# Lab 03: Understanding the Windows Workflow Runtime

**Lab Overview:** Now that your company has an automated system to submit orders, it is time to focus on the order fulfillment process. Currently this is a three-step process. As orders are received they are considered submitted. Once an account manager approves them, they move into the Approved step and are waiting to be shipped. Finally, when the order is shipped, the order moves into the closed state. At any point along the way, the order can be canceled.

At each stage of the process, the account managers and the shipping department will be notified the order is in the current state. Once the account manager or the shipping department completes their task, the order should advance through the process.

## Exercise 0: Setting up the project

### Open the starter **Visual Studio 2008** solution at **\Labs\Lab03\StarterFiles\OrderWorkflow.sln**.

## Exercise 1: Creating the ExternalDataExchange interfaces

### Create a new workflow runtime service named **OrderManagerService**.

#### Create a new class named **OrderManagerService** in the **OrderWorkflow** project

##### Right click on **OrderWorkflow** in the **Solution Explorer**.

##### Select **Add -> New Item**.

##### Select **Code** in the list on the left and select **Class** in the window on the right.

##### Enter **OrderManagerService.cs** as the name and click **Add**.

#### Drive the **OrderManagerService** from the **WorkflowRuntimeService** class so it can be hosted in the Windows Workflow services list.

using System.Workflow.Runtime.Hosting;  
  
namespace OrderWorkflow  
{  
 public class OrderManagerService : WorkflowRuntimeService  
 {

### Create an external communications interface named **IOrderManager**.

#### Apply the **ExternalDataExchange** attribute to the interface to identify it as an interface Windows Workflow can use to communicate with the hosting environment.

#### Add a method named **SetOrderStatus** to **IOrderManager** to be used when telling the hosting environment to update an order’s status.

[ExternalDataExchange]

public interface IOrderManager

{

void SetOrderStatus(string status);

}

### Make **OrderManagerService** an implementation of **IOrderManager**

#### In the **OrderManagerService.cs** file, derive the **OrderManagerService** from the **IOrderManager** interface.

public class OrderManagerService : WorkflowRuntimeService, IOrderManager

{

}

#### In the **OrderManagerService** class, implement the **IOrderManager.SetOrderStatus** method. This method will update the status of the order in the order database. If the order does not exist, it will be created.

public void SetOrderStatus(string status)

{

// lookup the current workflow instance id

Guid orderId = WorkflowEnvironment.WorkflowInstanceId;

// find the order based on the current workflow instance id

// FirstOrDefault return the order or null is none found

DataClassesDataContext context = new DataClassesDataContext();

Order order = context.Orders.FirstOrDefault(n => n.Id == orderId);

// if the order wasn’t found create a new order and insert it

if (order == null)

{

order = new Order();

order.Id = orderId;

context.Orders.InsertOnSubmit(order);  
 }

// update the status and commit the changes

order.Status = status;

context.SubmitChanges();

}

### Add an event to the **OrderManagerService** that will be used to communicate order updates from the service to the user interface.

#### Add an **OrderChanged** event to the **OrderManagerService** class.

public event EventHandler OrderChanged;

#### Modify **SetOrderStatus** so **OrderChanged** is called anytime an order is updated.

// fire the order changed event  
 if (OrderChanged != null)

OrderChanged(this, EventArgs.Empty);

}

### Create another workflow runtime service named **TaskManagerService**.

#### Create a new class named **TaskManagerService** in the **OrderWorkflow** project

#### Modify the definition of **TaskManagerService** to make it a public class.

public class TaskManagerService

{

}

### Create an external communications interface named **ITaskManager**.

#### Apply the **ExternalDataExchange** attribute to the interface to identify it as an interface Windows Workflow can use to communicate with the hosting environment.

#### Add methods named **CreateTask** and **RemoveTask** to **ITaskManager** to be used to ask the hosting environment to create and delete user tasks.

#### Add two events named **Complete** and **Cancel** to **ITaskManager** to be used when sending events from the hosting environment into the workflow engine.

[ExternalDataExchange]

public interface ITaskManager

{

int CreateTask(string type);

void RemoveTask(int taskId);

event EventHandler<ExternalDataEventArgs> Complete;

event EventHandler<ExternalDataEventArgs> Cancel;

}

### Make **TaskManagerService** an implementation of **ITaskManager**

#### In the **TaskManagerService.cs** file, derive the **TaskManagerService** from the **ITaskManager** interface.

public class TaskManagerService : ITaskManager

{

#### Add a **TaskChanged** event to the **TaskManagerService** class.

public event EventHandler TaskChanged;

#### In the **TaskManagerService** class, implement the **ITaskManager.CreateTask** method. This method will create a new task in the tasks database table.

public int CreateTask(string type)

{

// lookup the current workflow instance id

Guid orderId = WorkflowEnvironment.WorkflowInstanceId;

// create the new task

Task task = new Task();

task.OrderId = orderId;

task.Type = type;

// add the task to the context

DataClassesDataContext context = new DataClassesDataContext();

context.Tasks.InsertOnSubmit(task);

context.SubmitChanges();

// fire the task changed event

if (TaskChanged != null)

TaskChanged(this, EventArgs.Empty);

// return the new task’s id

return task.Id;

}

#### In the **TaskManagerService** class, implement the **ITaskManager.RemoveTask** method. This method will remove a task when it is complete.

public void RemoveTask(int taskId)

{

// find the task by id

DataClassesDataContext context = new DataClassesDataContext();

Task task = context.Tasks.FirstOrDefault(n => n.Id == taskId);

// if the task was found, delete it

if (task != null)

context.Tasks.DeleteOnSubmit(task);

context.SubmitChanges();

// fire the task changed event

if (TaskChanged != null)

TaskChanged(this, EventArgs.Empty);

}

#### Add two helper methods named **OnComplete** and **OnCancel**. These methods will wrap the **ITaskManager’s** events and make them easier to call.

public void OnComplete(Guid orderId) {

if (Complete != null)

Complete(null, new ExternalDataEventArgs(orderId));

}

public void OnCancel(Guid orderId) {

if (Cancel != null)

Cancel(null, new ExternalDataEventArgs(orderId));

}

## Exercise 2: Setup Hosting Environment

### Host the **Windows Workflow** runtime in the **ShoppingApplication** class.

#### View the code for the **ShoppingApplication** class.

##### Right click **ShoppingApplication.cs** in the **Solution Explorer** and click **View Code**.

#### Add a new member variable **\_runtime** of type **WorkflowRuntime**.

WorkflowRuntime \_runtime = new WorkflowRuntime();

### Attach the **ExternalDataExchangeService** and the previously created services to the new **WorkflowRuntime** object.

#### In the **ShoppingApplication\_Load** method, add a new instance of the **ExternalDataExchangeService** class to the workflow runtime using the **AddService** method

// add the external data exchange service

\_runtime.AddService(new ExternalDataExchangeService());

#### Add new instances of the **OrderManagerService** and **TaskManagerService** to the newly created **ExternalDataExchangeService**. Use the **GetService** method of the **WorkflowRuntime** to get a reference to the **ExternalDataExchangeService**.

// add the order manager and task manager services to the runtime

\_runtime.GetService<ExternalDataExchangeService>().AddService(

new OrderManagerService());

\_runtime.GetService<ExternalDataExchangeService>().AddService(

new TaskManagerService());

### Attach event handlers to the **OrderChanged** and **TaskChanged** events in the newly created **OrderManagerService** and **TaskManagerService**.

// register two event handlers that will update the UI

\_runtime.GetService<OrderManagerService>().OrderChanged +=

new EventHandler(ShoppingApplication\_OrderChanged);

\_runtime.GetService<TaskManagerService>().TaskChanged +=

new EventHandler(ShoppingApplication\_TaskChanged);

**Note: The ShoppingApplication\_OrderUpdated and ShoppingApplication\_TaskUpdated methods have already been implemented for you.**

### Add code to start the workflow runtime when the initialization of the workflow runtime is completed.

// start the runtime

\_runtime.StartRuntime();

### Shutdown the workflow runtime and clean up when the form closes.

#### Locate the **ShoppingApplication\_FormClosing** method and add the shut down and dispose code.

// shutdown the runtime

\_runtime.StopRuntime();

\_runtime.Dispose();

## Exercise 3: Create the primary workflow path

### Create the three order states: **Submitted**, **Approved**, and **Closed**

#### Create a new state machine workflow in the project named **OrderWorkflow**

##### Right click **OrderWorkflow** in the **Solution Explorer** and click **Add -> New Item**.

##### Select **Workflow** in the tree view on the left and select **State Machine Workflow (code)** in the window on the right.

##### Enter **OrderPipeline.cs** as the name and click **Add**.

#### Define several constants used in the state machine.

##### Right click **OrderPipeline.cs** in the **Solution Explorer** and click **View Code**.

##### Add the following code to the class.

public const string SubmittedStatus = "Submitted";

public const string ApprovedStatus = "Approved";

public const string CompletedStatus = "Completed";

public const string CancelledStatus = "Cancelled";

public const string ApprovalType = "Approval";

public const string ShipType = "Ship";

#### Drag three **State** activities onto the canvas named **SubmittedState**, **ApprovedState**, **ClosedState**.

##### Right click on the **State Machine** designer canvas and click **Add State**.

##### In the properties pane, set the **Name** property to **SubmittedState**.

##### Repeat previous two steps for **ApprovedState** and **ClosedState**.

#### Right click on the canvas and select **Properties**

##### In the properties pane, set the **CompletedStateName** property to **ClosedState**

##### Set the **InitialStateName** property to **SubmittedState**

### Define the transition between the **SubmittedState** and the **ApprovedState**.

#### Drag an **EventDriven** activity from the toolbox onto the **SubmittedState** activity and name it **orderApproved**.

##### In the properties pane, set the **Name** property to **orderApproved**.

##### Double click the new **orderApproved** activity to open its design window.

#### Drag a new **HandleExternalEvent** activity onto the canvas and name it **handleOrderApproved**.

##### In the properties pane, set the **Name** property to **handleOrderApproved**.

##### Set the **InterfaceType** to **OrderWorkflow.ITaskManager**

##### Set **EventName** to **Complete**

#### Place a new **SetState** activity following **handleOrderApproved** and name it **setStateApproved**.

##### In the properties pane, set the **Name** to **setStateApproved**.

##### Set the **TargetStateName** property to **ApprovedState**.

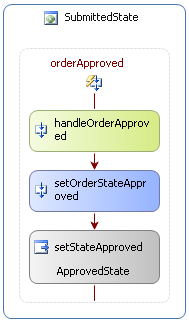
#### Add a **CallExternalMethod** activity named **setOrderStateApproved** between the existing activities to initiate communication with the hosting environment.

##### In the properties pane, set the **Name** property to **setOrderStateApproved**.

##### Set the **InterfaceType** property to **OrderWorkflow.IOrderManager**.

##### Set the **MethodName** property to **SetOrderStatus**.

##### Bind the **status** property to the **OrderPipeline.ApprovedStatus** property.



#### Switch back to the state view by clicking the **OrderPipeline** link in the upper left corner of the designer.

### Define the transition between the **ApprovedState** and the **ClosedState**.

#### Drag an **EventDriven** activity from the toolbox onto the **ApprovedState** and name it **orderCompleted**.

##### In the properties pane, set the **Name** property to **orderCompleted**.

#### Drag a new **HandleExternalEvent** activity onto the canvas and name it **handleOrderCompleted**.

##### In the properties pane, set the **Name** property to **handleOrderCompleted**.

##### Set the **InterfaceType** property to **OrderWorkflow.ITaskManager**

##### Set the **EventName** property to **Complete**

#### Place a new **SetState** activity following **handleOrderCompleted** and name it **setStateCompleted**. .

##### In the properties pane, set the **Name** property to **setStateCompleted**.

##### Set the **TargetStateName** property to **ClosedState**.

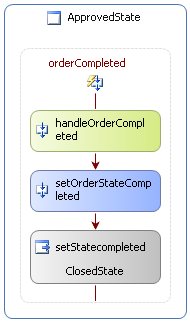
#### Add a **CallExternalMethod** activity between the existing activities and name it **setOrderStateCompleted**.

##### In the properties pane, set the **Name** property to **setOrderStateCompleted**.

##### Set the **InterfaceType** property to **OrderWorkflow.IOrderManager**.

##### Set the **MethodName** property to **SetOrderStatus**.

##### Bind the **status** property to the **OrderPipeline.CompletedStatus** property.



#### Switch back to the state view by clicking the **OrderPipeline** link in the upper left corner of the designer.

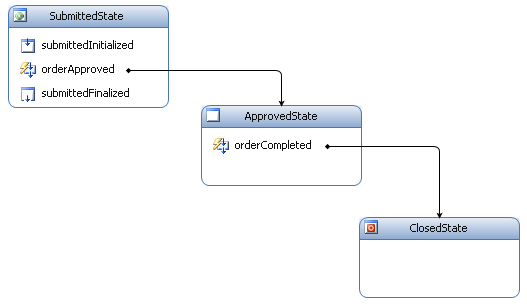
### Manage approval tasks whenever flow transitions into and out of the submitted state.

#### Drag a **StateInitialization** activity onto the **SubmittedState** activity.

##### Name the new activity **submittedInitialized**.

#### Drag a **StateFinalized** activity onto the **SubmittedState** activity.

##### Name the new activity **submittedFinalized**.



#### Double click the **submittedInitialized** activity to open its design window.

#### Set the initial state of the order by dragging a **CallExternalMethod** activity into **submittedInitialized** activity and name it **setOrderStatus**.

##### In the properties pane, set the **Name** property to **setOrderStatus**.

##### Set the **InterfaceType** property to **OrderWorkflow.IOrderManager**.

##### Set the **MethodName** property to **SetOrderStatus**.

##### Bind the **status** property to the **OrderPipeline.SubmittedStatus** property.

#### Create the approval task by dragging a **CallExternalMethod** activity onto the **submittedInitialized** activity and name it **createApproveTask**.

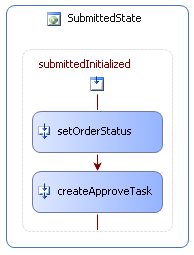
##### In the properties pane, set the **Name** property to **createApproveTask**.

##### Set the **InterfaceType** property to **OrderWorkflow.ITaskManager**.

##### Set the **MethodName** property to **CreateTask**.

##### Bind the **type** property to the **OrderPipeline.ApprovalType** property.

##### Bind the **(ReturnValue)** property to a new **TaskId** property.



#### Switch back to the state view by clicking the **OrderPipeline** link in the upper left corner of the designer.

#### Double click the **submittedFinalized** activity to open its design window.

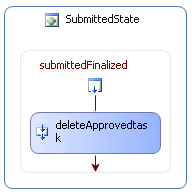
#### Delete the completed approval task by dragging a **CallExternalMethod** activity onto the **submittedFinalized** activity.

##### Name the new activity **deleteApprovedTask**.

##### Set the **InterfaceType** property to **OrderWorkflow.ITaskManager**.

##### Set the **MethodName** property to **RemoveTask**.

##### Bind the **taskId** property to the **OrderPipeline.TaskId** property.



#### Switch back to the state view by clicking the **OrderPipeine** link in the upper left corner of the designer.

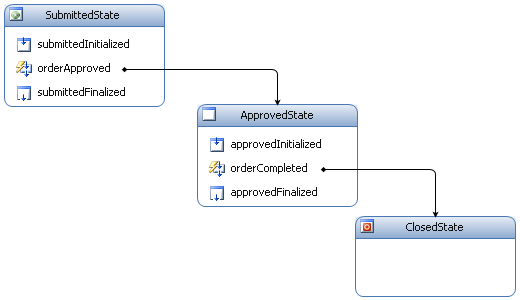
### Manage ship tasks whenever flow transitions into and out of the approved state.

#### Drag a **StateInitialization** activity onto the **ApprovedState** activity.

##### In the properties pane, set the **Name** property to **approvedInitialized**.

#### Drag a **StateFinalized** activity onto the **ApprovedState** activity.

##### In the properties pane, set the **Name** property to **approvedFinalized**.



#### Double click the **approvedInitialized** activity to open its design window.

#### Create the approval task by dragging a **CallExternalMethod** activity onto the **approvedInitialized** activity and naming it **createShipTask**.

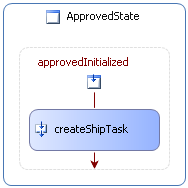
##### In the properties pane, set the **Name** property to **createShipTask**.

##### Set the **InterfaceType** property to **OrderWorkflow.ITaskManager**.

##### Set the **MethodName** property to **CreateTask**.

##### Bind the **type** property to the **OrderPipeline.ShipType** property.

##### Bind the **(ReturnValue)** property to the **OrderPipeline.TaskId** property.



#### Switch back to the state view by clicking the **OrderPipeline** link in the upper left corner of the designer.

#### Double click the **approvedFinalized** activity to open its design window.

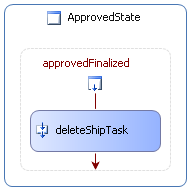
#### Delete the completed approval task by dragging a **CallExternalMethod** activity onto the **approvedFinalized** activity and name it **deleteShipTask**.

##### In the properties pane, set the **Name** property to **deleteShipTask**.

##### Set the **InterfaceType** property to **OrderWorkflow.ITaskManager**.

##### Set the **MethodName** property to **RemoveTask**.

##### Bind the **taskId** property to the **OrderPipeline.TaskId** property.



## Exercise 4: Connect the workflow to the user interface

### Implement the handler for the **Submit New Order** button to start a new **OrderPipeline** workflow.

#### Right click on the **ShoppingApplication.cs** file in the solution explorer and choose **View Code**.

#### Locate the **btnSubmitOrder\_Click** method and use the **CreateWorkflow** method to create a new workflow instance of type **OrderPipeline**.

WorkflowInstance instance = \_runtime.CreateWorkflow(typeof(OrderPipeline));

#### Start the new instance using its **Start** method.

instance.Start();

### When a task is displayed in the user interface, a button will be used to communicate with the workflow engine that the task is completed.

#### Locate the **gridTasks\_CellContentClick** and add code to call the **OnComplete** method of the **TaskManagerService**.

Task task = gridTasks.Rows[e.RowIndex].DataBoundItem as Task;

\_runtime.GetService<TaskManagerService>().OnComplete(task.OrderId);

**Note: Most of the code in this method has been implemented for you as it does not relate to workflow. The first line in the above code segment is already implemented and is included for orientation purposes only.**

### When the workflow is completed, the runtime fires a **WorkflowCompleted** event. Use this to display a nice message to the user.

#### Locate the **ShoppingApplication\_Load** method and register a handler for the runtime’s **WorkflowCompletedEvent**.

##### Register the method at the top of the function

// register the event handlers

\_runtime.WorkflowCompleted += new EventHandler<WorkflowCompletedEventArgs>(Runtime\_WorkflowCompleted);

#### Create a new method named **Runtime\_WorkflowCompleted**.

void Runtime\_WorkflowCompleted(object sender, WorkflowCompletedEventArgs e)

{

}

#### Add code to access the database and loads the details of the order that was just completed.

// find the order that just completed

DataClassesDataContext context = new DataClassesDataContext();

Order order = context.Orders.FirstOrDefault(n => n.Id == e.WorkflowInstance.InstanceId);

#### If the order was found, display a message box with an order summary, then deletes the order.

// if the order exists, display a message and remove the order

if (order != null) {

// display the message

MessageBox.Show(

string.Format("Order '{0}' was '{1}'.",

e.WorkflowInstance.InstanceId, order.Status));

// delete the order

context.Orders.DeleteOnSubmit(order);

context.SubmitChanges();

}

#### Finally update the order data source.

// update the order display

this.Invoke(new ThreadStart(delegate() {

orderBindingSource.DataSource = context.Orders; }));

### Test the application by passing an order through the system.

#### When the application has loaded, click the **Submit New Order** to place an order in the pipeline.

#### A task of type approve should appear in the tasks list. Click the **Complete** button related to the task to indicate the order was approved.

#### This will change the status of the order and add a new task of type **Ship**. Click the task’s **complete** button to indicate the order was shipped.

#### A message box will confirm the final status of the order and the lists should be cleared. The order has been successfully processed.

## Challenge: Add the ability to cancel orders

### Add the global **orderCancelled** **EventDriven** activity to the **OrderPipeline**.

#### View the design for the **OrderPipeline** workflow

##### Right click **OrderPipeline.cs** in the solution explorer and select **View Designer**.

#### Drag an **EventDriven** activity onto the canvas outside any existing **State** activities.

##### Name the new activity **orderCancelled**

**Note: By placing this activity outside a state, it is active in all states.**

#### Drag a new **HandleExternalEvent** activity onto the canvas and name it **handleOrderCancelled**.

##### In the properties pane, set the **Name** property to **handleOrderCancelled**.

##### Set the **InterfaceType** property to **OrderWorkflow.ITaskManager**

##### Set the **EventName** property to **Cancel**

#### Place a new **SetState** activity following **handleOrderCancelled** and name it **setStateCancelled**.

##### In the properties pane, set the **Name** property to **setStateCancelled**.

##### Set the **TargetStateName** property to **ClosedState**.

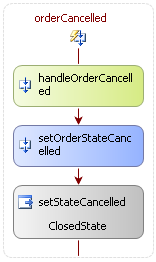
#### Add a **CallExternalMethod** activity between the existing activities and name it **setOrderStateCancelled**.

##### In the properties pane, set the **Name** property to **setOrderStateCancelled**.

##### Set the **InterfaceType** property to **OrderWorkflow**.**IOrderManager**.

##### Set the **MethodName** property to **SetOrderStatus**.

##### Bind the **status** property to the **OrderPipeline.CancelledStatus** property.



### When an order is displayed in the user interface, a cancel button is associated with it. Connect the cancel button with the **ITaskManager.OnCancel** method.

#### Right click **ShoppingApplication.cs** in the **Solution Explorer** and click **View Code**.

#### Locate the **gridOrders\_CellContentClick** and add code to call the **OnCancel** method of the **TaskManagerService**.

Order order = gridOrders.Rows[e.RowIndex].DataBoundItem as Order;

\_runtime.GetService<TaskManagerService>().OnCancel(order.Id);

**Note: Most of the code in this method has been implemented for you as it does not relate to workflow. The first line in the above code segment is already implemented and is included for orientation purposes only.**

### Test the application by passing an order through the system and cancelling it.

#### When the application has loaded, click the **Submit New Order** to place an order in the pipeline.

#### An order should appear in the order list. Click the **cancel** button to cancel the order.

#### A message box will confirm the final status of the order and the lists should be cleared. The order has been successfully canceled.

## Challenge: Use persistence for long running processes

### Build a configuration file entry that will cause the persistence service to be loaded into the workflow runtime. The persistence service will store running workflows in a pre initialized database.

#### Open the **app.config** file and add the following line to the **configuration/configSections** node.

<section name=”WorkflowRuntime” type=”System.Workflow.Runtime.Configuration.WorkflowRuntimeSection, System.Workflow.Runtime, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35” />

#### Add the following section to the configuration node.

<WorkflowRuntime Name="SampleApplication">

<Services>

<add type="System.Workflow.Runtime.Hosting.SqlWorkflowPersistenceService,

System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,

PublicKeyToken=31bf3856ad364e35"

ConnectionString="Data Source=.\SQLEXPRESS;

AttachDbFilename=|DataDirectory|Database.mdf;

Integrated Security=True;Connect Timeout=30;

User Instance=True"

UnloadOnIdle="true"/>

</Services>

</WorkflowRuntime>

### Update the creation of the **WorkflowRuntime** object to use the config file when loading.

#### Open the code for **ShoppingApplication** form.

##### Right click **ShoppingApplication.cs** in the solution explorer and click **View Code**.

#### Change the code creating the **\_runtime** to the following code.

WorkflowRuntime \_runtime = new WorkflowRuntime(“WorkflowRuntime”);

### Add a handler to the workflow runtime what will log whenever the workflow instance is persisted to the database.

#### Add another line of code following the registering of the **WorkflowCompleted** handler that handles the **WorkflowPersistence** event.

// register the event handlers

\_runtime.WorkflowCompleted += new EventHandler<WorkflowCompletedEventArgs>(Runtime\_WorkflowCompleted);

\_runtime.WorkflowPersisted += new EventHandler<WorkflowEventArgs>(Runtime\_WorkflowPersisted);

### Test the application by passing an order through the system. Notice that after each step, the workflow is persisted.