to the farm using the **deploysolution** command in **stsadm**.

```
stsadm -o deploysolution -name DocumentArchiveWorkflowPart4.wsp -
allowgacdeployment -local
```

5) Verify the feature is still available and activated.

- Using **Internet Explorer** navigate to the **Demo** site collection at **http://litwareinc.com/sites/Demo**.
- Open the features list by clicking **Site Actions -> Site Settings**.
- On the **Site Settings** page, click **Site collection features** in the **Site Collection Administration** section.
- Verify the **Document Archive – Part 4** feature is active.



Challenge:

This scenario implies that the file will be moved from one location to another. For the purposes of simplicity, this action is not outlined in the labs. For an additional challenge, use a **Code Activity** to add code to the workflow that will copy the file to the appropriate list. To gain access to the current item, use the **WorkflowProperties** property of the workflow, look at the **AssocInitData** property to find information about the archive folder and finally use the **SPListItem.CopyTo** method to perform the copy.

Lab 08: Integrating InfoPath Forms into a SharePoint Workflow Template

Lab Overview: Designing workflow forms in ASP.NET is not a task easily delegated to web designers. There is no simple way to provide web designers with a visual design environment; therefore, workflow form design is currently the responsibility of developers. This is not a good scenario for forms that may need minor layout and branding adjustments that don't change the way they work.

The solution to this problem is to use InfoPath. InfoPath provides a functional user interface that interacts with XML data. This is perfect for interacting with SharePoint Workflow Association, Initiation, Task, and Modification data. In this lab, you'll be building a simple approval workflow using InfoPath as the user interface.

Exercise 0: Setup

1) If you did not complete lab 4, you will need to create the **Demo** site collection.

   Open SharePoint and browse to the Demo site collection

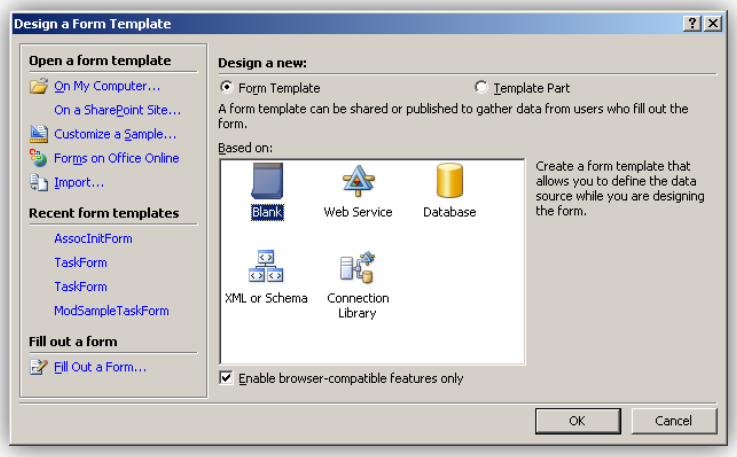
   *The url is **http://litwareinc.com/sites/Demo**.*

   *If the site collection does not exist, create it using the **CreateDemo.bat** file in the **C:\Labs\Files folder**.*

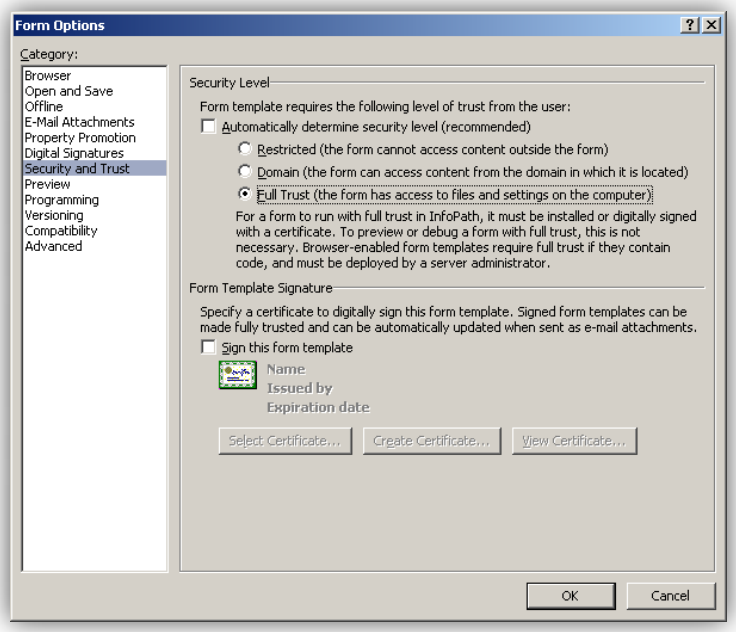
2) Open the starter **VS 2008** solution at **\Labs\Lab08\StarterFiles\InfoPathApprovalWorkflow.sln**.

Exercise 1: InfoPath Association Forms

1) Create a new InfoPath form named **AssocInitForm.xsn** in the project's **Forms** folder.
   - Open InfoPath by clicking the **Start Button** and selecting **All Programs -> Microsoft Office -> Microsoft Office InfoPath 2007**.
   - In the **Getting Started** window, click the **Design a Form Template** link in the **Design a form** section.
   - In the **Design a Form Template** window, select a **Blank** form template and make sure the **Enable browser-compatible features only** check box is checked and click **OK**.



2) Update the form's security options to grant the form full trust when hosted in Forms Services.
   - Click the **Tools -> Form Options** item in the menu bar.
   - In the **Form Options** dialog, select **Security and Trust** in the **Category** list box.
   - Uncheck **Automatically determine security level** and select **Full Trust**.
   - Click **OK** to close the dialog.



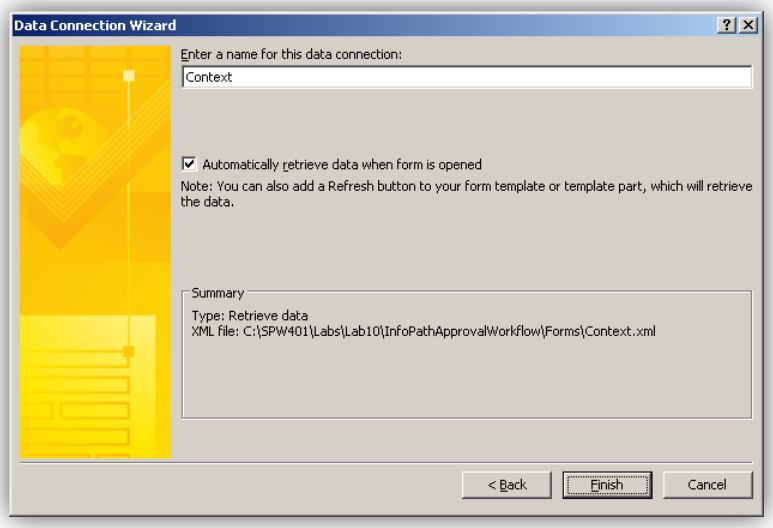
3) Add a secondary data source that will allow the hosting environment to provide extra data to the form. This is required to use the **Contact Selector** in the next step.
   - In **Visual Studio 2008**, add a new file named **Context.xml** to the **Forms** folder.

     *Right click on the **Forms** folder in the **Solution Explorer** and select **Add -> New Item**.*
     *Select the **Data** category in the left side of the screen and then select **Xml File** in the **Template** list.*
     *Enter a name of **Context.xml** and click **Add**.*
   - Enter the following XML into the new **Context.xml** file.

```
<Context
  isStartWorkflow="true"
  isRunAtServer="false"
  provideAllFields="true"
  siteUrl="" />
```

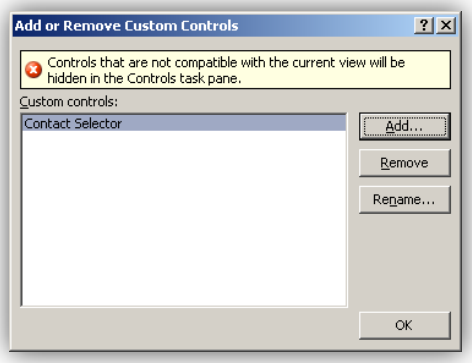
   - In InfoPath, click **Tools -> Data Connections** in the menu bar.
   - In the **Data Connections** dialog, click the **Add** button.
   - Select the **Create a new connection to** option and then select **Receive data** and click **Next**.
   - On the next page, choose to receive data from an **XML document**.
   - Browse to the **Context.xml** file just created and click **Next**.
   - On the next page, select to include the file as a resource and click **Next**.

- On the final page, click **Finish** to create the data source.
- In the **Data Connections** dialog, click **Close** to close the window.



4) Add a **Contact Selector** ActiveX control to the controls list for adding to the form later.
   - Locate the **Controls** task pane on the right hand side of the page.

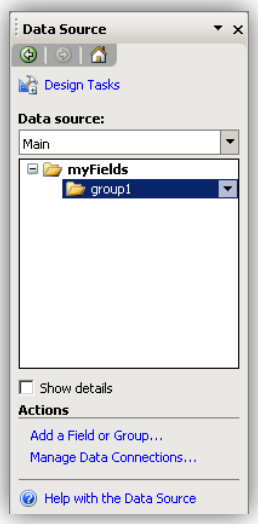
     *If it is not visible, click the drop down menu at the top of the task pane and select **Controls**.*
   - Click **Add or Remove Custom Controls** to add the **Contact Selector** control.
   - In the **Add or Remove Custom Controls** dialog, click the **Add** button.
   - In the first page of the wizard, choose **ActiveX Control** and click **Next**.
   - Select **Contact Selector** in the control list and click **Next**.
   - Choose to not include a cab file and click **Next**.
   - Select a binding property of **Value** and click **Next**.
   - In the **Field or group type** drop down list, choose **Field or Group (any data type)** and click **Finish**.
   - On the summary page, click **Close**.
   - Now the **Contact Selector** should be visible in the **Add or Remove Custom Controls** dialog. Close the dialog when you are done.



5) Add a **Contact Selector** to the form and bind it to the data source.

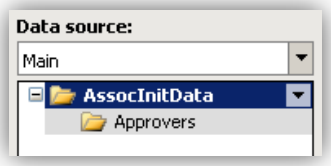
- Drag a **Contact Selector** from the **Controls** task pane onto the design surface.
- Switch to the **Data Source** view using the drop down menu at the top of the task pane.

  *Notice the **Contact Selector** control has already created a new group named **group1**.*



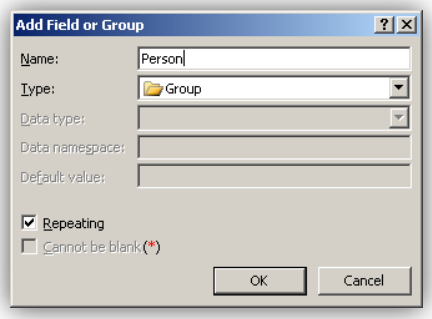
- Rename the **myFields** and **group1** groups to **AssocInitData** and **Approvers**.

  *Right click **myFields** and select **Properties**.*
  *In the **Field or Group Properties** dialog, change the name to **AssocInitData**.*
  *Click **OK** to close the dialog.*
  *Repeat to change **group1** to **Approvers**.*



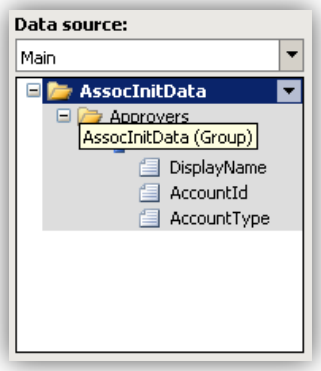
- Add a **Person** repeating group as a child of the **Approvers** group.

  *Right click the **Approvers** group and select **Add**.*
  *In the **Add Field or Group** dialog, enter a name of **Person**.*
  *Change the **Type** to **Group**.*
  *Check the **Repeating** checkbox.*
  *Click **OK** to create the new group.*



- Add three more string fields named **DisplayName**, **AccountId**, **AccountType** as children of the **Person** group.

  *Right click the **Person** group and click **Add**.*
  *In the **Add Field or Group** dialog, enter a name of **DisplayName**.*
  *Verify the **Data type** is **string**.*
  *Click **OK** to create the new field.*
  *Repeat the above steps for **AccountId** and **AccountType**.*

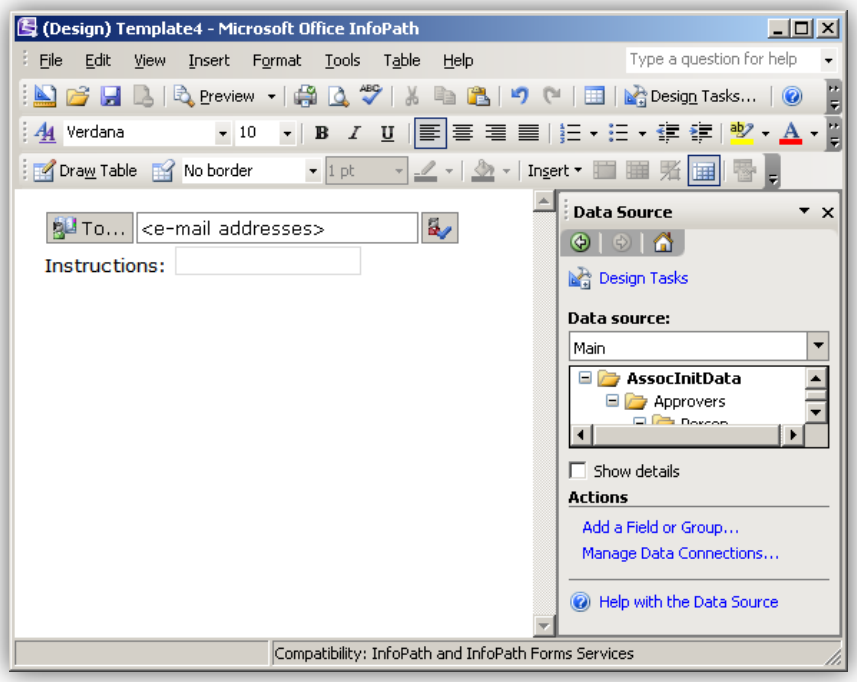


6) Add a **Text Box** control to the form to be used to enter any instructions to the approver.

- Add an **Instructions** field to the **AssocInitData** group.

  *In the **Data Source** task pane, right click the **AssocInitData** group and select **Add**.*
  *In the **Add Field or Group** dialog, enter a name of **Instructions***
  *Verify the **Data type** is **string**.*
  *Click **OK** to create the new field.*

- Drag the **Instructions** field onto the design canvas to create a new text box.

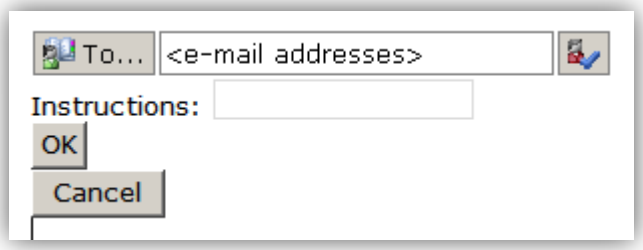


7) Add two buttons to the page for the **OK** and **Cancel** operations.

- Switch to the **Controls** task pane and drag two button controls onto the design canvas.
- Change the name of one button to **OK**.

  *Right click the button and select **Button Properties**.*
  *In the **Button Properties** dialog, change the **Label** value to **OK**.*
  *Click **OK** to close the dialog.*
- Change the name of the other button to **Cancel**.

  *Right click the button and select **Button Properties**.*
  *In the **Button Properties** dialog, change the **Label** value to **Cancel**.*
  *Click **OK** to close the dialog.*



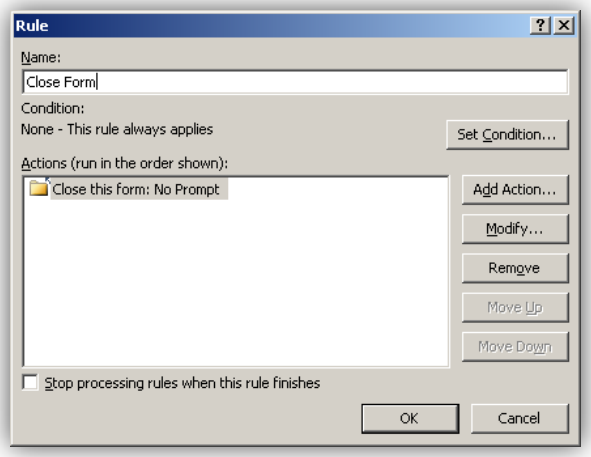
8) Define rules for the **Cancel** button that will cause it to close the form.

- Open the **Rules** dialog to manage the rules that will execute when the button is clicked.

  *Right click the cancel button and select **Button Properties**.*
  *Click the **Rules** button to open the **Rules** dialog.*
- Add a new rule named **Close Form** that will be executed when the button is clicked.

  *Click the **Add** button to add a new rule.*
  *Change the **Name** property in the **Rule** dialog to **Close Form**.*
- Add a close form action that will close the current form.

  *Click **Add Action** to open the **Action** dialog.*
  *In the **Action** dialog, select an action of **Close the form***
  *Click **OK** to close the dialog.*



- Close all of the open dialogs.

9) Define rules for the **OK** button that will cause it to submit and then close the form.

- Open the **Rules** dialog to manage the rules that will execute when the button is clicked.

  *Right click the cancel button and select **Button Properties**.*
  *Click the **Rules** button to open the Rules dialog.*

- Add a new rule named **Submit and Close Form** that will be executed when the button is clicked.

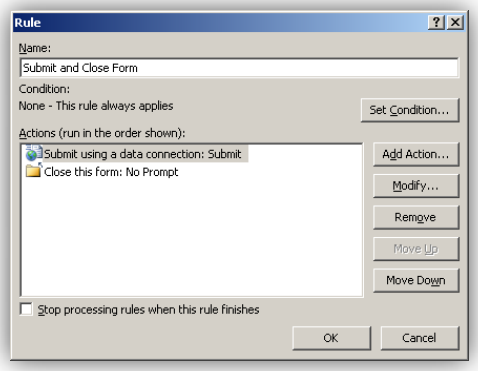
  *Click the **Add** button to add a new rule.*
  *Change the **Name** property in the **Rule** dialog to **Submit and Close Form**.*

- Add a submit form action that will submit the data to Forms Services.

  *Click **Add Action** to open the **Action** dialog.*
  *In the **Action** dialog, select an action of **Submit using a data connection**.*
  *Click the **Add** button to add a new submission data connection.*
  *In the **Data Connection Wizard** create a new **Submit** data connection and click **Next**.*
  *Select **To the hosting environment** and click **Next**.*
  *Keep the default name of **Submit** and click **Finish**.*
  *Back in the **Action** dialog, verify the data connection **Submit** is selected and click **OK** to close the dialog.*

- Add a close form action that will close the current form.

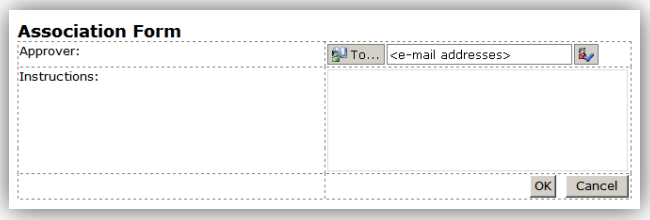
  *Click **Add Action** to open the **Action** dialog.*
  *In the **Action** dialog, select an action of **Close the form***
  *Click **OK** to close the dialog.*



- Close all of the open dialogs.

10) Update the layout of the form to look nicer. Add a title of Associat**ion Form** that is size 14pt and bold and place the controls in a layout table.

- Using InfoPath like a word processor, add the text **Association Form** to the top of the page. Use a font size of 14pt and bold.
- Create a layout grid of 2x3 and place the approver and instructions controls in the top two rows. Adjust the height of the Instructions text box so it can contain multiple lines.
- Place the buttons in the lower right cell of the layout table and right justify them.



11) Save the new form template into the **Forms** folder of the Visual Studio Project.

- Click the **Save** toolbar button.
- Save the form in **C:\Labs\Lab08\StarterFiles\InfoPathApprovalWorkflow\Forms** with a name of **AssocInitForm.xsn** and close InfoPath.

12) Extract the schema file form the new InfoPath form.

- Locate the **AssocInitForm.xsn** file in the **Windows Explorer** and rename it **AssocInitForm.xsn.cab**.
- Open the new cab file and copy the **myschema.xsd** file contained within it into the root folder of the **InfoPathApprovalWorkflow** project.
- Change the name of the **AssocInitForm.xsn.cab** back to **AssocInitForm.xsn**.

13) Update the **AssocInitData** class by using the **xsd.exe** tool to regenerate the **AssocInitData** class.

- In the **InfoPathApprovalWorkflow** folder, rename the **myschema.xsd** to **AssocInitData.xsd**.
- Open a **Visual Studio 2008 Command Prompt** in the **InfoPathApprovalWorkflow** folder.

  *Click the **Start Button** and click **All Programs -> Microsoft Visual Studio 2008 -> Visual Studio Tools -> Visual Studio 2008 Command Prompt**.*

  *In the command prompt, navigate to **C:\Labs\Lab08\StarterFiles\InfoPathApprovalWorkflow**.*
- Execute **xsd.exe** to regenerate the class that will serialize xml to the format defined in the InfoPath schema.

```
xsd.exe AssocInitData.xsd /c /n:InfoPathApprovalWorkflow
```

- Open the generated **AssocInitData.cs** in **Visual Studio 2008**.

  *Comment out the **anyAttrField** field and property if it exists*

Exercise 2: InfoPath Instantiation Form using Views

1) Add a new view named **Instantiation View** to the **InfoPath** form

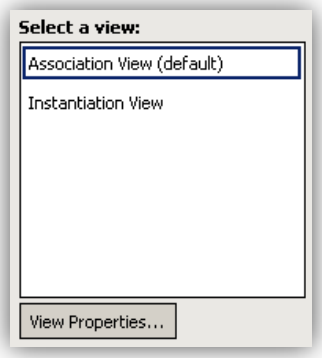
- Design **AssocInitForm.xsn** in InfoPath.
- Select the **Views** task pane using the drop down list at the top of the task pages window.
- Rename the default view to **Association View**.

  *Select the default view and click the **View Properties** button.*

  *Change the **View name** to **Association View** and click **OK**.*
- Add a new view named **Instantiation View** to the form.

  *Click the **Add a New View** link on the **Views** task pane.*

  *In the **Add View** dialog, enter the name of **Instantiation View** and click **OK**.*
- Switch back to the **Association View** by clicking it in the list of **Views**.



2) Copy all of the controls from the **Association View** to the **Instantiation View** and update them to display an instantiation view

- Copy and paste the contents of the **Association View** into the **Instantiation View**.

  *Select the entire content of the **Association View** in the designer and click **Edit -> Copy**.*

  *Switch to the **Instantiation View** using the **Views** task pane.*

  *Click **Edit -> Paste** to paste the contents into the new view.*

- Change the header from **Association Form** to **Instantiation Form**.
- Change the text of the **OK** button to **Start**.

  *Right click the **OK** button and click **Button Properties**.*

  *In the **Button Properties** dialog, change the name from **OK** to **Start**.*

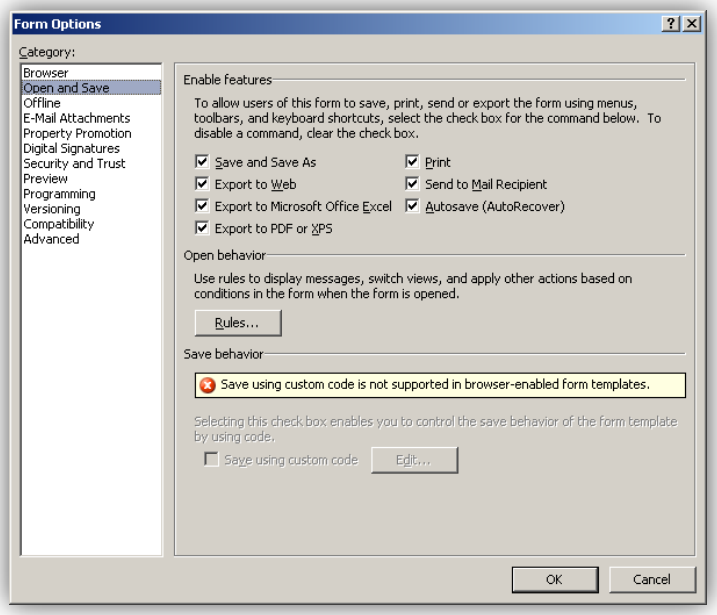
  *Click **OK** to close the dialog.*

3) Add form load rules that will choose the appropriate view based on the **isStartWorkflow** field in the Context data source.

- Using the **Form Options** dialog, open the **Open** behavior rules for the form.

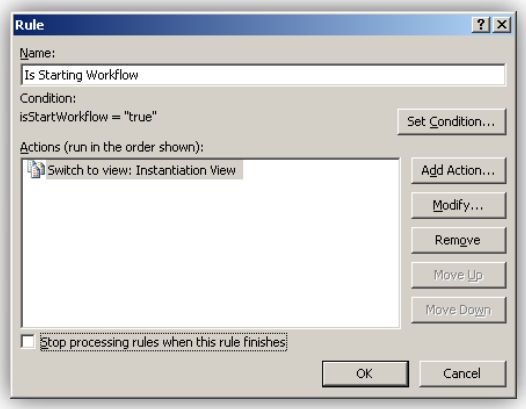
  *Click **Tools -> Form Options** in the menu bar.*

  *Switch to the **Open and Save** category and click the **Rules** button.*



- Add rule to switch to association form if **isStartWorkflow == "false"**.

  *Click the **Add** button to add a new rule.*
  *Set the **Name** of the new rule to **Is Not Starting Workflow**.*
  *Click the **Set Condition** button to set the conditions under which the actions will execute.*
  *On the left hand side drop down list, click **Select a Field or Group**.*
  *In the **Select a Field or Group** dialog, choose the **Context** data source and the **isStartWorkflow** field and click **OK**.*
  *In the center drop down list, verify **is equal to** is selected.*
  *In the right drop down list, click **type text**, enter the text **false**, and click **OK**.*
  *Click **OK** again to close the condition.*
  *Click **Add Action** to add a new switch view action.*
  *Select an action of **Switch Views** and a view of **Association View**.*
  *Click **OK** to close the dialogs back to the **Rules** dialog.*

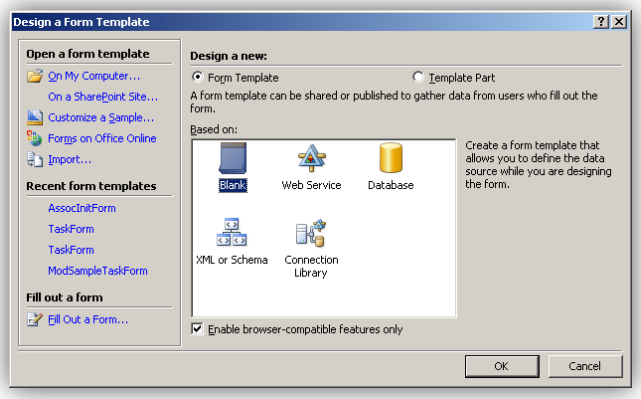


- Add rule to switch to instantiation form if **isStartWorkflow == "true"**.

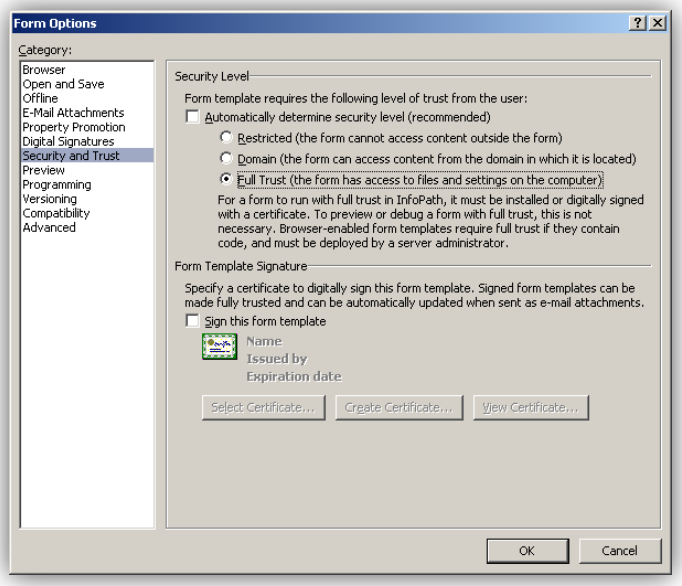
  *Click the **Add** button to add a new rule.*
  *Set the **Name** of the new rule to **Is Starting Workflow**.*
  *Click the **Set Condition** button to set the conditions under which the actions will execute.*
  *On the left hand side drop down list, click **Select a Field or Group**.*
  *In the **Select a Field or Group** dialog, choose the **Context** data source and the **isStartWorkflow** field and click **OK**.*
  *In the center drop down list, verify **is equal to** is selected.*
  *In the right drop down list, click **type text** and enter the text **true** and click **OK**.*
  *Click **OK** again to close the condition.*
  *Click **Add Action** to add a new switch view action.*
  *Select an action of **Switch Views** and a view of **Instantiation View**.*
  *Click **OK** to close the dialogs back to the designer.*

Exercise 3: InfoPath Task Forms

1) Create a new InfoPath form named **TaskForm.xsn**.
   - Open **InfoPath** by clicking the **Start Button** and selecting **All Programs -> Microsoft Office -> Microsoft Office InfoPath 2007**.
   - In the **Getting Started** window, click the **Design a Form Template** link in the **Design a form** section.
   - In the **Design a Form Template** window, select a **Blank form** template and make sure the **Enable browser-compatible features only** check box is checked and click **OK**.



2) Update the form's security options so the form will be granted full trust when hosted in Forms Services.
   - Click the **Tools -> Form Options** item in the menu bar.
   - In the **Form Options** dialog, select **Security and Trust** in the **Category** list box.
   - Uncheck **Automatically determine security level** and select **Full Trust**.
   - Click **OK** to close the dialog.



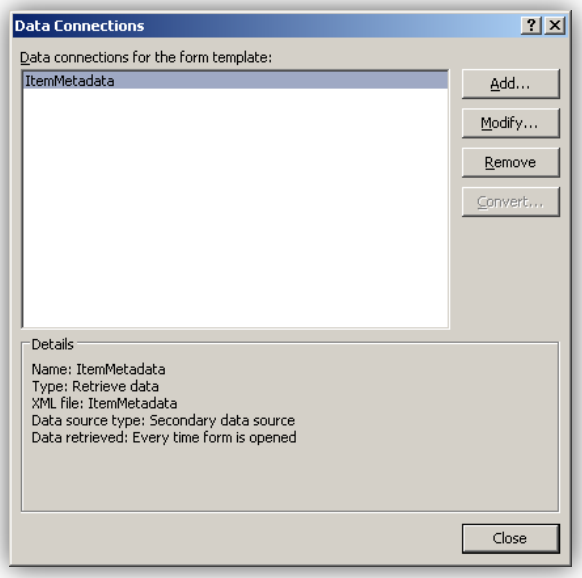
3) Add a secondary data source that will allow the form to access fields in the current task.
   - In **Visual Studio 2008**, add a new file named **ItemMetadata.xml** to the **Forms** folder.

     *Right click on the **Forms** folder in the **Solution Explorer** and select **Add -> New Item**.*
     *Select the **Data** category in the left side of the screen and then select **Xml File** in the **Template** list.*
     *Enter a name of **ItemMetadata.xml** and click **Add**.*
   - Enter the following XML into the new **ItemMetadata.xml** file.

```
<z:row xmlns:z="#RowsetSchema"
  ows_Instructions="" />
```

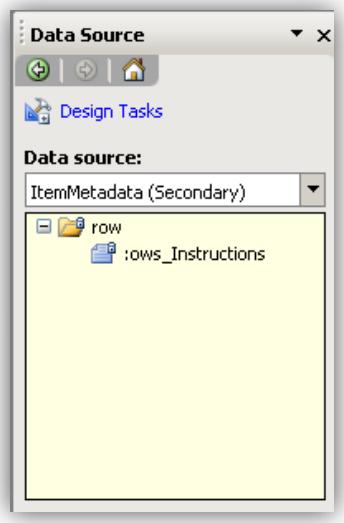
- Add a new data connection using the **ItemMetadata.xml** file.

  *In InfoPath, click **Tools -> Data Connections** in the menu bar.*
  *In the **Data Connections** dialog, click the **Add** button.*
  *Select the **Create a new connection to** option and then select **Receive data** and click **Next**.*
  *On the next page, choose to receive data from an **XML document**.*
  *Browse to the **ItemMetadata.xml** file just created and click **Next**.*
  *On the next page select to include the file as a resource and click **Next**.*
  *On the final page, click **Finish** to create the data source.*
  *In the **Data Connections** dialog, click **Close** to close the window.*



- Using the new data source, add a new control using the **ows_Instructions** field.

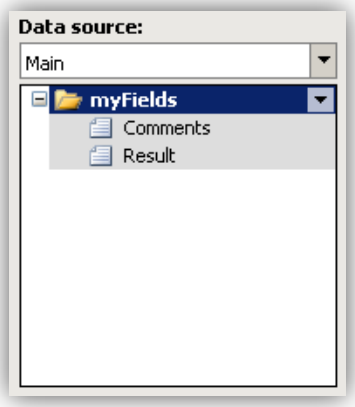
  *In the **Data Source** task pane, use the **Data source** drop down to change to the **ItemMetadata** data source.*
  *Drag the **ows_Instructions** field onto the designer to create an **Instructions** control.*



4) Define the output data that will be transmitted back to the workflow when the task is submitted.
   - Add a new **Comments** field of type string to the primary data source.

     *In the **Data Source** task pane, right click the **myFields** group and select **Add**.*
     *In the **Add Field or Group** dialog, enter a name of **Comments**.*
     *Verify the **Data type** is **string**.*
     *Click **OK** to create the new field.*
   - Add a new **Result** field of type string to the primary data source.

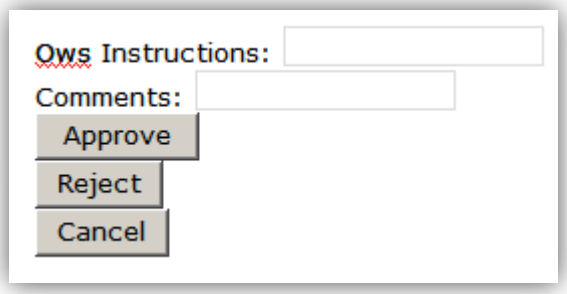
     *In the **Data Source** task pane, right click the **myFields** group and select **Add**.*
     *In the **Add Field or Group** dialog, enter a name of **Result**.*
     *Verify the **Data type** is **string**.*
     *Click **OK** to create the new field.*



   - Drag the **Comments** field onto the design canvas to create a new text box.

5) Add three buttons to the page for the **Approve**, **Reject**, and Cancel **operations**.
   - Switch to the **Controls** task pane and drag two button controls onto the design canvas.
   - Change the name of one button to **Approve**.

     *Right click the button and select **Button Properties**.*
     *In the **Button Properties** dialog, change the **Label** value to **Approve**.*
     *Click **OK** to close the dialog.*
   - Repeat the previous step for the **Reject** and **Cancel** buttons.



6) Define rules for the **Cancel** button that will cause it to close the form.
   - Open the **Rules** dialog to manage the rules that will execute when the button is clicked.

     *Right click the **cancel** button and select **Button Properties**.*
     *Click the **Rules** button to open the **Rules** dialog.*

- Add a new rule named **Close Form** that will be executed when the button is clicked.

  *Click the **Add** button to add a new rule.*
  *Change the **Name** property in the **Rule** dialog to **Close Form**.*

- Add a close form action that will close the current form.

  *Click **Add Action** to open the **Action** dialog.*
  *In the **Action** dialog, select an action of **Close the form***
  *Click **OK** to close the dialog.*
  Close all of the open dialogs.

7) Define rules for the **Approve** button that will cause it to submit and then close the form.

- Open the **Rules** dialog to manage the rules that will execute when the button is clicked.

  *Right click the cancel button and select **Button Properties**.*
  *Click the **Rules** button to open the Rules dialog.*

- Add a new rule named **Approve and Close Form** that will be executed when the button is clicked.

  *Click the **Add** button to add a new rule.*
  *Change the **Name** property in the **Rule** dialog to **Approve and Close Form**.*

- Add a **Set a field's value** action that will set the **Result** field to **Approved**.

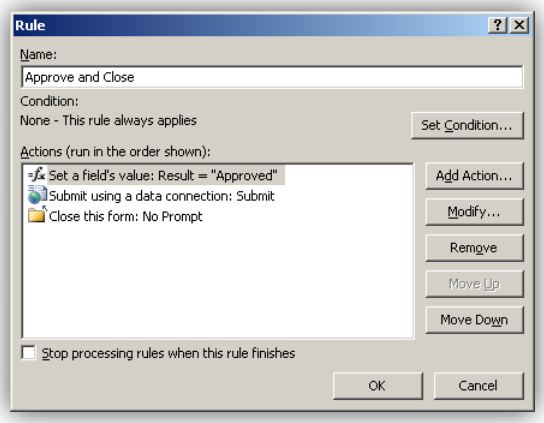
  *Click **Add Action** to open the **Action** dialog.*
  *In the **Action** dialog, select an action of **Set a field's value**.*
  *Click the button following the **Field** text box and choose the **Result** field.*
  *In the **Value** text box, enter **Approved** and click **OK** to create the action.*
  *Back in the **Action** dialog, verify the data connection **Submit** is selected and click **OK** to close the dialog.*

- Add a submit form action that will submit the data to Forms Services.

  *Click **Add Action** to open the **Action** dialog.*
  *In the **Action** dialog, select an action of **Submit using a data connection**.*
  *Click the **Add** button to add a new submission data connection.*
  *In the **Data Connection Wizard** create a new **Submit** data connection and click **Next**.*
  *Select **To the hosting environment** and click **Next**.*
  *Keep the default name of **Submit** and click **Finish**.*
  *Back in the **Action** dialog, verify the data connection **Submit** is selected and click **OK** to close the dialog.*

- Add a close form action that will close the current form.

  *Click **Add Action** to open the **Action** dialog.*
  *In the **Action** dialog, select an action of **Close the form***
  *Click **OK** to close the dialog.*

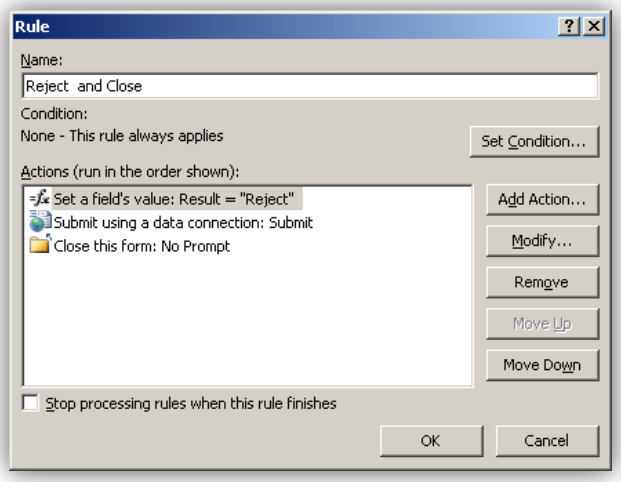


8) Repeat the previous step to define the rules for the **Rejected** button.

- Perform the previous steps with the following changes:

  *Use the rule name of **Reject and Close Form**.*

  *Set the **Result** field value to **Rejected**.*



9) Update the layout of the form to look nicer. Add a title of **Task Form** that is size 14pt, bold, and place the controls in a layout table.

- Using InfoPath like a word processor, add the text Asso**ciation Form** to the top of the page. Use a font size of 14pt and bold.
- Create a layout grid of 2x3 and place the instructions and comments controls in the top two rows. Adjust the height of the text boxes so they can contain multiple lines.
- Place the buttons in the lower right cell of the layout table and right justify them.

10) Save the new form template into the **Forms** folder of the Visual Studio Project.

- Click the **Save** toolbar button.
- Save the form in **C:\Labs\Lab08\StarterFiles\InfoPathApprovalWorkflow\Forms** with a name of **TaskForm.xsn**.

11) Update the workflow to support providing values to the task form and receiving results back.

- In **CreateApprovalTask_Invoking**, add an extended property named Instructions getting its value from the **AssocInitData** property.

  *Right click **Workflow.cs** in the **Solution Explorer** and click **View Code**.*

  *Locate the **CreateApprovalTask_Invoking** and add an Instructions property to the task properties.*

  *Get the value of the extended property from the **AssocInitData** property of the workflow.*

```
createApprovalTask.TaskProperties.ExtendedProperties.Add(
    "Instructions", InitData.Instructions);
```

- In **ApprovalTaskChanged_Invoked** retrieve the extended property with the Guid id of **{52578fc3-1f01-4f4d-b016-94ccbcf428cf}** representing the **Comments** column.

  *Locate the **ApprovalTaskChanged_Invoked** method.*

  Add the line to retrieve the comments extended property.

```
this.ApprovalTaskComments = args.afterProperties.ExtendedProperties[
    new Guid("52578fc3-1f01-4f4d-b016-94ccbcf428cf")] as string;
```

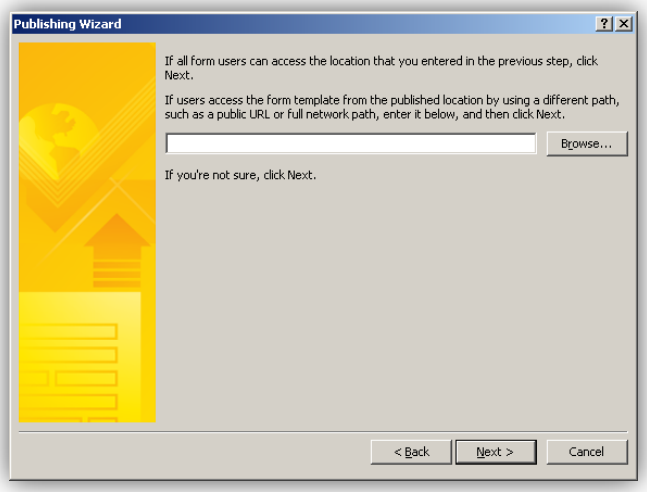
- In **ApprovalTaskChanged_Invoked** retrieve the **Result** expended property and convert it into a **TaskResult** enum value.

```
string result = args.afterProperties.ExtendedProperties["Result"] as string;
this.ApprovalTaskResult = (TaskResult)Enum.Parse(typeof(TaskResult), result);
```

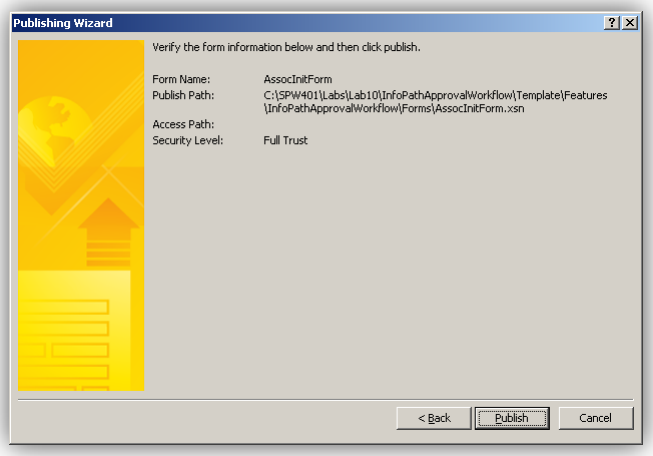
Exercise 4: Deploying InfoPath Workflow Forms

1) Publish the **AssocInitForm.xsn** form to the **Template\Features\InfoPathApprovalWorkflow\Forms** folder in the project.

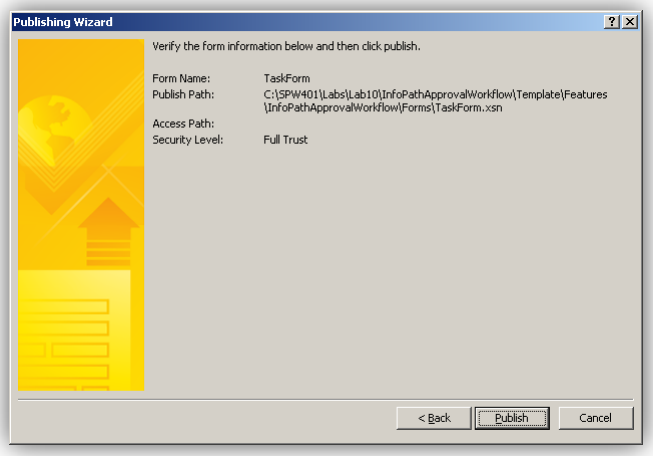
- Design the **Forms\AssocInitForm.xsn** form template in InfoPath.
- Publish the form by clicking **File -> Publish** in the menu bar.
- In the **Publishing Wizard**, choose to publish to a network location and click **Next**.
- Publish the document to the **C:\Labs\Lab08\StarterFiles\InfoPathApprovalWorkflow\Template\Features\InfoPathApprovalWorkflow\Forms\AssocInitForm.xsn** location.
- When asked for the public URL make sure the text box is empty and click **Next**.



- On the final page of the wizard, verify the results and click **Publish** and then **Close**.



2) Using the process in the previous step, publish the **Task.xsn** form to the **Template\Features\InfoPathApprovalWorkflow\Forms** folder in the project.



3) Update the **feature.xml** file to indicate the location of the InfoPath forms and include a receiver assembly that will install those forms to Forms Services.
   - Open the **Feature.xml** file in the **Template\Features\InfoPathApprovalWorkflow** folder.
   - Add a **ReceiverAssembly** and **ReceiverClass** to the **Feature** element.

```
ReceiverAssembly="Microsoft.Office.Workflow.Feature, Version=12.0.0.0,
Culture=neutral, PublicKeyToken=71e9bce111e9429c"
ReceiverClass="Microsoft.Office.Workflow.Feature.WorkflowFeatureReceiver"
```

   - Add a **Property** element to the **Properties** element that will set a property of **RegisterForms** to the **Forms*.xsn** value.

```
<Properties>
  <Property Key="GloballyAvailable" Value="true" />
  <Property Key="RegisterForms" Value="Forms\*.xsn" />
</Properties>
```

4) Update the **workflow.xml** file to define new form Urls and content types for tasks.

- Open the **Workflow.xml** file in the **Template\Features\InfoPathApprovalWorkflow** folder.
- Add an **AssociationUrl** and **InstantiationUrl** to the **Workflow** element.

  These values are special Forms Services hosting pages.

```
AssociationUrl="_layouts/CstWrkflIP.aspx"
InstantiationUrl="_layouts/IniWrkflIP.aspx">
```

- Add a TaskListContentTypeId attribute to the Workflow element.

  *This is a special **Forms Services** content type with the appropriate view and edit forms already defined.*

```
TaskListContentTypeId="0x01080100C9C9515DE4E24001905074F980F93160"
```

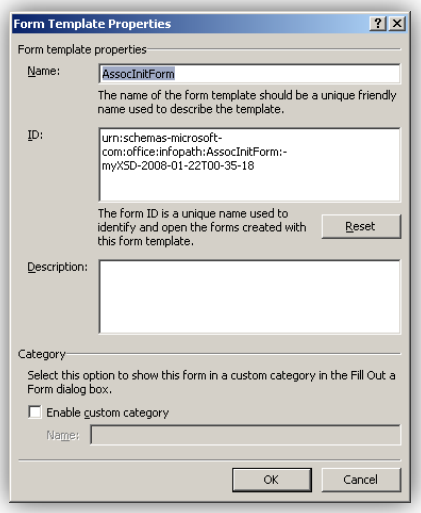
5) Add form registration URNs that are used by the special Forms Services pages to identify the InfoPath form to load.

- Lookup the URN of the **AssocInitForm.xsn** using the form's **Properties** window.

  *Right click the form in **Windows Explorer** and select **Design**.*

  *When the form is loaded, click **File -> Properties**.*

  *Copy the **ID** property from the dialog.*



- Use the URN for the form to populate the Association and Instantiation Form URNs in the workflow metadata.

  *Open **Workflow.xml** and add the following metadata into the form.*

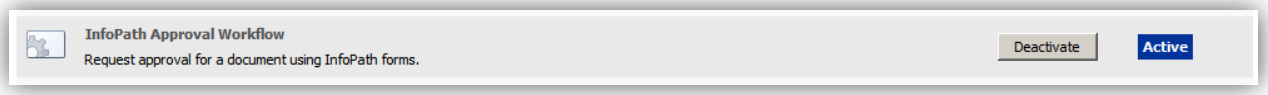
  *Replace the values in **Association_FormURN** and **Instantiation_FormURN** with the URN you copied from InfoPath.*

```
<MetaData>
  <Association_FormURN>urn:schemas-microsoft-com:office:infopath:AssocInitForm:-myXSD-2008-01-22T00-35-18</Association_FormURN>
  <Instantiation_FormURN>urn:schemas-microsoft-com:office:infopath:AssocInitForm:-myXSD-2008-01-22T00-35-18</Instantiation_FormURN>
  <Task0_FormURN>urn:schemas-microsoft-com:office:infopath:TaskForm:-myXSD-2008-01-22T03-21-33</Task0_FormURN>
</MetaData>
```

- Repeat the previous steps populate Task0_FormURN with the URN of **TaskForm.xsn**.

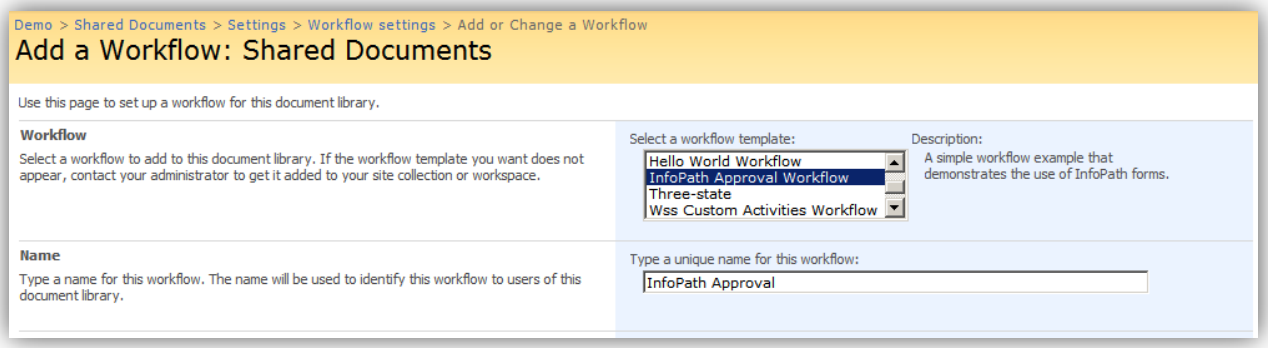
6) Rebuild the project to force redeployment of the forms.
   - Right click on the project in the **Solution Explorer** and click **Rebuild**.

7) Activate the new **InfoPathApprovalWorkflow** feature.
   - Using **Internet Explorer** navigate to the **Demo** site collection at **http://litwareinc.com/sites/Demo**.
   - Open the features list by clicking **Site Actions -> Site Settings**.
   - On the **Site Settings** page, click **Site collection features** in the **Site Collection Administration** section.
   - Click the **Activate** button next to the **InfoPath Approval Workflow** feature.



8) Create an association between the **Shared Documents** document library and the new **InfoPath Approval Workflow** workflow.
   - Navigate to the **Shared Documents** document library in the **Demos** site.
   - Click **Settings -> Document Library Settings** to load the settings page.
   - Click the **Workflow settings** link in the **Permissions and Management section**.
   - Create a new workflow using the **InfoPath Approval Workflow** workflow template and a name of **InfoPath Approval**.

     Use the default values for both list and startup options.



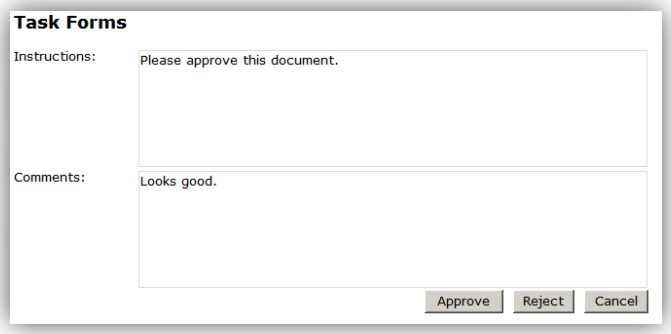
   - In the custom association form, enter Administrator as the approver and add “Please approve this document.” as the instructions.

9) Run the **InfoPath Approval** workflow on a document.

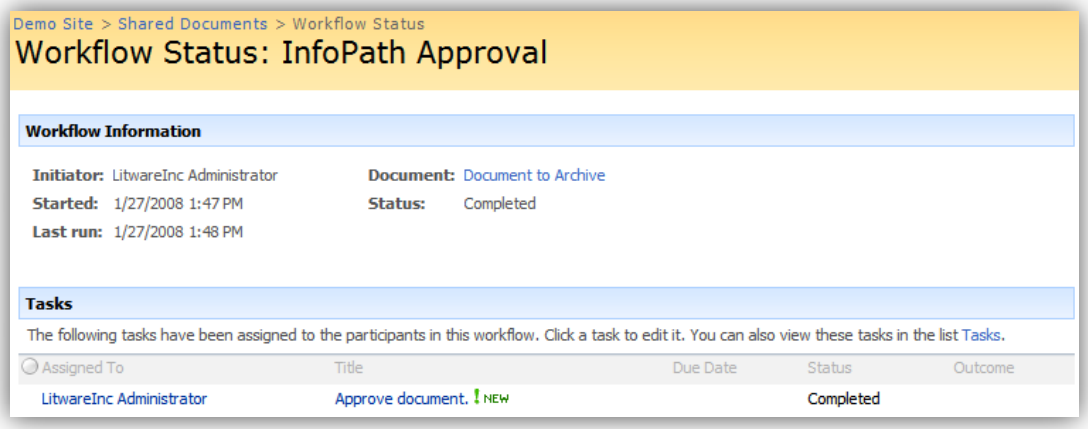
- Navigate to the Shared Documents document library and hover over the new document and select **Workflows** from the drop down menu.
- In the workflows page, click the **InfoPath Approval** to start the workflow.
- In the **Shared Documents** document library, verify the workflow is running.
- Click the **In Progress** link to view the workflow status and verify the started message was logged to the workflow's history.

10) Approve the document using the Custom Task Form.

- Click the task's title in the workflow status page.
- Notice that the **Instructions** text box defaults to the value set in the **Instantiation Form**.
- Click the **Approve** button to approve the task.



11) Verify that the workflow is now completed and the task is marked as completed.



Lab 09: Developing Custom Activities for a SharePoint Workflow Template

Lab Overview: Often there are pieces of code used many times in a solution. In a traditional .NET application, you encapsulate them as a class or a function. In Windows Workflow, you encapsulate them as Activities.

In this lab, you will be creating two custom activities that will provide database access. To do this you will need to create activities as well as validators and designers to ensure the activities are use properly.

Exercise 1: Building a simple Activity

1) Create a new **Workflow Activity Library** project to the solution named **SqlDatabaseActivities**.
   - Right click the solution in the **Solution Explorer** and click **Add -> New Project**.
   - Select the **Workflow** option in the **Project types** list box and select **Workflow Activity Library** in the Templates list.
   - Enter a name of **SqlDatabaseActivities** and click **OK**.

2) Create a new **Activity** named **SqlScalarQueryActivity** in the new project.
   - Right click on the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Add -> Activity**.
   - Enter a name of **SqlScalarQueryActivity.cs** and click **OK**.
   - Modify the class definition to change the base class to **Activity** instead of **SequenceActivity**.

```
public partial class SqlScalarQueryActivity : Activity
{
}
```

3) Define the dependency properties named **Parameters** and **Result**.
   - Define the static **DependencyProperty** object for **Parameters**.

     *Create a new **public static DependencyProperty** object named **ParametersProperty**.*

     *Set its value using the **DependencyProperty.Register** static method.*

     *Define its **name** as **Parameters**, it's **propertyType** as **typeof(SqlParameter[])**, and it's **ownerType** of **typeof(SqlScalarQueryActivity)**.*

```
public static DependencyProperty ParametersProperty =
    DependencyProperty.Register("Parameters",
        typeof(SqlParameter[]), typeof(SqlScalarQueryActivity));
```

   - Define a helper property that wraps access to the new dependency property.

     *Define a string **Parameters** property with a get and set seciton.*

     *Implement get using the **base.GetValue** method and casting the result to a string.*

     *Implement set using the **base.SetValue** method.*

     *Add a **Category** attribute to the property with a value of **Parameters** (this places the property in the **Parameters** group in the designer).*

```
[Category("Parameters")]
public SqlParameter[] Parameters
{
    get { return base.GetValue(ParametersProperty) as SqlParameter[]; }
    set { base.SetValue(ParametersProperty, value); }
}
```

- Repeat the process to create a **Result** dependency property and wrapping property definition. The only difference is the type. Make **Result** an object.

```
public static DependencyProperty ResultProperty =
   DependencyProperty.Register("Result",
       typeof(System.Object), typeof(SqlScalarQueryActivity));

[Category("Parameters")]
public object Result
{
    get { return base.GetValue(ResultProperty); }
    set { base.SetValue(ResultProperty, value); }
}
```

4) Define the **Query** dependency property. Make it a **MetaData** property so it can only be modified at design time.

- Define the static **DependencyProperty** object for **Query**.

  *Create the new **public static DependencyProperty** just like before.*

  *Add a fourth parameter to the call to **DependencyObject.Register** that is a **PropertyMetadata** object with an **options** parameter of **DependencyPropertyOptions.Metadata**.*

```
public static DependencyProperty QueryProperty =
    DependencyProperty.Register("Query",
        typeof(System.String), typeof(SqlScalarQueryActivity),
        new PropertyMetadata(DependencyPropertyOptions.Metadata));
```

- Define the wrapper property for **Query** the same as any other dependency property wrapper property.

```
[Category("Parameters")]
public string Query
{
    get { return base.GetValue(QueryProperty) as string; }
    set { base.SetValue(QueryProperty, value); }
}
```

5) Add two events to the activity, **Invoking** and **Invoked**, which can be bound using the same mechanism as the properties.

- Define the static **DependencyProperty** object for **Invoking**.

  *Create a new **public static DependencyProperty** object named **InvokingEvent**.*

  *Set its value using the **DependencyProperty.Register** static method.*

  *Define its **name** as **Invoked**, it's **propertyType** as **typeof(EventHandler)**, and it's **ownerType** of **typeof(SqlScalarQueryActivity)**.*

```
public static DependencyProperty InvokingEvent =
```

```
    DependencyProperty.Register("Invoking",
        typeof(EventHandler), typeof(SqlScalarQueryActivity));
```

- Define the wrapper event for **Invoking** that will provide a standard mechanism for attaching and removing event handlers.

  *Define an event named **Invoked** of type **EventHandler** with an add and remove seciton.*
  *Implement **add** using the **base.AddHandler** method.*
  *Implement **remove** using the **base.RemoveHandler** method.*
  *Add a **Category** attribute to the property with a value of **Handlers**.*

```
[Category("Handlers")]
public event EventHandler Invoking
{
    add { base.AddHandler(InvokingEvent, value); }
    remove { base.RemoveHandler(InvokingEvent, value); }
}
```

- Define a second dependency property and event for the **Invoked** event.

```
public static DependencyProperty InvokedEvent =
    DependencyProperty.Register("Invoked",
        typeof(EventHandler), typeof(SqlScalarQueryActivity));

[Category("Handlers")]
public event EventHandler Invoked
{
    add { base.AddHandler(InvokedEvent, value); }
    remove { base.RemoveHandler(InvokedEvent, value); }
}
```

6) Implement the **Execute** method so the query is made and the result is stored in the **Result** property.

- Override the **Execute** method of the **Activity** class.

```
protected override ActivityExecutionStatus Execute(ActivityExecutionContext
executionContext)
{
}
```

- Raise the **Invoking** event using the **Activity.RaiseEvent** method.

```
// raise the invoking event
this.RaiseEvent(InvokingEvent, this, EventArgs.Empty);
```

- Open the **SqlConnection** using a hardcoded connection string (this will be changed later)

```
using (SqlConnection connection = new SqlConnection("Data
Source=.\\SQLEXPRESS;AttachDbFilename=C:\\Labs\\Files\\Litware.mdf;Integrated
Security=True;Connect Timeout=30;User Instance=True"))
{
    connection.Open();
}
```

- While the connection is open, create and execute the new **SqlCommand** based on the **Query** and **Parameters** dependency properties.

```
// create the sql command
SqlCommand command = new SqlCommand(this.Query, connection);
if (this.Parameters != null)
    foreach (SqlParameter parameter in this.Parameters)
        command.Parameters.Add((parameter as ICloneable).Clone() as
SqlParameter);
this.Result = command.ExecuteScalar();
```

- After the **SqlConnection's** using scope, raise the **Invoked** event to indicate the activity is completed.

```
// raise the Invoked event
this.RaiseEvent(InvokedEvent, this, EventArgs.Empty);
```

- Return an **ActivityExecutionStatus** of **Closed** to let the executor know this activity is completed and it should move on to the next activity.

```
// return the closed activity status
return ActivityExecutionStatus.Closed;
```

7) Build the **SqlDatabaseActivities** project.

- Right click the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Build**.

8) Create a new **Sequential Workflow Console Application** project to the solution named **SqlDatabaseActivitiesConsole**.

- Click **File -> New Project** in the menu bar.
- Select the **Workflow** option in the **Project types** list box and select **Sequential Workflow Console Application** in the **Templates** list.
- Select **Add to Solution** in the **Solution** drop down list.
- Enter a name of **SqlDatabaseActivitiesConsole** and click **OK**.

9) Add a new **SqlScalarQueryActivity** to **Workflow1** and use it to make a database query.

- View **Workflow1.cs** in design mode by right clicking it in the **Solution Explorer** and selecting **View Designer**.
- Drag a new **SqlScalarQueryActivity** onto the canvas and name it **LookupComments**.

  *Drag a **SqlScalarQueryActivity** onto the canvas.*

  *In the properties pane, set the **Name** property to **LookupComments**.*
- Define the query as a lookup into the **UserProfiles** table.

  *In the properties pane, set the **Query** to the following query.*

```
SELECT Comments FROM UserProfiles WHERE Email = @email
```

- Define an **Invoking** method that will be used to define the **@email** parameter.

  *In the properties pane, set the **Invoking** property to **LookupComments_Invoking** and press **Enter**.*

  *In the **LookupComments_Invoking** method, cast the sender parameter to a **SqlScalarActivityQuery**.*

  *Set the activities **Parameters** property to an array containing a single **SqlParameter** named **@email** with a value of **Administrator@litwareinc.com**.*

```
private void LookupComments_Invoking(object sender, EventArgs e)
```

```
{
    SqlScalarQueryActivity activity = sender as SqlScalarQueryActivity;

    activity.Parameters = new SqlParameter[]
    {
        new SqlParameter("@email", "Administrator@litwareinc.com")
    };
}
```

- Define an **Invoked** event handler that will be used to display the activity's Result property to the console.

  *Back in the designer's properties pane, set the **Invoked** property to **LookupComments_Invoked** and press **Enter**.*
  *In the **LookupComments_Invoked** method, cast the sender parameter to a **SqlScalarActivityQuery**.*
  *Use **Console.WriteLine** to display the activity's **Result** property to the console.*

```
private void LookupComments_Invoked(object sender, EventArgs e)
{
    SqlScalarQueryActivity activity = sender as SqlScalarQueryActivity;

    Console.WriteLine("Result - {0}", activity.Result);
}
```

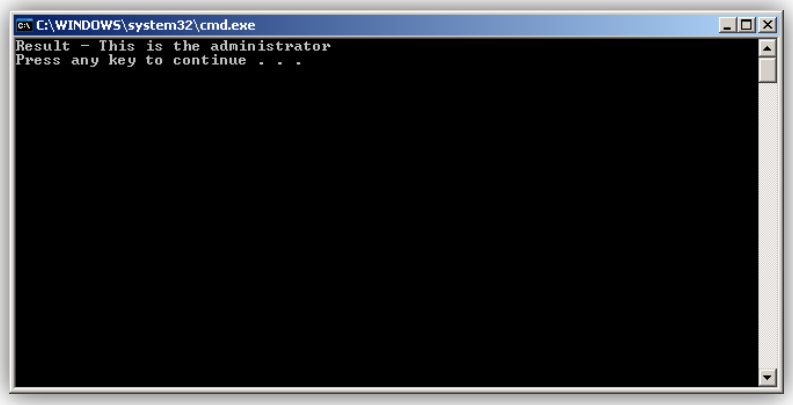
10) Run the workflow and verify that the output displays the appropriate result.

- Verify the **SqlDatabaseActivitiesConsole** is bold in the **Solution Explorer**.

  *If not, right click it and select **Set as Startup Project**.*

- Build and execute the workflow.

  *Click **Debug -> Start Without Debugging**.*



Exercise 2: Creating a Composite Activity

1) Create a new **Activity** named **SqlConnectionActivity** in the **SqlDatabaseActivities** project.
   - Right click on the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Add -> Activity**.
   - Enter a name of **SqlConnectionActivity.cs** and click **OK**.

2) Define the **SqlConection** property that will be used to expose the connection to other components.
   - Define a property with a public getter and private setter.

```
public SqlConnection Connection { get; private set; }
```

   - Add a **Browsable** attribute to the property so it does not show up in the designer.

```
[Browsable(false)]
public SqlConnection Connection { get; private set; }
```

3) Define the dependency property named **ConnectionString** of type **string**.

```
public static DependencyProperty ConnectionStringProperty =
    DependencyProperty.Register("ConnectionString",
        typeof(System.String), typeof(SqlConnectionActivity));

[Category("Parameters")]
public string ConnectionString
{
    get { return base.GetValue(ConnectionStringProperty) as string; }
    set { base.SetValue(ConnectionStringProperty, value); }
}
```

4) Implement the execute method to start the processing of the child activities.
   - Override the **Execute** method of the **Activity** class.

```
protected override ActivityExecutionStatus Execute(ActivityExecutionContext
executionContext)
{
}
```

   - If no child activities exist that are enabled, return with an **ActivityExecutionStatus** of **Closed**.

```
// check if there's any activities to execute
if (base.EnabledActivities.Count == 0)
    return ActivityExecutionStatus.Closed;
```

   - Create and open a new **SqlConnection** using the **ConnectionString** dependency property.

```
// open the database connection
if (this.Connection == null)
{
    this.Connection = new SqlConnection(this.ConnectionString);
    this.Connection.Open();
}
```

   - Find the first activity in the list of enabled activities, start it and register for status change message from that activity.

*Access and store the first item in the **base.EnabledActivities** collection*

*Use the **Activity.RegisterForStatusChange** method to register for the **ClosedEvent**.*

*Use the **ActivityExecutionContext.ExecuteActivity** to start the first activity executing.*

```
// start the first child activity and register for it's state change event
Activity firstEnabledActivity = base.EnabledActivities[0];
firstEnabledActivity.RegisterForStatusChange(Activity.ClosedEvent, this);
executionContext.ExecuteActivity(firstEnabledActivity);
```

- Return an **ActivityExecutionStatus** of **Executing** to notify the runtime that this activity is still running.

```
// tell the runtime this activity is still executing
return ActivityExecutionStatus.Executing;
```

5) Implement the **IActivityEventListener<ActivityExecutionStatusChangedEventArgs>** interface so the activity can receive activity events from its child activities.

- Modify the class to derive from **IActivityEventListener<ActivityExecutionStatusChangedEventArgs>**.

```
public partial class SqlConnectionActivity: SequenceActivity,
    IActivityEventListener<ActivityExecutionStatusChangedEventArgs>
```

- Implement the **OnEvent** method to handle the receipt of a status change event from a child activity.

```
public void OnEvent(object sender, ActivityExecutionStatusChangedEventArgs e)
{ }
```

- Validate that the sender parameter of the **OnEvent** method is an **ActivityExecutionContext**.

```
// validate the sender
ActivityExecutionContext executionContext = sender as ActivityExecutionContext;
if (executionContext == null)
    throw new ArgumentException("sender");
```

- Using the **Activity.UnregisterForStatusChange** method, cancel the subscription to events from the activity that has closed.

```
// unregister the handler for status events
e.Activity.UnregisterForStatusChange(Activity.ClosedEvent, this);
```

- If the current activity's **ExecutionStatus** property indicates a **Canceling** or **Faulting** state, close the activity.

```
// if the activity is cancelling or faulting, cleanup
if ((this.ExecutionStatus == ActivityExecutionStatus.Canceling) ||
(this.ExecutionStatus == ActivityExecutionStatus.Faulting)) {
    executionContext.CloseActivity();
}
```

- If the current activity's **ExecutionStatus** property indicates that the activity is in the **Executing** state, find the next activity and execute it.

*Use the **FindNextChild** method to lookup the next child activity to be executed (this will be implemented in the next step.*

If another activity was found, register to the closed event and tell the executor to execute the activity.

*If no more activities were found, tell the execution context to close the **SqlConnectionActivity**.*

```
// if the activity is executing, schedule the next child
else if (this.ExecutionStatus == ActivityExecutionStatus.Executing) {
    // find the next activity in the sequence
    Activity nextActivity = FindNextChild();
    if (nextActivity != null)  {
        // register and start the next activity
        nextActivity.RegisterForStatusChange(Activity.ClosedEvent, this);
        executionContext.ExecuteActivity(nextActivity);
    }
    else  {
        // close this activity
        executionContext.CloseActivity();
    }
}
```

6) Create a **FindNextActivity** method that will find the next child activity to execute.

- Define a private method named **FindNextChild** that returns an **Activity** object.

```
private Activity FindNextChild()
{
}
```

- Check if there are any enabled activities in the list, if not return null.

```
// check if any activities are in the list
if (base.EnabledActivities.Count == 0)
    return null;
```

- Find the last activity in the list that has a status of closed.

```
// find the last activity in the list that is closed
Activity lastClosedActivity = base.EnabledActivities.LastOrDefault(
    n => n.ExecutionStatus == ActivityExecutionStatus.Closed);
```

- If the last closed activity is the last activity in the list, return **null**.

```
// if this is the last closed activity, return false
int indexOfLastClosedActivity =
    base.EnabledActivities.IndexOf(lastClosedActivity);
if (indexOfLastClosedActivity == base.EnabledActivities.Count - 1)
    return null;
```

- Return the activity following the last closed activity.

```
// return try to indicate a task was found
return base.EnabledActivities[indexOfLastClosedActivity + 1];
```

7) Handle the activity's **OnClosed** method to make sure the **SqlConnection** is closed.

- Override the **OnClosed** method of the **Activity** class.

```
protected override void OnClosed(IServiceProvider provider)
```

```
{
}
```

- If the connection is not null, close it (if necessary) and set it to null.

```
// close the database connection
if (this.Connection != null)
{
    if (this.Connection.State != System.Data.ConnectionState.Closed)
        this.Connection.Close();
    this.Connection = null;
}
```

- Call the base class's implementation.

```
// call the base implementation
base.OnClosed(provider);
```

8) Update the **SqlScalarQueryActivity** to use the parent **SqlConnectionActivity's SqlConnection** instead of creating its own.

- Open the code for **SqlScalarQueryActivity** by right clicking it in the **Solution Explorer** and selecting **View Code**.
- Add a private method named **FindParentConnection** that accepts searches up the activity tree and finds the first **SqlConnectionActivity**.

  *Use the **Activity.Parent** property to find the parent.*
  *If the parent is a **SqlConnectionActivity**, return it.*
  *Set the **activity** variable to its parent, and keep looking.*

```
private SqlConnectionActivity FindParentConnection(Activity current)
{
    while (current.Parent != null)
    {
        if (current.Parent is SqlConnectionActivity)
            return current.Parent as SqlConnectionActivity;
        current = current.Parent;
    }
    return null;
}
```

- In the **Execute** method, remove all code that creates or opens the **SqlConnection** object.
- Immediately following the call where the **InvokingEvent** is raised, add the code to find the parent **SqlConnectionActivity** using the **FindParentConnection**.

```
// find the parent SqlConnectionActivity
SqlConnectionActivity connectionActivity = FindParentConnection(this);
if (connectionActivity == null)
    throw new InvalidOperationException("No parent connection was found.");
```

- Using the parent **SqlConnectionActivity's** Connection property, store the **SqlConnection** in a variable named connection.

```
// store the connection for later
```

```
SqlConnection connection = connectionActivity.Connection;
```

9) Build the **SqlDatabaseActivities** project.

- Right click the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Build**.

10) Add a new **SqlConnectionActivity** to **Workflow1** and move the **SqlScalarQueryAction** into it.

- View **Workflow1.cs** in design mode by right clicking it in the **Solution Explorer** and selecting **View Designer**.
- Drag a new **SqlConnectionActivity** onto the canvas and name it **SqlConnection**.

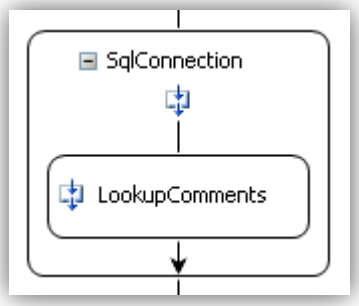
  *Drag a **SqlConnectionActivity** onto the canvas.*

  *In the properties pane, set the **Name** property to **SqlConnection**.*
- Define the connection string property as a lookup into the **UserProfiles** table.

  *In the properties pane, set the **ConnectionString** property to the following value.*

```
Data Source=.\SQLEXPRESS;AttachDbFilename=C:\Labs\Files\Litware.mdf;Integrated
Security=True;Connect Timeout=30;User Instance=True
```

- Drag the existing **SqlScalarQueryActivity** into the new **SqlConnection** activity.

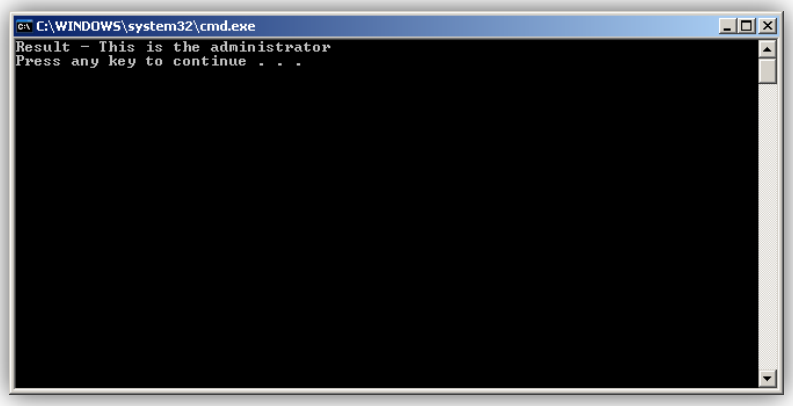


11) Run the workflow and verify that the output displays the appropriate result.

- Verify the **SqlDatabaseActivitiesConsole** is bold in the **Solution Explorer**.

  *If not, right click it and select **Set as Startup Project**.*
- Build and execute the workflow. The result should be the same as before, but now the connection is shared.

  *Click **Debug -> Start Without Debugging**.*



Exercise 3: Adding Validation to the Activities

1) Create a new **class** named **SqlScalarQueryActivityValidator** in the new project.
   - Right click on the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Add -> Class**.
   - Enter a name of **SqlScalarQueryActivityValidator.cs** and click **OK**.
   - Modify the class definition so it is **public** and derives from **ActivityValidator**.

```
public class SqlScalarQueryActivityValidator : ActivityValidator
{
}
```

   - Open the code for the **SqlScalarQueryActivity** and attach the **Validator** attribute defining the activity's validator.

     *Open the code for **SqlScalarQueryActivity.cs** by right clicking it in the **Solution Explorer** and clicking **View Code**.*

     *Attach the **Validator** attribute to the **SqlScalarQueryActivity** class using the type of the new. **SqlScalarQueryActivityValidator** class as the parameter.*

```
[ActivityValidator(typeof(SqlScalarQueryActivityValidator))]
public partial class SqlScalarQueryActivity : Activity
```

2) Validate that the **Query** property is not **null** or an empty string.
   - Override the **Validate** method from the **ActivityValidator** class.

```
public override ValidationErrorCollection Validate(ValidationManager manager,
object obj)
{ }
```

   - Cast the **obj** parameter to a **SqlScalarQueryActivity** object and store it for later.

```
SqlScalarQueryActivity activity = obj as SqlScalarQueryActivity;
```

   - Run the base validator and store any errors in a new errors variable of type **ValidationErrorCollection**.

```
// build the errors collection using the base validation method
ValidationErrorCollection errors = base.Validate(manager, obj);
```

   - If the validator is running in a real environment (not as part of the activity's designer), validate that the **Query** property is not null or empty.

     *Check if the **activity.Parent** attribute is not null. If it isn't, continue the validation.*

     *Check if the **activity.Query** property is null or empty using **string.IsNullOrEmpty** method.*

     *If the query property is empty, create a new validation error using the **ValidationError.GetNotSetValidationError** method and add it to the **error** collection.*

```
// only perform validation if in design or runtime mode
if (activity.Parent != null) {
    // validate that a query was defined
    if (string.IsNullOrEmpty(activity.Query))
        errors.Add(ValidationError.GetNotSetValidationError("Query"));
}
```

   - Return the collection of errors.

```
// return the errors
return errors;
```

3) Verify the **SqlScalarQueryActivity** has a parent **SqlConnectionActivity**.

- Create a new method named **HasSqlConnectionParent** that looks through an activity's parents to find an object of a specific type.

  *Use the **Activity.Parent** property to find the parent.*
  *If the parent is a **SqlConnectionActivity**, return **true**.*
  Set the activity variable to its parent, and keep looking.

```
private bool HasSqlConnectionParent(Activity current) {
    while (current.Parent != null) {
        if (current.Parent is SqlConnectionActivity)
            return true;
        current = current.Parent;
    }
    return false;
}
```

- In the **Validate** method, immediately after validating the **Query** parameter, check if the activity has a **SqlConnectionActivity** as a parent.

```
// validate the activity has an ancestor that is a SqlConnectionActivity
if (!HasSqlConnectionParent(activity))
    errors.Add(new ValidationError(
        "SqlScalarQueryActivities must be palaced in a SqlConnectionActivity",
        100, false));
```

4) Create a new class named **SqlConnectionActivityValidator** in the **SqlDatabaseActivities** project.

- Right click on the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Add -> Class**.
- Enter a name of **SqlConnectionActivityValidator.cs** and click **OK**.
- Modify the class definition so it derives from **ActivityValidator**.

```
public class SqlConnectionActivityValidator : CompositeActivityValidator
{ }
```

- Open the code for the **SqlConnectionActivity** and attach the **Validator** attribute.

  *Open the code for **SqlConnectionActivity.cs** by right clicking it in the **Solution Explorer** and clicking **View Code**.*
  *Attach the **Validator** attribute to the **SqlConnectionActivity** class using the type of the new **SqlConnectionActivityValidator** class as the parameter.*

```
[ActivityValidator(typeof(SqlConnectionActivityValidator))]
public partial class SqlConnectionActivity : CompositeActivity
```

5) Validate that the **ConnectionString** property is not null or an empty string.

- Override the **Validate** method from the **ActivityValidator** class.

```
public override ValidationErrorCollection Validate(ValidationManager manager,
object obj)
{
}
```

- Cast the **obj** parameter to a **SqlScalarQueryActivity** object and store it for later.

```
SqlConnectionActivity activity = obj as SqlConnectionActivity;
```

- Run the base validator and store any errors in a new errors variable of type **ValidationErrorCollection**.

```
// build the errors collection using the base validation method
ValidationErrorCollection errors = base.Validate(manager, obj);
```

- If the validator is running in a real environment (not as part of the activity's designer), validate that the **ConnectionString** property is not null or empty.

  *Check if the **activity.Parent** attribute is not null. If it isn't, continue the validation.*

  *Check if the **activity.ConnectionString** property is null or empty using **string.IsNullOrEmpty** method.*

  *If the connection stringproperty is empty, create a new validation error using the **ValidationError.GetNotSetValidationError** method and add it to the **error** collection.*

```
// only perform validation if in design or runtime mode
if (activity.Parent != null)
{
    // validate that a connection string was defined
    if (string.IsNullOrEmpty(activity.ConnectionString))
    errors.Add(ValidationError.GetNotSetValidationError("ConnectionString"));
}
```

- Return the collection of errors.

```
// return the errors
return errors;
```

6) Verify the **SqlConnectionActivity** does not have any child activities that derive from **IEventHandler**.

- Create a new method named **HasChildIEventActivity** that looks through a composite activity's child activities and checks if any derive from **IEventHandler**.

```
private bool HasChildIEventActivity(CompositeActivity activity)
{
}
```

- Loop through each enabled child activity and check if the activity derives from **IEventHandler**. If so, return **true**.
- Check each child activity to determine if it is a **CompositeActivity**. If so, call **HasChildIEventActivity** and return true if it returns **true**.

```
// loop through each child activity that is enabled
foreach (Activity childActivity in activity.EnabledActivities) {
    // if the child activity is an IEventActivity, log an error
    if (childActivity is IEventActivity)
        return true;

    // if the child activity is a composite activity, check its children
    CompositeActivity compositeActivity = childActivity as CompositeActivity;
    if (compositeActivity != null)
        if (HasChildIEventActivity(compositeActivity))
            return true;
}
```

- I no child activities are **IEventActivities**, return **false**.

```
// no child activities are IEventActivity
```

```
return false;
```

- In the **Validate** method, immediately after validating the **ConnectionString** parameter, check if the activity has a **SqlConnectionActivity** as a parent.

```
// validate the activity has has no child IEventActivity activities
if (HasChildIEventActivity(activity))
    errors.Add(new ValidationError(
        "SqlConnectionActivities cannot contain IEventActivity activities. ",
        100, false));
```

7) Build the **SqlDatabaseActivities** project.

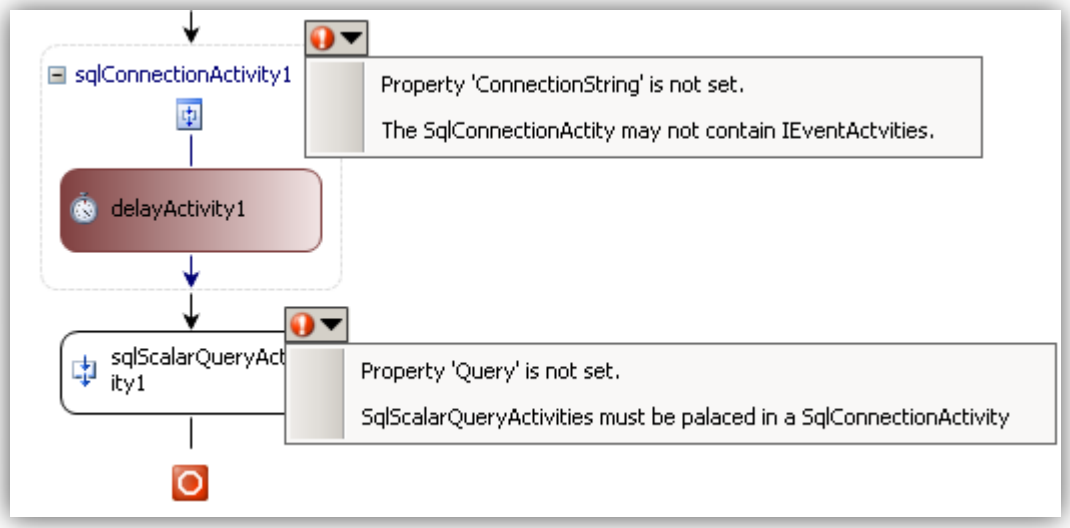
- Right click the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Build**.

8) Test the validators by trying to create activities that are missing key properties.

- Open **Workfow1.cs** in design mode by right clicking it and clicking **View Designer**.
- Drag a new **SqlConnectionActivity** onto the canvas.
- Drag a new **Delay** activity into the new **SqlConnectionActivity**.
- Drag a new **SqlScalarQueryActivity** onto the canvas.
- Verify each of the validators has fired.

  *The **SqlConnectionActivity** should have an error indicating an empty **ConnectionString** and containing an **IEventActiity**.*

  *The **SqlScalarQueryActivity** should have an error indicating an empty query and no parent **SqlConnectionActivity**.*



- Clean up by deleting the activities just added.

Exercise 4: Adding custom designers to the Activities

1) Create a new class named **SqlScalarQueryActivityDesigner** in the **SqlDatabaseActivities** project.

- Right click on the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Add -> Class**.
- Enter a name of **SqlScalarQueryActivityDesigner.cs** and click **OK**.
- Modify the class definition so it is **public** and derives from **ActivityDesigner**.

```
public class SqlScalarQueryActivityDesigner : ActivityDesigner
{
}
```

- Open the code for the **SqlScalarQueryActivity** and attach the **Designer** attribute defining the activity's designer.

  *Open the code for **SqlScalarQueryActivity.cs** by right clicking it in the **Solution Explorer** and clicking **View Code**.*

  *Attach the **Designer** attribute to the **SqlScalarQueryActivity** class using the type of the new **SqlScalarQueryActivityDesigner** class as the parameter.*

```
[Designer(typeof(SqlScalarQueryActivityDesigner))]
public partial class SqlScalarQueryActivity : Activity
```

2) Restrict **SqlScalarQueryActivities** from being placed on the canvas unless it is in a **SqlConnectionActivity**.

- Create a new method named **HasSqlConnectionParent** that looks through an activity's parents to find an object of a specific type.

  *Use the **Activity.Parent** property to find the parent.*

  *If the parent is a **SqlConnectionActivity**, return **true**.*

  *Set the **activity** variable to its parent, and keep looking.*

```
private bool HasSqlConnectionParent(Activity current)
{
    while (current.Parent != null)
    {
        if (current.Parent is SqlConnectionActivity)
            return true;
        current = current.Parent;
    }
    return false;
}
```

- Override the **CanBeParentedTo** method to restrict the activities that this activity can be contained in.

  *Define the **CanBeParentedTo** method.*

  *If the **Activity** property in the parent designer object is of type **SqlConnectionActivity**, return **true**.*

  *If not, check if any of the parent objects are of type **SqlConnectionActivity**.*

```
public override bool CanBeParentedTo(CompositeActivityDesigner
parentActivityDesigner)
{
    // if the parent activity is a SqlConnectionActivity
    if (parentActivityDesigner.Activity is SqlConnectionActivity)
        return true;

    // if not, check if a parent is a SqlConnectionActivity
    return HasSqlConnectionParent(parentActivityDesigner.Activity);
}
```

3) Create a new class named **SqlConnectionActivityDesigner** in the **SqlDatabaseActivities** project.

- Right click on the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Add -> Class**.

- Enter a name of **SqlConnectionActivityDesigner.cs** and click **OK**.
- Modify the class definition so it is **public** and derives from **SequentialActivityDesigner**.

```
public class SqlConnectionActivityDesigner : SequentialActivityDesigner
{
}
```

- Open the code for the **SqlDesignerActivity** and attach the **Designer** attribute defining the activity's designer.

  *Open the code for **SqlConnectionActivity.cs** by right clicking it in the **Solution Explorer** and clicking **View Code**.*

  *Attach the **Designer** attribute to the **SqlConnectionActivity** class using the type of the new **SqlConnectionActivityDesigner** class as the parameter.*

```
[Designer(typeof(SqlConnectionActivityDesigner))]
public partial class SqlConnectionActivity : SequenceActivity
```

4) Restrict the activities that can be placed in the **SqlConnectionActivity**.
   - Override the **CanInsertActivities** method in the **SqlConnectionActivityDesigner**.
   - If any of the activities in **activitiesToInsert** parameter derive from **IEventActivity**, return **false**.

```
public override bool CanInsertActivities(HitTestInfo insertLocation,
ReadOnlyCollection<Activity> activitiesToInsert)
{
    // check if any activities are of type IEventActivity
    return !activitiesToInsert.Any(n => n is IEventActivity);
}
```

5) Add a new verb to the designer that will allow workflow developers to right click and add a new **SqlScalarQueryActivity** to the **SqlConnectionActivity**.
   - Create a private event handler named **AddScalarQuery_Click**.

```
private void AddScalarQuery_Click(object sender, EventArgs e) { }
```

   - Cast the designer's activity to a **CompositeActivity** variable for later.

```
//Access the designer's activity using the base classes Activity property.
CompositeActivity activity = base.Activity as CompositeActivity;
```

   - Create a list of new Activity objects and create a new **SqlScalarQueryActivity** in the list.

```
// create the list of new activites
List<Activity> activities = new List<Activity>();
activities.Add(new SqlScalarQueryActivity());
```

   - Create a **ConnectorHitTestInfo** object identifying where to add the activity. In this case add it as the last activity in the **SqlConnectionActivity**.

```
// determine where to add the activities
ConnectorHitTestInfo location =
    new ConnectorHitTestInfo(this,
        HitTestLocations.Designer, activity.Activities.Count);
```

   - Add the child activities using the base class's **InsertActivities** method.

```
// add the activities to the designer
base.InsertActivities(location, activities.AsReadOnly());
```

- Override the **Verbs** property.

```
protected override ActivityDesignerVerbCollection Verbs { get {} }
```

- Create a new **ActivityDesignerVerbCollection** and populate it by calling the base class's implementation.

```
// load the verbs from the base class
ActivityDesignerVerbCollection verbs = new ActivityDesignerVerbCollection();
verbs.AddRange(base.Verbs);
```

- Add the new verb named **Add Scalar Query** the calls the **AddScalarQuery_Click**.

```
// add the new verb
verbs.Add(
    new ActivityDesignerVerb(this,DesignerVerbGroup.Actions, "Add Scalar Query",
        new EventHandler(AddScalarQuery_Click)));
```

- Return the new list of verbs.

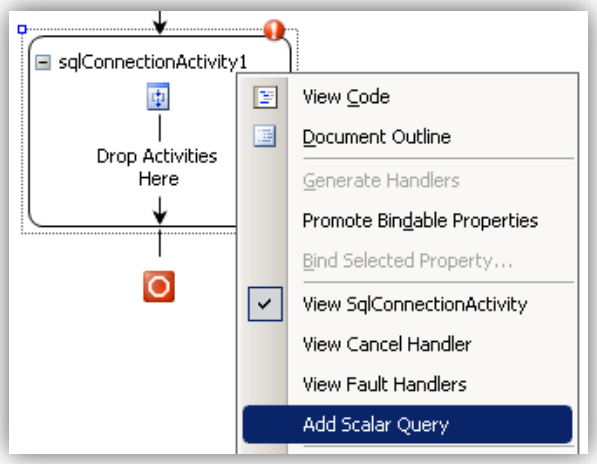
```
// return the list of verbs
return verbs;
```

6) Build the **SqlDatabaseActivities** project.

- Right click the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Build**.

7) Test the designers by trying to place activities in positions that are not valid

- Open **Workfow1.cs** in design mode by right clicking it and clicking **View Designer**.
- Drag a new **SqlConnectionActivity** onto the canvas.
- Try to drag a new **Delay** activity into the new **SqlConnectionActivity** and verify it will not work.
- Try to drag a new **SqlScalarQueryActivity** outside of the **SqlConnectionActivity** and verify it will not work.
- Right click on the **SqlConnectionActivity** and select **Add Scalar Query** to add a new **SqlScalarQueryActivity** to the **SqlConnectionActivity**.



Challenge: Designer Themes and Toolbox Items

Add a designer theme to the **SqlConnectionActivityDesigner** that changes the background color of the **SqlConnectionActivity**. To do this you will need to create a class derived from

CompositeDesignerTheme and set the theme's properties in the constructor. Once the theme is complete, attach it to the designer using the **ActivityDesignerTheme** attribute.

Add a toolbox item to the **SqlConnectionActivity** that adds a **SqlConnectionActivity** and an embedded **SqlScalarQueryActivity** when a **SqlConnectionActivity** is added to the canvas. To do this you will need to create a class derived from **ActivityToolboxItem** and override **CreateComponentsCore**. Once the toolbox item is complete, attach it to the activity using the **ToolboxItem** attribute.

Lab 10: Extending SharePoint Designer with Custom Activities

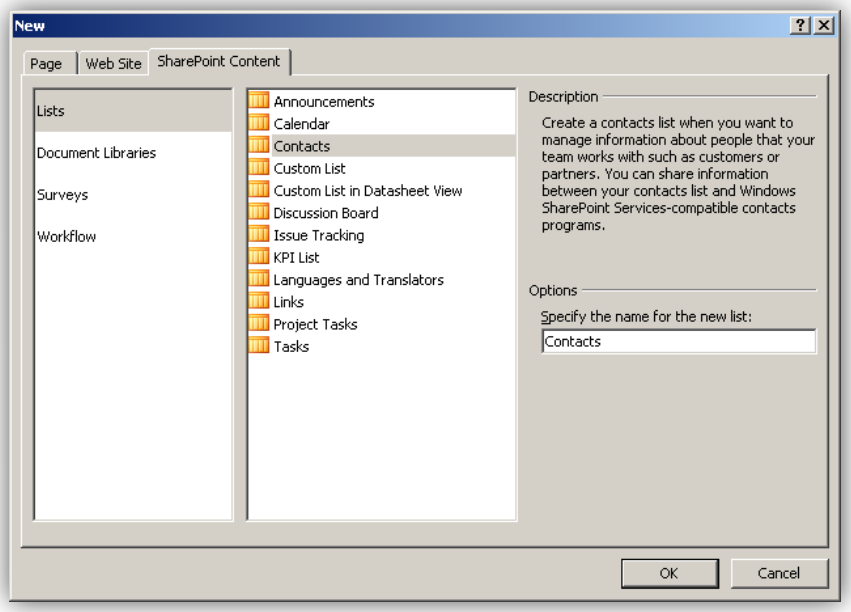
Lab Overview: Litware Inc has a need to integrate contact information that has currently been stored in a database into a **Contacts** list on a SharePoint site. A two-step process is designed. First, it checks if a new or updated contact has an email address and then looks up notes based on that email address. In the event no value exists for the email address, a user is assigned a task to provide them. The notes are then assigned to the Contact in its notes field.

This solution is fluid enough that the design team does not want a developer to manage it. Using the SharePoint designer is the best choice as it allows each department to manage their needs. To do this, a developer will need to create some custom conditions and actions that allow the IT departments to design the workflows themselves using the SharePoint Designer.

Exercise 1: Creating a simple workflow in SharePoint Designer

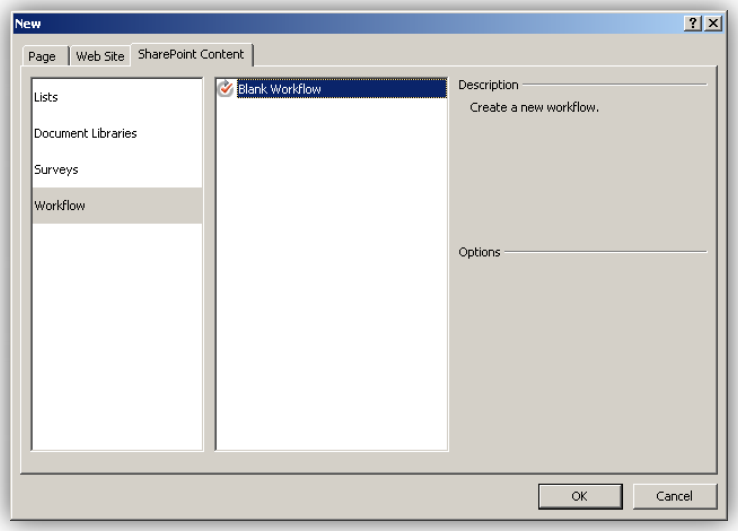
1) Start the **SharePoint Designer** on open the site at **http://litwareinc.com/sites/Demo**.
   - Start the **SharePoint Designer** by clicking the **Start Button** and selecting **All Programs -> Microsoft Office -> Microsoft Office SharePoint Designer 2007**.
   - Open the **Demos** site collection by clicking **File -> Open Site** and entering a site name of **http://litwareinc.com/sites/Demo**.

2) Create a new list in the site named **Contacts** using list type **Contacts**.
   - In the **Folder List** on the left hand side, right click the site at the top of the tree and click **New -> SharePoint Content.**
   - In the **New** dialog, select **Lists** in the list box on the left and select **Contacts** on the center list box.
   - Enter a name of **Contacts** and click **OK**.



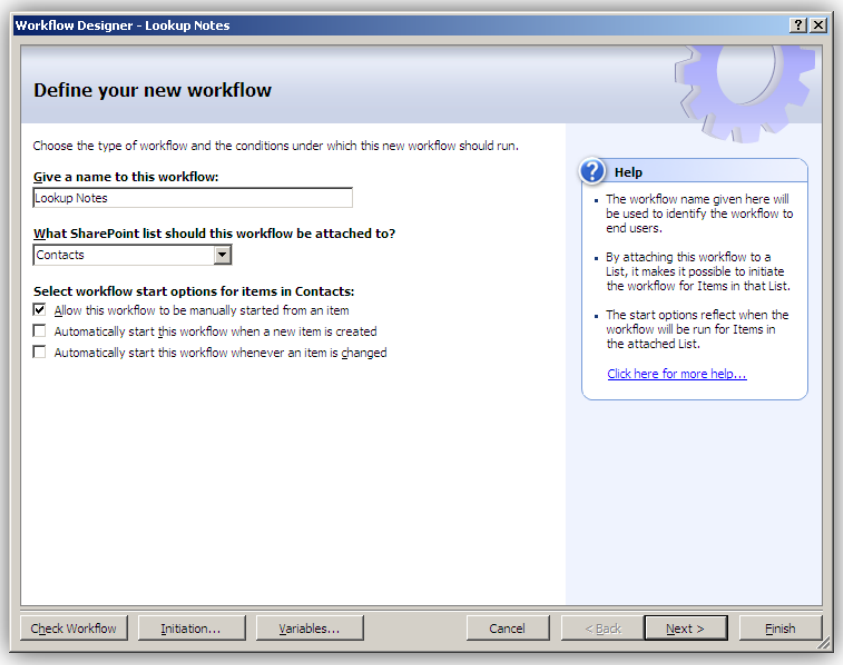
3) Add a new workflow named **Lookup Notes** to the **Contacts** list.

- In the **Folder List** on the left hand side, right click the site at the top of the tree and click **New -> SharePoint Content.**
- In the **New** dialog, select **Workflow** in the list box on the left and select **Blank Workflow** on the center list box.
- Click **OK** to create the new workflow.



4) Name the new workflow **Lookup Notes** and attach the new workflow to the Con**t**acts lists.

- In the **Workflow Designer Wizard**, set the name of the workflow to **Lookup Notes**.
- Choose the **Contacts** list as the list the workflow should be attached to.
- Click **Next** to move on to defining the workflow steps.

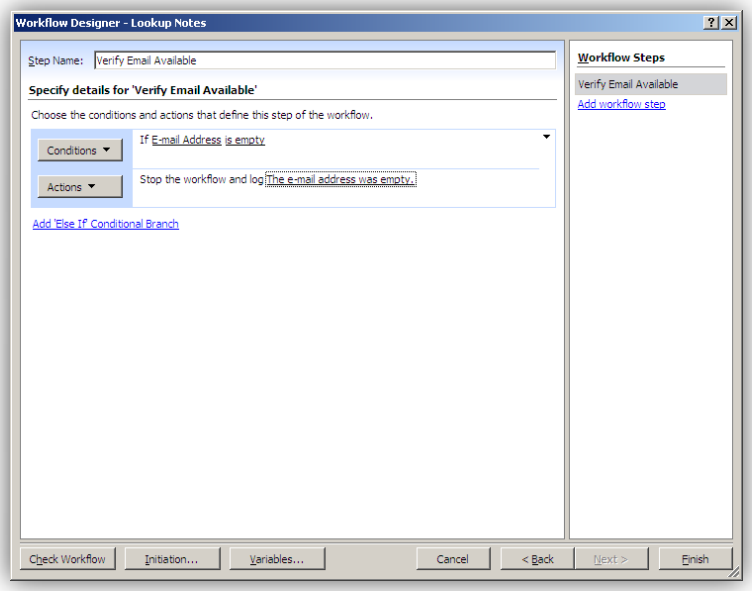


5) Create a step that will terminate the workflow if the contact's email field is empty.

- Set the **Step Name** to **Verify Email Available**.
- Add a condition of type **Compare Contacts Field**.

  *Click the **Condition** button and select **Compare Contacts Field**.*
  *Click the **field** link in the condition and select the **E-mail Address** field.*
  *Click the **equals** link and select **is empty**.*
- Add an action that of **Stop Workflow** to terminate the workflow.

  *Click the **Actions** button and select **More Actions**.*
  *In the **Workflow Actions** dialog, select the **Core Actions** category.*
  *Select the **Stop Workflow** action from the list and click **Add**.*
  *Click the **this message** link and enter "**The e-mail address was empty.**"*



6) Add a new step that will be used to retrieve the notes from the user.

- Click the **Add workflow step** link on the left hand side of the dialog.
- Set the **Step Name** to **Retrieve Contact Notes**.
- Add an action that will request notes from a user.

  *Click the **Actions** button and select **More actions**.*
  *In the **Workflow Actions** dialog, select the **Task Actions** category.*
  *Select the **Collect Data from a User** action from the list and click **Add**.*
  *Click the **data** link to start the **Custom Task Wizard**.*
  *Click **Next** to move to the next step.*
  *Enter a name of "**Enter notes for contact**" and click **Next**.*
  *Click **Add** to add a single field named **CustomNotes** of type **Single line of text** and then click **Next**.*
  *Click **Finish** on both dialogs to complete the wizard.*
  *Click the **this user** link and select a user of **Administrator**.*

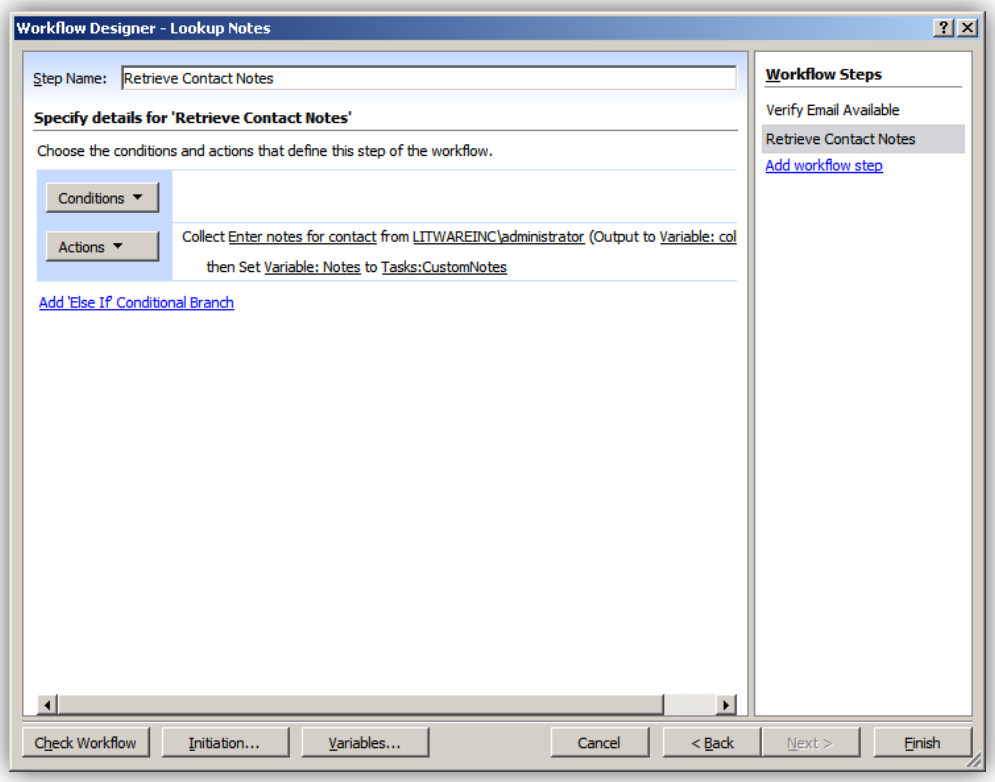
7) Store the results of the user activity into a workflow variable so it can be used later.

- Add an action that will store the result of the task in a workflow variable.

  *Click the **Actions** button and select **More actions**.*
  *In the **Workflow Actions** dialog, select the **Core Actions** category.*
  *Select the **Set Workflow Variable** action from the list and click **Add**.*
  *Click the **workflow variable** link in the action and select the **Create a new variable** link*
  *Give the variable a name of **Notes** and a type of **String** and click **OK**.*
  *Click the **value** link in the action and click the **Fx** button.*

- Select the field from the task that was just completed.

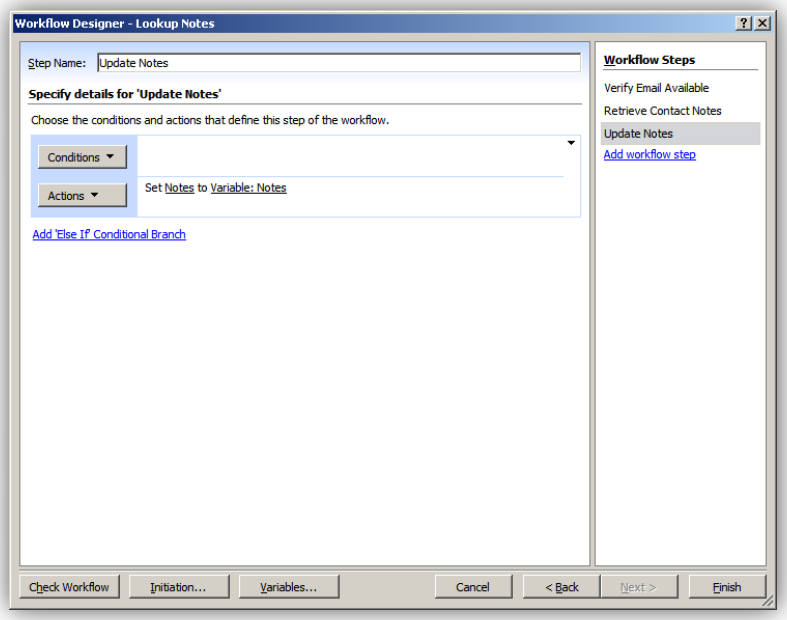
  *In the **Lookup Details** section's **Source** drop down list, select the **Tasks** list.*
  *In the **Lookup Details** section's **Field** drop down list, select **CustomNotes**.*
  *In the **Find the List Item** section's **Field** drop down list, select **Tasks:ID**.*
  *In the **Find the List Item** section's **Value** drop down list, click the **Fx** button.*
  *Select a **Source** of **Workflow Data** and a field of **Variable: collect** and click **OK**.*
  *Click **OK** again to complete the action.*



8) Using the **Notes** workflow variable, set the **Notes** field of the current item.

- Click the **Add workflow step** link on the left hand side of the dialog.
- Set the **Step Name** to **Update Notes**.
- Add an action that will update a field on the current item.

  *Click the **Actions** button and select **More actions**.*
  *In the **Workflow Actions** dialog, select the **Core Actions** category.*
  *Select the **Set Field in Current Item** action from the list and click **Add**.*
  *Click the **field** link in the action and select the **Notes** field.*
  *Click the **value** link in the action and click the **Fx** button.*
  *In the **Lookup Details** section's **Source** drop down list, select the **Workflow Data** item.*
  *In the **Lookup Details** section's **Field** drop down list, select **Variable: Notes** and click **OK**.*
  *Click **OK** again to complete the action.*



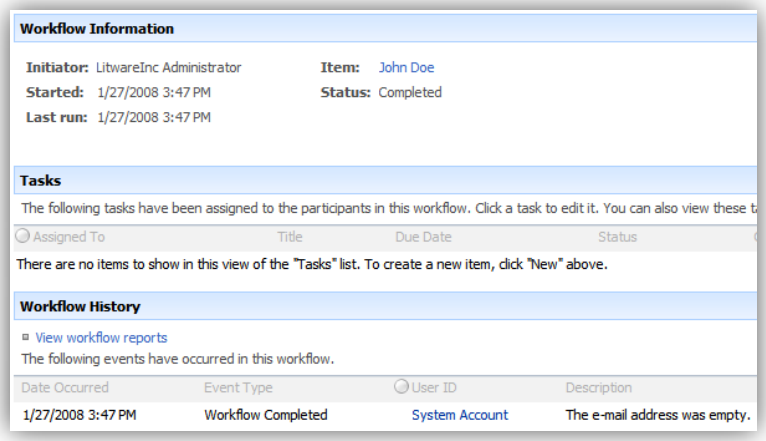
9) Save the workflow by clicking the **Finish** button in the Workflow Designer.

10) Test the workflow in the browser by creating a new contact with no email and starting the **Lookup Notes** workflow.

- Navigate to the **Contacts** list using **Internet Explorer**.

  *Navigate to **http://litwareinc.com/sites/Demos**.*
  *Click the **Contacts** link on the left hand navigation bar.*
- Create a new contact with an empty email address.
- Start the workflow on the new contact.

  *Click **Workflows** in the item's drop down menu and start a workflow of type **Lookup Notes**.*
  *On the instantiation page, click **Start**.*
- On the **Contacts** list page, click the **Completed** link under **Lookup Notes** to see the workflow status.
- Verify the workflow is completed and there is a history event indicating the email address was empty.



11) Test the workflow with a contact that has an email address.

- Edit the existing contact and provide an email address of **Administrator@litwareinc.com**.
- Start the workflow on the existing contact.

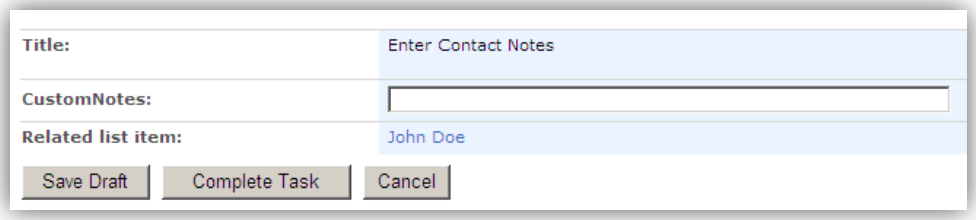
  *Click **Workflows** in the item's drop down menu and start a workflow of type **Lookup Notes**.*

  *On the instantiation page, click **Start**.*
- On the **Contacts** list page, click the **In Progress** link under **Lookup Notes** to see the workflow status.
- Approve the notes task

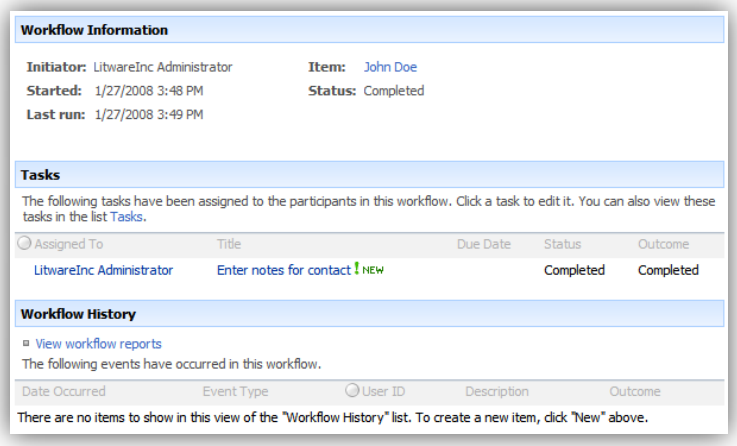
  *Click the **Edit Item** menu item on the task's title.*

  Enter notes for the current contact.

  *Click **Complete Task** to approve the task.*



- Verify the workflow is completed and all of the tasks are completed as well.



Exercise 2: Creating a custom SharePoint Designer condition

1) Add a new **SqlConditions** class to the **SqlDatabaseActivities** project.
   - In **Visual Studio 2008**, right click **SqlDatabaseActivities** in the **Solution Explorer** and click **Add -> Class**.
   - Name the class **SqlConditions.cs** and click **Add**.
   - Modify the class's definition to make it **public**.

2) Add a static method named **ItemExists** that will check for an item in the database.
   - Create a public static **ItemExists** method that returns a **bool** and accepts a **WorkflowContext**, list id, and item id along with the method specific parameters.

```
public static bool ItemExists(WorkflowContext context, string listId, int
itemId, string table, string queryField, object queryValue, string
connectionString)
{ }
```

   - Add code that opens the **SqlConnection** using the **connectionString** property.

```
// create the SqlConnection object
using (SqlConnection connection = new SqlConnection(connectionString)) {
    connection.Open();
}
```

   - Format the query and execute the query using the **table**, **queryField**, and **queryValue** parameters. Place the code in the previous **using** statement.

```
// build the query using the parameters
string query =
    string.Format("SELECT TOP 1 COUNT(*) FROM {0} WHERE {1} = @queryValue",
        table, queryField);

// create the SqlCommand and the parameters
SqlCommand command = new SqlCommand(query, connection);
command.Parameters.Add(new SqlParameter("@queryValue", queryValue));

// execute the query and return true if an item was found
return ((int)command.ExecuteScalar() == 1);
```

3) Create an **.ACTIONS** file in the **Template/1033/Workflow** folder.
   - Create a **Template** folder in the project.

     *Right click the project in the **Solution Explorer** and click **Add -> New Folder**.*
     *Change the name of the new folder to **Template**.*
   - Create a **1033** folder in the new **Template** folder.

     *Right click the **Template** folder in the **Solution Explorer** and click **Add -> New Folder**.*
     *Change the name of the new folder to **1033**.*
   - Create a **Workflow** folder in the new **1033** folder.

     *Right click the **1033** folder in the **Solution Explorer** and click **Add -> New Folder**.*
     *Change the name of the new folder to **Workflow**.*
   - Create a new **SqlDatabaseActivites.ACTIONS** file in the new **Workflow** folder.

     *Right click the **Workflow** folder in the **Solution Explorer** and click **Add -> New Item**.*

*In the **Add New Item** dialog, select **Data** in the **Categories** list and **Xml File** in the **Templates** list.*
*Set the name to **SqlDatabaseActivities.ACTIONS** and click **Add**.*

- In the **Actions** file, add the following XML to define the new condition.

*The condition defines the **Functioname**, **ClassName**, and **Assembly** of the condition method.*
*The **Sentence** and **FieldBind** defines the designer experience and references method parameters using the **_1_** syntax.*
*The **Parameters** define the type of each parameter in the condition method.*

```
<WorkflowInfo>
  <Conditions And="and" Or="or" Not="not" When="If" Else="Else if">
    <Condition Name="Exists in database" FunctionName="ItemExists"
ClassName="SqlDatabaseActivities.SqlConditions" Assembly="SqlDatabaseActivities,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=2848957be3dd88b2"
AppliesTo="all" UsesCurrentItem="true">
      <RuleDesigner Sentence="Exists in %1 in %2 where %3 equals %4">
        <FieldBind Id="1" Field="_1_" Text="table" />
        <FieldBind Id="2" Field="_4_" Text="database" />
        <FieldBind Id="3" Field="_2_" Text="field" />
        <FieldBind Id="4" Field="_3_" Text="value" />
      </RuleDesigner>
      <Parameters>
        <Parameter Name="_1_" Type="System.String, mscorlib" Direction="In" />
        <Parameter Name="_2_" Type="System.String, mscorlib" Direction="In" />
        <Parameter Name="_3_" Type="System.Object, mscorlib" Direction="In" />
        <Parameter Name="_4_" Type="System.String, mscorlib" Direction="In" />
      </Parameters>
    </Condition>
  </Conditions>
</WorkflowInfo>
```

4) Update the **web.config** file to allow the custom activities to be loaded by SharePoint's workflow engine.

- Open the **web.config** file at **C:\Inetpub\LitwarePublicSite** in **Visual Studio 2008**.
- Locate the **System.Workflow.ComponentModel.WorkflowCompiler/authorizedTypes**.
- Add another **authorizedType** referencing the **SqlDependencyActivities** assembly.

```
<authorizedType Assembly="SqlDatabaseActivities, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=2848957be3dd88b2"
Namespace="SqlDatabaseActivities" TypeName="*" Authorized="True" />
```

5) Add post build steps to install the assembly in the gac, copy files to the **Template** folder, and reset the app pool.

- Right click the project in the **Solution Explorer** and click **Properties**.
- Select the **Build Events** tab and enter the following **Post Build** steps

```
xcopy "$(ProjectDir)\TEMPLATE" "C:\Program Files\Common Files\Microsoft
Shared\web server extensions\12\TEMPLATE" /E /Y
"$(DevEnvDir)\..\..\SDK\v2.0\bin\gacutil.exe" /i "$(TargetPath)" /f
%windir%\system32\cscript.exe c:\windows\system32\iisapp.vbs /a
"SharePointDefaultAppPool" /r
```

Exercise 3: Creating a custom SharePoint Designer activity

1) Create a new **SqlGetValueActivity** in the **SqlDatabaseActivities** project.
   - Right click the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Add -> Activity**.
   - Enter a name of **SqlGetValueActivity.cs** and click **Add**.

2) Define the dependency properties the **SharePoint Designer** will use to communicate with the **activity**.
   - The **ConnectionString** property will define the connection string to use.
   - The **Table** property defines the table to use in the query.
   - The **QueryField** and **QueryValue** properties will define the row to use.
   - The **ResultField** will define the column to return.
   - The **Result** property will hold the result.

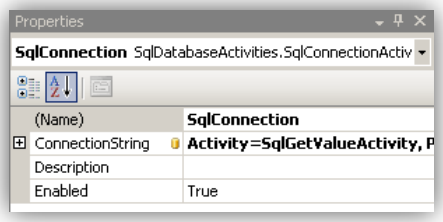
```
public static DependencyProperty ConnectionStringProperty =
    DependencyProperty.Register("ConnectionString",
      typeof(System.String), typeof(SqlDatabaseActivities.SqlGetValueActivity));

[Category("Parameters")]
public String ConnectionString
{
    get { return ((string)(base.GetValue(ConnectionStringProperty))); }
    set { base.SetValue(ConnectionStringProperty, value); }
}

public static DependencyProperty ResultFieldProperty =
    DependencyProperty.Register("ResultField",
      typeof(System.String), typeof(SqlDatabaseActivities.SqlGetValueActivity));

[Category("Parameters")]
public String ResultField
{
    get { return ((string)(base.GetValue(ResultFieldProperty))); }
    set { base.SetValue(ResultFieldProperty, value); }
}

public static DependencyProperty TableProperty =
    DependencyProperty.Register("Table",
      typeof(System.String), typeof(SqlDatabaseActivities.SqlGetValueActivity));

[Category("Parameters")]
public String Table
{
    get { return ((string)(base.GetValue(TableProperty))); }
    set { base.SetValue(TableProperty, value); }
}

public static DependencyProperty QueryFieldProperty =
```

```
    DependencyProperty.Register("QueryField",
      typeof(System.String), typeof(SqlDatabaseActivities.SqlGetValueActivity));

[Category("Parameters")]
public String QueryField
{
    get { return ((string)(base.GetValue(QueryFieldProperty))); }
    set { base.SetValue(QueryFieldProperty, value); }
}

public static DependencyProperty QueryValueProperty =
    DependencyProperty.Register("QueryValue",
      typeof(System.String), typeof(SqlDatabaseActivities.SqlGetValueActivity));

[Category("Parameters")]
public String QueryValue
{
    get { return ((string)(base.GetValue(QueryValueProperty))); }
    set { base.SetValue(QueryValueProperty, value); }
}

public static DependencyProperty ResultProperty =
    DependencyProperty.Register("Result",
      typeof(System.String), typeof(SqlDatabaseActivities.SqlGetValueActivity));

[Category("Parameters")]
public String Result
{
    get { return ((string)(base.GetValue(ResultProperty))); }
    set { base.SetValue(ResultProperty, value); }
}
```

3) Add a new **SqlConnection** activity to the **SqlGetValueActivity** designer.

- Right click **SqlGetValueActivity.cs** in the **Solution Explorer** and click **View Designer**.
- Drag a new **SqlConnection** activity onto the canvas and name it **SqlConnection**.

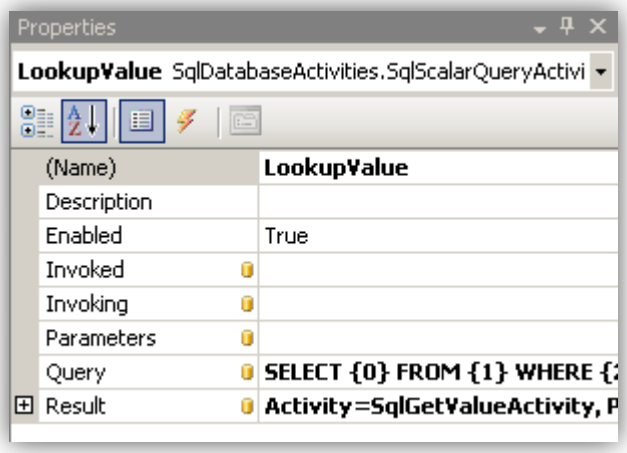
*Drag a new **SqlConnection** activity onto the canvas.*
*In the properties pane, set the **Name** property to **SqlConnection**.*
*Select the **ConnectionString** property and click the **…** button.*
*Select the **ConnectionString** property from the parent's property list.*



4) Define the query embedded in the new **SqlConnection** activity.

- Select the **SqlScalarQueryActivity** inside the **SqlConnection** activity and change its name to **LookupValue**.

  *Right click the **SqlScalarQueryActivity** in the **SqlConnection** activity and click **Properties**.*
  *In the properties pane, set the **Name** property to **LookupValue**.*
  *Bind the **Result** property to the parent's **Result** property.*
  *Set the **Query** property to **SELECT {0} FROM {1} WHERE {2} = @queryValue**.*



- Add an Invoking handler to the activity that will initialize the **Query** and **Parameters** property.

  *In the properties pane, set the **Invoking** property of **LookupValue_Invoking** and press **Enter**.*
  *Cast the sender property to a **SqlScalarQueryActivity** object.*
  *Use **string.Format** to replace the tokens in the query with the **ResultField**, **Table**, and **QueryField** properties.*
  *Create a new **SqlParameter** object to map the **QueryValue** property to the **@queryValue** query parameter.*
  *Put the new **SqlParameter** into an array of **SqlParameters** and assign it to the activity's **Parameters** property.*

```
private void LookupValue_Invoking(object sender, EventArgs e)
{
    SqlScalarQueryActivity activity = sender as SqlScalarQueryActivity;

    activity.Query = string.Format(
        activity.Query, this.ResultField, this.Table, this.QueryField);
    activity.Parameters = new SqlParameter[]
    {
        new SqlParameter("@queryValue", this.QueryValue)
    };
}
```

5) Update the **.ACTIONS** file to define the new activity's integration with the **SharePoint Designer**.

- Add the **Actions** and **Action** elements to the file.

  *The **ClassName** and **Assembly** attributes define the location of the activity.*

```
<Actions Sequential="then" Parallel="and">
  <Action Name="Sql Value Lookup"
      ClassName="SqlDatabaseActivities.SqlGetValueActivity"
      Assembly="SqlDatabaseActivities, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=2848957be3dd88b2"
      AppliesTo="all" Category="Sql" >
  </Action>
</Actions>
```

- Add the interface definition for the action as a **RuleDesigner** element embedded in the **Action** element.

  *The **sentence** defines the format of the action in the **SharePoint Designer**.*

  *The **FieldBind** elements map named properties in the activity to indexed tokens in the sentence.*

  *The **DesignerType** attribute defines the user interface provided for choosing the value.*

```
<RuleDesigner Sentence="Set %6 using the %1 field from the %2 table in database
%5 where the %3 field equals %4">
  <FieldBind Field="ResultField" Id="1" />
  <FieldBind Field="Table" Id="2" />
  <FieldBind Field="QueryField" Id="3" />
  <FieldBind Field="QueryValue" Id="4" />
  <FieldBind Field="ConnectionString" Id="5" />
  <FieldBind Field="Result" DesignerType="ParameterNames" Id="6" />
</RuleDesigner>
```

- Define the actual types for the named parameters using a **Parameters** element immediately following the **RuleDesigner** element.

  *Each field referenced in the **FieldBind** elements has a corresponding parameter element.*

  *The **Name** and **Type** match the properties of the activity.*

  *The **Direction** determines if the action provides or receives the value.*

```
<Parameters>
  <Parameter Name="ConnectionString" Type="System.String, mscorlib"
             Direction="In" />
  <Parameter Name="ResultField" Type="System.String, mscorlib" Direction="In" />
  <Parameter Name="Table" Type="System.String, mscorlib" Direction="In" />
  <Parameter Name="QueryField" Type="System.String, mscorlib" Direction="In" />
  <Parameter Name="QueryValue" Type="System.String, mscorlib" Direction="In" />
  <Parameter Name="Result" Type="System.String, mscorlib" Direction="Out" />
</Parameters>
```

6) Rebuild the project in **Visual Studio 2008**.

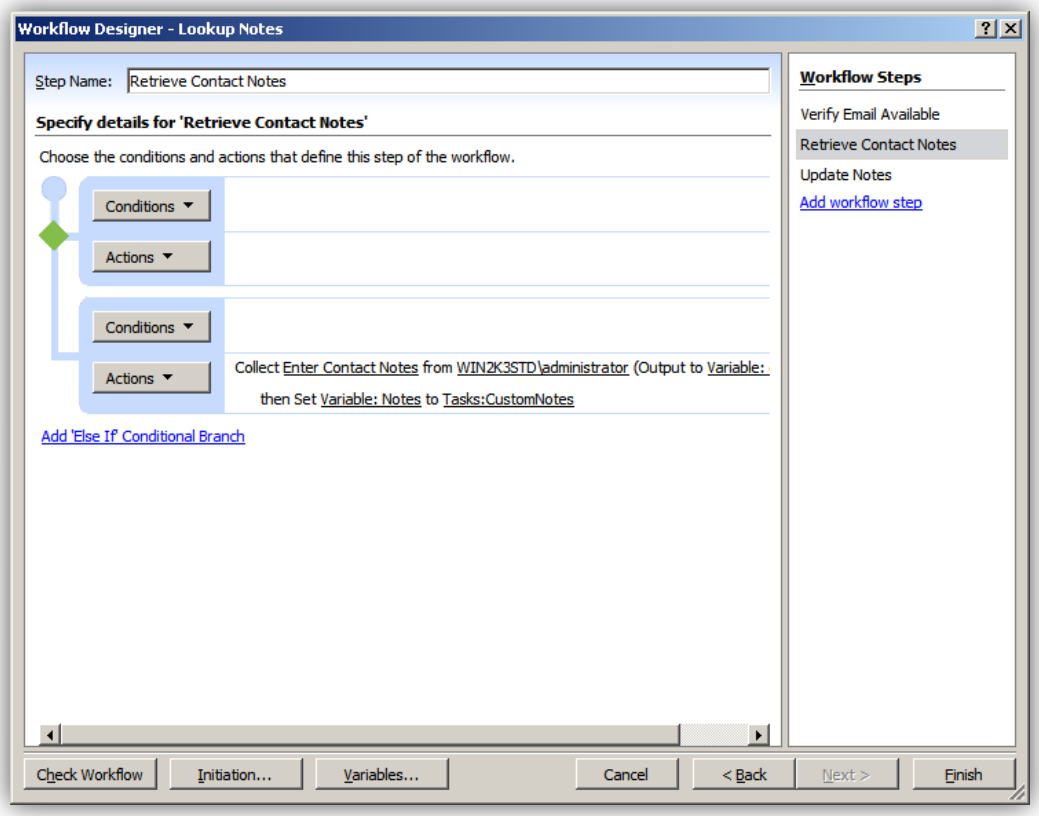
- Right click the project in the solution explorer and click **Rebuild**.

Exercise 4: Updating SharePoint Designer Workflow using new Activities

1) Close and reopen the SharePoint Designer to force it to unload the **SqlDatabaseActivity** assembly.
   - Start the SharePoint Designer by clicking the **Start Button** and selecting **All Programs -> Microsoft Office -> Microsoft Office SharePoint Designer 2007**.
   - Open the **Demos** site collection by clicking **File -> Open Site** and entering a site name of **http://litwareinc.com/sites/Demos**.

2) Open the existing **Lookup Notes** workflow.
   - Expand the **Workflow** node in the **Folder List**.
   - Expand the Lookup Notes in the Folder List
   - Double click **Lookup Notes.xoml** to open the workflow designer.

3) Add an **Else If** conditional branch o the **Retrieve Contact Notes**
   - Select the **Retreive Contact Notes** step in the **Workflow Steps** section of the **Workflow Designer**.
   - Click the **Add 'Else If' Conditional Branch** in the **Workflow Designer**.
   - Move the new branch up by clicking its drop down menu in the upper left corner and selecting **Move Branch Up**.



4) Add a condition that performs a lookup into the database to determine if the email address is available.

- Add a condition to the top branch of type **Exists in database**.

  *Click the **Condition** button and select **Exists in database**.*

  *Click the **table** link in the condition and enter **UserProfiles**.*

  *Click the **database** link in the condition and enter **Data. Source=.\SQLEXPRESS;AttachDbFilename=C:\Labs\Files\Litware.mdf;Integrated Security=True;Connect Timeout=30;User Instance=True**.*

  *Click the **field** link in the condition and enter **Email**.*

  *Click the **value** link in the action and click the **Fx** button.*

  *In the **Lookup Details** section's **Source** drop down list, select the **Current Item** item.*

  *In the **Lookup Details** section's **Field** drop down list, select **E-mail Address** and click **OK**.*

5) Add an action that will perform the lookup in the database and store the notes in the **Notes** variable.

- Add an action to the top branch of type **Sql Value Lookup**.

  *Click the **Actions** button and select **More actions**.*

  *In the **Workflow Actions** dialog, select the **Sql** category.*

  *Select the **Sql Value Lookup** action from the list and click **Add**.*

  *Click the **first** link in the action and select the **Variable: Notes** item.*

  *Click the **second** link in the action and set the value to **Comments**.*

  *Click the **third** link in the action and set the value to **UserProfiles**.*

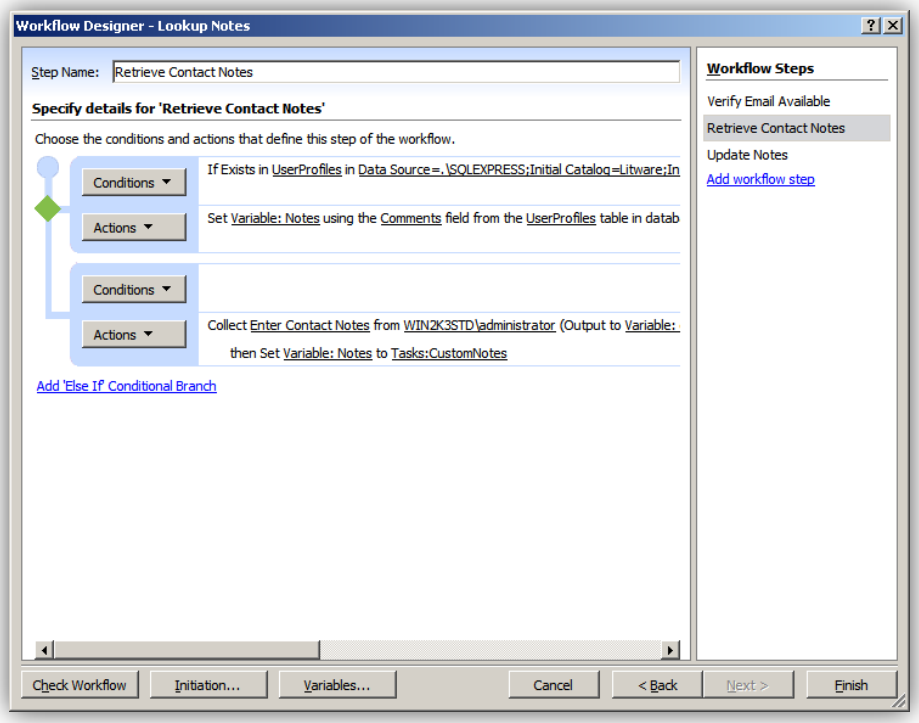
  *Click the **fourth** link in the action and set the value to **Data. Source=.\SQLEXPRESS;AttachDbFilename=C:\Labs\Files\Litware.mdf;Integrated Security=True;Connect Timeout=30;User Instance=True**.*

  *Click the **fifth** link in the action and set the value to **Email**.*

  *Click the **sixth** link in the action and click the **Fx** button.*

  *In the **Lookup Details** section's **Source** drop down list, select the **Current Item** item.*

  *In the **Lookup Details** section's **Field** drop down list, select **E-mail Address** and click **OK**.*



6) Save the workflow by clicking the **Finish** button in the **Workflow Designer**.

7) Test the workflow in the browser by creating a new contact with an email of **Administrator@litwareinc.com** and starting the **Lookup Notes** workflow.

- Navigate to the **Contacts** list using **Internet Explorer**.

  *Navigate to **http://litwareinc.com/sites/Demos**.*
  *Click the **Contacts** link on the left hand navigation bar.*

- Create a new contact with an email address of **Administrator@litwareinc.com**.
- Start the workflow on the new contact.

  *Click **Workflows** in the item's drop down menu and start a workflow of type **Lookup Notes**.*
  *On the instantiation page, click **Start**.*

- On the **Contacts** list page, click the **Completed** link under **Lookup Notes** to see the workflow status.
- Verify that the workflow is completed and that the contact's notes are now populated using the value in the database.