

Windows Workflow Foundation

Hands-On Lab

Lab Manual

Lab 03 – Hosting the Windows Workflow Foundation Runtime in C#

Information in this document, including URL and other Internet Web site references, is subject to change without notice. This document supports a preliminary release of software that may be changed substantially prior to final commercial release, and is the proprietary information of Microsoft Corporation.

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, AS TO THE INFORMATION IN THIS DOCUMENT.

The entire risk of the use or the results from the use of this document remains with the user. Complying with all applicable copyright laws is the responsibility of the user.  Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document.  Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2005 Microsoft Corporation.  All rights reserved.

Microsoft, MS-DOS, Windows, Windows NT, Windows Server, SQL Server, Visual C# and Visual Studioare either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

**Contents**

[Overview 1](#_Toc219026680)

[Objectives 1](#_Toc219026681)

[System Requirements 1](#_Toc219026682)

[Setup 1](#_Toc219026683)

[Physical Folder Structure 2](#_Toc219026684)

[Code Snippets 2](#_Toc219026685)

[Starting Material 2](#_Toc219026686)

[Acronyms Used in this Lab 2](#_Toc219026687)

[Scenario 2](#_Toc219026688)

[Exercise 1 – Hosting the Workflow Runtime 3](#_Toc219026689)

[Task 1 – Create a new Console Application Project 3](#_Toc219026690)

[Task 2 – Listen for Runtime Events 6](#_Toc219026691)

[Task 3 – Configuring Runtime Services 10](#_Toc219026692)

[Exercise 2 – Using the default SQL Services 16](#_Toc219026693)

[Task 1 – Create a new Console Application Project. 17](#_Toc219026694)

[Task 2 – Adding the SQLTrackingService to the host environment 19](#_Toc219026695)

[Task 3 – Adding the SQLPersistenceService to the host environment 23](#_Toc219026696)

[Task 4 – Adding a WorkflowSchedulerService to the host environment 29](#_Toc219026697)

[Exercise 3 – Building a Custom TrackingService for Workflow Monitoring 33](#_Toc219026698)

[Task 1 – Create a new Console Application Project 33](#_Toc219026699)

[Task 2 – Create the ConsoleTrackingChannel 35](#_Toc219026700)

[Task 3 – Create our ConsoleTrackingService 40](#_Toc219026701)

[Task 4 – Run ConsoleTrackingServiceSample 44](#_Toc219026702)

[Lab Summary 46](#_Toc219026703)

Overview

Estimated time to complete this lab: **60 minutes**

This lab introduces several concepts of working with the Windows Workflow Foundation runtime. In order to make effective use of long-running workflows, experience with the WF runtime is needed. The scenarios presented in these exercises are simplistic “Hello World” style programs in order to focus on simple mechanics of using the WF components being learned.

## Objectives

This lab introduces hosting the Windows Workflow Foundation runtime. After completing this lab, you will be able to:

* Host the workflow runtime in a simple Windows console application and run a workflow using the workflow runtime.
* Run the console application to verify that workflow was loaded, ran successfully, and the runtime events are raised.
* Use the **SQL Tracking**, **SQL Persistence,** and **WorkflowSchedulerService** service to log workflow and runtime related events.
* Use an app.config file to tailor the runtime environment to meet your needs.

More information about Windows Workflow Foundation can be found at <http://msdn.microsoft.com/workflow>

## System Requirements

* Microsoft Visual Studio 2008
* SQL Server Express or other commercial version of SQL Server 2005

## Setup

1. Unzip the lab to your local hard drive. Everything needed to complete the lab is in the zip file.
2. Execute the following command file:  
   *C:\WF\WF 3.5 Labs\Lab03\Resources\CreateDatabaseObjects.cmd*   
   This installs the appropriate database tables into a local instance of SQL Server Express

**NOTE:** If you are working with a different instance of SQL Server, edit the command file to provide the correct server name to the OSQL commands.

### Physical Folder Structure

File paths referenced in this lab assume the lab is installed in the following folder:

*C:\WF\WF 3.5 Labs\Lab[Number]*

Within the ***Lab[Number]*** folder, several child folders are available:

* ***CSharp*** – The lab written for C#
* ***VB*** – The lab written for VB
* ***resources*** –Any files referenced in the lab can be found in the Resources subdirectory, including source code for custom assemblies referenced in the exercises.

Within each *[Language]* folder, several child folders are available:

* ***before*** – The work area for completing the HOL
* ***after*** – The fully completed HOL

### Code Snippets

All code required for this lab consisting of more than 2 lines is available as code snippets. To learn more about code snippets including how to install them and how to use them, see the snippet guide document for the language of your choice in the folder:

*C:\WF\WF 3.5 Labs\Snippets*

## Starting Material

### Acronyms Used in this Lab

* WF – Windows Workflow Foundation

### Scenario

The scenarios presented in these exercises are simplistic “Hello World” style programs in order to focus on simple mechanics of using the WF components being learned.

# Exercise 1 – Hosting the Workflow Runtime

In this exercise you will create a simple console application to instantiate the WF runtime and start a workflow. The runtime host is necessary to the execution of a workflow as it invokes the workflow.

## Task 1 – Create a new Console Application Project

#### Creating the new project and solution

1. Open Visual Studio 2008 by going to the   
   **Start Menu | All Programs | Microsoft Visual Studio 2008 | Microsoft Visual Studio 2008**
2. In Visual Studio 2008, select the **File** | **New** | **Project** menu command.
3. Visual Studio will display the New Project dialog window.
4. In the New Project dialog window, expand **Visual C# | Workflow** in the Project Types tree on the left side.
5. Select **Sequential Workflow Console Application** change the location to   
   *C:\WF\WF 3.5 Labs\Lab03\CSharp\before*

**Name:** *HostingWorkflowRuntime*

**Location:** *C:\WF\WF 3.5 Labs\Lab03\CSharp\before*

1. Click **OK** to create the new solution.

#### Modifying the workflow

1. Right click on *Workflow1.cs* in the solution explorer and **Rename** it to *SimpleWorkflow*.cs.

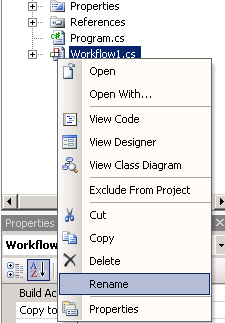


Figure : Rename workflow

1. Press **Yes** if prompted to rename all instances of *Workflow1*

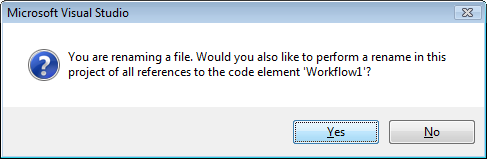


Figure : Confirm rename

1. Double click on *SimpleWorkflow.cs* to view the designer
2. Drop a **Code** activity on the workflow where it states **Drop Activities to create a Sequential Workflow**.
3. In the **Properties** window, set the **(Name)** property on the new **Code** activity to *working* as shown in Figure 3: the Working Code activity*.*

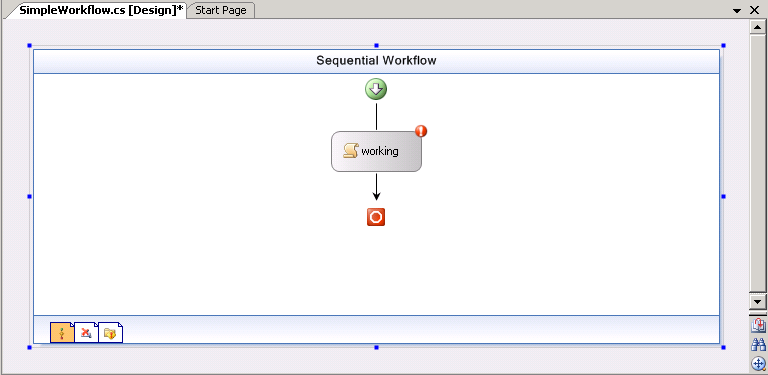


Figure : The Working Code activity

1. Double click on the *working* **Code** activity, to produce a default event handler.
2. When the workflow is executed the code handler for *working* will be called.
3. Add a simple **Console.WriteLine** statement to *working\_ExecuteCode* so that you know the function was called. Add the highlighted line from the following code:

|  |
| --- |
| **private void working\_ExecuteCode(object sender, EventArgs e)**  **{**  Console.WriteLine("Here in working\_ExecuteCode.");  **}** |

1. Select the console application code by clicking on *Program.cs* in **Solution Explorer** and pressing the **View Code** button on the toolbar.

New Picture (1)

Figure : Solution Explorer action icons

The code already generated in *Main* is all that is needed to create an instance of the workflow and run it. However, it will complete and exit before you have a chance to see what it displayed.

1. Insert the following highlighted two lines of code after starting the workflow instance and waiting. Remember to insert some white space at the place where you wish the snippet to go.

|  |
| --- |
| **Snippet:** WFLab03\_Ex01\_Task01\_PauseExecute |
| **WorkflowInstance instance = workflowRuntime.CreateWorkflow(**  **typeof(HostingWorkflowRuntime.SimpleWorkflow));**  **instance.Start();**  **waitHandle.WaitOne();**  Console.WriteLine("Workflow Completed - press ENTER to continue");  Console.Read(); |

1. Now select the **Build | Build Solution** menu command to compile and build your project. Verify that there are no compiler errors or warnings.
2. If the solution builds properly, press **F5** to run the console application. Figure 5: Simple workflow output shows an example of what you should see.

As you have inserted a **Console.Read();** in as the last line inthe application the console will wait for you to press **Enter** before it will exits.

CropperCapture[22]

Figure : Simple workflow output

## Task 2 – Listen for Runtime Events

This task looks at some of the runtime events you can subscribe to. In addition you will look at the logging built into the runtime that you can use to keep track of what is going on, starting with runtime events.

There are a number of things you can do in the host to either control the runtime environment or be informed about what is happening. In this task you will subscribe to some runtime events.

#### Subscribing to the runtime’s Started and Stopped events

1. Add the code highlighted below to the *Main* method below the existing delegates. Remember to insert some white space at the place where you wish the snippet to go.

|  |
| --- |
| **Snippet:** WFLab03\_Ex01\_Task02\_Delegates1 |
| **AutoResetEvent waitHandle = new AutoResetEvent(false);**  **workflowRuntime.WorkflowCompleted += delegate(object sender, . . .**  **workflowRuntime.WorkflowTerminated += delegate(object sender, . . .**  **{**  **Console.WriteLine(e.Exception.Message);**  **waitHandle.Set();**  **};**  workflowRuntime.Started += new EventHandler<WorkflowRuntimeEventArgs>  (workflowRuntime\_Started);  workflowRuntime.Stopped += new EventHandler<WorkflowRuntimeEventArgs>  (workflowRuntime\_Stopped); |

1. Next, insert the event handler methods for the above delegates inside the *Program* class below the *Main* method. As shown in the code below.

|  |
| --- |
| **Snippet:** WFLab03\_Ex01\_Task02\_EventHandler1 |
| static void workflowRuntime\_Stopped(object sender, WorkflowRuntimeEventArgs e)  {  Console.WriteLine("Runtime stopped event.");  }  static void workflowRuntime\_Started(object sender, WorkflowRuntimeEventArgs e)  {  WorkflowRuntime w = (WorkflowRuntime)sender;  ICollection<object> services = w.GetAllServices(typeof(object));  foreach (object o in services)  {  Console.WriteLine("Service of type " + o.ToString() + " started.");  }  } |

The *workflowRuntime\_Started*method casts the **sender** parameter to a **WorkflowRuntime** variable so that you can get a collection of all the services loaded by the workflow runtime. A **foreach** then prints them to the console.

The *workflowRuntime\_Stopped* method just prints a marker to the screen.

1. Press **F5** to run the console application, typically you will see something like the output show in Figure 6.

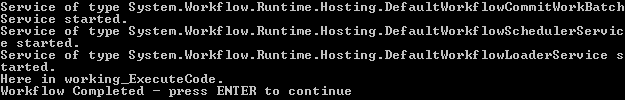


Figure : Workflow output

1. Add some additional events you want to be notified about by subscribing to the **WorkflowResumed** and **WorkflowSuspended** events. There are others you can look at but for now let's just look at these additional two.

Insert the following code below the existing delegates to subscribe to these event handlers:

|  |
| --- |
| **Snippet:** WFLab03\_Ex01\_Task02\_Delegates2 |
| workflowRuntime.Started += new EventHandler<WorkflowRuntimeEventArgs>( . . .  workflowRuntime.Stopped += new EventHandler<WorkflowRuntimeEventArgs>( . . .  workflowRuntime.WorkflowResumed += new EventHandler<WorkflowEventArgs>  (workflowRuntime\_WorkflowResumed);  workflowRuntime.WorkflowSuspended+= new EventHandler<WorkflowSuspendedEventArgs>  (workflowRuntime\_WorkflowSuspended); |

1. Next insert the event handler methods for the above delegates inside the *Program* class.

|  |
| --- |
| **Snippet:** WFLab03\_Ex01\_Task02\_EventHandler2 |
| static void workflowRuntime\_WorkflowSuspended(object sender,  WorkflowSuspendedEventArgs e)  {  Console.WriteLine("In WorkflowSuspended, reason: " + e.Error);  }  static void workflowRuntime\_WorkflowResumed(object sender, WorkflowEventArgs e)  {  Console.WriteLine("In WorkflowResumed");  } |

1. Now make a few changes to *Main*. Add the two lines of highlighted code:

|  |
| --- |
| **Snippet:** WFLab03\_Ex01\_Task02\_SuspendResume |
| WorkflowInstance instance = workflowRuntime.CreateWorkflow(typeof(HostingWorkflowRuntime.SimpleWorkflow));  instance.Start();  instance.Suspend("Reason we are suspending the workflow.");  instance.Resume();  waitHandle.WaitOne(); |

The call to **Suspend** takes a **string** as a parameter. This **string** will be available in the **WorkflowSuspended** event as part of the **WorkflowSuspendedEventArgs**.

1. Compile and run the application and you should see something like the output shown in Figure 7.

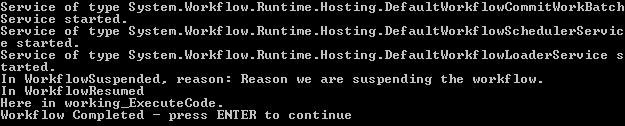


Figure : Console output

## Task 3 – Configuring Runtime Services

|  |
| --- |
| Hosting the WF Runtime There are several ways the host can set up the runtime environment. It can use an *App.Config* file that contains the settings for the runtime to use. Alternatively it can use the **WorkflowRuntime.AddService** and **WorkflowRuntime.RemoveService** methods. |

#### Changing the behavior of the runtime with a config file

1. Right click on *HostingWorkflowRuntime* project**,** select **Add | New Item**.
2. Choose **Application Configuration File**
3. Overwrite the contents of the *App.Config* file with the contents of the following file:

*C:\WF\WF 3.5 Labs\Lab03\Resources\app.config.txt*

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <configuration>  <configSections>  <section name="ConsoleApplication"  type="System.Workflow.Runtime.Configuration.WorkflowRuntimeSection,  System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,  PublicKeyToken=31bf3856ad364e35" />  </configSections>  <ConsoleApplication Name="WorkflowConsoleApplication1">  <CommonParameters>  <add name="ConnectionString" value="Initial Catalog=TrackingStore;  Data Source=localhost\SQLExpress;  Integrated Security=SSPI;" />  </CommonParameters>  <Services>  <add type="System.Workflow.Runtime.Hosting.DefaultWorkflowSchedulerService,  System.Workflow.Runtime,  Version=3.0.00000.0, Culture=neutral,  PublicKeyToken=31bf3856ad364e35" />  <add type="System.Workflow.Runtime.Hosting. SharedConnectionWorkflowCommitWorkBatchService,  System.Workflow.Runtime,  Version=3.0.00000.0, Culture=neutral,  PublicKeyToken=31bf3856ad364e35" />  <add type="System.Workflow.Runtime.Hosting.SqlWorkflowPersistenceService,  System.Workflow.Runtime,  Version=3.0.00000.0, Culture=neutral,  PublicKeyToken=31bf3856ad364e35" />  <add type="System.Workflow.Runtime.Tracking.SqlTrackingService,  System.Workflow.Runtime,  Version=3.0.00000.0, Culture=neutral,  PublicKeyToken=31bf3856ad364e35" IsTransactional="true"  UseDefaultProfile="true" />  </Services>  </ConsoleApplication>  <system.diagnostics>  <switches>  <add name="WorkflowTraceToDefault" value="1" />  <add name="Host" value="All" />  <add name="Runtime" value="All" />  <add name="Tracking" value="All" />  <add name="Activity" value="All" />  </switches>  <trace autoflush="true" indentsize="4">  <listeners>  <add name="myListener"  type="System.Diagnostics.TextWriterTraceListener"  initializeData="WFHOLTrace.log" />  </listeners>  </trace>  </system.diagnostics>  </configuration> |

1. In order to use an *app.config* file you need to add a reference to **System.Configuration**. This is standard in the .NET environment and not unique to WF.

**Right-click** on *HostingWorkflowRuntime* in Solution Explorer and select “**Add Reference”**

1. Select the **.NET** tab and select **System.Configuration** as shown in Figure 8:

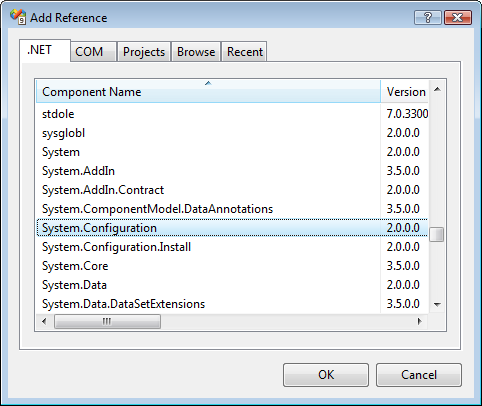


Figure : Adding a reference to System.Configuration

|  |
| --- |
| Configuration Error Warning If you don't add the above reference you will see an error like this when you compile:  The type System.Configuration.ConfigurationSection is defined in an assembly that  is not referenced. You must add a reference to assembly System.Configuration |

1. The *<listeners>* section of the app.config file allows specifying a file for logging information. For example, the following writes trace information to *WFHOLTrace.log* in the same directory as the executable. This file is not truncated on each run of your application so it will grow in size.

|  |
| --- |
| **<trace autoflush="true" indentsize="4">**  **<listeners>**  **<add name="myListener"**  **type="System.Diagnostics.TextWriterTraceListener"**  **initializeData="WFHOLTrace.log" />**  **</listeners>**  **</trace>** |

You may select the type of information that gets logged by selectively turning on or off 5 classes of information. The WorkflowRuntime logs the following types of information, **Runtime**, **Tracking**, **Host**, **Activity** and **LogTraces**.

Tracing is off by default so in order to turn it on you need the *<switches>* section in your .config file.

|  |
| --- |
| **<system.diagnostics>**  **<switches>**  **<add name="WorkflowTraceToDefault" value="1" />**  **<add name="Host" value="All" />**  **<add name="Runtime" value="All" />**  **<add name="Tracking" value="All" />**  **<add name="Activity" value="All" />**  **</switches>**  **<trace autoflush="true" indentsize="4">**  **<listeners>**  **<add name="myListener"**  **type="System.Diagnostics.TextWriterTraceListener"**  **initializeData="WFHOLTrace.log" />**  **</listeners>**  **</trace>**  **</system.diagnostics>** |

#### Using the new app.config file

1. Make a slight change when declaring and initializing the **WorkflowRuntime** variable. Change the first line of the *Main* method in *Program.cs* from this:

|  |
| --- |
| **using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())** |

to this:

|  |
| --- |
| **using ( WorkflowRuntime workflowRuntime = new WorkflowRuntime**  **("ConsoleApplication"))** |

This tells the runtime to look in the config file for a *<section>* with the name *"ConsoleApplication"*.

Within the *<Services>* tag you can add individual services you want available along with some of their configuration information such as a connection string for Microsoft SQL Server.

|  |
| --- |
| **<configSections>**  **<section name="ConsoleApplication"**  **type="System.Workflow.Runtime.Configuration.WorkflowRuntimeSection,**  **System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,**  **PublicKeyToken=31bf3856ad364e35" />**  **</configSections>** |

|  |
| --- |
| A note on SQL Server If SQL Server or SQL Express is not running, the window will stay blank for 15 seconds until the database connection times out. Also, if the database and database objects have not been set up you see other errors.  The SQL scripts for all the database objects needed for Lab03 are in the Lab03\Resources directory. The batch file CreateDatabaseObjects.cmd will create all the database objects you need so running it should correct any errors you receive if the database has not been created. |

1. Run the application by pressing **F5**.

When you run your application using a config file and print out the services loaded you will see something like Figure 9:

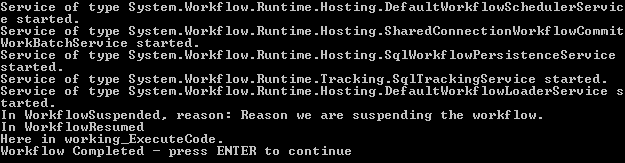


Figure : Workflow console output

#### Using SQL Tracking Service without a configuration file

The SQL Tracking Service may used directly from code without specifying the functionality in an app.config file.

1. Delete the Application Configuration file.
2. Remove the “ConsoleApplication” parameter from the WorkflowRuntime instantiation.

|  |
| --- |
| **WorkflowRuntime workflowRuntime = new WorkflowRuntime("ConsoleApplication")** |

1. Add the following code inside the *using* statement.

|  |
| --- |
| **Snippet:** WFLab03\_Ex01\_Task03\_SqlTrackingAddService |
| **using (WorkflowRuntime workflowRuntime = new WorkflowRuntime("ConsoleApplication"))**  **{**  string connectionString = "Initial Catalog=TrackingStore; " +  "Data Source=localhost\\SQLEXPRESS; " +  "Integrated Security=SSPI;";    workflowRuntime.AddService(new  System.Workflow.Runtime.Tracking.SqlTrackingService  (connectionString)); |

1. Press **F5**. The output should look like Figure 10:

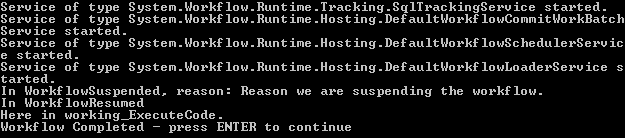


Figure : Console output

# Exercise 2 – Using the default SQL Services

In this exercise you will show you how to use the **SqlTrackingService** with your workflows. The **SqlTrackingService** logs relevant information to a database which you can retrieve later.

Window Workflow Foundation provides a SQL Server based tracking service. During execution of a workflow certain events are emitted by the runtime. The tracking service can catch these events and store that event information in a SQL Server database.

There are three types of events:

* **Workflow Events –** These events are emitted by the engine when a workflow instance enters into a particular state. For example, an event is raised when the workflow instance is 'Initialized' or when the workflow instance is 'Executing'.
* **Activity Level Events –** A workflow is made up of activities and each activity goes through a lifecycle. During its lifecycle the **activity** passes through multiple states. Whenever an **activity** reaches a particular state an event is raised.
* **User Tracked events –** If a workflow author would like to add additional information he can add custom tracking events in a workflow's code beside.

Along with the events, data from the workflow data context can also be emitted and given to the tracking service.

All the events for all the states of the instance as well as activities are not always of interest. You can specify the events you are interested in along with the data that is of interest to you in a tracking profile. The SQL tracking service has a default tracking profile. This example demonstrates the use of the SQL tracking service with the default tracking profile.

# Task 1 – Create a new Console Application Project.

#### Creating the new workflow project

1. Open **Visual Studio 2008** by going to the

**Start Menu | All Programs | Microsoft Visual Studio 2008 | Microsoft Visual Studio 2008**.

1. In Visual Studio 2008, select the **File** | **New** | **Project** menu command.
2. Visual Studio will display the **New Project** dialog window.
3. In the **New Project** dialog window, expand **Visual C# | Workflow** in the **Project Types** tree on the left side.
4. Select **“Sequential Workflow Console Application”**.
5. Set the following values in the **New Project** dialog*.*

**Name:** *DefaultSQLServices*

**Location:** *C:\WF\WF 3.5 Labs\Lab03\before*

1. Click **OK** to create the new solution.

#### Renaming the workflow

1. Rename *Workflow1.cs* to *SimpleWorkflow.cs* (**Right click | Rename**).

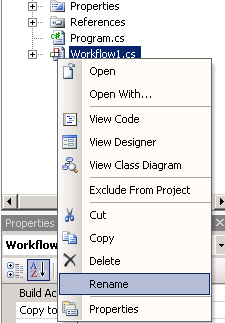


Figure : Renaming the Workflow class file

1. Press **Yes** if prompted to rename all instances of *Workflow1* as shown in Figure 2.

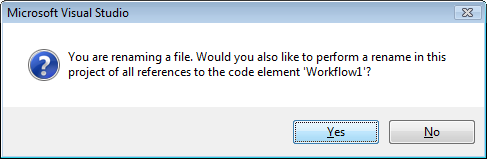


Figure : Confirmation dialog

#### Add a code activity to the workflow

1. Open *SimpleWorkflow.cs* in the visual designer (double-click the file in **Solution Explorer**).
2. Open the Visual Studio Toolbox (**CTRL + ALT + X**).
3. Drop a **Code** activity on the workflow where it states:  
   **Drop Activities to create a Sequential Workflow**.
4. Right click on *codeActivity1*, select **Properties.**
5. In the **Properties** window, set the **(Name)** property to *working.*
6. Double click on the *working* Code activity, to

This produces a default event handler for the code activity.

1. When the workflow is executed the code handler for *working* will be called. Add a simple **Console.WriteLine** statement to *working\_ExecuteCode* so that you know the function was called, as shown below in the highlighted code.

|  |
| --- |
| private void working\_ExecuteCode(object sender, EventArgs e)  {  Console.WriteLine("GOT IT: working\_ExecuteCode");  } |

## Task 2 – Adding the SQLTrackingService to the host environment

1. Double click on *Program.cs* in the solution explorer to bring it into the code editor.
2. Inside *Program.cs* insert the following **using** directives above the class definition using the provided code snippet.

|  |
| --- |
| **Snippet:** Lab03\_Ex02\_Task02\_Usings |
| using System.Workflow.Runtime.Hosting;  using System.Workflow.Runtime.Tracking;  using System.Data;  using System.Data.SqlClient;  using System.Data.SqlTypes; |

1. Create a static member within the *Program* class that contains the connection string to the **TrackingStore.**

|  |
| --- |
| class Program  {  static string connectionString = "Initial Catalog=TrackingStore;" +  "Data Source=localhost\\SQLExpress; Integrated Security=SSPI;";  static void Main(string[] args)  {  . . . |

#### Adding the SqlTrackingService to the workflow

1. Within the *workflowRuntime* instansiation block, add the following code.

|  |
| --- |
| static void Main(string[] args)  {  using (WorkflowRuntime workflowRuntime = new WorkflowRuntime())  {  workflowRuntime.AddService(  new SqlTrackingService(connectionString));  . . . |

1. Remove the **waitHandle.WaitOne()** invocation at the end of the *Main* method.

|  |
| --- |
| WorkflowInstance instance = workflowRuntime.CreateWorkflow(  typeof(DefaultSQLServices.SimpleWorkflow));  instance.Start();  waitHandle.WaitOne(); |

1. Insert the following highlighted code into the *Main* method after the workflow instance invocation.

|  |
| --- |
| **Snippet:** WFLab03\_Ex02\_Task02\_InvokeTracking |
| WorkflowInstance instance = workflowRuntime.CreateWorkflow(  typeof(DefaultSQLServices.SimpleWorkflow));  instance.Start();  instance.Suspend("Reason we are suspending the workflow.");  instance.Resume();  waitHandle.WaitOne();  GetInstanceTrackingEvents(instance.InstanceId);  Console.WriteLine("Workflow Completed - press ENTER to continue");  Console.Read(); |

1. The *GetInstanceTrackingEvents* method will pull information about the workflow itself. Insert this method below the *Main* method in the *Program* class.

|  |
| --- |
| **Snippet:** WFLab03\_Ex02\_Task02\_InstanceTracking |
| static void GetInstanceTrackingEvents(Guid instanceId)  {  Console.WriteLine("\r\nInstance Tracking Events :");  SqlTrackingQuery sqlTrackingQuery =  new SqlTrackingQuery(connectionString);  SqlTrackingWorkflowInstance sqlTrackingWorkflowInstance;  sqlTrackingQuery.TryGetWorkflow(instanceId,  out sqlTrackingWorkflowInstance);  try  {  foreach (WorkflowTrackingRecord workflowTrackingRecord  in sqlTrackingWorkflowInstance.WorkflowEvents)  {  Console.WriteLine("EventDescription: {0} DateTime: {1}",  workflowTrackingRecord.TrackingWorkflowEvent, workflowTrackingRecord.EventDateTime);  }  }  catch (Exception)  {  Console.WriteLine("No Instance Tracking Events Found");  }  } |

1. Press **F5** to run the application and you should see output similar to that shown in :

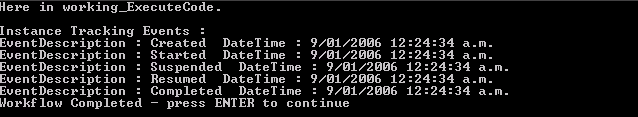


Figure : Console output

## Task 3 – Adding the SQLPersistenceService to the host environment

Many business processes that can be implemented as workflows take long periods of time to complete (up to many months or even years). Holding the workflow in memory is not only impractical (what happens if the server needs to be restarted?), but also prevents scaling because an instance must be processed on a single server and takes up memory and other resources on that server.

Many of these long-running workflows are not actively executing flow or process logic and are effectively idle, waiting for input from users or other systems. By unloading an idle instance, the host application can save memory and enable scalability across processing servers.

This task demonstrates how to unload an idle instance with a simple workflow that waits for five seconds before completing.

#### Modifying the existing Workflow to output the current time before and after a delay

1. Double click on *SimpleWorkflow.cs* to bring it into the designer
2. Delete the *working* **CodeActivity** from *SimpleWorkflow* by right-clicking on it and selecting **Delete**.
3. Drop two **Code Activities** on the Workflow design surface, set their **(Name)** properties to *timeBefore* and *timeAfter* respectfully.
4. Drop one **Delay Activity** in-between them and set its **(Name)** property to *delay.*

Your Workflow design surface should look like Figure 4.

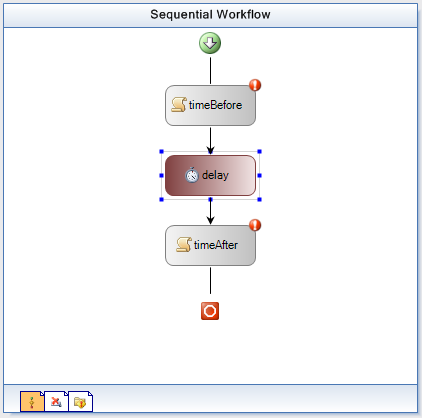


Figure : Delay action in the workflow

#### Adding Console Writes to track timings in the workflow

1. Double-click on *timeBefore* to create a event handler, and add the following highlighted code.

|  |
| --- |
| **private void timeBefore\_ExecuteCode(object sender, EventArgs e)**  **{**  Console.WriteLine("before delay: \'{0}\'",  DateTime.Now.ToLongTimeString());  **}** |

1. Double click on *SimpleWorkflow.cs* to view the designer.
2. Double click on *timeAfter* to create a event handler, and add the following code.

|  |
| --- |
| **private void timeAfter\_ExecuteCode(object sender, EventArgs e)**  **{**  Console.WriteLine("after delay: \'{0}\'",  DateTime.Now.ToLongTimeString());  **}** |

#### Setting delay timing

1. Double click on *SimpleWorkflow.cs* to view the designer.
2. **Right click** on *delay* select **Properties**

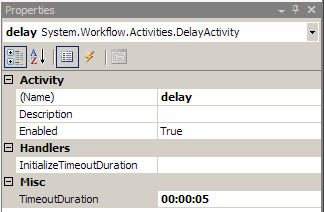


Figure : Setting Timeout Duration

1. Set **TimeoutDuration** on *delay* to 5 seconds as shown in Figure 5.

#### Adding a SQL Commit Work Batch Service reference

1. Double click on *Program.cs* to bring it into the code window.

|  |
| --- |
| The SQL Commit Work Batch Service The workflow runtime has a default **SQL Commit Work Batch Service** for accessing the database in a consistent way. You will add a service that replaces it and assumes that all database connections are the same. This means that it will avoid using the **DTC** and improve our database performance. This is useful when you are using both the **SQL Persistence Service** and the **SQL Tracking Service** with an identical database connection string. |

1. Add the following below the **WorkflowRuntime** instantiation.

|  |
| --- |
| **using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())**  **{**  workflowRuntime.AddService( new  SharedConnectionWorkflowCommitWorkBatchService(connectionString));  **. . .** |

#### Adding a SqlWorkflowPersistenceService reference

1. Add the following call to add a **SqlWorkflowPersistenceService** below the existing **AddService** calls in the *Main* method.   
   The **SqlWorkflowPersistenceService** instantiation allows persistence of workflows to the database.

|  |
| --- |
| **workflowRuntime.AddService( new**  **SharedConnectionWorkflowCommitWorkBatchService(connectionString));**  **workflowRuntime.AddService(new SqlTrackingService(connectionString));**  workflowRuntime.AddService(new SqlWorkflowPersistenceService(  connectionString, true,  new TimeSpan(0, 0, 0, 10, 0),  new TimeSpan(0, 0, 0, 10, 0))); |

#### Handling other workflow events

1. Add the following highlighted delegates into the *Main* method below the existing delegates.

|  |
| --- |
| **Snippet:** WFLab03\_Ex02\_Task03\_Delegates |
| **workflowRuntime.WorkflowTerminated += delegate(object sender,**  **WorkflowTerminatedEventArgs e)**  **{**  **Console.WriteLine(e.Exception.Message);**  **waitHandle.Set();**  **};**  workflowRuntime.WorkflowLoaded += new  EventHandler<WorkflowEventArgs>(workflowRuntime\_WorkflowLoaded);  workflowRuntime.WorkflowIdled += new  EventHandler<WorkflowEventArgs>(workflowRuntime\_WorkflowIdled);  workflowRuntime.WorkflowPersisted += new  EventHandler<WorkflowEventArgs>(workflowRuntime\_WorkflowPersisted);  workflowRuntime.WorkflowUnloaded += new  EventHandler<WorkflowEventArgs>(workflowRuntime\_WorkflowUnloaded); |

1. Add the following event handlers into the *Program* class after the *Main* method. These are the handlers for the EventHandlers you just created.

|  |
| --- |
| **Snippet:** WFLab03\_Ex02\_Task03\_EventHandlers |
| **static void workflowRuntime\_WorkflowIdled(object sender, WorkflowEventArgs e)**  **{**  **Console.WriteLine("Workflow {0} idled", e.WorkflowInstance.InstanceId);**  **ThreadPool.QueueUserWorkItem(UnloadInstance, e.WorkflowInstance);**  **}**  **static void workflowRuntime\_WorkflowUnloaded(object sender, WorkflowEventArgs e)**  **{**  **Console.WriteLine("Workflow {0} unloaded", e.WorkflowInstance.InstanceId);**  **}**  **static void workflowRuntime\_WorkflowPersisted(object sender,**  **WorkflowEventArgs e)**  **{**  **Console.WriteLine("Workflow {0} persisted",**  **e.WorkflowInstance.InstanceId);**  **}**  **static void workflowRuntime\_WorkflowLoaded(object sender, WorkflowEventArgs e)**  **{**  **Console.WriteLine("Workflow {0} loaded", e.WorkflowInstance.InstanceId);**  **}** |

1. You may have noticed a call to *UnloadInstance* during the **WorkflowIdled** event. This is a call to a method that attempts unload the current workflow instance. The method needs to be provided.

Add this code below the *Main* method inside the *Program* class.

|  |
| --- |
| **Snippet:** WFLab03\_Ex02\_Task03\_UnloadInstance |
| **static void UnloadInstance(object workflowInstance)**  **{**  **WorkflowInstance instance = (WorkflowInstance)workflowInstance;**  **try**  **{**  **Console.WriteLine("UnloadInstance: attempting to unload \'{0}\'",**  **instance.InstanceId);**  **instance.Unload();**  **}**  **catch (Exception ex)**  **{**  **Console.WriteLine("UnloadInstance: failed \r\n{0}", ex);**  **}**  **}** |

1. Make sure *SQL Server* or *SQL Express* is running, press F5 and you should see output similar to this:

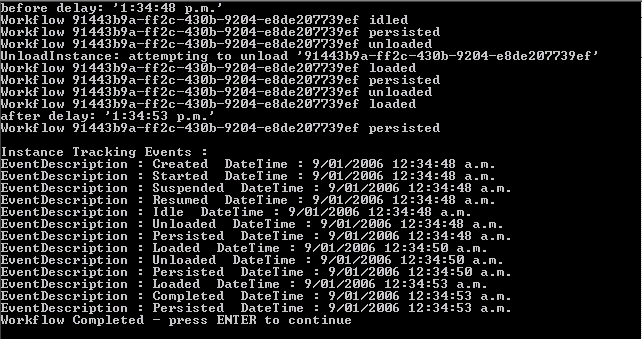


Figure : Sample application output

## Task 4 – Adding a WorkflowSchedulerService to the host environment

|  |
| --- |
| About the Workflow Scheduler Services A **WorkflowSchedulerService** decides how and when to load and persist running workflows. By default **Windows Workflow Foundation** will use the **DefaultWorkflowSchedulerService,** however you can choose to use anything that inherits from **WorkflowSchedulerService**.  WF already comes with two types of SchedulerServices, **DefaultWorkflowSchedulerService** and **ManualWorkflowSchedulerService**.  note if you do not specify a type of **WorkflowSchedulerService** through use of an application configuration file or manually making a call to **AddService**; **DefaultWorkflowSchedulerService** will be used. |

#### Starting with a new code activity

1. Double click on *SimpleWorkflow.cs* to bring it into the designer.
2. Delete *timeBefore*, *delay*, and *timeAfter* by right clicking and selecting delete for each activity.
3. Drop a new **code** activity on the designer and set the **(Name)** property to *code*, your design surface should look like Figure 7.

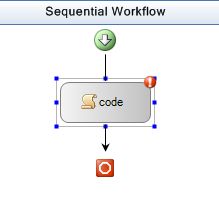


Figure : New Code activity

1. Double click on *code* to create a default event handler.
2. Add the following highlighted code into the code execute handler.

|  |
| --- |
| **private void working\_ExecuteCode(object sender, EventArgs e)**  **{**  Console.WriteLine("Here in code\_ExecuteCode.");  System.Threading.Thread.Sleep(new TimeSpan(0, 0, 2));  **}** |

#### Adding the scheduler service

1. Double click on *Program.cs* to bring it into the code window.
2. Insert the following call to add a **DefaultWorkflowSchedulerService** below the existing **AddService** calls.

**NOTE:** The **1** in the **DefaultWorkflowSchedulerService** instantiation ensures that it will only allow a single Workflow to be executed at any single point in time.

|  |
| --- |
| workflowRuntime.AddService(new DefaultWorkflowSchedulerService(1)); |

#### Spooling up multiple workflows

1. Delete the following lines.

|  |
| --- |
| **WorkflowInstance instance =**  **workflowRuntime.CreateWorkflow(**  **typeof(DefaultSQLServices.SimpleDelayWorkflow));**  **instance.Start();**  **instance.Suspend("Reason we are suspending the workflow.");**  **instance.Resume();**  **waitHandle.WaitOne();**  **GetInstanceTrackingEvents(instance.InstanceId);** |

1. Insert in place of the deleted code the following snippet, which starts up 5 instances of our *simpleWorkflow* workflow.

|  |
| --- |
| **Snippet:** WFLab03\_Ex02\_Task04\_InvokeMulti |
| for (int workflowCount = 0; workflowCount < 5; workflowCount++)  {  WorkflowInstance simpleWorkflowInstance =  workflowRuntime.CreateWorkflow(typeof(  DefaultSQLServices.SimpleWorkflow));  simpleWorkflowInstance.Start();  }  System.Threading.Thread.Sleep(new TimeSpan(0, 0, 20)); |

#### Running the workflow

1. Press **F5** and you should see output similar to this:

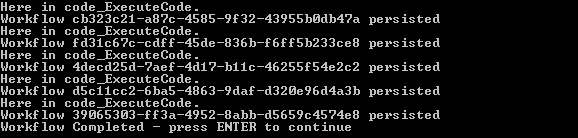


Figure : Workflow output

As expected you can see that the workflows are being executed one after the other, which is because you have specified that our **DefaultWorkflowScheduler** only allow a single workflow to execute at any point in time.

1. Remove the parameter “**1**” from the constructor of the **DefaultWorkflowShedulerService** class leaving the code as shown below. This enables the default scheduler to execute the workflows at the same time.

|  |
| --- |
| workflowRuntime.AddService(new DefaultWorkflowSchedulerService()); |

1. Press **F5** and you should see output similar to this:

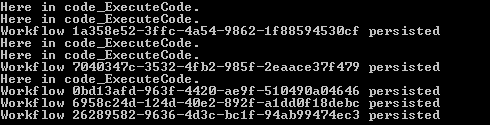


Figure : Multiple executing workflows

The workflows are being executed concurrently as our **DefaultWorkflowScheduler** doesn’t have any specific limit to concurrent threads to run.

# Exercise 3 – Building a Custom TrackingService for Workflow Monitoring

Windows Workflow Foundation provides a tracking infrastructure that enables applications to track workflow state changes and data. In addition, it gives you the flexibility to create the most appropriate tracking services for your business needs and use those tracking services in the runtime.

To write a tracking service, you need to implement a **TrackingChannel** and a **TrackingService.** The **TrackingChannel** receives the various tracking records sent by the runtime. The **TrackingService** provides the runtime with tracking profiles based on specific parameters and conditions. It is also responsible for providing a **TrackingChannel** that receives the data sent by the runtime.

## Task 1 – Create a new Console Application Project

#### Creating the new Visual Studio project

1. Open **Visual Studio 2008** by going to the

**Start Menu | All Programs | Microsoft Visual Studio 2008 | Microsoft Visual Studio 2008.**

1. In Visual Studio 2008, select the **File** | **New** | **Project** menu command.

Visual Studio will display the New Project dialog window.

1. In the **New Project** dialog window, expand **Visual C# | Workflow** in the **Project Types** tree on the left side.
2. Select **Sequential Workflow Console Application**.

**Name***: ConsoleTrackingServiceSample*

**Location:** *C:\WF\WF 3.5 Labs\Lab03\CSharp\before*

1. Click **OK** to create the project.

#### Configuring the new workflow

1. Right click on *Workflow1.cs* in the solution explorer and Rename it to *SimpleWorkflow.cs*.

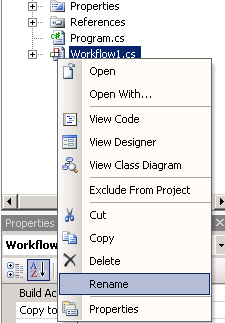


Figure : Renaming the workflow

1. Press **Yes** if prompted to rename all instances of *Workflow1*

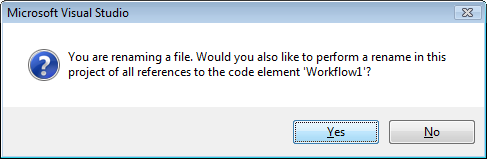


Figure : Confirmation dialog

1. Double click on *SimpleWorkflow* in the **Solution Explorer** to view the designer
2. Drop a **Code** activity on the workflow where it states **Drop Activities to create a Sequential Workflow**.
3. Right Click on*codeActivity1* select **Properties** set the **(Name)** property to *working.*
4. Double click on *working* **Code** activity, to produce a default event handler.

When the workflow is executed the code handler for *working* will be called.

1. Add a simple **Console.WriteLine** statement to *working \_ExecuteCode* so that you know the function was called. Add the line of code highlighted below:

|  |
| --- |
| **private void working\_ExecuteCode(object sender, EventArgs e)**  **{**  this.TrackData("Hello - this is a UserTrackPoint");  **}** |

## Task 2 – Create the ConsoleTrackingChannel

The **TrackingChannel** abstract class serves as a conduit that receives tracking events and data for a single workflow instance. The purpose of **TrackingChannel** is to provide a mechanism for tracking service writers to receive tracking information without concern for thread safety. Because each workflow instance has only one thread of execution there is never more than one thread active in a **TrackingChannel** object at any given time alleviating the need for data synchronization.

#### Adding the Tracking Channel

1. Right click on the *ConsoleTrackingServiceSample* project select **Add | New Item.**
2. Select **Code File** and change the name from *CodeFile1.cs* to *ConsoleTrackingChannel.cs.*
3. **Replace the entire contents** of the code file with the code snippet below.

Any Tracking Channel must inherit from the **TrackingChannel** class. The code below declares our *ConsoleTrackingChannel* as a subclass of **TrackingChannel** and sets up necessary constructors.

|  |
| --- |
| **Snippet:** WFLab03\_Ex03\_Task02\_TrackingChannel |
| using System;  using System.IO;  using System.Workflow.Runtime;  using System.Workflow.Runtime.Tracking;  using System.Workflow.ComponentModel;  using System.Workflow.Runtime.Hosting;  namespace ConsoleTrackingServiceSample  {  public class ConsoleTrackingChannel : TrackingChannel  {  private TrackingParameters parameters = null;  protected ConsoleTrackingChannel()  {  }  public ConsoleTrackingChannel(TrackingParameters parameters)  {  this.parameters = parameters;  }  }  } |

1. The runtime tracking infrastructure calls **Send** to deliver a **TrackingRecord** on the **TrackingChannel** when it matches a track point in a **TrackingProfile**.

Override the **Send** method by adding the following code inside the *ConsoleTrackingChannel* class.

|  |
| --- |
| **Snippet:** WFLab03\_Ex03\_Task02\_SendLogic |
| protected override void Send(TrackingRecord record)  {  // filter on record type  if (record is WorkflowTrackingRecord)  {  WriteWorkflowTrackingRecord((WorkflowTrackingRecord)record);  }  if (record is ActivityTrackingRecord)  {  WriteActivityTrackingRecord((ActivityTrackingRecord)record);  }  if (record is UserTrackingRecord)  {  WriteUserTrackingRecord((UserTrackingRecord)record);  }  } |

1. Add the following supporting code into the *ConsoleTrackingChannel* class.

For each of the differing types of **TrackingRecord** you will be outputting some details about that record.

|  |
| --- |
| **Snippet:** WFLab03\_Ex03\_Task02\_Support |
| #region support-code  private static void WriteTitle(string title)  {  Console.WriteLine("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  Console.WriteLine("\t" + title);  Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  }  private static void WriteWorkflowTrackingRecord(WorkflowTrackingRecord workflowTrackingRecord)  {  WriteTitle("Workflow Tracking Record");  Console.WriteLine("EventDateTime: " +  workflowTrackingRecord.EventDateTime.ToString());  Console.WriteLine("Status: " +  workflowTrackingRecord.TrackingWorkflowEvent.ToString());  }  private static void WriteActivityTrackingRecord(  ActivityTrackingRecord activityTrackingRecord)  {  WriteTitle("Activity Tracking Record");  Console.WriteLine("EventDateTime: " +  activityTrackingRecord.EventDateTime.ToString());  Console.WriteLine("QualifiedName: " +  activityTrackingRecord.QualifiedName.ToString());  Console.WriteLine("Type: " + activityTrackingRecord.ActivityType);  Console.WriteLine("Status: " +  activityTrackingRecord.ExecutionStatus.ToString());  }  private static void WriteUserTrackingRecord(  UserTrackingRecord userTrackingRecord)  {  WriteTitle("User Activity Record");  Console.WriteLine("EventDateTime: " +  userTrackingRecord.EventDateTime.ToString());  Console.WriteLine("QualifiedName: " +  userTrackingRecord.QualifiedName.ToString());  Console.WriteLine("ActivityType: " +  userTrackingRecord.ActivityType.FullName.ToString());  Console.WriteLine("Args: " + userTrackingRecord.UserData.ToString());  }  #endregion |

1. *InstanceCompletedOrTerminated* is called by the tracking runtime to indicate that the workflow instance finished running, regardless of the tracking profile associated with that instance.

Implement this method by writing a message to the console.

Add this code inside the *ConsoleTrackingChannel* class.

|  |
| --- |
| **Snippet:** WFLab03\_Ex03\_Task02\_CompleteTerminated |
| protected override void InstanceCompletedOrTerminated()  {  WriteTitle("Workflow Instance Completed or Terminated");  } |

## Task 3 – Create our ConsoleTrackingService

Tracking services are designed to monitor workflow execution through the exchange of data that can then be persisted to a storage medium or output on a specified stream, the **System.Console** for instance.

#### Adding the ConsoleTrackingService class

1. Right click on *ConsoleTrackingServiceSample* select **Add | Class**.
2. Name the class *ConsoleTrackingService.cs.*

Replace the contents of the code file with the code snippet below. This code declares the **ConsoleTrackingService** as a descendant of **TrackingService**.

|  |
| --- |
| **Snippet:** WFLab03\_Ex03\_Task03\_TrackingService |
| using System;  using System.IO;  using System.Workflow.Runtime;  using System.Workflow.Runtime.Tracking;  using System.Workflow.ComponentModel;  using System.Workflow.Runtime.Hosting;  namespace ConsoleTrackingServiceSample  {  public class ConsoleTrackingService : TrackingService  {  }  } |

#### Implementing the abstract class

You can take advantage of built in Visual Studio functionality to help implement the inherited functionality.

1. Place your cursor on the **TrackingService** class name.
2. Select Implement abstract class **TrackingService** from the Visual Studio .NET smart tag.

If you can’t make the smart tag appear, you can right click on **TrackingService** to see the menu below.

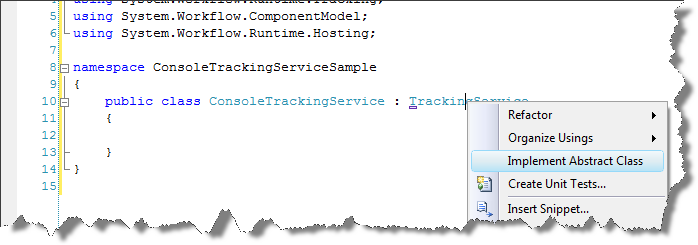


Figure : Implementing the abstract class

This produces 5 new methods, each overridden from the abstract base. One by one, you will write code to implement these stubs, adding code into their specific methods.

1. **GetProfile(Guid workflowInstanceId)**

This class won’t support reloading / instance profiles so throw an exception.

|  |
| --- |
| **protected override TrackingProfile GetProfile(Guid workflowInstanceId)**  **{**  **throw new NotImplementedException**  **("The method or operation is not implemented.");**  **}** |

1. **GetProfile(Type workflowType, Version profileVersionId)**   
   Return a version of the tracking profile that the runtime requests.

|  |
| --- |
| protected override TrackingProfile GetProfile(Type workflowType,  Version profileVersionId)  {  return GetProfile();  } |

1. **GetTrackingChannel(TrackingParameters parameters)**   
   Return the *ConsoleTrackingChannel.*

|  |
| --- |
| **protected override TrackingChannel**  **GetTrackingChannel(TrackingParameters parameters)**  **{**  return new ConsoleTrackingChannel(parameters);  **}** |

1. **TryGetProfile(Type workflowType, out TrackingProfile profile)**

Depending on the workflow type, you can return different tracking profiles. This lab returns the same profile for all running types.

|  |
| --- |
| **protected override bool TryGetProfile(Type workflowType,**  **out TrackingProfile profile)**  **{**  profile = GetProfile();  return true;  **}** |

1. **TryReloadProfile(Type workflowType, Guid workflowInstanceId, out TrackingProfile profile)**

Return false to indicate that there are no new profiles.

|  |
| --- |
| **protected override bool TryReloadProfile(Type workflowType,**  **Guid workflowInstanceId,**  **out TrackingProfile profile)**  **{**  profile = null;  return false;  **}** |

1. You may have noticed a call to a method called **GetProfile**. This method (code below) is responsible for creating a **TrackingProfile**, setting up events for notification of **ActivityTrackPoint**, **WorkflowTrackPoint** Status Events, and **UserTrackPoint**s.

Add the method to the class.

|  |
| --- |
| **Snippet:** WFLab03\_Ex03\_Task03\_GetProfile |
| private static TrackingProfile GetProfile()  {  // Create a Tracking Profile  TrackingProfile profile = new TrackingProfile();  profile.Version = new Version("3.0.0");  // Add a TrackPoint to cover all activity status events  ActivityTrackPoint activityTrackPoint = new ActivityTrackPoint();  ActivityTrackingLocation activityLocation = new  ActivityTrackingLocation(typeof(Activity));  activityLocation.MatchDerivedTypes = true;  foreach (ActivityExecutionStatus status in Enum.GetValues(  typeof(ActivityExecutionStatus)))  {  activityLocation.ExecutionStatusEvents.Add(status);  }  activityTrackPoint.MatchingLocations.Add(activityLocation);  profile.ActivityTrackPoints.Add(activityTrackPoint);  // Add a TrackPoint to cover all workflow status events  WorkflowTrackPoint workflowTrackPoint = new WorkflowTrackPoint();  workflowTrackPoint.MatchingLocation = new WorkflowTrackingLocation();  foreach (TrackingWorkflowEvent workflowEvent in Enum.GetValues(  typeof(TrackingWorkflowEvent)))  {  workflowTrackPoint.MatchingLocation.Events.Add(workflowEvent);  }  profile.WorkflowTrackPoints.Add(workflowTrackPoint);  // Add a TrackPoint to cover all user track points  UserTrackPoint userTrackPoint = new UserTrackPoint();  UserTrackingLocation userLocation = new UserTrackingLocation();  userLocation.ActivityType = typeof(Activity);  userLocation.MatchDerivedActivityTypes = true;  userLocation.ArgumentType = typeof(object);  userLocation.MatchDerivedArgumentTypes = true;  userTrackPoint.MatchingLocations.Add(userLocation);  profile.UserTrackPoints.Add(userTrackPoint);  return profile;  } |

## Task 4 – Run ConsoleTrackingServiceSample

Now that the service exists, it must be added to the workflow runtime to be used.

#### Adding the console tracking service

1. Double click on *Program.cs* to bring up the code window.
2. Add the following line of code just below the **WorkflowRuntime** using directive.

|  |
| --- |
| **using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())**  **{**  workflowRuntime.AddService(new ConsoleTrackingService());  **AutoResetEvent waitHandle = new AutoResetEvent(false);**  **. . .** |

1. Add the following code at the end of the *main* method.

|  |
| --- |
| **. . .**  **waitHandle.WaitOne();**  Console.WriteLine("Workflow Completed - press ENTER to continue");  Console.Read();  **}** |

1. Select the **Debug | Start Debugging** menu command.
2. You should see output similar to Figure 23:

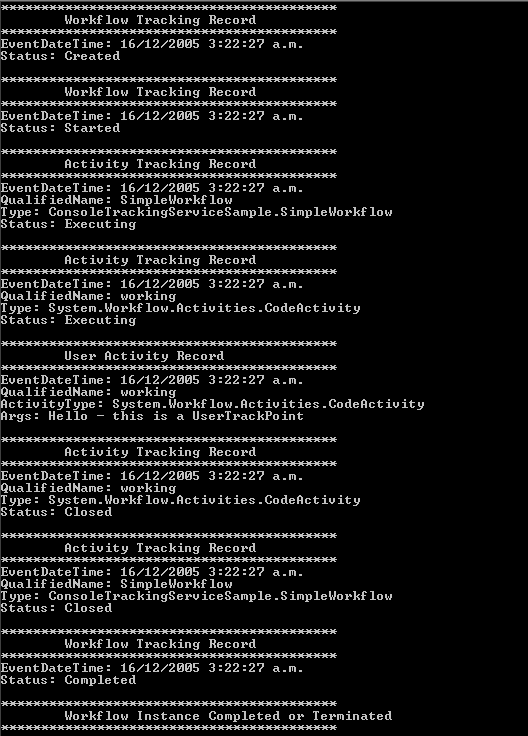


Figure : Console output

# Lab Summary

The objective of this lab was to introduce hosting the Workflow Foundation runtime and responding to some of its events.

In the lab you learned how to:

* Host Workflow Foundation in a simple Windows console application and start a workflow using the Workflow Foundation runtime.
* Run the console application to verify that workflow was loaded, ran successfully and that the runtime events were raised.
* Used the **SQLTracking** service to log runtime and instance events and retrieve them from the database.
* Used the **SQLPersistenceService** to persist the state of our workflows to a database and have them retrieved when they were required.
* Used the **WorkflowSchedulerService** to customize the execution of a group of workflows
* Built a Custom **TrackingService** for Workflow Monitoring