THE EXPERT'S VOICE® IN .NET

# Pro WF

## Windows Workflow in .NET 3.0

*Use Windows Workflow Foundation to develop next-generation, workflow-enabled applications*

Bruce Bukovics

Apress®

# Pro WF

## Windows Workflow in .NET 3.0

Bruce Bukovics

Apress®

**Pro WF: Windows Workflow in .NET 3.0**



ISBN-13 (pbk): 978-1-59059-778-1

ISBN-10 (pbk): 1-59059-778-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1



Lead Editor: Ewan Buckingham
Technical Reviewer: Sylvain Groulx
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Beth Christmas
Copy Edit Manager: Nicole Flores
Copy Editor: Jennifer Whipple
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Susan Glinert
Proofreader: Nancy Riddiough
Indexer: Julie Grady
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.



The source code for this book is available to readers at http://www.apress.com in the Source Code/Download section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 8

# Workflow Persistence

An important capability of workflows is that they can be *persisted* (saved and reloaded at a later time). Workflow persistence is especially important when developing applications that coordinate human interactions, since those interactions could take a long period of time. But persistence is also applicable to other types of applications. Without persistence, the lifetime of your workflows is limited. When the host application is shut down, any workflow instances simply cease to exist.

Workflow persistence is implemented as one of the optional core workflow services. You need to load a persistence service only if your application requires it.

After taking a brief look at reasons to use persistence, this chapter examines how persistence works in the Windows Workflow Foundation (WF) world. WF includes a standard persistence service that works with SQL Server. This chapter includes an example that demonstrates how to use this persistence service.

The WF persistence logic was implemented as external services in order to permit you to provide your own persistence implementation. The second half of this chapter demonstrates how to implement your own custom persistence service and use it in an application.

## Understanding Persistence

Before jumping into a live example of workflow persistence, this section of the chapter provides an overview of how persistence works in WF. This overview provides background information that will make the examples that follow it easier to understand.

This section begins with a review of several reasons to use workflow persistence.

### Why Persist Workflows?

Up to this point, you have seen workflows that perform only short-lived tasks. You have seen that multiple instances of a workflow can be started by an application, but each workflow exists only in memory, within the workflow runtime engine. While these in-memory workflows are very useful and can be used to accomplish many tasks, they are limited. When the host application or the workflow runtime is stopped, the workflow instances cease to exist. Their lifetime is tightly bound to a single host application. Workflows that only exist in memory are limited in several ways.

*Workflow persistence* means to save the complete state of a workflow to a durable store such as a SQL database or file. Once persisted, the workflow can be removed from memory and reloaded at a later time. Here are some of the reasons to use persistence with your workflows:

- *Human interaction tasks*: Workflows that are designed to interact with humans are typically long running. They may take minutes, hours, days, or weeks to complete. It isn't practical to keep such a workflow alive in memory for that length of time. Persistence of the workflow provides a way to unload it while it is waiting for the next human event, and then reload it when the event is received.
- *State machine workflows*: These workflows are designed around a set of defined states, with interactions that cause a transition to a different state. The kinds of tasks that are modeled with a state machine workflow usually require persistence of the workflow while waiting for the next transition.
- *Scalability*: In-memory workflows are limited to execution within a single application host. To provide scalability, some applications may require multiple workflow runtime hosts, perhaps running on multiple servers. A persisted workflow can be loaded and executed on a different server than the one that started it.
- *Resource consumption*: Without a persistence mechanism, workflows must stay in memory. They have nowhere else to go. If a workflow is waiting for an external input, such as an event, it is actually idle. With a persistence mechanism in place, the idled workflow can be persisted and unloaded from memory. Once the external event is received, the workflow is loaded back into memory and processing can continue. Swapping workflows in and out of memory like this frees resources (memory and workflow threads), making them available for workflows that are actively executing.
- *Application flexibility*: An in-memory workflow can be executed only by the application that started it. Perhaps you have a web-based application that starts a workflow, but you also want to use more traditional applications (e.g., Windows Forms) to work with the workflow. Persisting the workflow allows it to be reloaded by an entirely different class of application.

Not every application requires persistence. If you require only short-lived workflows that execute within a single application, you can probably do without persistence. On the other hand, if your workflows model long-running tasks that are designed around human interactions, or if you need the flexibility to tune the performance of your application, then you likely need persistence.

## Persistence Overview

WF implements persistence as one of the optional core workflow services. Instead of providing a static persistence mechanism that is implemented internally by the workflow runtime engine, the WF designers decided to make the implementation externally pluggable. This provides you with total control when it comes to persistence. If you don't need it, you don't have to load the persistence service. If you do need persistence, you load the persistence service that provides the implementation that meets your needs.

WF provides a standard persistence implementation that works with a SQL Server database (`SqlWorkflowPersistenceService` in the `System.Workflow.Runtime.Hosting` namespace). You create your own SQL Server database for persistence, but WF provides the scripts to create the necessary tables and stored procedures that this service expects.

If you don't want to use the standard SQL Server persistence service, you can provide your own implementation. WF includes a base class (`WorkflowPersistenceService` in the `System.Workflow.Runtime.Hosting` namespace) that you must derive from when implementing your own persistence service. You are permitted to implement the persistence service using any type of durable store that you want. You can persist workflows to binary files, to XML, or to a relational database with your own schema. But you can load only a single persistence service at a time for an instance of the workflow runtime.

You load a persistence service just once, when you are first initializing the workflow runtime engine. Once the service is loaded, the workflow runtime engine uses it at designated times in the life of a workflow to save or load the state of a workflow instance. Persisting a workflow instance doesn't require any manual intervention from you; it is something that is controlled by the workflow runtime engine and occurs automatically.

This is one of the real benefits of using persistence with WF. It eliminates the need to implement your own persistence mechanism to save the state of your application. If your application state is made up of one or more workflow instances, the state is saved for you. Because you are utilizing WF, you get persistence for free.

If a persistence service is loaded, the *state* of the workflow is persisted in these situations:

- When a workflow becomes idle. An example is when a workflow is waiting for an external event or executes a `DelayActivity`.
- When a workflow completes or terminates.
- When a `TransactionScopeActivity` completes. A `TransactionScopeActivity` identifies a logical unit of work that is ended when the activity completes.
- When a `CompensatableSequenceActivity` completes. A `CompensatableSequenceActivity` identifies a set of child activities that are compensatable. *Compensation* is the ability to undo the actions of a completed activity.
- When a custom activity that is decorated with the `PersistOnCloseAttribute` completes.
- When you manually invoke one of the methods on a `WorkflowInstance` that cause a persistence operation. Examples are `Unload` and `TryUnload`. The `Load` method results in a previously unloaded and persisted workflow being retrieved and loaded back into memory.

---

■ **Note** It is important to make a distinction between saving a workflow and saving the *state* of a workflow. Not all persistence operations result in a new serialized copy of a workflow being saved. For instance, when a workflow completes or terminates, the standard SQL Server persistence service (`SqlWorkflowPersistenceService`) actually removes the persisted copy of the workflow. It persisted the workflow in the sense that it updated the durable store with the state of the workflow. If you implement your own persistence service, you may choose to do something else when a workflow completes.

---

When any of these situations occur, the workflow runtime engine invokes the appropriate methods of the persistence service to save the workflow state. The workflow runtime doesn't know (or care) how the persistence method was implemented. It only knows that a persistence operation was invoked on the service. Obviously, if no persistence service has been loaded, these methods are not invoked and no persistence takes place.

When a workflow must be reloaded into memory, other methods on the persistence service are invoked. These methods retrieve the workflow state from a durable store and deserialize it back into an `Activity` object that the workflow engine can use. This occurs most often when a workflow that was previously idle is now ready for execution. Perhaps the workflow received an external event that it was waiting to receive, or the time span for a `DelayActivity` expired.

To handle a `DelayActivity`, a persistence service must also save the time when the `DelayActivity` is expected to expire. The service will then periodically poll the persisted workflows to see if any have a timer that has expired. If so, the workflow is ready to be reloaded and execution resumes. The `SqlWorkflowPersistenceService` has a `LoadingInterval` property that permits you to control the frequency of this polling interval.

In most scenarios, you will want to unload a workflow from memory when it becomes idle and after it has been persisted. A workflow might become idle if it is waiting to receive an external event.

Unloading an idle workflow makes sense since it frees valuable resources (memory and workflow threads), making them available to workflows that are actively executing. And since you don't know exactly how long the workflow will be idle, it's usually more efficient to unload it.

However, unloading workflows when they become idle is a persistence service option. You can choose to persist idle workflows but continue to keep them in memory. You might do this in an application where workflows are frequently waiting for external events, but the events are received very quickly. In this case, the overhead of unloading and then immediately reloading a workflow might be greater than simply keeping the workflow in memory.

---

■ **Note** Persistence and unloading of idle workflows are related but separate options. You can choose to persist workflows and unload them from memory when they become idle. Or you can persist them and choose to keep them in memory. You cannot choose to unload an idle workflow unless you also persist it; that is not an option.

---

In the next section, you will learn how to use the standard persistence service for SQL Server (`SqlWorkflowPersistenceService`).

# Using the SqlWorkflowPersistenceService

WF includes the `SqlWorkflowPersistenceService` class. This class contains the implementation for a persistence service that uses SQL Server as its durable store. Using this service is, by far, the easiest way to introduce persistence to your workflows. The primary requirement is that you must have a version of SQL Server installed (SQL Server 2000 or greater). Since SQL Server Express is now available for free, it is very easy to meet this requirement.

To use `SqlWorkflowPersistenceService`, you follow these steps:

1. If you haven't already done so, install a version of SQL Server (SQL Server 2000 or greater). This includes the full purchased versions of SQL Server 2000 or SQL Server 2005, or the free versions of SQL Server Express or Microsoft SQL Server Desktop Engine (MSDE).
2. Create a database to use for workflow persistence. This can be a new database that will contain only the persistence tables, or an existing database that is already used by your application.
3. Create the database tables, stored procedures, and other objects that `SqlWorkflowPersistenceService` expects. WF includes the SQL scripts that create these objects. You simply need to execute them once against your target database.
4. In your host application, create an instance of `SqlWorkflowPersistenceService` and add it to the workflow runtime engine as a core service. This is done just once during initialization of the workflow runtime. The constructor for `SqlWorkflowPersistenceService` is where you provide the database connection string, along with other parameters that control the service. You can also load this service using entries in your `App.Config` file.

That's it. Once you've completed these steps, workflows will be persisted automatically according to the workflow runtime rules outlined in the previous section.

In the sections that follow, you will walk through the development of an application that demonstrates workflow persistence. The application is a Windows Forms application that allows you to start multiple workflow instances and monitor their current status (e.g., idle, persisted, unloaded, complete). The demonstration workflow is extremely simple. It contains a `WhileActivity` that executes until a workflow property is set to `true`. Within the `WhileActivity`, a set of two external events is handled. The first event doesn't really do anything useful. But when that event is raised, it allows the workflow to execute the next iteration of the `WhileActivity`. That causes the workflow

to cycle through the states again (idle, persisted, unloaded) as you watch from the host application. The second event sets the workflow property so that the `WhileActivity` and the workflow end.

Since the workflows are persisted, you can also stop and restart the application without losing any workflows. When the application starts, it obtains a list of all workflows that are persisted. You can then reload and execute one of the available workflows by raising one of the external events.

## Preparing a Database for Persistence

Before you begin development of the demonstration application, you should prepare a database that will be used for workflow persistence.

If you haven't already done so, install a version of Microsoft SQL Server. If you don't have a full retail version available, you can use SQL Server Express, which is distributed with Visual Studio 2005.

Once you have SQL Server installed, you can create a database and populate it with the objects required by `SqlWorkflowPersistenceService` by following the steps outlined in Chapter 4. Within that chapter, the sidebar titled "Setup Steps for Persistence Testing" outlines the steps necessary to prepare a persistence database.

After you've performed all of these steps, a local database named `WorkflowPersistence` should now be populated with the tables and stored procedures that are required for workflow persistence. The demonstration application assumes that this is the database name and that it is available on the local instance of SQL Server Express (`localhost\SQLEXPRESS`). If you change the database or server name, you'll need to adjust the example code accordingly.

## Implementing the Local Service

This application requires the ability to raise two different events that are handled by the workflow. To raise these events, you need to implement a local service. Start by creating a new project using the Empty Workflow Project template and name the project `SharedWorkflows`. This creates a DLL assembly that can be referenced by the Windows Forms demonstration application.

You'll need to first define an `interface` for the local service. Add a new C# `interface` to the `SharedWorkflows` project and name it `IPersistenceDemo`. Listing 8-1 shows the complete contents of the `IPersistenceDemo.cs` file.

**Listing 8-1.** *Complete IPersistenceDemo.cs File*

```
using System;
using System.Workflow.Activities;

namespace SharedWorkflows
{
    /// <summary>
    /// Events exposed by the PersistenceDemoService
    /// </summary>
    [ExternalDataExchange]
    public interface IPersistenceDemo
    {
        event EventHandler<ExternalDataEventArgs> ContinueReceived;

        event EventHandler<ExternalDataEventArgs> StopReceived;
    }
}
```

The `interface` is decorated with the `ExternalDataExchangeAttribute` to identify it as the local service `interface`. The two events that are defined both use `ExternalDataEventArgs` as their event arguments. This is a requirement in order to handle these events in a workflow.

Next, add a new C# class to the SharedWorkflows project and name it PersistenceDemoService. This is the local service that implements the IPersistenceDemo interface. It is almost as simple as the interface. Listing 8-2 shows the complete PersistenceDemoService.cs file.

**Listing 8-2.** *Complete PersistenceDemoService.cs File*

```
using System;
using System.Workflow.Activities;

namespace SharedWorkflows
{
    /// <summary>
    /// Implements events that are handled by workflow instances
    /// </summary>
    public class PersistenceDemoService : IPersistenceDemo
    {
        #region IPersistenceDemo Members

        public event EventHandler<ExternalDataEventArgs> ContinueReceived;

        public event EventHandler<ExternalDataEventArgs> StopReceived;

        #endregion

        #region Members used by the host application

        /// <summary>
        /// Raise the ContinueReceived event
        /// </summary>
        /// <param name="args"></param>
        public void OnContinueReceived(ExternalDataEventArgs args)
        {
            if (ContinueReceived != null)
            {
                ContinueReceived(null, args);
            }
        }

        /// <summary>
        /// Raise the StopReceived event
        /// </summary>
        /// <param name="args"></param>
        public void OnStopReceived(ExternalDataEventArgs args)
        {
            if (StopReceived != null)
            {
                StopReceived(null, args);
            }
        }

        #endregion
    }
}
```

In addition to implementing the two IPersistenceDemo events, the service includes two methods to raise the events. These methods will be invoked by the host application and are not available to workflows since they are not part of the interface.

## Implementing the Workflow

To implement the example workflow, add a new sequential workflow to the SharedWorkflows project and name it PersistenceDemoWorkflow. The workflow requires one property that will be referenced by a condition of the WhileActivity, so it's best to add the property before you begin the visual design of the workflow. Listing 8-3 shows the PersistenceDemoWorkflow.cs file after the property has been added.

**Listing 8-3.** *PersistenceDemoWorkflow.cs After Adding a Property*

```
using System;
using System.Workflow.Activities;

namespace SharedWorkflows
{
    /// <summary>
    /// A workflow that demonstrates the behavior of
    /// persistence services
    /// </summary>
    public sealed partial class PersistenceDemoWorkflow :
        SequentialWorkflowActivity
    {
        private Boolean _isComplete = false;

        public Boolean IsComplete
        {
            get { return _isComplete; }
            set { _isComplete = value; }
        }

        public PersistenceDemoWorkflow()
        {
            InitializeComponent();
        }

    }
}
```

The visual design of the workflow is fairly simple. Here's the big picture: The workflow will start with a WhileActivity, with a CompensatableSequenceActivity as its child. Under the CompensatableSequenceActivity, there will be a ListenActivity with two EventDrivenActivity branches. Each EventDrivenActivity will contain a single HandleExternalEventActivity that listens for an event from the local service.

To implement the workflow, first switch to the visual workflow designer view, and then drag and drop a WhileActivity onto the empty workflow. Add a declarative rule condition to the WhileActivity and give it a ConditionName of checkIsComplete. Enter an Expression that checks the value of the IsComplete workflow property like this:

```
!this.IsComplete
```

This condition causes the `WhileActivity` to repeat its child activity until the `IsComplete` property is set to `true`.

Next, add a `CompensatableSequenceActivity` as the direct child of the `WhileActivity`.

---

■ **Note** We don't actually need the `CompensatableSequenceActivity` for this workflow to operate correctly. However, adding it here causes additional persistence service methods to be invoked each time the `CompensatableSequenceActivity` completes. You will see these persistence service methods in action later in this chapter when you develop your own persistence service. That example uses this same workflow to demonstrate a custom persistence service. Adding the `CompensatableSequenceActivity` now is easier than adding it later.

---

You can now add a `ListenActivity` to the `CompensatableSequenceActivity`, and add a `HandleExternalEventActivity` under each `EventDrivenActivity`. Set the `InterfaceType` for both of the `HandleExternalEventActivity` instances to `SharedWorkflows.IPersistenceDemo`.

Rename the `HandleExternalEventActivity` instance on the left to `handleContinueReceived` and set the `EventName` property to `ContinueReceived`. You don't need to add any code to handle this event. The act of raising the event is enough to demonstrate the behavior of the persistence service when an event is received.

Rename the `HandleExternalEventActivity` instance on the right to `handleStopReceived` and set the `EventName` to `StopReceived`. When this event is received, you need to set the `IsComplete` workflow property to `true`. To do this, double-click the activity to add a handler for the `Invoked` event named `handleStopReceived_Invoked`. You will add code to this handler in the next step. This completes the visual design of the workflow. Figure 8-1 shows the completed workflow.

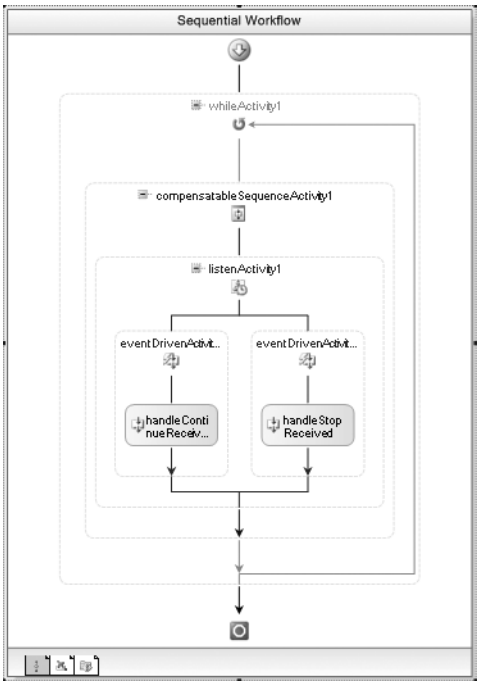


**Figure 8-1.** *Complete PersistenceDemoWorkflow*

Listing 8-4 shows the completed PersistenceDemoWorkflow.cs file, including the event handling code you need to add to the handleStopReceived_Invoked method.

**Listing 8-4.** *Complete PersistenceDemoWorkflow.cs File*

```
using System;
using System.Workflow.Activities;

namespace SharedWorkflows
{
    /// <summary>
    /// A workflow that demonstrates the behavior of
    /// persistence services
    /// </summary>
    public sealed partial class PersistenceDemoWorkflow :
        SequentialWorkflowActivity
    {
        private Boolean _isComplete = false;

        public Boolean IsComplete
        {
            get { return _isComplete; }
            set { _isComplete = value; }
        }

        public PersistenceDemoWorkflow()
        {
            InitializeComponent();
        }

        private void handleStopReceived_Invoked(
            object sender, ExternalDataEventArgs e)
        {
            //tell the WhileActivity to stop
            _isComplete = true;
        }
    }
}
```

# Implementing the Host Application

The host application is a Windows Forms application that will allow you to start and then interact with multiple workflow instances. The form will have a DataGridView that lists available workflow instances along with a status message for each instance. The DataGridView is bound to a collection of objects that contain the workflow instance ID and the status message. The user interface includes buttons that permit you to start a new workflow instance, or interact with an existing workflow instance by raising the ContinueReceived or StopReceived events.

To begin the host application, add a new Windows Application project to the current solution and name the project PersistenceDemo. Since this project isn't created from a workflow project template, you'll need to add references to these assemblies yourself:

- System.Workflow.Activities
- System.Workflow.ComponentModel

- System.Workflow.Runtime
- SharedWorkflows
- Bukovics.Workflow.Hosting

The SharedWorkflows and Bukovics.Workflow.Hosting references are project references. The Bukovics.Workflow.Hosting project was originally developed in Chapter 4 and contains a set of workflow manager classes that assist with hosting duties.

## Implementing a Class for Displaying Workflow Status

Before you begin the visual design of the host application, there is one nonvisual class that you need to add. The application will use a DataGridView to display status information for the available workflows. You need to implement a class that contains the information to display. Instances of this class will be created and placed in a collection that is databound to the DataGridView.

Add a new C# class to the PersistenceDemo project and name it Workflow. Listing 8-5 shows the completed code for this class.

**Listing 8-5.** *Complete Workflow.cs File*

```
using System;

namespace PersistenceDemo
{
    /// <summary>
    /// Used for display of workflow status in a DataGridView
    /// </summary>
    public class Workflow
    {
        private Guid _instanceId = Guid.Empty;
        private String _statusMessage = String.Empty;
        private Boolean _isCompleted;

        public Guid InstanceId
        {
            get { return _instanceId; }
            set { _instanceId = value; }
        }

        public String StatusMessage
        {
            get { return _statusMessage; }
            set { _statusMessage = value; }
        }

        public Boolean IsCompleted
        {
            get { return _isCompleted; }
            set { _isCompleted = value; }
        }
    }
}
```

## Designing the User Interface

The visual design of `Form1` in the `PersistenceDemo` project looks like Figure 8-2.

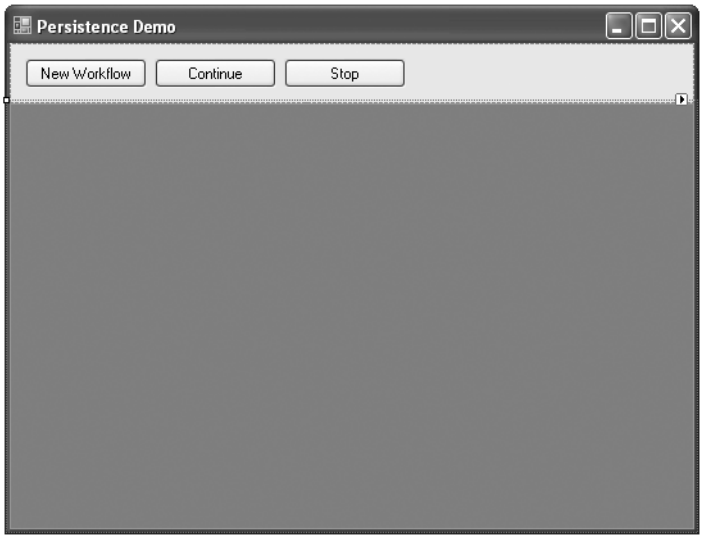


**Figure 8-2.** *Windows Forms designer for PersistenceDemo Form1*

Table 8-1 lists the set of visual controls that you'll need to add to the form.

**Table 8-1.** *Form1 Controls*

| Control Type | Name | Events to Handle | Description |
|---|---|---|---|
| `DataGridView` | `dataGridView1` | `SelectionChanged` | Bound to a collection of objects that displays the list of available workflow instances along with their status. The `DataGridView` is the large gray area shown in the lower portion of the form. |
| `Button` | `btnNewWorkflow` | `Click` | Starts a new workflow instance. |
| `Button` | `btnContinue` | `Click` | Raises the `ContinueReceived` event. |
| `Button` | `btnStop` | `Click` | Raises the `StopReceived` event. |

After adding and naming the visual controls, add event handles for the events listed in Table 8-1. If you use the suggested control names, the names for the event handlers should match the example code shown in Listing 8-6.

## Adding Code to the Form

Unlike many of the other examples in this book, this host application contains a fair amount of code. To make it clearer how all of the pieces of code work together, Listing 8-6 shows the completed code for the `Form1.cs` file with comments embedded within the listing to highlight some of the more interesting sections of code.

A number of instance variables are declared in the `Form1` class shown in Listing 8-6. Table 8-2 identifies the variables and their usage.

**Table 8-2.** *Form1 Instance Variables*

| Variable | Description |
|---|---|
| `_workflowManager` | Holds a reference to the `WorkflowManager` object that wraps the `WorkflowRuntime`. |
| `_persistence` | Holds a reference to the persistence service. |
| `_persistenceDemoService` | Holds a reference to the local service that implements the events. |
| `_workflows` | A dictionary of `Workflow` objects that are keyed by the workflow instance ID. This collection is bound to the `DataGridView` for display. |
| `_selectedWorkflow` | The `Workflow` object that is currently selected in the `DataGridView`. |

**Listing 8-6.** *Complete Form1.cs File*

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Windows.Forms;

using System.Workflow.Activities;
using System.Workflow.Runtime;
using System.Workflow.Runtime.Hosting;

using Bukovics.Workflow.Hosting;
using SharedWorkflows;

namespace PersistenceDemo
{
    /// <summary>
    /// The persistence demo application
    /// </summary>
    public partial class Form1 : Form
    {
        private WorkflowRuntimeManager _workflowManager;
        private WorkflowPersistenceService _persistence;
        private PersistenceDemoService _persistenceDemoService;
        private Dictionary<Guid, Workflow> _workflows
            = new Dictionary<Guid, Workflow>();
        private Workflow _selectedWorkflow;
```

```
public Form1()
{
    InitializeComponent();
}

#region Initialization and shutdown

/// <summary>
/// Initialize the workflow runtime during startup
/// </summary>
/// <param name="e"></param>
protected override void OnLoad(EventArgs e)
{
    base.OnLoad(e);
    //create workflow runtime and manager
    _workflowManager = new WorkflowRuntimeManager(
        new WorkflowRuntime());

    //add services to the workflow runtime
    AddServices(_workflowManager.WorkflowRuntime);

    _workflowManager.WorkflowRuntime.WorkflowCreated
        += new EventHandler<WorkflowEventArgs>(
            WorkflowRuntime_WorkflowCreated);
    _workflowManager.WorkflowRuntime.WorkflowCompleted
        += new EventHandler<WorkflowCompletedEventArgs>(
            WorkflowRuntime_WorkflowCompleted);
    _workflowManager.WorkflowRuntime.WorkflowPersisted
        += new EventHandler<WorkflowEventArgs>(
            WorkflowRuntime_WorkflowPersisted);
    _workflowManager.WorkflowRuntime.WorkflowUnloaded
        += new EventHandler<WorkflowEventArgs>(
            WorkflowRuntime_WorkflowUnloaded);
    _workflowManager.WorkflowRuntime.WorkflowLoaded
        += new EventHandler<WorkflowEventArgs>(
            WorkflowRuntime_WorkflowLoaded);
    _workflowManager.WorkflowRuntime.WorkflowIdled
        += new EventHandler<WorkflowEventArgs>(
            WorkflowRuntime_WorkflowIdled);

    //initially disable these buttons until a workflow
    //is selected in the data grid view
    btnContinue.Enabled = false;
    btnStop.Enabled = false;

    //start the runtime prior to checking for any
    //existing workflows that have been persisted
    _workflowManager.WorkflowRuntime.StartRuntime();

    //load information about any workflows that
    //have been persisted
    RetrieveExistingWorkflows();
}
```

The initialization of the workflow runtime is done in the overridden `OnLoad` method. This method is invoked when the form is first loaded. The code in this method creates an instance of the `WorkflowRuntime` and wraps it in a `WorkflowRuntimeManager`. `WorkflowRuntimeManager` was developed in Chapter 4 and assists with some of the routine hosting tasks.

The code in the `OnLoad` method also adds handlers for a number of the `WorkflowRuntime` events. Events such as `WorkflowCreated`, `WorkflowPersisted`, and `WorkflowUnloaded` are handled in order to update the display when the status of a workflow instance changes.

During startup the `OnLoad` method executes the `AddServices` method. This is where an instance of the `SqlWorkflowPersistenceService` class is created and added to the workflow runtime.

```
/// <summary>
/// Add any services needed by the runtime engine
/// </summary>
/// <param name="instance"></param>
private void AddServices(WorkflowRuntime instance)
{
    //use the standard SQL Server persistence service
    String connStringPersistence = String.Format(
        "Initial Catalog={0};Data Source={1};Integrated Security={2};",
        "WorkflowPersistence", @"localhost\SQLEXPRESS", "SSPI");
    _persistence =
        new SqlWorkflowPersistenceService(connStringPersistence, true,
            new TimeSpan(0, 2, 0), new TimeSpan(0, 0, 5));
    instance.AddService(_persistence);

    //add the external data exchange service to the runtime
    ExternalDataExchangeService exchangeService
        = new ExternalDataExchangeService();
    instance.AddService(exchangeService);

    //add our local service
    _persistenceDemoService = new PersistenceDemoService();
    exchangeService.AddService(_persistenceDemoService);
}
```

The `AddServices` method first constructs a SQL Server connection string. This code assumes that the name of the database is `WorkflowPersistence` and that it is managed by a local copy of SQL Server Express. It also assumes that integrated security is used, eliminating the need to specify a SQL Server login and password.

Several overloaded constructors are available for the `SqlWorkflowPersistenceService` class. The one used here expects a connection string followed by a `Boolean` value and two `TimeSpan` values. The `Boolean` value is the `unloadOnIdle` parameter. A value of `true` instructs the persistence service to unload any idle workflows from memory after they have been persisted.

The first `TimeSpan` value indicates the amount of time that an instance of `SqlWorkflowPersistenceService` should maintain a lock on a workflow. This is primarily used in situations where you have multiple workflow hosts that are all processing workflows from a common persistence database. For this demonstration application, the value entered here doesn't really matter.

The second `TimeSpan` value sets the `loadingInterval` property and determines how often the persistence service polls for workflows that have an expired `DelayActivity`. The code sets this to five seconds, which is a fairly aggressive value; every five seconds, the persistence service will examine the persisted workflows that have been unloaded due to a `DelayActivity`. If the delay has expired, they are candidates to be loaded back into memory and execution will resume.

■ **Tip** This application loads the persistence service in code, but you can also accomplish the same thing with entries in the application configuration file (`App.Config`). For further information on how to do this, refer to Chapter 4.

After loading the persistence service, an instance of the `PersistenceDemoService` class is also created and added to the workflow runtime engine. This is the local service providing the external events that are handled by the workflow.

```
/// <summary>
/// Perform cleanup during application shutdown
/// </summary>
/// <param name="e"></param>
protected override void OnFormClosed(FormClosedEventArgs e)
{
    base.OnFormClosed(e);
    //cleanup the workflow runtime
    if (_workflowManager != null)
    {
        _workflowManager.Dispose();
    }
}

#endregion

#region Workflow event handling

void WorkflowRuntime_WorkflowCreated(object sender,
    WorkflowEventArgs e)
{
    UpdateDisplay(e.WorkflowInstance.InstanceId, "Created");
}

void WorkflowRuntime_WorkflowIdled(object sender,
    WorkflowEventArgs e)
{
    UpdateDisplay(e.WorkflowInstance.InstanceId, "Idled");
}

void WorkflowRuntime_WorkflowLoaded(object sender,
    WorkflowEventArgs e)
{
    UpdateDisplay(e.WorkflowInstance.InstanceId, "Loaded");
}

void WorkflowRuntime_WorkflowUnloaded(object sender,
    WorkflowEventArgs e)
{
    UpdateDisplay(e.WorkflowInstance.InstanceId, "Unloaded");
}
```

```
void WorkflowRuntime_WorkflowPersisted(object sender,
    WorkflowEventArgs e)
{
    UpdateDisplay(e.WorkflowInstance.InstanceId, "Persisted");
}

void WorkflowRuntime_WorkflowCompleted(object sender,
    WorkflowCompletedEventArgs e)
{
    UpdateCompletedWorkflow(e.WorkflowInstance.InstanceId);
    UpdateDisplay(e.WorkflowInstance.InstanceId, "Completed");
}

private delegate void UpdateDelegate();

/// <summary>
/// Update the status message for a workflow
/// </summary>
/// <param name="instanceId"></param>
/// <param name="statusMessage"></param>
private void UpdateDisplay(Guid instanceId, String statusMessage)
{
    UpdateDelegate theDelegate = delegate()
    {
        Workflow workflow = GetWorkflow(instanceId);
        workflow.StatusMessage = statusMessage;
        RefreshData();
        //slow things down so you can see the status changes
        System.Threading.Thread.Sleep(1000);
    };

    //execute the anonymous delegate on the UI thread
    this.Invoke(theDelegate);
}

/// <summary>
/// Updating the bindings for the DataGridView
/// </summary>
private void RefreshData()
{
    //setup binding for DataGridView
    BindingSource source = new BindingSource();
    dataGridView1.DataSource = source;
    source.DataSource = _workflows.Values;

    dataGridView1.Columns[0].MinimumWidth = 250;
    dataGridView1.Columns[1].MinimumWidth = 140;
    dataGridView1.Columns[2].MinimumWidth = 40;

    dataGridView1.Refresh();
}
```

The `UpdateDisplay` and `UpdateCompletedWorkflow` methods both refresh the display by updating the bound collection of `Workflow` objects. The `UpdateDisplay` method is invoked by all of the handlers

for the workflow runtime event (e.g., `WorkflowCreated`, `WorkflowPersisted`, `WorkflowCompleted`). It locates the workflow instance in the internal collection (if it exists) using the private `GetWorkflow` method. It then updates the status message for the `Workflow` object.

Notice that the code to update the data bound collection is executed on the UI thread. This is important since you can update a UI control only from the thread that created it. The actual code to execute is wrapped in an anonymous delegate in order to keep the code all in one place.

The call to `Thread.Sleep(1000)` is used to slow down the display so that you can actually see the status changes. Each workflow will typically cycle through a number of status changes very quickly and you would not see those changes without this short delay.

---

■ **Caution** Be careful when updating the user interface from workflow runtime event handling code. The workflow events are raised from a workflow thread, not the user interface thread.

---

```
/// <summary>
/// Mark a workflow as completed
/// </summary>
/// <param name="instanceId"></param>
private void UpdateCompletedWorkflow(Guid instanceId)
{
    UpdateDelegate theDelegate = delegate()
    {
        Workflow workflow = GetWorkflow(instanceId);
        workflow.IsCompleted = true;
    };

    //execute the anonymous delegate on the UI thread
    this.Invoke(theDelegate);
}
```

The `UpdateCompletedWorkflow` method is only invoked by the handler for the `WorkflowCompleted` event. It updates the `IsCompleted` property of the `Workflow` object to indicate that the workflow has completed. This prevents the application from raising additional events for the workflow.

Both of these methods call the `RefreshData` method, which binds the collection of `Workflow` objects to the `DataGridView`.

```
#endregion

#region UI event handlers and management

/// <summary>
/// Start a new workflow
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnNewWorkflow_Click(object sender, EventArgs e)
{
    //start a new workflow instance
    _workflowManager.StartWorkflow(
        typeof(PersistenceDemoWorkflow), null);
}
```

The Click event handlers for the three buttons are used to start new instances of PersistenceDemoWorkflow or raise one of the local service events. The btnNewWorkflow_Click method is responsible for starting a new workflow instance.

```
/// <summary>
/// Raise the Continue event through the local service
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnContinue_Click(object sender, EventArgs e)
{
    if (_selectedWorkflow != null)
    {
        _persistenceDemoService.OnContinueReceived(
            new ExternalDataEventArgs(_selectedWorkflow.InstanceId));
    }
}
```

The btnContinue_Click method raises the local service ContinueReceived event.

```
/// <summary>
/// Raise the Stop event through the local service
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnStop_Click(object sender, EventArgs e)
{
    if (_selectedWorkflow != null)
    {
        _persistenceDemoService.OnStopReceived(
            new ExternalDataEventArgs(_selectedWorkflow.InstanceId));
    }
}
```

The btnStop_Click looks very similar to the code for btnContinue_click. The only difference is the name of the local service method that it executes. Both of these methods reference the _selectedWorkflow variable. This variable contains the Workflow object that was selected in the DataGridView and saved by the dataGridView1_SelectionChanged method.

```
/// <summary>
/// The selected workflow has changed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void dataGridView1_SelectionChanged(
    object sender, EventArgs e)
{
    //save the selected workflow instance
    if (dataGridView1.SelectedRows.Count > 0)
    {
        DataGridViewRow selectedRow = dataGridView1.SelectedRows[0];
        _selectedWorkflow = selectedRow.DataBoundItem as Workflow;
        SetButtonState();
    }
}
```

Before you can raise the ContinueReceived or StopReceived events, you must first identify the workflow instance that should receive the event. This is accomplished by selecting one of the available workflow instances shown in the DataGridView control. The code that saves the selected workflow instance is in the SelectionChanged event handler for the DataGridView (method dataGridView1_SelectionChanged).

```
        /// <summary>
        /// Enable / Disable buttons
        /// </summary>
        private void SetButtonState()
        {
            if (_selectedWorkflow != null)
            {
                btnContinue.Enabled = !(_selectedWorkflow.IsCompleted);
                btnStop.Enabled = !(_selectedWorkflow.IsCompleted);
            }
            else
            {
                btnContinue.Enabled = false;
                btnStop.Enabled = false;
            }
        }

        #endregion

        #region Collection Management

        /// <summary>
        /// Retrieve a workflow from our local collection
        /// </summary>
        /// <param name="instanceId"></param>
        /// <returns></returns>
        private Workflow GetWorkflow(Guid instanceId)
        {
            Workflow result = null;
            if (_workflows.ContainsKey(instanceId))
            {
                result = _workflows[instanceId];
            }
            else
            {
                //create a new instance
                result = new Workflow();
                result.InstanceId = instanceId;
                _workflows.Add(result.InstanceId, result);
            }
            return result;
        }

        /// <summary>
        /// Identify all persisted workflows
        /// </summary>
```

```
        private void RetrieveExistingWorkflows()
        {
            _workflows.Clear();
            //retrieve a list of workflows that have been persisted

            foreach (SqlPersistenceWorkflowInstanceDescription workflowDesc
                in ((SqlWorkflowPersistenceService)_persistence).GetAllWorkflows())
            {
                Workflow workflow = new Workflow();
                workflow.InstanceId = workflowDesc.WorkflowInstanceId;
                workflow.StatusMessage = "Unloaded";
                _workflows.Add(workflow.InstanceId, workflow);
            }

            if (_workflows.Count > 0)
            {
                RefreshData();
            }
        }
```

The `RetrieveExistingWorkflows` method is executed from the `OnLoad` method. The purpose of this method is to retrieve a list of all workflow instances that are currently persisted. This is done in order to present a list of available workflows when the application first starts.

The `SqlWorkflowPersistenceService` contains a method named `GetAllWorkflows` that returns a collection of `SqlPersistenceWorkflowInstanceDescription` objects. Each one of these objects represents a persisted workflow. This information is used to build a collection of `Workflow` objects that are bound to the `DataGridView` and displayed.

```
        #endregion

    }
}
```

## Testing the Application

Once all of these individual application pieces are in place, you should be able to start the `PersistenceDemo` application. Initially, there are no workflows shown, so begin the demonstration by clicking the New Workflow button. This starts a workflow and updates the display to show the workflow instance ID and status message. The application looks like Figure 8-3 after I start a new workflow.

---

■ **Note** Since the workflow instance ID is a `Guid` and should be globally unique, you should always see different instance IDs than the ones shown here.

---

Although the StatusMessage column in Figure 8-3 shows a status of Unloaded, the workflow actually cycled through several status messages before it was unloaded. If you watch very closely as you start a new workflow, you will see that the workflow status quickly cycles through Created, Idled, Persisted, and finally Unloaded. These status messages displayed as `WorkflowRuntime` events are received for the instance.

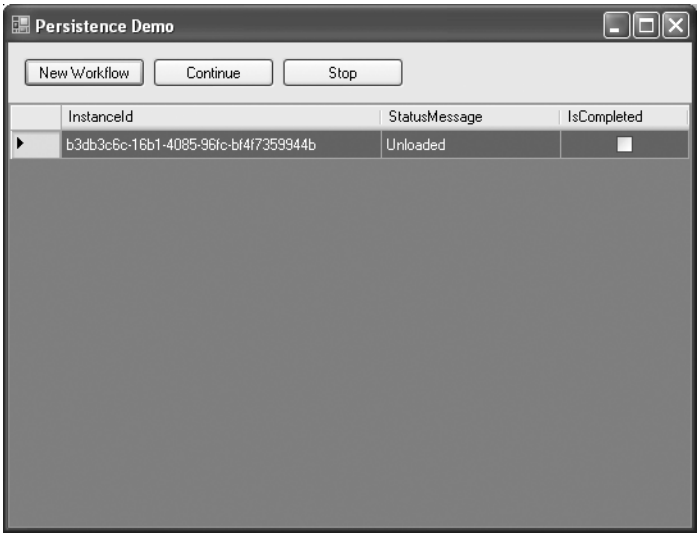


**Figure 8-3.** *PersistenceDemo application with one workflow*

If you remember the design of the workflow, this series of events makes sense. After the workflow is created, it executes the `ListenActivity`, which contains activities to handle two different external events. As the workflow is waiting for these events, it is considered idle. This accounts for the immediate move from Created to Idled. Since a persistence service has been loaded, idled workflows are automatically persisted, so the workflow then displays the Persisted status message. Finally, because the persistence service is constructed with a value that instructed it to unload idled workflows, the final status message is Unloaded.

Now that you have a workflow, you can raise the `ContinueReceived` event by clicking the Continue button. When you do so, watch the StatusMesssage again. You should see the workflow quickly cycle between Loaded, Idled, Persisted, and Unloaded. What would account for this series of events?

Remember that the workflow contains a `WhileActivity` that continues to execute until the `IsComplete` property is set to `true`. That property is set in the workflow when the `StopReceived` event is raised. When you raised the `ContinueReceived` event, the persistence service loaded the workflow back into memory in order to process the event. After handling the event, the workflow was again considered idle and was persisted and unloaded once again. This same processing will continue each time you click the Continue button.

To complete the workflow, click the Stop button. This time, the workflow cycles through status messages of Loaded, Persisted, and finally Completed. The application should now look like Figure 8-4.

The really interesting resultsoccur when you start multiple workflows and then close the application without stopping them with the Stop button. Click the New Workflow button three times, waiting for each instance to reach the Unloaded state before starting the next workflow. The application should now look like Figure 8-5.

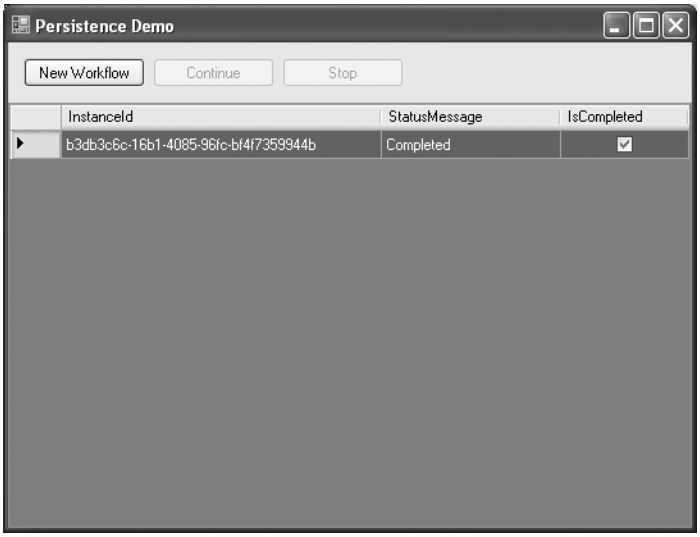


**Figure 8-4.** *PersistenceDemo application with completed workflow*

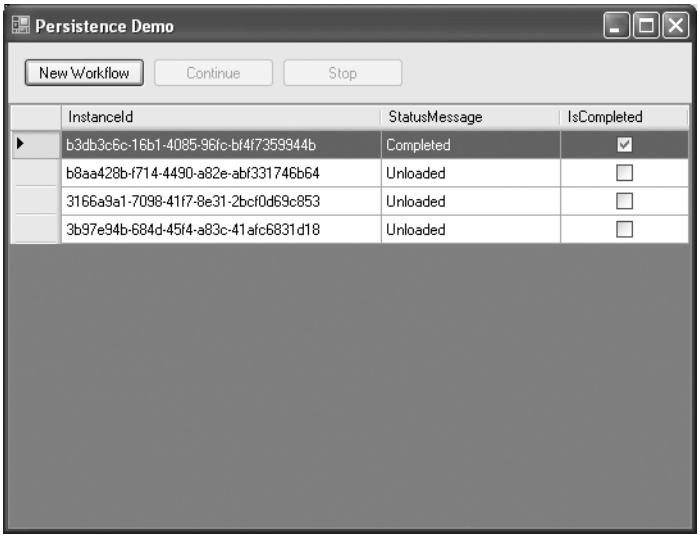


**Figure 8-5.** *PersistenceDemo application with four workflows*

Now shut down the application and immediately restart it. The application should look like Figure 8-6, showing the three incomplete workflows.

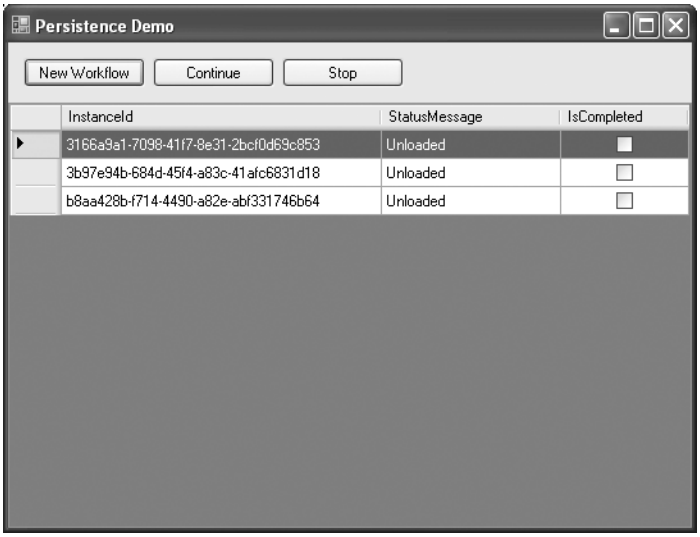


**Figure 8-6.** *PersistenceDemo application with incomplete workflows*

This list of workflows was retrieved during startup by calling the `GetAllWorkflows` method of the persistence service.

---

■ **Note** The `GetAllWorkflows` method doesn't load all of the workflows back into memory. It simply identifies the workflows that are persisted. Each one of these workflows can then be loaded back into memory by raising one of the local service events.

---

You can now raise events for these three workflows in the same way you did for the first workflow. Just select a workflow, and then click the Continue or Stop button. If you click Continue, the selected workflow will cycle through the same set of status messages you saw previously. If you click the Stop button, the selected workflow will receive the `StopReceived` event and will complete.

This simple application illustrates several things about the behavior of the SQL Server persistence service.

- Once the persistence service is loaded, workflows are automatically persisted according to a set of internal WF rules.
- A workflow is removed by the persistence service when it is complete. The workflow that you previously completed by clicking the Stop button is now gone when you restart the application.
- Persisted workflows are not dependent on a particular instance of an application. They now live in the durable store that is managed by the persistence service and can be reloaded into memory for execution by any workflow application.
- Workflows that are waiting to receive an event are considered idle. As far as the persistence service and the workflow runtime are concerned, it doesn't matter how long these workflows are idle. It could take minutes, days, weeks, or even longer for you to raise one of the local service events. While they are waiting, they are safely persisted.

# Implementing a Custom Persistence Service

The easiest way to add persistence to your workflows is to use the standard SQL Server persistence service that was just demonstrated. However, this service might not always meet your needs. Perhaps you can't or don't want to use SQL Server. You might already be developing your application around another relational database engine (e.g., MySQL, Oracle). Or you might want to use SQL Server, but need to use a different database schema, perhaps using application tables that you've already defined. Or you might need persistence, but don't want the overhead of a relational database at all; your application might work just fine persisting workflows as binary or XML files in a local directory.

The solution is to develop your own persistence service. There are really just a few steps that you need to follow in order to implement and use your own persistence service.

1. Derive your custom persistence class from `WorkflowPersistenceService` (in the `System.Workflow.Runtime.Hosting` namespace).
2. Provide implementations for all of the abstract methods in the base class. Each of these methods is called at various times during the life of a workflow to save or retrieve the state of an entire workflow or part of it.
3. Load the custom persistence service into the workflow runtime in the same manner as the standard service.

## Understanding the Abstract Methods

The largest task you face is providing implementations for all of the abstract methods defined by the `WorkflowPersistenceService` class. For this reason, each of the abstract methods is briefly described in the sections that follow.

### SaveWorkflowInstanceState

This method is called by the workflow runtime engine to persist the state of the entire workflow. The root `Activity` of the workflow is one of the arguments passed to this method. To save the state of the workflow, you must call the `Save` method on this root `Activity`. This returns a `Stream`, which you can then persist using any type of durable store that you like.

Calling the `Save` method is an important step, because it guarantees that the workflow will be persisted in a form that you can later pass to the static `Load` method of the `Activity` class. This takes place in the `LoadWorkflowInstanceState` method.

When you persist the entire workflow, you will need a primary key to use when referencing the workflow in your durable store. The obvious choice for this key is the workflow instance ID. This `Guid` uniquely identifies the workflow, and using it will avoid any potential problems associated with some other identification scheme. The workflow instance ID is also passed back to you when the workflow runtime requests that you reload a workflow instance.

This method is also called when a workflow is completed or terminates. This final call is your last chance to perform additional processing with the persisted copy of the workflow. If you follow the same approach as the standard persistence service for SQL Server, you will remove the workflow from the durable store when it is completed or terminated. But if your application requires the retention of completed workflows, you can move the workflow to a different set of database tables or archive it in some other way. You can determine the workflow status by calling the static `GetWorkflowStatus` method of the base `WorkflowPersistenceService` class.

The `SaveWorkflowInstanceState` method is also passed a `Boolean` argument named `unlock`. This parameter indicates whether you should unlock the workflow instance after it is persisted. It is your choice to implement workflow locking or not. If implemented, a locking mechanism would record the fact that a workflow instance has been loaded by a particular instance of the workflow runtime

and is actively executing. A lock that you record for the workflow instance would prevent other instances of the workflow runtime from working with that workflow instance.

Whenever you are working with a durable store such as a database or a set of files, you should be careful to perform updates in a way that will guarantee the consistency of the data. For databases, this usually means applying individual updates as part of a larger batch of work (a transaction) and then committing all of the changes at once.

---

■ **Note** WF includes a framework for managing and committing batches of work. It works by adding work items to `WorkflowEnvironment.WorkBatch` and then later committing them. Using this mechanism guarantees that updates to your durable store are coordinated with the internal state of the workflow. Use of this WF framework is discussed in Chapter 10. In order to keep the focus of the discussion on workflow persistence, the custom persistence service in this chapter does not use this batching mechanism.

---

If you are unable to persist the state of a workflow, you should throw an exception of type `PersistenceException`.

### LoadWorkflowInstanceState

This method is called to load the previously saved state of a workflow instance. You are passed the instance ID of the workflow to load. Once you retrieve the workflow state from your durable store, you must use the static `Load` method of the `Activity` class to recreate the workflow instance. The return value of this method is the workflow (with a return type of `Activity`) that was loaded.

### SaveCompletedContextActivity

The `SaveWorkflowInstanceState` method described previously is called to persist the state of an entire workflow. In contrast, this method saves just a portion of the workflow. Its purpose is to save the activity execution context for completed activities that support compensation.

Every activity runs within an execution context. The context is simply an execution environment that contains one or more activities. The context determines the set of parent and child activities that you can safely reference. Activities that have looping or repeating behavior generate new execution contexts for each of their iterations. For example, the `WhileActivity` loops continuously as long as a specified condition is `true`. But a new activity execution context is created with each iteration. The purpose of this method is to save the state of each of those activity contexts as they complete. Think of each of these contexts as an additional checkpoint in the life of a workflow.

However, individual activity contexts are saved only if the activity supports compensation. *Compensation* is the process of rolling back changes from activities that have successfully completed. This might be needed if the individual activities successfully complete, but later in the workflow the decision is made to undo all of the updates from the individual activities. In order to roll back changes that have already completed, you need the checkpoints that have been saved by this method.

As an example, your workflow might contain a `WhileActivity` that has a `SequenceActivity` as its child (a `WhileActivity` only supports a single child). Assume for this example that the `SequenceActivity` contains several `CodeActivity` instances. As the `WhileActivity` iterates, new activity contexts are created. However, this method will not be called for each context since none of these activities support compensation. If you replace the `SequenceActivity` with a `CompensatableSequenceActivity`, the situation changes. As the name implies, `CompensatableSequenceActivity` supports compensation. Now at the end of each iteration, this method is called, providing you with the opportunity to save a context checkpoint.

If the workflow requires compensation (the actual undoing of the updates), the `LoadCompletedContextActivity` method is called to reload each activity context that was saved by this method.

Each activity context has its own unique context ID (a `Guid`) that is different from the workflow instance ID. When saving a completed context with this method, you should provide a mechanism to associate each context ID with the workflow instance ID. This is needed when it is time to reload a persisted context and also to remove all persisted data for a workflow when it is completed.

Just as you do in the `SaveWorkflowInstanceState` method, you must call the `Save` method of the activity to serialize the activity to a format that can be reloaded.

### LoadCompletedContextActivity

This method is the reciprocal of `SaveCompletedContextActivity`. It loads completed activity contexts that have been previously saved. It is invoked if and when a workflow requires compensation (undoing) of completed changes.

This method is passed an `outerActivity` parameter that is the parent `Activity` of the context to be loaded. You must use the static `Load` method of the `Activity` class to load the activity. However, you need to pass this `outerActivity` to the method in order to properly associate the loaded context with the correct parent activity.

### UnlockWorkflowInstanceState

This method is called to unlock a workflow instance outside of the normal load and save methods. For instance, if a workflow is aborted, this method would be called to clean up any locks that you have for the workflow instance.

Since this is an abstract method, you must provide an implementation for it in your derived class. However, you only need to provide actual unlocking logic if you choose to implement a locking mechanism.

### UnloadOnIdle

This method returns a `Boolean` value that indicates whether the workflow that is passed in as an argument should be unloaded when it becomes idle. You can choose to always return `true` or `false`, or make your decision based on each individual workflow.

## Implementing the Service

In this section, you will develop a custom persistence service. This service is extremely simple, serializing the workflow state to a set of files in the current directory. Although it is simple, this example service does demonstrate the basic steps necessary to implement your own service.

Your custom persistence service might need to be much more elaborate than this one. In particular, this service does not implement workflow instance locking, so it doesn't support multiple workflow hosts that work with the same set of persisted workflows. If you are using a database as your durable store, you should also add support for transactions and batching of work to maintain the consistency of your data.

---

■ **Note** Please refer to Chapter 10 for more information on transactions and batching of work.

---

To implement the service, add a new C# class to the `SharedWorkflows` project and name it `FileWorkflowPersistenceService`. Derive the class from `WorkflowPersistenceService` and provide

implementations for all of the abstract methods discussed previously. Listing 8-7 presents the completed code for the `FileWorkflowPersistenceService.cs` file. Selected portions of the code are discussed within the listing.

**Listing 8-7.** *Complete FileWorkflowPersistenceService.cs File*

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Workflow.Runtime;
using System.Workflow.Runtime.Hosting;
using System.Workflow.ComponentModel;

namespace SharedWorkflows
{
    /// <summary>
    /// A file-based workflow persistence service
    /// </summary>
    public class FileWorkflowPersistenceService
        : WorkflowPersistenceService
    {
        private String _path = Environment.CurrentDirectory;

        #region Abstract method implementations

        /// <summary>
        /// Persist the current state of the entire workflow
        /// </summary>
        /// <param name="rootActivity"></param>
        /// <param name="unlock"></param>
        protected override void SaveWorkflowInstanceState(
            Activity rootActivity, bool unlock)
        {
            //get the workflow instance ID
            Guid instanceId = WorkflowEnvironment.WorkflowInstanceId;

            //determine the status of the workflow
            WorkflowStatus status =
                WorkflowPersistenceService.GetWorkflowStatus(rootActivity);
            switch (status)
            {
                case WorkflowStatus.Completed:
                case WorkflowStatus.Terminated:
                    //delete the persisted workflow
                    DeleteWorkflow(instanceId);
                    break;
                default:
                    //save the workflow
                    Serialize(instanceId, Guid.Empty, rootActivity);
                    break;
            }
        }
```

The SaveWorkflowInstanceState method contains the code needed to persist the state of the entire workflow. This method calls the static WorkflowEnvironment.WorkflowInstanceId method to retrieve the instance ID for the current workflow. You will see later in the code that this ID is used as part of the file name when persisting the workflow.

This method also determines if it should save the workflow state or delete it based on the workflow status. The status is retrieve by calling the static GetWorkflowStatus method of the base WorkflowPersistenceService class. If the workflow status is Completed or Terminated, any files that have been persisted for this workflow are deleted. A private Serialize method is called to handle the actual persistence of the workflow state.

```
/// <summary>
/// Load an entire workflow
/// </summary>
/// <param name="instanceId"></param>
/// <returns></returns>
protected override Activity LoadWorkflowInstanceState(
    Guid instanceId)
{
    Activity activity = Deserialize(instanceId, Guid.Empty, null);
    if (activity == null)
    {
        ThrowException(instanceId,
            "Unable to deserialize workflow", null);
    }
    return activity;
}
```

The LoadWorkflowInstanceState method loads the requested workflow instance by calling the private Deserialize method.

```
/// <summary>
/// Persist a completed activity context
/// </summary>
/// <remarks>
/// This persists completed activities that were part
/// of an execution scope. Example:  Activities
/// within a CompensatableSequenceActivity.
/// </remarks>
/// <param name="activity"></param>
protected override void SaveCompletedContextActivity(
    Activity activity)
{
    //get the workflow instance ID
    Guid instanceId = WorkflowEnvironment.WorkflowInstanceId;

    //get the context ID, which identifies the activity scope
    //within the workflow instance
    Guid contextId = (Guid)activity.GetValue(
        Activity.ActivityContextGuidProperty);

    //persist the activity for this workflow
    Serialize(instanceId, contextId, activity);
}
```

The SaveCompletedContextActivity method is responsible for saving a completed activity execution context if the activity supports compensation. This method requires two different identifiers to save the context. It retrieves the workflow instance ID and also a separate activity context ID. Both IDs are passed to the private Serialize method that saves the context.

```
/// <summary>
/// Load an activity context
/// </summary>
/// <param name="scopeId"></param>
/// <param name="outerActivity"></param>
/// <returns></returns>
protected override Activity LoadCompletedContextActivity(
    Guid scopeId, Activity outerActivity)
{
    //get the workflow instance ID
    Guid instanceId = WorkflowEnvironment.WorkflowInstanceId;

    Activity activity = Deserialize(instanceId, scopeId, outerActivity);
    if (activity == null)
    {
        ThrowException(instanceId,
            "Unable to deserialize activity", null);
    }
    return activity;
}
```

The LoadCompletedContextActivity retrieves a completed activity context and loads it as a child of the outerActivity parameter that was passed to the method.

```
protected override void UnlockWorkflowInstanceState(
    Activity rootActivity)
{
    //locking not implemented
}

protected override bool UnloadOnIdle(Activity activity)
{
    //always unload on idle
    return true;
}
```

This service always returns true from the UnloadOnIdle method. This means that any workflows that are idled will be removed from memory after they are persisted.

```
#endregion

#region Persistence and File Management

/// <summary>
/// Serialize the workflow or an activity context
/// </summary>
/// <param name="instanceId"></param>
/// <param name="contextId"></param>
/// <param name="activity"></param>
```

```
private void Serialize(
    Guid instanceId, Guid contextId, Activity activity)
{
    try
    {
        String fileName = GetFilePath(instanceId, contextId);
        using (FileStream stream = new FileStream(
            fileName, FileMode.Create))
        {
            activity.Save(stream);
        }
    }
    catch (ArgumentException e)
    {
        ThrowException(instanceId,
            "Serialize: Path has invalid argument", e);
    }
    catch (DirectoryNotFoundException e)
    {
        ThrowException(instanceId,
            "Serialize: Directory not found", e);
    }
    catch (Exception e)
    {
        ThrowException(instanceId,
            "Serialize: Unknown exception", e);
    }
}
```

The `Serialize` method is responsible for actually saving the workflow state or a completed activity context to a file. This method retrieves the file name using a private `GetFilePath` method. The file name always uses the workflow instance ID as the high-order part of the file name. If the entire workflow state is being saved, the complete file name is the instance ID and a `.wf` extension. If the method is saving a completed activity context, the file name contains the workflow instance ID and the context ID with a `.wfc` extension. This naming convention logically associates all of the saved context files with the parent workflow. This makes it easier to delete all files for a workflow when the workflow is completed.

```
/// <summary>
/// Deserialize a workflow or an activity context
/// </summary>
/// <param name="instanceId"></param>
/// <param name="contextId"></param>
/// <param name="rootActivity"></param>
/// <returns></returns>
private Activity Deserialize(
    Guid instanceId, Guid contextId, Activity rootActivity)
{
    Activity activity = null;
    try
    {
        String fileName = GetFilePath(instanceId, contextId);
        using (FileStream stream = new FileStream(
            fileName, FileMode.Open))
        {
```

```
                activity = Activity.Load(stream, rootActivity);
            }
        }
        catch (ArgumentException e)
        {
            ThrowException(instanceId,
                "Deserialize: Path has invalid argument", e);
        }
        catch (FileNotFoundException e)
        {
            ThrowException(instanceId,
                "Deserialize: File not found", e);
        }
        catch (DirectoryNotFoundException e)
        {
            ThrowException(instanceId,
                "Deserialize: Directory not found", e);
        }
        catch (Exception e)
        {
            ThrowException(instanceId,
                "Deserialize: Unknown exception", e);
        }
        return activity;
    }
```

The `Deserialize` method reverses the process and retrieves the requested file. It uses the static `Load` method of the `Activity` class to reload the previously saved file.

```
    /// <summary>
    /// Delete a workflow and any related activity context files
    /// </summary>
    /// <param name="instanceId"></param>
    private void DeleteWorkflow(Guid instanceId)
    {
        try
        {
            String[] files = Directory.GetFiles(
                _path, instanceId.ToString() + "*");

            foreach (String file in files)
            {
                if (File.Exists(file))
                {
                    File.Delete(file);
                }
            }
        }
        catch (ArgumentException e)
        {
            ThrowException(instanceId,
                "Delete: Path has invalid argument", e);
        }
```

```
        catch (DirectoryNotFoundException e)
        {
            ThrowException(instanceId,
                "Delete: Directory not found", e);
        }
        catch (Exception e)
        {
            ThrowException(instanceId,
                "Delete: Unknown exception", e);
        }
    }

    /// <summary>
    /// Determine the full file path
    /// </summary>
    /// <param name="instanceId"></param>
    /// <param name="contextId"></param>
    /// <returns></returns>
    private String GetFilePath(Guid instanceId, Guid contextId)
    {
        String fullPath = String.Empty;
        if (contextId == Guid.Empty)
        {
            //create a path for the entire workflow.
            //Naming convention is [instanceId].wf
            fullPath = Path.Combine(_path, String.Format("{0}.{1}",
                instanceId, "wf"));
        }
        else
        {
            //create a path for a single activity context
            //within the workflow.
            //naming convention is [instanceId].[contextId].wfc
            fullPath = Path.Combine(_path, String.Format("{0}.{1}.{2}",
                instanceId, contextId, "wfc"));
        }
        return fullPath;
    }

    #endregion

    #region Existing Workflow Management

    /// <summary>
    /// Return a list of all workflow IDs that are persisted
    /// </summary>
    /// <returns></returns>
    public List<Guid> GetAllWorkflows()
    {
        List<Guid> workflows = new List<Guid>();
        String[] files = Directory.GetFiles(_path, "*.wf");
```

```
            foreach (String file in files)
            {
                //turn the file name into a Guid
                Guid instanceId = new Guid(
                    Path.GetFileNameWithoutExtension(file));
                workflows.Add(instanceId);
            }

            return workflows;
        }
```

The `GetAllWorkflows` was added to the class in order to return a list of workflow instance IDs that are persisted. The host application can call this method to retrieve a list of available workflows.

```
        #endregion

        #region Common Error handling

        /// <summary>
        /// Throw an exception due to an error
        /// </summary>
        /// <param name="instanceId"></param>
        /// <param name="message"></param>
        /// <param name="inner"></param>
        private void ThrowException(Guid instanceId, String message,
            Exception inner)
        {
            if (inner == null)
            {
                throw new PersistenceException(
                    String.Format("Workflow: {0} Error: {1}",
                        instanceId, message));
            }
            else
            {
                throw new PersistenceException(
                    String.Format("Workflow: {0} Error: {1}: Inner: {2}",
                        instanceId, message, inner.Message), inner);
            }
        }

        #endregion
    }
}
```

## Testing the Custom Service

You can test this new persistence service using the same `PersistenceDemo` application that was used with the SQL Server persistence service earlier in the chapter. A few minor changes are needed to the code and are outlined in this section.

Open the `Form1.cs` file of the `PersistenceDemo` project and locate the `AddService` method. Remove the references to the `SqlWorkflowPersistenceService` class and the SQL Server connection string since they are no longer needed. In their place, add code to create an instance of the

FileWorkflowPersistenceService class and add it to the workflow runtime. The revised code for the AddService method is shown here:

```
private void AddServices(WorkflowRuntime instance)
{
    //use the custom file-based persistence service
    _persistence = new FileWorkflowPersistenceService();
    instance.AddService(_persistence);

    //add the external data exchange service to the runtime
    ExternalDataExchangeService exchangeService
        = new ExternalDataExchangeService();
    instance.AddService(exchangeService);

    //add our local service
    _persistenceDemoService = new PersistenceDemoService();
    exchangeService.AddService(_persistenceDemoService);
}
```

The other necessary change is to the RetrieveExistingWorkflows method. This method retrieves the list of available workflows that have been persisted. The custom persistence service implements a GetAllWorkflows method, but the return value is different from the SQL Server persistence service. The custom service returns a collection of Guid values. The revised code for the RetrieveExistingWorkflows is shown here:

```
private void RetrieveExistingWorkflows()
{
    _workflows.Clear();
    //retrieve a list of workflows that have been persisted

    foreach (Guid instanceId
        in ((FileWorkflowPersistenceService)_persistence).GetAllWorkflows())
    {
        Workflow workflow = new Workflow();
        workflow.InstanceId = instanceId;
        workflow.StatusMessage = "Unloaded";
        _workflows.Add(workflow.InstanceId, workflow);
    }

    if (_workflows.Count > 0)
    {
        RefreshData();
    }
}
```

That's it. After rebuilding the PersistenceDemo project, you should be able to run it and see the same type of behavior as the previous example that used the SQL Server persistence service.

As you start and interact with workflows, you should see the files used for persistence in the same directory as the executable. For example, when I start a new workflow, I see this file appear in the directory:

```
e9cdb9d0-ec23-4d05-86a4-9b30b977f259.wf
```

If I raise the `ContinueReceived` event by clicking the Continue button, I see an additional file created:

```
e9cdb9d0-ec23-4d05-86a4-9b30b977f259.wf
e9cdb9d0-ec23-4d05-86a4-9b30b977f259.c6893b50-9e89-48c0-ae9d-1887615fe384.wfc
```

The second file represents the completed activity context for the `CompensatableSequenceActivity` that is included in the workflow. Additional files are created each time I click the Continue button. When I click the Stop button for the workflow, all of the files for the workflow instance are deleted.

■ **Note** Remember that since these are `Guid` values, the file names will be different on your machine.

I can also create multiple workflow instances and then stop and restart the application. Just like the example that used SQL Server, the application presents a list of workflows that have been persisted. I can then interact with them individually by raising events.

Using the standard SQL Server persistence service provides you with easy, out-of-the-box support for persistence. But implementing your own custom persistence service provides you with ultimate flexibility; you are able to implement the service to meet the exact needs of your particular application.

# Summary

This chapter focused on workflow persistence. Workflow persistence in WF is implemented as one of the optional core services. This architecture allows you to use the standard SQL Server persistence service that is provided with WF, or implement your own custom persistence service.

Several good reasons to use persistence were covered in the overview section of this chapter. Following the overview of how persistence works in WF, an example application was developed that demonstrated the behavior of the standard SQL Server persistence service.

After the SQL Server persistence example, a custom persistence service was developed and used by the same application. This custom service used simple files as its durable store instead of a relational database such as SQL Server. To illustrate the ability to plug in different implementations of core services, the demo application was modified to use the custom service with minimal effort.

In the next chapter, you will learn about state machine workflows that allow you to more naturally model processes that involve human interaction.