

# Contents

## Windows Workflow Foundation

[Guide to the Windows Workflow Documentation](#)

[What's New in Windows Workflow Foundation](#)

[What's New in Windows Workflow Foundation in .NET 4.5](#)

[Deprecated types in Windows Workflow Foundation](#)

[Windows Workflow Foundation Feature Specifics](#)

[Windows Workflow Conceptual Overview](#)

[Windows Workflow Overview](#)

[Fundamental Windows Workflow Concepts](#)

[Windows Workflow Architecture](#)

[Getting Started Tutorial](#)

[How to: Create an Activity](#)

[How to: Create a Workflow](#)

[How to: Create a Flowchart Workflow](#)

[How to: Create a Sequential Workflow](#)

[How to: Create a State Machine Workflow](#)

[How to: Run a Workflow](#)

[How to: Create and Run a Long Running Workflow](#)

[How to: Create a Custom Tracking Participant](#)

[How to: Host Multiple Versions of a Workflow Side-by-Side](#)

[How to: Update the Definition of a Running Workflow Instance](#)

## Windows Workflow Foundation Programming

[Designing Workflows](#)

[Flowchart Workflows](#)

[Procedural Workflows](#)

[State Machine Workflows](#)

[Authoring Workflows, Activities, and Expressions Using Imperative Code](#)

[Using and Creating Activities](#)

[Built-In Activity Library](#)

[Control Flow](#)

[Pick Activity](#)

[Flowchart](#)

[State Machine](#)

[Runtime](#)

[Primitives](#)

[Transaction](#)

[Collection](#)

[Error Handling](#)

[Migration](#)

[Designing and Implementing Custom Activities](#)

[Activity Authoring Options](#)

[Activity Base Class](#)

[CodeActivity Base Class](#)

[NativeActivity Base Class](#)

[Using a custom activity](#)

[Creating Asynchronous Activities](#)

[Error handling in asynchronous activities](#)

[Configuring Activity Validation](#)

[Required Arguments and Overload Groups](#)

[Imperative Code-Based Validation](#)

[Declarative Constraints](#)

[Invoking Activity Validation](#)

[Creating an Activity at Runtime](#)

[Workflow Execution Properties](#)

[Using Activity Delegates](#)

[Using Activity Extensions](#)

[Consuming OData Feeds from a Workflow](#)

[Activity Definition Scoping and Visibility](#)

[Creating custom flow control activities](#)

[Windows Workflow Foundation Data Model](#)

[Variables and Arguments](#)

- [Expressions](#)
- [C# Expressions](#)
- [Properties vs. Arguments](#)
- [Exposing data with CacheMetadata](#)
- [Waiting for Input in a Workflow](#)
- [Bookmarks](#)
- [Exceptions, Transactions, and Compensation](#)
  - [Exceptions](#)
  - [Transactions](#)
  - [Compensation](#)
  - [Cancellation](#)
  - [Debugging Workflows](#)
- [Hosting Workflows](#)
  - [Workflow Hosting Options](#)
  - [Using WorkflowInvoker and WorkflowApplication](#)
  - [Activity Tree Inspection](#)
  - [Serializing Workflows and Activities to and from XAML](#)
  - [Using WorkflowIdentity and Versioning](#)
- [Dynamic Update](#)
- [Contract First Workflow Service Development](#)
- [How to: Create a workflow service that consumes an existing service contract](#)
- [Workflow Persistence](#)
  - [SQL Workflow Instance Store](#)
    - [Properties of SQL Workflow Instance Store](#)
      - [Instance Encoding Option](#)
      - [Instance Completion Action](#)
      - [Instance Locked Exception Action](#)
      - [Host Lock Renewal Period](#)
      - [Runnable Instances Detection Period](#)
      - [Connection String and Connection String Name](#)
    - [How to: Enable SQL Persistence for Workflows and Workflow Services](#)
    - [Instance Activation](#)

[Support for Queries](#)

[Store Extensibility](#)

[Security](#)

[SQL Server Persistence Database](#)

[Persistence Database Schema](#)

[How to: Deserialize Instance Data Properties](#)

[How to: Query for Non-persisted Instances](#)

[Instance Stores](#)

[How to: Enable Persistence for Workflows and Workflow Services](#)

[How to: Create a Custom Instance Store](#)

[Persistence Participants](#)

[How to: Create a Custom Persistence Participant](#)

[Persistence Best Practices](#)

[Non-Persisted Workflow Instances](#)

[Pausing and Resuming a Workflow](#)

[Migration Guidance](#)

[Migration Guidance](#)

[Using .NET Framework 3.0 WF Activities in .NET Framework 4 with the Interop Activity](#)

[Workflow Tracking and Tracing](#)

[Tracking Records](#)

[Tracking Profiles](#)

[Tracking Participants](#)

[Configuring Tracking for a Workflow](#)

[Using Tracking to Troubleshoot Applications](#)

[Workflow Tracing](#)

[Tracking Events Reference](#)

[100 - WorkflowInstanceRecord](#)

[101 - WorkflowInstanceUnhandledExceptionRecord](#)

[102 - WorkflowInstanceAbortedRecord](#)

[103 - ActivityStateRecord](#)

[104 - ActivityScheduledRecord](#)

[105 - FaultPropagationRecord](#)

106 - CancelRequestRecord  
107 -- BookmarkResumptionRecord  
108 - CustomTrackingRecordInfo  
110 - CustomTrackingRecordWarning  
111 - CustomTrackingRecordError  
112 - WorkflowInstanceSuspendedRecord  
113 - WorkflowInstanceTerminatedRecord  
114 - WorkflowInstanceRecordWithId  
115 - WorkflowInstanceAbortedRecordWithId  
116 - WorkflowInstanceSuspendedRecordWithId  
117 - WorkflowInstanceTerminatedRecordWithId  
118 - WorkflowInstanceUnhandledExceptionRecordWithId  
119 - WorkflowInstanceUpdatedRecord  
225 - TraceCorrelationKeys  
440 - StartSignpostEvent1  
441 - StopSignpostEvent1  
1001 - WorkflowApplicationCompleted  
1002 - WorkflowApplicationTerminated  
1003 - WorkflowInstanceCanceled  
1004 - WorkflowInstanceAborted  
1005 - WorkflowApplicationIdled  
1006 - WorkflowApplicationUnhandledException  
1007 - WorkflowApplicationPersisted  
1008 - WorkflowApplicationUnloaded  
1009 - ActivityScheduled  
1010 - ActivityCompleted  
1011 - ScheduleExecuteActivityWorkItem  
1012 - StartExecuteActivityWorkItem  
1013 - CompleteExecuteActivityWorkItem  
1014 - ScheduleCompletionWorkItem  
1015 - StartCompletionWorkItem  
1016 - CompleteCompletionWorkItem

1017 - ScheduleCancelActivityWorkItem  
1018 - StartCancelActivityWorkItem  
1019 - CompleteCancelActivityWorkItem  
1020 - CreateBookmark  
1021 - ScheduleBookmarkWorkItem  
1022 - StartBookmarkWorkItem  
1023 - CompleteBookmarkWorkItem  
1024 - CreateBookmarkScope  
1025 - BookmarkScopeInitialized  
1026 - ScheduleTransactionContextWorkItem  
1027 - StartTransactionContextWorkItem  
1028 - CompleteTransactionContextWorkItem  
1029 - ScheduleFaultWorkItem  
1030 - StartFaultWorkItem  
1031 - CompleteFaultWorkItem  
1032 - ScheduleRuntimeWorkItem  
1033 - StartRuntimeWorkItem  
1034 - CompleteRuntimeWorkItem  
1035 - RuntimeTransactionSet  
1036 - RuntimeTransactionCompletionRequested  
1037 - RuntimeTransactionComplete  
1038 - EnterNoPersistBlock  
1039 - ExitNoPersistBlock  
1040 - InArgumentBound  
1041 - WorkflowApplicationPersistableIdle  
1101 - WorkflowActivityStart  
1102 - WorkflowActivityStop  
1103 - WorkflowActivitySuspend  
1104 - WorkflowActivityResume  
1124 - InvokeMethodIsStatic  
1125 - InvokeMethodIsNotStatic  
1126 - InvokedMethodThrewException

1131 - InvokeMethodUseAsyncPattern  
1132 - InvokeMethodDoesNotUseAsyncPattern  
1140 - FlowchartStart  
1141 - FlowchartEmpty  
1143 - FlowchartNextNull  
1146 - FlowchartSwitchCase  
1147 - FlowchartSwitchDefault  
1148 - FlowchartSwitchCaseNotFound  
1150 - CompensationState  
1223 - SwitchCaseNotFound  
1449 - WfMessageReceived  
1450 - WfMessageSent  
2021 - ExecuteWorkItemStart  
2022 - ExecuteWorkItemStop  
2023 - SendMessageChannelCacheMiss  
2024 - InternalCacheMetadataStart  
2025 - InternalCacheMetadataStop  
2026 - CompileVbExpressionStart  
2027 - CacheRootMetadataStart  
2028 - CacheRootMetadataStop  
2029 - CompileVbExpressionStop  
2576 - TryCatchExceptionFromTry  
2577 - TryCatchExceptionDuringCancelation  
2578 - TryCatchExceptionFromCatchOrFinally  
3501 - InferredContractDescription  
3502 - InferredOperationDescription  
3503 - DuplicateCorrelationQuery  
3507 - ServiceEndpointAdded  
3508 - TrackingProfileNotFound  
3550 - BufferOutOfOrderMessageNoInstance  
3551 - BufferOutOfOrderMessageNoBookmark  
3552 - MaxPendingMessagesPerChannelExceeded

[3557 - TransactedReceiveScopeEndCommitFailed](#)

[4201 - EndSqlCommandExecute](#)

[4202 - StartSqlCommandExecute](#)

[4203 - RenewLockSystemError](#)

[4205 - FoundProcessingError](#)

[4206 - UnlockInstanceException](#)

[4207 - MaximumRetriesExceededForSqlCommand](#)

[4208 - RetryingSqlCommandDueToSqlError](#)

[4209 - TimeoutOpeningSqlConnection](#)

[4210 - SqlExceptionCaught](#)

[4211 - QueuingSqlRetry](#)

[4212 - LockRetryTimeout](#)

[4213 - RunnableInstancesDetectionError](#)

[4214 - InstanceLocksRecoveryError](#)

[39456 - TrackingRecordDropped](#)

[39457 - TrackingRecordRaised](#)

[39458 - TrackingRecordTruncated](#)

[39459 - TrackingDataExtracted](#)

[39460 - TrackingValueNotSerializable](#)

[57398 - MaxInstancesExceeded](#)

[Custom Tracking Records](#)

[Variable and Argument Tracking](#)

[Workflow Security](#)

[Windows Workflow Foundation 4 Performance](#)

[Extending Windows Workflow Foundation](#)

[Customizing the Workflow Design Experience](#)

[Using Custom Activity Designers and Templates](#)

[How to: Create a Custom Activity Designer](#)

[How to: Create a Custom Activity Template](#)

[Using the ModelItem Editing Context](#)

[Binding a custom activity property to a designer control](#)

[Rehosting the Workflow Designer](#)

[Task 1: Create a New Windows Presentation Foundation Application](#)

[Task 2: Host the Workflow Designer](#)

[Task 3: Create the Toolbox and PropertyGrid Panes](#)

[How to: Display Validation Errors in a Rehosted Designer](#)

[Support for New Workflow Foundation 4.5 Features in the Rehosted Workflow Designer](#)

[Using a Custom Expression Editor](#)

[Windows Workflow Foundation Glossary](#)

[Windows Workflow Samples](#)

# Windows Workflow Foundation

3/9/2019 • 2 minutes to read • [Edit Online](#)

This section describes the programming model, samples, and tools of the Windows Workflow Foundation (WF).

## In This Section

### [Guide to the Windows Workflow Documentation](#)

A set of suggested topics to read, depending upon your familiarity (novice to well-acquainted), and requirements.

### [What's New in Windows Workflow Foundation](#)

Discusses the changes in several development paradigms from previous versions.

### [What's New in Windows Workflow Foundation in .NET 4.5](#)

Describes the new features in Windows Workflow Foundation in .NET Framework 4.6.1.

### [Windows Workflow Foundation Feature Specifics](#)

Describes the new features in Windows Workflow Foundation in .NET Framework 4

### [Windows Workflow Conceptual Overview](#)

A set of topics that discusses the larger concepts behind Windows Workflow Foundation.

### [Getting Started Tutorial](#)

A set of walkthrough topics that introduce you to programming Windows Workflow Foundation applications.

### [Windows Workflow Foundation Programming](#)

A set of primer topics that you should understand to become a proficient WF programmer.

### [Extending Windows Workflow Foundation](#)

A set of topics that discusses how to extend or customize Windows Workflow Foundation to suit your needs.

### [Windows Workflow Foundation Glossary for .NET Framework 4.5](#)

Defines a list of terms that are specific to WF.

### [Windows Workflow Samples](#)

Contains sample applications that demonstrate WF features and scenarios.

# Guide to the Windows Workflow Documentation

3/9/2019 • 2 minutes to read • [Edit Online](#)

This topic contains information about how to use the Windows Workflow Foundation documentation. The linked documents are recommended starting points grouped according to specific interests and levels of expertise.

## New to Windows Workflow Foundation Programming

- If you are new to programming with Windows Workflow Foundation and you just want to see some sample applications that work, see the topics listed under [Windows Workflow Samples](#).
- For a discussion of the various Windows Workflow Foundation flow-control models, see [Designing Workflows](#).
- For a tutorial that walks through the basic steps of creating a Windows Workflow Foundation application, see [Getting Started Tutorial](#).
- If you are interested in the concepts behind Windows Workflow Foundation, see the topics in the [Windows Workflow Conceptual Overview](#) section.
- For a list of terms used in the Windows Workflow Foundation documentation, see [Windows Workflow Foundation Glossary for .NET Framework 4.5](#).
- For a list of new concepts and functionalities, see [What's New in Windows Workflow Foundation](#).

## Programming In-Depth

- For an in-depth discussion of the Windows Workflow Foundation object model, see [Windows Workflow Architecture](#).
- If you are ready to start developing an application, see [Windows Workflow Foundation Programming](#).
- If you would like to extend or customize Windows Workflow Foundation to suit your requirements, see [Extending Windows Workflow Foundation](#).

# What's New in Windows Workflow Foundation

3/9/2019 • 2 minutes to read • [Edit Online](#)

Windows Workflow Foundation (WF) in .NET Framework version 4 changes several development paradigms from previous versions. Workflows are now easier to create, execute, and maintain, and implement a host of new functionality. For more information about migrating .NET 3.0 and .NET 3.5 workflow applications to use the latest version, see [Migration Guidance](#).

## Workflow Activity Model

The activity is now the base unit of creating a workflow, rather than using the [SequentialWorkflowActivity](#) or [StateMachineWorkflowActivity](#) classes. The [Activity](#) class provides the base abstraction of workflow behavior. Activity authors can then implement either [CodeActivity](#) for basic custom activity functionality, or [NativeActivity](#) for custom activity functionality that uses the breadth of the runtime. [Activity](#) is a class used by activity authors to express new behaviors declaratively in terms of other [NativeActivity](#), [CodeActivity](#), [AsyncCodeActivity](#), or [DynamicActivity](#) objects, whether they are custom-developed or included in the [Built-In Activity Library](#).

## Rich Composite Activity Options

[Flowchart](#) is a powerful new control flow activity that allows authors to model arbitrary loops and conditional branching. [Flowchart](#) provides an event-driven programming model that was previously only able to be implemented with [StateMachineWorkflowActivity](#). Procedural workflows benefit from new flow-control activities that model traditional flow-control structures, such as [TryCatch](#) and [Switch<T>](#).

## Expanded Built-In Activity Library

New features of the activity library include:

- New flow control activities, such as, [DoWhile](#), [Pick](#), [TryCatch](#), [ForEach<T>](#), [Switch<T>](#), and [ParallelForEach<T>](#).
- Activities for manipulating member data, such as [Assign](#) and collection activities such as [AddToCollection<T>](#).
- Activities for controlling transactions, such as [TransactionScope](#) and [Compensate](#).
- New messaging activities such as [SendContent](#) and [ReceiveReply](#).

## Explicit Activity Data Model

.NET Framework 4 includes new options for storing or moving data. Data can be stored in an activity using [Variable](#). When moving data in and out of an activity, specialized argument types are used to determine which direction data is moving. These types are [InArgument](#), [InOutArgument](#), and [OutArgument](#). For more information, see [Windows Workflow Foundation Data Model](#).

## Enhanced Hosting, Persistence, and Tracking Options

.NET Framework 4 contains persistence enhancements such as the following:

- There are more options for running workflows, including [WorkflowServiceHost](#), [WorkflowApplication](#), and [WorkflowInvoker](#).

- Workflow state data can be explicitly persisted using the [Persist](#) activity.
- A host can persist an [ActivityInstance](#) without unloading it.
- A workflow can specify no-persist zones while working with data that cannot be persisted, so that persistence is postponed until the no-persist zone exits.
- Transactions can be flowed into a workflow using [TransactionScope](#).
- Tracking is more easily accomplished using [TrackingParticipant](#).
- Tracking to the system event log is provided using [EtwTrackingParticipant](#).
- Resuming a pending workflow is now managed using a [Bookmark](#) object.

## Easier Ability to Extend WF Designer Experience

The new WF Designer is built on Windows Presentation Foundation (WPF) and provides an easier model to use when rehosting the WF Designer outside of Visual Studio and also provides easier mechanisms for creating custom activity designers. For more information, see [Customizing the Workflow Design Experience](#).

# What's New in Windows Workflow Foundation in .NET 4.5

3/9/2019 • 12 minutes to read • [Edit Online](#)

Windows Workflow Foundation (WF) in .NET Framework 4.5 introduces many new features, such as new activities, designer capabilities, and workflow development models. Many, but not all, of the new workflow features introduced in .NET Framework 4.5 are supported in the re-hosted workflow designer. For more information about the new features that are supported, see [Support for New Workflow Foundation 4.5 Features in the Rehosted Workflow Designer](#). For more information about migrating .NET 3.0 and .NET 3.5 workflow applications to use the latest version, see [Migration Guidance](#). This topic provides an overview of the new workflow features introduced in .NET Framework 4.5.

## WARNING

The new Windows Workflow Foundation features introduced in .NET Framework 4.5 are not available for projects that target previous versions of the framework. If a project that targets .NET Framework 4.5 is re-targeted to a previous version of the framework, several issues can occur.

- C# expressions will be replaced in the designer with the message **Value was set in XAML**.
- Many build errors will occur, including the following error.

**The file format is not compatible with current targeting framework. To convert the file format, please explicitly save the file. This error message will go away after you save the file and reopen the designer.**

## Workflow Versioning

.NET Framework 4.5 introduced several new versioning features based around the new [WorkflowIdentity](#) class. [WorkflowIdentity](#) provides workflow application authors a mechanism for mapping a persisted workflow instance with its definition.

- Developers using [WorkflowApplication](#) hosting can use [WorkflowIdentity](#) to enable hosting multiple versions of a workflow side-by-side. Persisted workflow instances can be loaded using the new [WorkflowApplicationInstance](#) class, and then the [DefinitionIdentity](#) can be used by the host to provide the correct version of the workflow definition when instantiating the [WorkflowApplication](#). For more information, see [Using WorkflowIdentity and Versioning](#) and [How to: Host Multiple Versions of a Workflow Side-by-Side](#).
- [WorkflowServiceHost](#) is now a multi-version host. When a new version of a workflow service is deployed, new instances are created using the new service, but existing instances complete using the previous version. For more information, see [Side by Side Versioning in WorkflowServiceHost](#).
- Dynamic update is introduced which provides a mechanism for updating the definition of a persisted workflow instance. For more information, see [Dynamic Update](#) and [How to: Update the Definition of a Running Workflow Instance](#).
- A `SqlWorkflowInstanceStoreSchemaUpgrade.sql` database script is provided to upgrade persistence databases created using the .NET Framework 4 database scripts. This script updates .NET Framework 4 persistence databases to support the new versioning capabilities introduced in .NET Framework 4.5. The persisted workflow instances in the database are given default versioning values, and can participate in side-by-side execution and dynamic update. For more information, see [Upgrading .NET Framework 4](#)

## Activities

The built-in activity library contains new activities and new features for existing activities.

### NoPersist Scope

[NoPersistScope](#) is a new container activity that prevents a workflow from being persisted when the NoPersistScope's child activities are executing. This is useful in scenarios where it is not appropriate for the workflow to be persisted, such as when the workflow is using machine-specific resources such as file handles, or during database transactions. Previously, to prevent persistence from occurring during an activity's execution, a custom [NativeActivity](#) that used a [NoPersistHandle](#) was required.

### New Flowchart Capabilities

Flowcharts are updated for .NET Framework 4.5 and have the following new capabilities:

- The `DisplayName` property of a [FlowSwitch<T>](#) or [FlowDecision](#) activity is editable. This will let the activity designer show more information about the activity's purpose.
- Flowcharts have a new property called [ValidateUnconnectedNodes](#); the default for this property is `False`. If this property is set to `True`, then unconnected flowchart nodes will produce validation errors.

## Support for Partial Trust

Workflows in .NET Framework version 4 required a fully trusted application domain. In .NET Framework 4.5, workflows can operate in a partial trust environment. In a partial trust environment, third-party components can be used without granting them full access to the resources of the host. Some concerns about running workflows in partial trust are as follows:

1. Using legacy components (including Rules) contained in the [Interop](#) activity is not supported under partial trust.
2. Running workflows in partial trust in [WorkflowServiceHost](#) is not supported.
3. Persisting exceptions in a partial-trust scenario is a potential security threat. To disable persisting of exceptions, an extension of type [ExceptionPersistenceExtension](#) must be added to the project in order to opt out of persisting exceptions. The following code example demonstrates how to implement this type.

```
public class ExceptionPersistenceExtension
{
    public ExceptionPersistenceExtension()
    {
        this.PersistExceptions = false;
    }
    public bool PersistExceptions { get; set; }
}
```

If exceptions are not to be serialized, ensure that exceptions are used within a [NoPersistScope](#).

4. Activity authors should override [CacheMetadata](#) to avoid having the workflow runtime automatically execute reflection against the type. Arguments and child activities must be non-null, and [Bind](#) must be called explicitly. For more information on overriding [CacheMetadata](#), see [Exposing data with CacheMetadata](#). Also, instances of arguments that are of a type that is `internal` or `private` must be explicitly created in [CacheMetadata](#) to avoid being created by reflection.
5. Types will not use [ISerializable](#) or [SerializableAttribute](#) for serialization; types that are to be serialized must support [DataContractSerializer](#).

6. Expressions that use [LambdaValue<TResult>](#) require [RestrictedMemberAccess](#), and thus will not work under partial trust. Workflows that use [LambdaValue<TResult>](#) should replace those expressions with activities that derive from [CodeActivity<TResult>...](#)
7. Expressions cannot be compiled using [TextExpressionCompiler](#) or the Visual Basic hosted compiler in partial trust, but previously compiled expressions can be run.
8. A single assembly that uses [Level 2 Transparency](#) cannot be used in .NET Framework 4, .NET Framework 4.6.1 in full trust, and .NET Framework 4.6.1 in partial trust.

## New Designer Capabilities

### Designer Search

To make larger workflows more manageable, workflows can now be searched by keyword. This feature is only available in Visual Studio; this feature is not available in a rehosted designer. There are two kinds of searches available:

- Quick Find, initiated with either **Ctrl+F** or **Edit, Find and Replace, Quick Find**.
- Find in Files, initiated with either **Ctrl+Shift+F** or **Edit, Find and Replace, Find in Files**.

Note that Replace is not supported.

#### Quick Find

Keywords searched in workflows will match the following designer items:

- Properties of [Activity](#) objects, [FlowNode](#) objects, [State](#) objects, [Transition](#) objects, and other custom flow-control items.
- Variables
- Arguments
- Expressions

Quick Find is performed on the designer's [ModelItem](#) tree. Quick Find will not locate namespaces imported in the workflow definition.

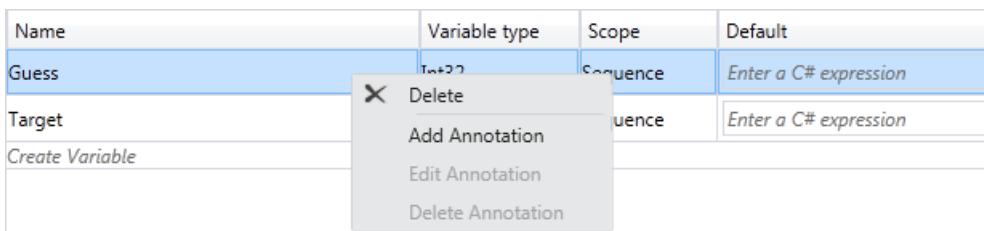
#### Find in Files

Keywords searched in workflows will match the actual content of the workflow files. The search results will be shown in Visual Studio Find Results view pane. Double-clicking the result item will navigate to the activity which contains the match in workflow designer.

### Delete context menu item in variable and argument designer

In .NET Framework 4, variables and arguments could only be deleted in the designer using the keyboard. Starting with .NET Framework 4.5, variables and arguments can be deleted using the context menu.

The following screenshot shows the variable and argument designer context menu.

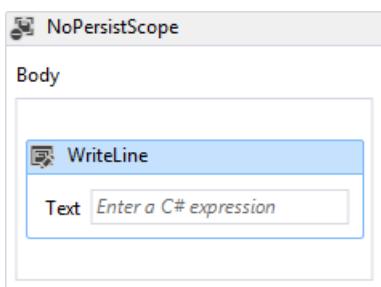


### Auto-surround with Sequence

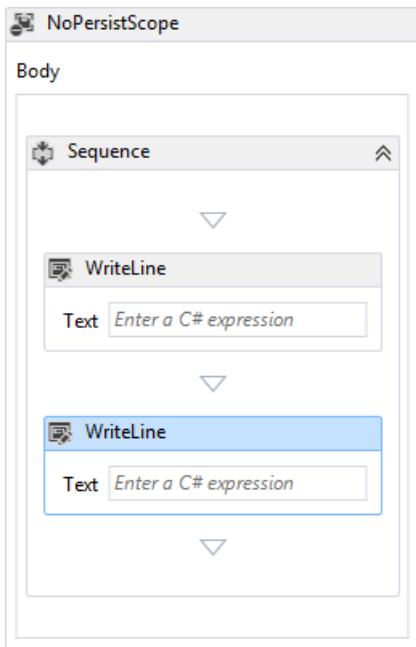
Since a workflow or certain container activities (such as [NoPersistScope](#)) can only contain a single body activity, adding a second activity required the developer to delete the first activity, add a [Sequence](#) activity, and then add

both activities to the sequence activity. Starting with .NET Framework 4.5, when adding a second activity to the designer surface, a `Sequence` activity will be automatically created to wrap both activities.

The following screenshot shows a `WriteLine` activity in the `Body` of a `NoPersistScope`.



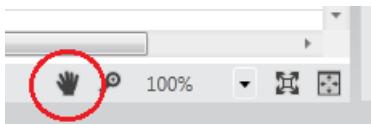
The following screenshot shows the automatically created `Sequence` activity in the `Body` when a second `WriteLine` is dropped below the first.



## Pan Mode

To more easily navigate a large workflow in the designer, pan mode can be enabled, allowing the developer to click and drag to move the visible portion of the workflow, rather than needing to use the scroll bars. The button to activate pan mode is in the lower right corner of the designer.

The following screenshot shows the pan button which is located at the bottom right corner of the workflow designer.



The middle mouse button or space bar can also be used to pan the workflow designer.

## Multi-select

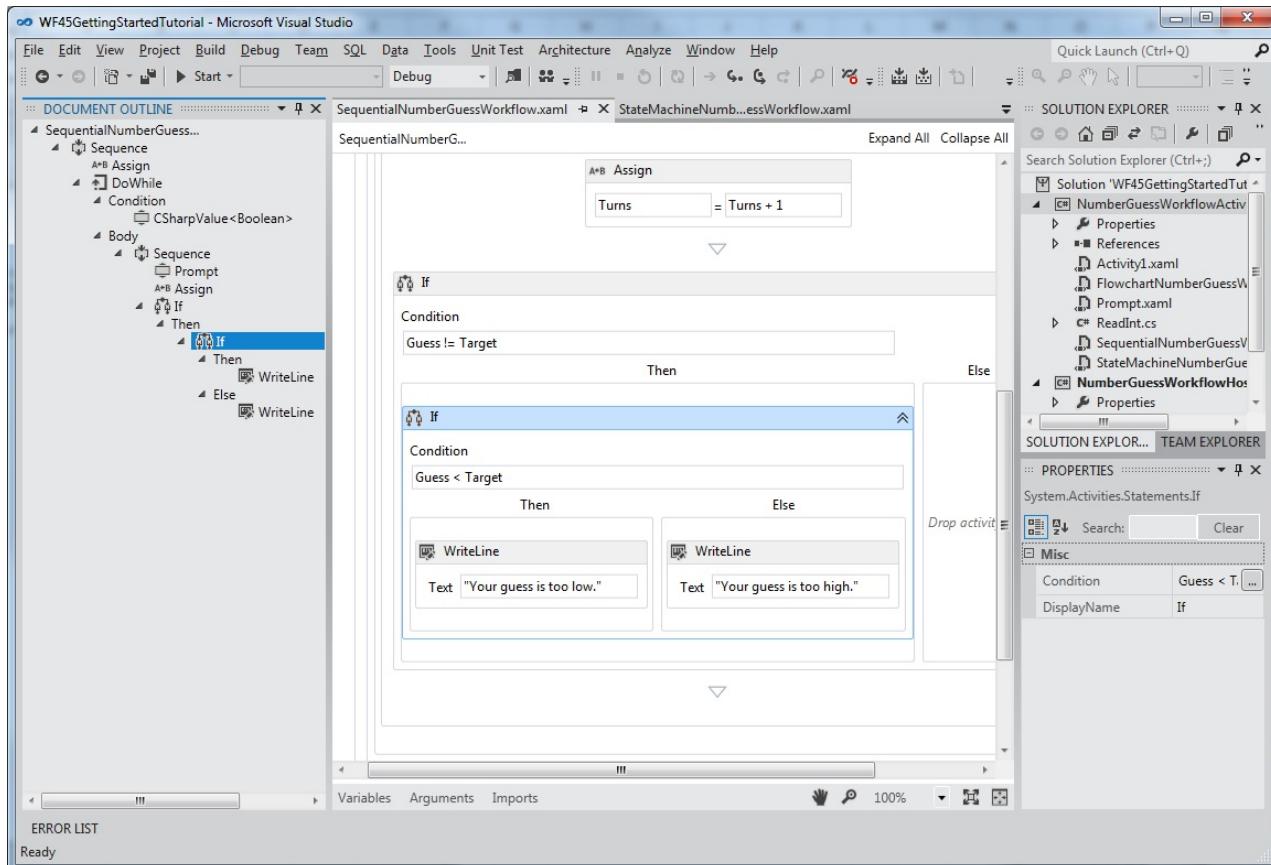
Multiple activities can be selected at one time, either by dragging a rectangle around them (when pan mode is not enabled), or by holding down `Ctrl` and click the desired activities one by one.

Multiple activity selections can also be dragged and dropped within the designer, and can also be interacted with using the context menu.

## Outline view of workflow items

In order to make hierarchical workflows easier to navigate, components of a workflow are shown in a tree-style outline view. The outline view is displayed in the **Document Outline** view. To open this view, from the top menu, select **View, Other Windows, Document Outline**, or press Ctrl W,U. Clicking on a node in outline view will navigate to the corresponding activity in the workflow designer, and the outline view will be updated to show activities that are selected in the designer.

The following screenshot of the completed workflow from the [Getting Started Tutorial](#) shows the outline view with a sequential workflow.



## C# Expressions

Prior to .NET Framework 4.5, all expressions in workflows could only be written in Visual Basic. In .NET Framework 4.5, Visual Basic expressions are only used for projects created using Visual Basic. Visual C# projects now use C# for expressions. A fully functional C# expression editor is provided which capabilities such as grammar highlighting and intellisense. C# workflow projects created in previous versions that use Visual Basic expressions will continue to work.

C# expressions are validated at design-time. Errors in C# expressions will be marked with a red wavy underline.

For more information about C# expressions, see [C# Expressions](#).

## More control of visibility of shell bar and header items

In a rehosted designer, some of the standard UI controls may not have meaning for a given workflow, and may be turned off. In .NET Framework 4, this customization is only supported by the shell bar at the bottom of the designer. In .NET Framework 4.5, the visibility of shell header items at the top of the designer can be adjusted by setting [WorkflowShellHeaderItemsVisibility](#) with the appropriate [ShellHeaderItemsVisibility](#) value.

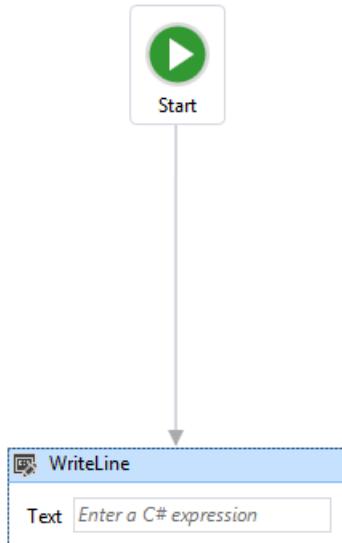
## Auto-connect and auto-insert in Flowchart and State Machine workflows

In .NET Framework 4, connections between nodes in a Flowchart workflow had to be added manually. In .NET Framework 4.5, Flowchart and State Machine nodes have auto-connect points that become visible when an activity is dragged from the toolbox onto the designer surface. Dropping an activity on one of these points automatically adds the activity along with the necessary connection.

The following screenshot shows the attachment points that become visible when an activity is dragged from the toolbox.

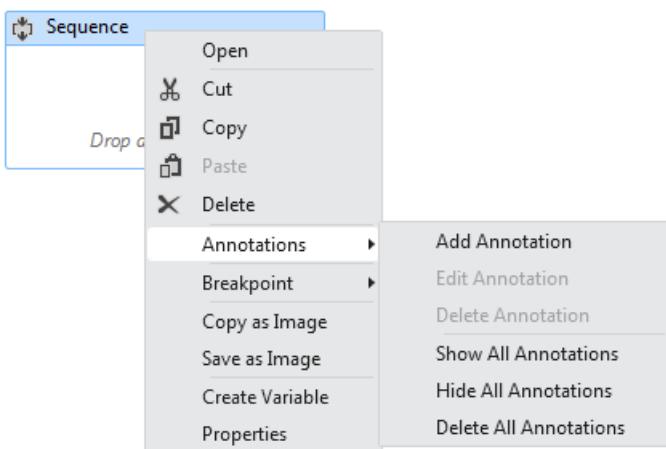


Activities can also be dragged onto connections between flowchart nodes and states to auto-insert the node between two other nodes. The following screenshot shows the highlighted connecting line where activities can be dragged from the toolbox and dropped.



## Designer Annotations

To facilitate developing larger workflows, the designer now supports adding annotations to help keep track of the design process. Annotation can be added to activities, states, flowchart nodes, variables and arguments. The following screenshot shows the context menu used to add annotations to the designer.



## Debugging states

In .NET Framework 4, non-activity elements could not support debug breakpoints since they were not units of execution. This release provides a mechanism for adding breakpoints to [State](#) objects. When a breakpoint is set on a [State](#), execution will break when the state is transitioned to, before its entry activities or triggers are scheduled.

## Define and consume ActivityDelegate objects in the designer

Activities in .NET Framework 4 used [ActivityDelegate](#) objects to expose execution points where other parts of the workflow could interact with a workflow's execution, but using these execution points usually required a fair

amount of code. In this release, developers can define and consume activity delegates using the workflow designer. For more information, see [How to: Define and consume activity delegates in the Workflow Designer](#).

### **Build-time validation**

In .NET Framework 4, workflow validation errors weren't counted as build errors during the build of a workflow project. This meant that building a workflow project could succeed even when there were workflow validation errors. In .NET Framework 4.5, workflow validation errors cause the build to fail.

### **Design-time background validation**

In .NET Framework 4, workflows were validated as a foreground process, which could potentially hang the UI during complex or time-consuming validation processes. Workflow validation now takes place on a background thread, so that the UI is not blocked.

### **View state located in a separate location in XAML files**

In .NET Framework 4, the view state information for a workflow is stored across the XAML file in many different locations. This is inconvenient for developers who want to read XAML directly, or write code to remove the view state information. In .NET Framework 4.5, the view state information in the XAML file is serialized as a separate element in the XAML file. Developers can easily locate and edit the view state information of an activity, or remove the view state altogether.

### **Expression extensibility**

In .NET Framework 4.5, we provide a way for developers to create their own expression and expression authoring experience that can be plugged into the workflow designer.

### **Opt-in for Workflow 4.5 features in rehosted designer**

To preserve backward compatibility, some new features included in .NET Framework 4.5 are not enabled by default in the rehosted designer. This is to ensure that existing applications that use the rehosted designer are not broken by updating to the latest version. To enable new features in the rehosted designer, either set `TargetFrameworkName` to ".NET Framework 4.5", or set individual members of `DesignerConfigurationService` to enable individual features.

## **New Workflow Development Models**

In addition to flowchart and sequential workflow development models, this release includes State Machine workflows, and contract-first workflow services.

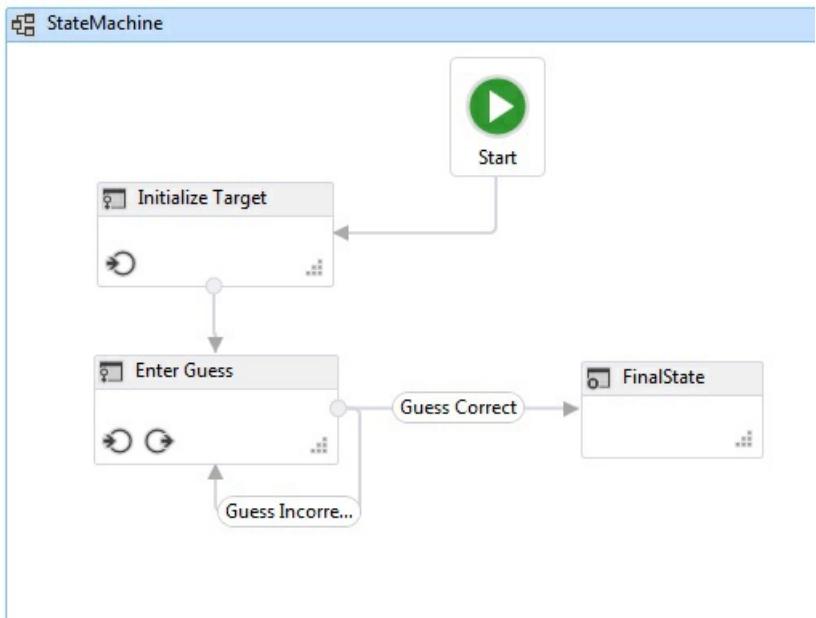
### **State machine workflows**

State machine workflows were introduced as part of the .NET Framework 4, version 4.0.1 in the [Microsoft .NET Framework 4 Platform Update 1](#). This update included several new classes and activities which allowed developers to create state machine workflows. These classes and activities have been updated for .NET Framework 4.5.

Updates include:

1. The ability to set breakpoints on states
2. The ability to copy and paste transitions in the workflow designer
3. Designer support for shared trigger transition creation
4. Activities used to create State Machine workflows, including: `StateMachine`, `State`, and `Transition`

The following screenshot shows the completed state machine workflow from the [Getting Started Tutorial](#) step [How to: Create a State Machine Workflow](#).



For more information on creating state machine workflows, see [State Machine Workflows](#).

### Contract-first workflow development

The contract-first workflow development tool allows the developer to design a contract in code first, then, with a few clicks in Visual Studio, automatically generate an activity template in the toolbox representing each operation. These activities are then used to create a workflow that implements the operations defined by the contract. The workflow designer will validate the workflow service to ensure that these operations are implemented and the signature of the workflow matches the contract signature. The developer can also associate a workflow service with a collection of implemented contracts. For more information on contract-first workflow service development, see [How to: Create a workflow service that consumes an existing service contract](#).

# Deprecated types in Windows Workflow Foundation

10/3/2018 • 2 minutes to read • [Edit Online](#)

In .NET 4 the Workflow Team released an all new Workflow engine in the [System.Activities](#) namespace. With the release of .NET 4.5 Beta we are marking most of the types in the "WF 3" [System.Workflow.Activities](#), [System.Workflow.ComponentModel](#), and [System.Workflow.Runtime](#) namespaces as obsolete.

## Obsolete namespaces and tools

The following assemblies have one or more public types that will be deprecated:

- System.Workflow.Activities.dll
- System.Workflow.ComponentModel.dll
- System.Workflow.Runtime.dll
- System.WorkflowServices.dll
- Microsoft.Workflow.DebugController.dll
- Microsoft.Workflow.Compiler.exe
- Wfc.exe

As a result, customers who are using the deprecated WF 3 APIs will encounter build warnings with a message similar to the following:

**Warning BC40000: X is obsolete: WF 3 types are deprecated. Please use WF 4 instead.** We will remove the types from the .NET Framework in a future release, but we have not yet determined that timeframe (not in 4.5). This current step allows us to communicate our direction to our customers and allow them plenty of time to move to the new WF4 model. We will, of course, continue to support these WF 3 types under the [Microsoft Support Lifecycle Policy](#). Existing WF3 applications will run without issue on .NET 4.5, and Visual Studio 2012 will support new and existing WF3-based solutions.

Rules related types in the [System.Workflow.Activities.Rules](#) namespace, which do not have a replacement in WF 4.5, have not been deprecated.

Customers who want to migrate their applications to WF 4 will find help in the [Workflow 4 Migration Guidance](#).

# Windows Workflow Foundation Feature Specifics

3/14/2019 • 14 minutes to read • [Edit Online](#)

.NET Framework version 4 adds a number of features to Windows Workflow Foundation. This document describes a number of the new features, and gives details about scenarios in which they may be useful.

## Messaging Activities

The messaging activities ([Receive](#), [SendReply](#), [Send](#), [ReceiveReply](#)) are used to send and receive WCF messages from your workflow. [Receive](#) and [SendReply](#) activities are used to form a Windows Communication Foundation (WCF) service operation that is exposed via WSDL just like standard WCF web services. [Send](#) and [ReceiveReply](#) are used to consume a web service similar to a WCF [ChannelFactory](#); an **Add Service Reference** experience also exists for Workflow Foundation that generates pre-configured activities.

### Getting Started with Messaging Activities

- In Visual Studio 2012, create a WCF Workflow Service Application project. A [Receive](#) and [SendReply](#) pair will be placed on your canvas.
- Right-click on the project and select **Add Service Reference**. Point to an existing web service WSDL and click **OK**. Build your project to show the generated activities (implemented using [Send](#) and [ReceiveReply](#)) in your toolbox.
- [Workflow services documentation](#)

### Messaging Activities Example Scenario

A `BestPriceFinder` service calls out to multiple airline services to find the best ticket price for a particular route. Implementing this scenario would require you to use the message activities to receive the price request, retrieve the prices from the back-end services, and reply to the price request with the best price. It would also require you to use other out-of-box activities to create the business logic for calculating the best price.

## WorkflowServiceHost

The [WorkflowServiceHost](#) is the out-of-box workflow host that supports multiple instances, configuration, and WCF messaging (although the workflows aren't required to use messaging in order to be hosted). It also integrates with persistence, tracking, and instance control through a set of service behaviors. Just like WCF's [ServiceHost](#), the [WorkflowServiceHost](#) can be self-hosted in a console/WinForms/WPF application or Windows service, or web-hosted (as a .xamlx file) in IIS or WAS.

### Getting Started with Workflow Service Host

- In Visual Studio 2010, create a WCF Workflow Service Application project: this project will be set up to use [WorkflowServiceHost](#) in a web-host environment.
- In order to host a non-messaging workflow, add a custom [WorkflowHostingEndpoint](#) that will create the instance based on a message.
- Workflow instances can be controlled (e.g. suspended or terminated) by adding a [WorkflowControlEndpoint](#) to the [WorkflowServiceHost](#) and then using a [WorkflowControlClient](#).
- Samples for the [WorkflowServiceHost](#) can be found in the following sections:
  - [Execution](#)
  - [Application: Suspended Instance Management](#)

- [Hosting Workflow services overview](#)

## **WorkflowServiceHost Scenario**

A BestPriceFinder service calls out to multiple airline services to find the best ticket price for a particular route. Implementing this scenario would require you to host the workflow in [WorkflowServiceHost](#). It would also use the message activities to receive the price request, retrieve the prices from the back-end services, and reply to the price request with the best price.

## **Correlation**

A correlation is one of two things:

- A way of grouping messages together; that is, the relationship between a request message and its reply.
- A way of mapping a piece of data to a service instance

### **Getting Started**

- To get started with correlation, create a new project in Visual Studio. Create a variable of type [CorrelationHandle](#).
- An example of correlation used to group messages together is a Request-Reply correlation that groups messages together.
  - On a [Receive](#) activity, click on the [CorrelationInitializers](#) property and add a [RequestReplyCorrelationInitializer](#) using the CorrelationHandle created in the first step above.
  - Create a [SendReply](#) activity by right-clicking on the [Receive](#) and clicking "Create SendReply". Paste it into your workflow after the [Receive](#) activity.
- An example of mapping a piece of data to a service instance is content-based correlation which maps a piece of data (for example, an order ID) to a particular workflow instance.
  - On any messaging activity, click on the [CorrelationInitializers](#) property and add a [QueryCorrelationInitializer](#) using the [CorrelationHandle](#) variable created above. Double-click on the desired property on the message (e.g. OrderID) from the drop-down menu. Set the [CorrelatesWith](#) property to the [CorrelationHandle](#) variable used above.
- [Correlation Conceptual Documentation](#)

### **Correlation Scenario**

An order-processing workflow is used to handle new order creation and updating existing orders that are in process. Implementing this scenario would require you to host the workflow in [WorkflowServiceHost](#) and use the messaging activities. It would also require correlation based on the [orderId](#) to ensure that updates are made to the correct workflow.

## **Simplified Configuration**

The WCF configuration schema is complex and provides users with many hard to find features. In .NET Framework 4.6.1, we have focused on helping WCF users configure their services with the following features:

- Removing the need for explicit per-service configuration. If you do not configure any <service> elements for your service, and your service does not define programmatically any endpoint, then a set of endpoints will be automatically added to your service, one per service base address and per contract implemented by your service.
- Enables the user to define default values for WCF bindings and behaviors, which will be applied to services with no explicit configuration.

- Standard endpoints define reusable preconfigured endpoints, which have fixed values for one or more of the endpoint properties (address, binding and contract), and allow defining custom properties.
- Finally, the [ConfigurationChannelFactory<TChannel>](#) allows you to do central management of WCF client configuration, useful in scenarios in which configuration is selected or changed after the application domain load time.

## Getting Started

- [A Developer's Guide to WCF 4.0](#)
- [Configuration Channel Factory](#)
- [Standard Endpoint Element](#)
- [Service configuration improvements in .NET Framework 4](#)
- [Common User Mistake in .NET 4: Mistyping the WF/WCF Service Configuration Name](#)

## Simplified Configuration Scenarios

- An experienced ASMX developer wants to start using WCF. However, WCF seems way too complicated! What is all that information that I need to write in a configuration file? In .NET 4, you can even decide to not have a configuration file at all.
- An existing set of WCF services are very difficult to configure and maintain. The configuration file has thousands of lines of XML code that are extremely dangerous to touch. Help is needed to reduce that amount of code to something more manageable.

## Data Contract Resolver

In .NET 3.5, there were a few limitations in the design of known types:

- Adding known types dynamically, during serialization or deserialization, was not possible.
- Serializers could not deal with unknown xsi:type information.
- It was not possible for users to specify what xsi:type they would like to have appear on the wire to, for instance, make the size of a serialization instance on the wire smaller.

The [DataContractResolver](#) solves these issues in .NET 4.5.

## Getting Started

- [Data Contract Resolver API documentation](#)
- [Introducing the Data Contract Resolver](#)
- Samples:
  - [DataContractResolver](#)
  - [KnownAssemblyAttribute](#)

## Data Contract Resolver Scenarios

- Avoiding having to declare tens of [KnownTypeAttribute](#) objects in a service.
- Reducing the size of the XML blob.

## Flowchart

Flowchart is a well-known paradigm to visually represent domain problems. It is a new control flow style we're introducing in .NET 4. A core characteristic of Flowchart is that only one activity is executed at any given time.

Flowcharts can express loops and alternative outcomes, but cannot natively express concurrent execution of multiple nodes.

## Getting Started

- In Visual Studio 2012, create a workflow console application. Add a Flowchart in the workflow designer.
- The flowchart feature uses the following classes:
  - [Flowchart](#)
  - [FlowNode](#)
  - [FlowDecision](#)
  - [FlowStep](#)
  - [FlowSwitch<T>](#)
- Samples:
  - [Fault Handling in a Flowchart Activity Using TryCatch](#)
  - [Hiring Process](#)
- Designer Documentation:
  - [Flowchart Activity Designers](#)

## Flowchart Scenarios

A flowchart activity can be used to implement a guessing game. The guessing game is very simple: the computer selects a random number and the player has to guess that number. When the player submits each guess, the computer shows him a hint (i.e. "try a lower number"). If the player finds the number in less than 7 attempts, he receives a special congratulation from the computer. This game can be implemented with a combination of the following procedural activities:

- [Sequence](#)
- [While](#)
- [Switch<T>](#)
- [TryCatch](#)
- [Assign<T>](#)
- [If](#)

## Procedural activities (Sequence, If, ForEach, Switch, Assign, DoWhile, While)

Procedural activities provide a mechanism to model sequential control flow using concepts that are familiar to programmers. These activities enable traditionally structured programming language constructs and, when appropriate, provide language parity with common procedural languages such as C#/VB.

## Getting Started

- In Visual Studio 2012, create a workflow console application. Add procedural activities in the workflow designer.
- Samples:
  - [Hiring Process](#)

- [Corporate Purchase Process](#)
- Designer Documentation:
  - [Parallel Activity Designer](#)
  - [ParallelForEach<T> Activity Designer](#)

## Procedural Activity Scenarios

- [Parallel](#): An intranet document management system has a document approval workflow. Documents need to be approved by people in several departments before they can be published to the intranet. There isn't an established order for the approvals; they can occur at any time while the document is in the "approval pending" phase. When a user submits a document for review it must be approved by her direct manager, the intranet administrator, and the internal communications manager.
- [ParallelForEach<T>](#): A WF application manages corporate buys within a large company. The corporate rules dictate that before planning any purchase operation, the valuations of three different vendors is required. An employee from the buying department selects three vendors from the company's vendor list. After these vendors have been selected and notified, the company will wait for their economic proposals. The proposals can come in any order. To implement this scenario in WF, we use a [ParallelForEach<T>](#) that will iterate through our collection of vendors and ask for their economic proposals. After all offers are gathered, the best one is selected and displayed.

## InvokeMethod

The [InvokeMethod](#) activity allows invoking public methods in objects or types in scope. It supports invoking instance and static methods with or without parameters (including parameter arrays), and generic methods. It also allows executing method synchronously and asynchronously.

### Getting Started

- In Visual Studio 2012, create a workflow console application. Add an [InvokeMethod](#) activity in the workflow designer, and configure static and instance methods on it.
- Designer Documentation: [InvokeMethod Activity Designer](#)

### InvokeMethod Scenarios

- A method in an object in scope needs to be invoked. For example, a value needs to be added to a dictionary. The Add method of the instance of the dictionary is invoked, and the key and value are provided.
- A method needs to be invoked on a legacy CLR object. Instead of creating a custom activity to wrap the call to that legacy class, if it is in scope during the workflow execution, [InvokeMethod](#) can be used.

## Error handling activities

The [TryCatch](#) activity provides a mechanism for catching exceptions that occur during the execution of a set of contained activities (similar to the Try/Catch construct in C#/VB). [TryCatch](#) provides exception handling at the workflow level. When an unhandled exception is thrown, the workflow is aborted and the Finally block won't be executed. This behavior is consistent with C#.

### Getting Started

- In Visual Studio 2012, create a workflow console application. Add a [TryCatch](#) activity in the workflow designer.
- Sample: [Fault Handling in a Flowchart Activity Using TryCatch](#)
- Designer Documentation: [Error Handling Activity Designers](#)

## Error handling scenarios

A set of activities needs to be executed, and specific logic needs to be executed when an error occurs. If during that error handling logic it is found that the error is not recoverable, the exception will be rethrown, and the parent activity (or the host) will deal with the problem.

## Pick activity

The [Pick](#) Activity provides event-based control flow modeling in WF. [Pick](#) contains many branches where each branch waits for a particular event to occur before running. In this setup, a [Pick](#) behaves similar to a [Switch<T>](#) to which the Activity will execute only one of the set of events it is listening. Each branch is event driven and the event that occurs runs the corresponding branch first. All other branches cancel and stop listening for events.

### Getting Started

- In Visual Studio 2012, create a workflow console application. Add a [Pick](#) activity in the workflow designer.
- Sample: [Using the Pick Activity](#)
- Designer documentation: [Pick Activity Designer](#)

### Pick Scenario

A user needs to be prompted for input. Under normal circumstances, the developer would use a method call like [ReadLine](#) to prompt for a user's input. The problem with this setup is that the program waits until the user enters something. In this scenario, a time-out is needed to unblock a blocking activity. A common scenario is one that requires a task to be completed within a given time duration. Timing out a blocking activity is a scenario where [Pick](#) adds a lot of value.

## WCF Routing Service

The Routing Service is designed to be a generic software Router that allows you to control how WCF messages flow in between your clients and services. The Routing Service allows you to decouple your clients from your services, which gives you much more freedom in terms of the configurations that you can support and the flexibility you have when considering how to host your services. In .NET 3.5, clients and services were tightly coupled; a client had to know about all of the services it needed to talk to and where they were located. In addition, WCF in .NET Framework 3.5 had the following limitations:

- Error handling was complex, as this logic had to be hard-coded into the client.
- Clients and services had to always use the same bindings.
- Services were rarely well factored: it is easier to have the client talk to one service which implements everything, rather than needing to choose between multiple services.

The routing service in .NET 4 is designed to make these problems easier to solve. The new routing service has the following features:

1. Content based routing ([MessageFilter](#) objects examine a message to determine where it should be sent.)
2. Protocol bridging (transport & message)
3. Error handling (the router catches communication exceptions and fails over to backup endpoints)
4. Dynamic (in memory) update of [MessageFilterTable<TFilterData>](#) and Routing Configuration.

### Getting Started

1. Documentation: [Routing](#)
2. Samples: [Routing Services \[WCF Samples\]](#)

### 3. Blog: [Routing Rules!](#)

#### **Routing Scenarios**

The routing service is useful in the following scenarios:

- Clients can talk to multiple services without having to address them all directly.
- Clients can perform additional logic on a client request to determine where to route it
- Decompose the operations a client performs into multiple service implementations without refactoring the client.
- Clients and services can speak different bindings with different security settings.
- Clients can be enabled to be more robust against failure or the unavailability of services.

## **WCF Discovery**

WCF Discovery is a framework technology that allows you to incorporate a discovery mechanism to your application infrastructure. You can use this to make your service discoverable, and configure your clients to search for services. Clients no longer need to be hard coded with endpoint, making your application more robust and fault tolerant. Discovery is the perfect platform to build auto-configuration capabilities into your application.

The product is built on top of the WS-Discovery standard. It's designed to be interoperable, extensible, and generic. The product supports two modes of operation:

1. Managed: where there is an entity on the network knowledgeable about existing services, clients query it directly for information. This is analogous to Active Directory.
2. Ad-hoc: where clients use multicast messages to locate services.

Furthermore, discovery messages are network protocol agnostic; you can use them on top any protocol that supports the mode requirements. For example, discovery multicast messages can be sent over the UDP channel or any other network that supports multicast messaging. These design points, combined with feature flexibility, allow you to adapt the discovery specifically to your solution.

#### **Getting Started**

- Documentation: [WCF Discovery](#)
- Samples: [Discovery \(Samples\)](#)

#### **Discovery Scenarios**

A developer doesn't want to hard code endpoints, since it is unknown when my service will be available. Instead, the developer wants to choose a service at runtime. More decoupling, robustness, and auto-configuration is needed between the components in the application.

## **Tracking**

Workflow tracking provides insight into the execution of a workflow instance. The tracking events are emitted from a workflow at the workflow instance level and when activities within the workflow execute. A workflow tracking participant needs to be added to the workflow host to subscribe to tracking records. The tracking records are filtered using a tracking profile. The .NET Framework provides an ETW (Event Tracing for Windows) tracking participant, and a basic profile is installed in the machine.config file.

#### **Getting Started**

1. In Visual Studio 2010, create a WCF Workflow Service Application project. A [Receive](#) and [SendReply](#) pair will be placed on your canvas to start.

2. Open the web.config and add an ETW tracking behavior with no profile.
  - a. The default profile is used.
  - b. Open event viewer and enable the analytic channel in the following node: **Event Viewer**, **Applications and Services Logs**, **Microsoft**, **Windows**, **Application Server-Applications**. Right-click **Analytic** and select **Enable Log**.
  - c. Run the workflow service.
  - d. Observe the workflow tracking events in event viewer.
3. Samples: [Tracking](#)
4. Conceptual documentation: [Workflow Tracking and Tracing](#)

## SQL Workflow Instance Store

The [SqlWorkflowInstanceStore](#) is a SQL Server-based implementation of an instance store. An instance store stores the state of a running instance together with all data necessary to load and resume that instance. The service host instructs the instance store to save the instance state if the workflow persists, and it instructs the instance store to load the instance state when a message arrives for that instance or a delay activity expires.

### Getting Started

1. In Visual Studio 2012, create a Workflow that contains an implicit or explicit [Persist](#) activity. Add the [SqlWorkflowInstanceStore](#) behavior to your workflow service host. This can be done in code or in the application configuration file.
2. Samples: [Persistence](#)
3. Conceptual documentation: [SQL Workflow Instance Store](#).

# Windows Workflow Conceptual Overview

3/9/2019 • 2 minutes to read • [Edit Online](#)

This section contains a set of topics discussing the larger concepts behind Windows Workflow Foundation (WF).

## In This Section

### [Windows Workflow Overview](#)

Describes the foundation that enables users to create system or human workflows in their applications written for Windows Vista, Windows XP, Windows Server 2003, and Windows Server 2008 operating systems.

### [Fundamental Windows Workflow Concepts](#)

Describes fundamental concepts used in Windows Workflow Foundation development that may be new to some developers.

### [Windows Workflow Architecture](#)

Describes components used in Windows Workflow Foundation development.

# Windows Workflow Overview

3/26/2019 • 2 minutes to read • [Edit Online](#)

A workflow is a set of elemental units called *activities* that are stored as a model that describes a real-world process. Workflows provide a way of describing the order of execution and dependent relationships between pieces of short- or long-running work. This work passes through the model from start to finish, and activities might be executed by people or by system functions.

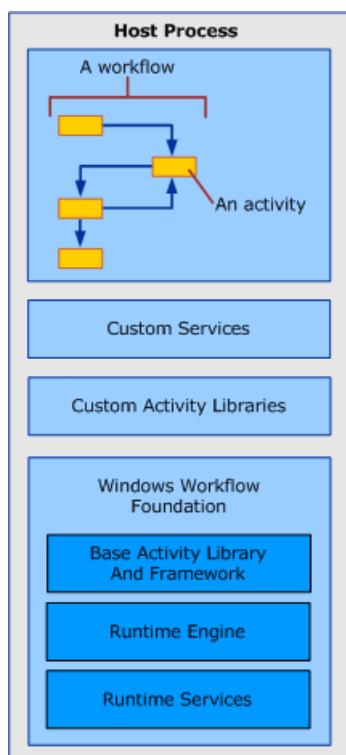
## Workflow Run-time Engine

Every running workflow instance is created and maintained by an in-process run-time engine that the host process interacts with through one of the following:

- A [WorkflowInvoker](#), which invokes the workflow like a method.
- A [WorkflowApplication](#) for explicit control over the execution of a single workflow instance.
- A [WorkflowServiceHost](#) for message-based interactions in multi-instance scenarios.

Each of these classes wraps the core activity runtime represented as a [ActivityInstance](#) responsible for activity execution. There can be several [ActivityInstance](#) objects within an application domain running concurrently.

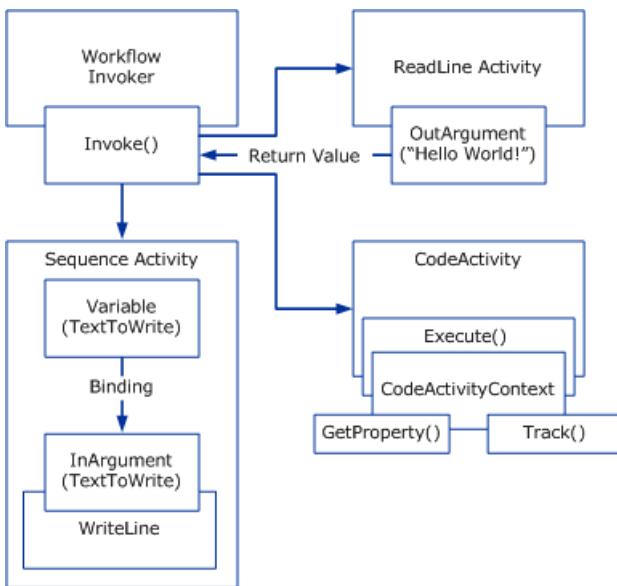
Each of the preceding three host interaction objects is created from a tree of activities referred to as a workflow program. Using these types or a custom host that wraps [ActivityInstance](#), workflows can be executed inside any Windows process including console applications, forms-based applications, Windows Services, ASP.NET Web sites, and Windows Communication Foundation (WCF) services.



Workflow components in the host process

## Interaction between Workflow Components

The following diagram demonstrates how workflow components interact with one another.



In the preceding diagram, the `Invoke` method of the `WorkflowInvoker` class is used to invoke several workflow instances. `WorkflowInvoker` is used for lightweight workflows that do not need management from the host; workflows that need management from the host (such as `Bookmark` resumption) must be executed using `Run` instead. It isn't required to wait for one workflow instance to complete before invoking another; the runtime engine supports running multiple workflow instances simultaneously. The workflows invoked are as follows:

- A `Sequence` activity that contains a `WriteLine` child activity. A `Variable` of the parent activity is bound to an `InArgument` of the child activity. For more information about on variables, arguments, and binding, see [Variables and Arguments](#).
- A custom activity called `ReadLine`. An `OutArgument` of the `ReadLine` activity is returned to the calling `Invoke` method.
- A custom activity that derives from the `CodeActivity` abstract class. The `CodeActivity` can access run-time features (such as tracking and properties) using the `CodeActivityContext` that is available as a parameter of the `Execute` method. For more information about these run-time features, see [Workflow Tracking and Tracing](#) and [Workflow Execution Properties](#).

## See also

- [BizTalk Server 2006 or WF? Choosing the Right Workflow Tool for Your Project](#)

# Fundamental Windows Workflow Concepts

3/9/2019 • 3 minutes to read • [Edit Online](#)

Workflow development in the .NET Framework 4.6.1 uses concepts that may be new to some developers. This topic describes some of the concepts and how they are implemented.

## Workflows and Activities

A workflow is a structured collection of actions that models a process. Each action in the workflow is modeled as an activity. A host interacts with a workflow by using [WorkflowInvoker](#) for invoking a workflow as if it were a method, [WorkflowApplication](#) for explicit control over the execution of a single workflow instance, and [WorkflowServiceHost](#) for message-based interactions in multi-instance scenarios. Because steps of the workflow are defined as a hierarchy of activities, the topmost activity in the hierarchy can be said to define the workflow itself. This hierarchy model takes the place of the explicit `SequentialWorkflow` and `StateMachineWorkflow` classes from previous versions. Activities themselves are developed as collections of other activities (using the [Activity](#) class as a base, usually defined by using XAML) or are custom created by using the [CodeActivity](#) class, which can use the runtime for data access, or by using the [NativeActivity](#) class, which exposes the breadth of the workflow runtime to the activity author. Activities developed by using [CodeActivity](#) and [NativeActivity](#) are created by using CLR-compliant languages such as C#.

## Activity Data Model

Activities store and share data by using the types shown in the following table.

| Variable   | Stores data in an activity.  |
|------------|--|
| Argument   | Moves data into and out of an activity.                              |
| Expression | An activity with an elevated return value used in argument bindings. |

## Workflow Runtime

The workflow runtime is the environment in which workflows execute. [WorkflowInvoker](#) is the simplest way to execute a workflow. The host uses [WorkflowInvoker](#) for the following:

- To synchronously invoke a workflow.
- To provide input to, or retrieve output from a workflow.
- To add extensions to be used by activities.

[ActivityInstance](#) is the thread-safe proxy that hosts can use to interact with the runtime. The host uses [ActivityInstance](#) for the following:

- To acquire an instance by creating it or loading it from an instance store.
- To be notified of instance life-cycle events.
- To control workflow execution.

- To provide input to, or retrieve output from a workflow.
- To signal a workflow continuation and pass values into the workflow.
- To persist workflow data.
- To add extensions to be used by activities.

Activities gain access to the workflow runtime environment by using the appropriate [ActivityContext](#) derived class, such as [NativeActivityContext](#) or [CodeActivityContext](#). They use this for resolving arguments and variables, for scheduling child activities, and for many other purposes.

## Services

Workflows provide a natural way to implement and access loosely-coupled services, using messaging activities. Messaging activities are built on WCF and are the primary mechanism used to get data into and out of a workflow. You can compose messaging activities together to model any kind of message exchange pattern you wish. For more information, see [Messaging Activities](#). Workflow services are hosted using the [WorkflowServiceHost](#) class. For more information, see [Hosting Workflow Services Overview](#). For more information about workflow services see [Workflow Services](#)

## Persistence, Unloading, and Long-Running Workflows

Windows Workflow simplifies the authoring of long-running reactive programs by providing:

- Activities that access external input.
- The ability to create [Bookmark](#) objects that can be resumed by a host listener.
- The ability to persist a workflow's data and unload the workflow, and then reload and reactivate the workflow in response to the resumption of [Bookmark](#) objects in a particular workflow.

A workflow continuously executes activities until there are no more activities to execute or until all currently executing activities are waiting for input. In this latter state, the workflow is idle. It is common for a host to unload workflows that have gone idle and to reload them to continue execution when a message arrives.

[WorkflowServiceHost](#) provides functionality for this feature and provides an extensible unload policy. For blocks of execution that use volatile state data or other data that cannot be persisted, an activity can indicate to a host that it should not be persisted by using the [NoPersistHandle](#). A workflow can also explicitly persist its data to a durable storage medium by using the [Persist](#) activity.

# Windows Workflow Architecture

3/9/2019 • 2 minutes to read • [Edit Online](#)

Windows Workflow Foundation (WF) raises the abstraction level for developing interactive long-running applications. Units of work are encapsulated as activities. Activities run in an environment that provides facilities for flow control, exception handling, fault propagation, persistence of state data, loading and unloading of in-progress workflows from memory, tracking, and transaction flow.

## Activity Architecture

Activities are developed as CLR types that derive from either [Activity](#), [CodeActivity](#), [AsyncCodeActivity](#), or [NativeActivity](#), or their variants that return a value, [Activity<TResult>](#), [CodeActivity<TResult>](#), [AsyncCodeActivity<TResult>](#), or [NativeActivity<TResult>](#). Developing activities that derive from [Activity](#) allows the user to assemble pre-existing activities to quickly create units of work that execute in the workflow environment. [CodeActivity](#), on the other hand, enables execution logic to be authored in managed code using [CodeActivityContext](#) primarily for access to activity arguments. [AsyncCodeActivity](#) is similar to [CodeActivity](#) except that it can be used to implement asynchronous tasks. Developing activities that derive from [NativeActivity](#) allows users to access the runtime through the [NativeActivityContext](#) for functionality like scheduling children, creating bookmarks, invoking asynchronous work, registering transactions, and more.

Authoring activities that derive from [Activity](#) is declarative and these activities can be authored in XAML. In the following example, an activity called `Prompt` is created using other activities for the execution body.

```
<Activity x:Class='Prompt'  
  xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml'  
  xmlns:z='http://schemas.microsoft.com/netfx/2008/xaml/schema'  
  xmlns:my='clr-namespace:XAMLActivityDefinition;assembly=XAMLActivityDefinition'  
  xmlns:s="clr-namespace:System;assembly=mscorlib"  
  xmlns="http://schemas.microsoft.com/2009/workflow">  
  <z:SchemaType.Members>  
    <z:SchemaType.SchemaProperty Name='Text' Type=' InArgument(s:String)' />  
    <z:SchemaType.SchemaProperty Name='Response' Type='OutArgument(s:String)' />  
  </z:SchemaType.Members>  
  <Sequence>  
    <my:WriteLine Text='[Text]' />  
    <my:ReadLine BookmarkName='r1' Result='[Response]' />  
  </Sequence>  
</Activity>
```

## Activity Context

The [ActivityContext](#) is the activity author's interface to the workflow runtime and provides access to the runtime's wealth of features. In the following example, an activity is defined that uses the execution context to create a bookmark (the mechanism that allows an activity to register a continuation point in its execution that can be resumed by a host passing data into the activity).

```

public sealed class ReadLine : NativeActivity<string>
{
    [RequiredArgument]
    public InArgument<string> BookmarkName { get; set; }

    protected override void Execute(NativeActivityContext context)
    {
        // Create a Bookmark and wait for it to be resumed.
        context.CreateBookmark(BookmarkName.Get(context),
            new BookmarkCallback(OnResumeBookmark));
    }

    // NativeActivity derived activities that do asynchronous operations by calling
    // one of the CreateBookmark overloads defined on System.Activities.NativeActivityContext
    // must override the CanInduceIdle property and return true.
    protected override bool CanInduceIdle
    {
        get { return true; }
    }

    public void OnResumeBookmark(NativeActivityContext context, Bookmark bookmark, object obj)
    {
        // When the Bookmark is resumed, assign its value to
        // the Result argument.
        Result.Set(context, (string)obj);
    }
}

```

## Activity Life Cycle

An instance of an activity starts out in the [Executing](#) state. Unless exceptions are encountered, it remains in this state until all child activities are finished executing and any other pending work ([Bookmark](#) objects, for instance) is completed, at which point it transitions to the [Closed](#) state. The parent of an activity instance can request a child to cancel; if the child is able to be canceled it completes in the [Canceled](#) state. If an exception is thrown during execution, the runtime puts the activity into the [Faulted](#) state and propagates the exception up the parent chain of activities. Following are the three completion states of an activity:

- **Closed:** The activity has completed its work and exited.
- **Canceled:** The activity has gracefully abandoned its work and exited. Work is not explicitly rolled back when this state is entered.
- **Faulted:** The activity has encountered an error and has exited without completing its work.

Activities remain in the [Executing](#) state when they are persisted or unloaded.

# Getting Started Tutorial

3/9/2019 • 2 minutes to read • [Edit Online](#)

This section contains a set of walkthrough topics that introduce you to programming Windows Workflow Foundation (WF) applications. By following the procedures in these topics, you will build an application that is a number guessing game. The first topic in the tutorial leads you through the steps to create the custom activities required for the workflow. In the second topic, these activities are assembled along with built-in workflow activities into a flowchart workflow. In the third topic, the host application is configured to run the workflow, and in the final topic persistence is introduced. Each step in this process depends on the previous steps, so we recommend that you complete them in order.

## In This Section

### [How to: Create an Activity](#)

Describes how to create a custom activity that derives from `NativeActivity<TResult>`, and how to compose this activity along with a built-in activity into a composite activity using the activity designer.

### [How to: Create a Workflow](#)

Describes how to create flowchart, sequential, and state machine workflows by using built-in activities and the custom activities from the preceding tutorial.

### [How to: Run a Workflow](#)

Describes how to invoke a workflow from a host environment, pass data into and out of a workflow, and how to resume bookmarks.

### [How to: Create and Run a Long Running Workflow](#)

Describes how to add persistence to a workflow application.

### [How to: Create a Custom Tracking Participant](#)

Describes how to create a custom tracking participant and tracking profile.

### [How to: Host Multiple Versions of a Workflow Side-by-Side](#)

Describes how to use `WorkflowIdentity` to host multiple versions of a workflow side-by-side.

### [How to: Update the Definition of a Running Workflow Instance](#)

Describes how to use dynamic update to modify running workflow instances.

## See also

- [Windows Workflow Foundation Programming](#)

# How to: Create an Activity

3/9/2019 • 5 minutes to read • [Edit Online](#)

Activities are the core unit of behavior in WF. The execution logic of an activity can be implemented in managed code or it can be implemented by using other activities. This topic demonstrates how to create two activities. The first activity is a simple activity that uses code to implement its execution logic. The implementation of the second activity is defined by using other activities. These activities are used in following steps in the tutorial.

## NOTE

To download a completed version of the tutorial, see [Windows Workflow Foundation \(WF45\) - Getting Started Tutorial](#).

## Create the activity library project

1. Open Visual Studio and choose **New > Project** from the **File** menu.
2. In the **New Project** dialog, under the **Installed** category, select **Visual C# > Workflow** (or **Visual Basic > Workflow**).

## NOTE

If you don't see the **Workflow** template category, you may need to install the **Windows Workflow Foundation** component of Visual Studio. Choose the **Open Visual Studio Installer** link on the left-hand side of the **New Project** dialog. In Visual Studio Installer, select the **Individual components** tab. Then, under the **Development activities** category, select the **Windows Workflow Foundation** component. Choose **Modify** to install the component.

3. Select the **Activity Library** project template. Type `NumberGuessWorkflowActivities` in the **Name** box and then click **OK**.
4. Right-click **Activity1.xaml** in **Solution Explorer** and choose **Delete**. Click **OK** to confirm.

## Create the ReadInt activity

1. Choose **Add New Item** from the **Project** menu.
2. In the **Installed > Common Items** node, select **Workflow**. Select **Code Activity** from the **Workflow** list.
3. Type `ReadInt` into the **Name** box and then click **Add**.
4. Replace the existing `ReadInt` definition with the following definition.

```

public sealed class ReadInt : NativeActivity<int>
{
    [RequiredArgument]
    public InArgument<string> BookmarkName { get; set; }

    protected override void Execute(NativeActivityContext context)
    {
        string name = BookmarkName.Get(context);

        if (string.IsNullOrEmpty(name))
        {
            throw new ArgumentException("BookmarkName cannot be an Empty string.", "BookmarkName");
        }

        context.CreateBookmark(name, new BookmarkCallback(OnReadComplete));
    }

    // NativeActivity derived activities that do asynchronous operations by calling
    // one of the CreateBookmark overloads defined on System.Activities.NativeActivityContext
    // must override the CanInduceIdle property and return true.
    protected override bool CanInduceIdle
    {
        get { return true; }
    }

    void OnReadComplete(NativeActivityContext context, Bookmark bookmark, object state)
    {
        this.Result.Set(context, Convert.ToInt32(state));
    }
}

```

```

Public NotInheritable Class ReadInt
    Inherits NativeActivity(Of Integer)

    <RequiredArgument()>
    Property BookmarkName() As InArgument(Of String)

    Protected Overrides Sub Execute(ByVal context As NativeActivityContext)
        Dim name As String
        name = BookmarkName.Get(context)

        If name = String.Empty Then
            Throw New ArgumentException("BookmarkName cannot be an Empty string.", "BookmarkName")
        End If

        context.CreateBookmark(name, New BookmarkCallback(AddressOf OnReadComplete))
    End Sub

    ' NativeActivity derived activities that do asynchronous operations by calling
    ' one of the CreateBookmark overloads defined on System.Activities.NativeActivityContext
    ' must override the CanInduceIdle property and return True.
    Protected Overrides ReadOnly Property CanInduceIdle As Boolean
        Get
            Return True
        End Get
    End Property

    Sub OnReadComplete(ByVal context As NativeActivityContext, ByVal bookmark As Bookmark, ByVal state As Object)
        Result.Set(context, Convert.ToInt32(state))
    End Sub

End Class

```

**NOTE**

The `ReadInt` activity derives from `NativeActivity<TResult>` instead of `CodeActivity`, which is the default for the code activity template. `CodeActivity<TResult>` can be used if the activity provides a single result, which is exposed through the `Result` argument, but `CodeActivity<TResult>` does not support the use of bookmarks, so `NativeActivity<TResult>` is used.

## Create the Prompt activity

1. Press **Ctrl+Shift+B** to build the project. Building the project enables the `ReadInt` activity in this project to be used to build the custom activity from this step.
2. Choose **Add New Item** from the **Project** menu.
3. In the **Installed > Common Items** node, select **Workflow**. Select **Activity** from the **Workflow** list.
4. Type `Prompt` into the **Name** box and then click **Add**.
5. Double-click `Prompt.xaml` in **Solution Explorer** to display it in the designer if it is not already displayed.
6. Click **Arguments** in the lower-left side of the activity designer to display the **Arguments** pane.
7. Click **Create Argument**.
8. Type `BookmarkName` into the **Name** box, select **In** from the **Direction** drop-down list, select **String** from the **Argument type** drop-down list, and then press **Enter** to save the argument.
9. Click **Create Argument**.
10. Type `Result` into the **Name** box that is underneath the newly added `BookmarkName` argument, select **Out** from the **Direction** drop-down list, select **Int32** from the **Argument type** drop-down list, and then press **Enter**.
11. Click **Create Argument**.
12. Type `Text` into the **Name** box, select **In** from the **Direction** drop-down list, select **String** from the **Argument type** drop-down list, and then press **Enter** to save the argument.

These three arguments are bound to the corresponding arguments of the `WriteLine` and `ReadInt` activities that are added to the `Prompt` activity in the following steps.

13. Click **Arguments** in the lower-left side of the activity designer to close the **Arguments** pane.
14. Drag a **Sequence** activity from the **Control Flow** section of the **Toolbox** and drop it onto the **Drop activity here** label of the **Prompt** activity designer.

**TIP**

If the **Toolbox** window is not displayed, select **Toolbox** from the **View** menu.

15. Drag a **WriteLine** activity from the **Primitives** section of the **Toolbox** and drop it onto the **Drop activity here** label of the **Sequence** activity.
16. Bind the **Text** argument of the **WriteLine** activity to the **Text** argument of the **Prompt** activity by typing `Text` into the **Enter a C# expression** or **Enter a VB expression** box in the **Properties** window, and then press the **Tab** key two times. This dismisses the IntelliSense list members window and saves the property value by moving the selection off the property. This property can also be set by typing `Text` into

the **Enter a C# expression** or **Enter a VB expression** box on the activity itself.

**TIP**

If the **Properties Window** is not displayed, select **Properties Window** from the **View** menu.

17. Drag a **ReadInt** activity from the **NumberGuessWorkflowActivities** section of the **Toolbox** and drop it in the **Sequence** activity so that it follows the **WriteLine** activity.
18. Bind the **BookmarkName** argument of the **ReadInt** activity to the **BookmarkName** argument of the **Prompt** activity by typing `BookmarkName` into the **Enter a VB expression** box to the right of the **BookmarkName** argument in the **Properties Window**, and then press the **Tab** key two times to close the IntelliSense list members window and save the property.
19. Bind the **Result** argument of the **ReadInt** activity to the **Result** argument of the **Prompt** activity by typing `Result` into the **Enter a VB expression** box to the right of the **Result** argument in the **Properties Window**, and then press the **Tab** key two times.
20. Press **Ctrl+Shift+B** to build the solution.

## Next steps

For instructions on how to create a workflow by using these activities, see the next step in the tutorial, [How to: Create a Workflow](#).

## See also

- [CodeActivity](#)
- [NativeActivity<TResult>](#)
- [Designing and Implementing Custom Activities](#)
- [Getting Started Tutorial](#)
- [How to: Create a Workflow](#)
- [Using the ExpressionTextBox in a Custom Activity Designer](#)

# How to: Create a Workflow

3/9/2019 • 2 minutes to read • [Edit Online](#)

Workflows can be constructed from built-in activities as well as from custom activities. The topics in this section step through creating a workflow that uses both built-in activities such as the [Flowchart](#) activity, and the custom activities from the previous [How to: Create an Activity](#) topic. The workflow models a number guessing game. Only one of the topics in this section is required to complete the tutorial; you should pick the style that interests you and follow that step. However, you may complete all of the topics if desired.

## NOTE

Each topic in the Getting Started tutorial depends on the previous topics. To complete this topic, you must first complete [How to: Create an Activity](#).

## NOTE

To download a completed version of the tutorial, see [Windows Workflow Foundation \(WF45\) - Getting Started Tutorial](#).

## In This Section

### [How to: Create a Sequential Workflow](#)

Describes how to create a sequential workflow using the [Sequence](#) activity.

### [How to: Create a Flowchart Workflow](#)

Describes how to create a flowchart workflow using the [Flowchart](#) activity.

### [How to: Create a State Machine Workflow](#)

Describes how to create a state machine workflow using the [StateMachine](#) activity.

## See also

- [Windows Workflow Foundation Programming](#)

# How to: Create a Flowchart Workflow

3/9/2019 • 6 minutes to read • [Edit Online](#)

Workflows can be constructed from built-in activities as well as from custom activities. This topic steps through creating a workflow that uses both built-in activities such as the **Flowchart** activity, and the custom activities from the previous [How to: Create an Activity](#) topic. The workflow models a number guessing game.

## NOTE

Each topic in the Getting Started tutorial depends on the previous topics. To complete this topic, you must first complete [How to: Create an Activity](#).

## NOTE

To download a completed version of the tutorial, see [Windows Workflow Foundation \(WF45\) - Getting Started Tutorial](#).

## To create the workflow

1. Right-click **NumberGuessWorkflowActivities** in **Solution Explorer** and select **Add, New Item**.
2. In the **Installed, Common Items** node, select **Workflow**. Select **Activity** from the **Workflow** list.
3. Type `FlowchartNumberGuessWorkflow` into the **Name** box and click **Add**.
4. Drag a **Flowchart** activity from the **Flowchart** section of the **Toolbox** and drop it onto the **Drop activity here** label on the workflow design surface.

## To create the workflow variables and arguments

1. Double-click **FlowchartNumberGuessWorkflow.xaml** in **Solution Explorer** to display the workflow in the designer, if it is not already displayed.
2. Click **Arguments** in the lower-left side of the workflow designer to display the **Arguments** pane.
3. Click **Create Argument**.
4. Type `MaxNumber` into the **Name** box, select **In** from the **Direction** drop-down list, select **Int32** from the **Argument type** drop-down list, and then press ENTER to save the argument.
5. Click **Create Argument**.
6. Type `Turns` into the **Name** box that is below the newly added `MaxNumber` argument, select **Out** from the **Direction** drop-down list, select **Int32** from the **Argument type** drop-down list, and then press ENTER.
7. Click **Arguments** in the lower-left side of the activity designer to close the **Arguments** pane.
8. Click **Variables** in the lower-left side of the workflow designer to display the **Variables** pane.
9. Click **Create Variable**.

## TIP

If no **Create Variable** box is displayed, click the **Flowchart** activity on the workflow designer surface to select it.

10. Type `Guess` into the **Name** box, select **Int32** from the **Variable type** drop-down list, and then press **ENTER** to save the variable.
11. Click **Create Variable**.
12. Type `Target` into the **Name** box, select **Int32** from the **Variable type** drop-down list, and then press **ENTER** to save the variable.
13. Click **Variables** in the lower-left side of the activity designer to close the **Variables** pane.

#### To add the workflow activities

1. Drag an **Assign** activity from the **Primitives** section of the **Toolbox** and hover it over the **Start** node, which is at the top of the flowchart. When the **Assign** activity is over the **Start** node, three triangles will appear around the **Start** node. Drop the **Assign** activity on the triangle that is directly below the **Start** node. This will link the two items together and designates the **Assign** activity as the first activity in the flowchart.

#### NOTE

Activities can also be indicated as the starting activity in the workflow by manually linking them activity to the start node. To do this, hover the mouse over the **Start** node, click one of the rectangles that appear when the mouse is over the **Start** node, and drag the connecting line down to the desired activity and drop it on one of the rectangles that appear. You can also designate an activity as the starting activity by right-clicking the it and choosing **Set as Start Node**.

2. Type `Target` into the **To** box and the following expression into the **Enter a C# Expression** or **Enter a VB expression** box.

```
New System.Random().Next(1, MaxNumber + 1)
```

```
new System.Random().Next(1, MaxNumber + 1)
```

#### TIP

If the **Toolbox** window is not displayed, select **Toolbox** from the **View** menu.

3. Drag a **Prompt** activity from the **NumberGuessWorkflowActivities** section of the **Toolbox**, drop it below the **Assign** activity from the previous step, and connect the **Prompt** activity to the **Assign** activity. There are three ways to connect the two activities. The first way is to connect them as you drop the **Prompt** activity on the workflow. As you are dragging the **Prompt** activity to the workflow, hover it over the **Assign** activity and drop it onto one of the four triangles that appear when the **Prompt** activity is over the **Assign** activity. The second way is to drop the **Prompt** activity onto the workflow at the desired location. Then, hover the mouse over the **Assign** activity and drag one of the rectangles that appears down to the **Prompt** activity. Drag the mouse so that the connecting line from the **Assign** activity connects to one of the rectangles of the **Prompt** activity, and then release the mouse button. The third way is very similar to the first way, except that instead of dragging the **Prompt** activity from the **Toolbox**, you drag it from its location on the workflow design surface, hover it over the **Assign** activity, and drop it onto one of the triangles that appears.
4. In the **Properties Window** for the **Prompt** activity, type `"EnterGuess"` including the quotes into the **BookmarkName** property value box. Type `Guess` into the **Result** property value box, and type the following expression into the **Text** property box.

```
"Please enter a number between 1 and " & MaxNumber
```

```
"Please enter a number between 1 and " + MaxNumber
```

**TIP**

If the **Properties Window** is not displayed, select **Properties Window** from the **View** menu.

5. Drag an **Assign** activity from the **Primitives** section of the **Toolbox** and connect it using one of the methods described in the previous step so that it is below the **Prompt** activity.
6. Type `Turns` into the **To** box and `Turns + 1` into the **Enter a C# expression** or **Enter a VB expression** box.
7. Drag a **FlowDecision** from the **Flowchart** section of the **Toolbox** and connect it below the **Assign** activity. In the **Properties Window**, type the following expression into the **Condition** property value box.

```
Guess = Target
```

```
Guess == Target
```

8. Drag another **FlowDecision** activity from the **Toolbox** and drop it below the first one. Connect the two activities by dragging from the rectangle that is labeled **False** on the top **FlowDecision** activity to the rectangle at the top of the second **FlowDecision** activity.

**TIP**

If you do not see the **True** and **False** labels on the **FlowDecision**, hover the mouse over the **FlowDecision**.

9. Click the second **FlowDecision** activity to select it. In the **Properties Window**, type the following expression into the **Condition** property value box.

```
Guess < Target
```

10. Drag two **WriteLine** activities from the **Primitives** section of the **Toolbox** and drop them so that they are side by side below the two **FlowDecision** activities. Connect the **True** action of the bottom **FlowDecision** activity to the leftmost **WriteLine** activity, and the **False** action to the rightmost **WriteLine** activity.

11. Click the leftmost **WriteLine** activity to select it, and type the following expression into the **Text** property value box in the **Properties Window**.

```
"Your guess is too low."
```

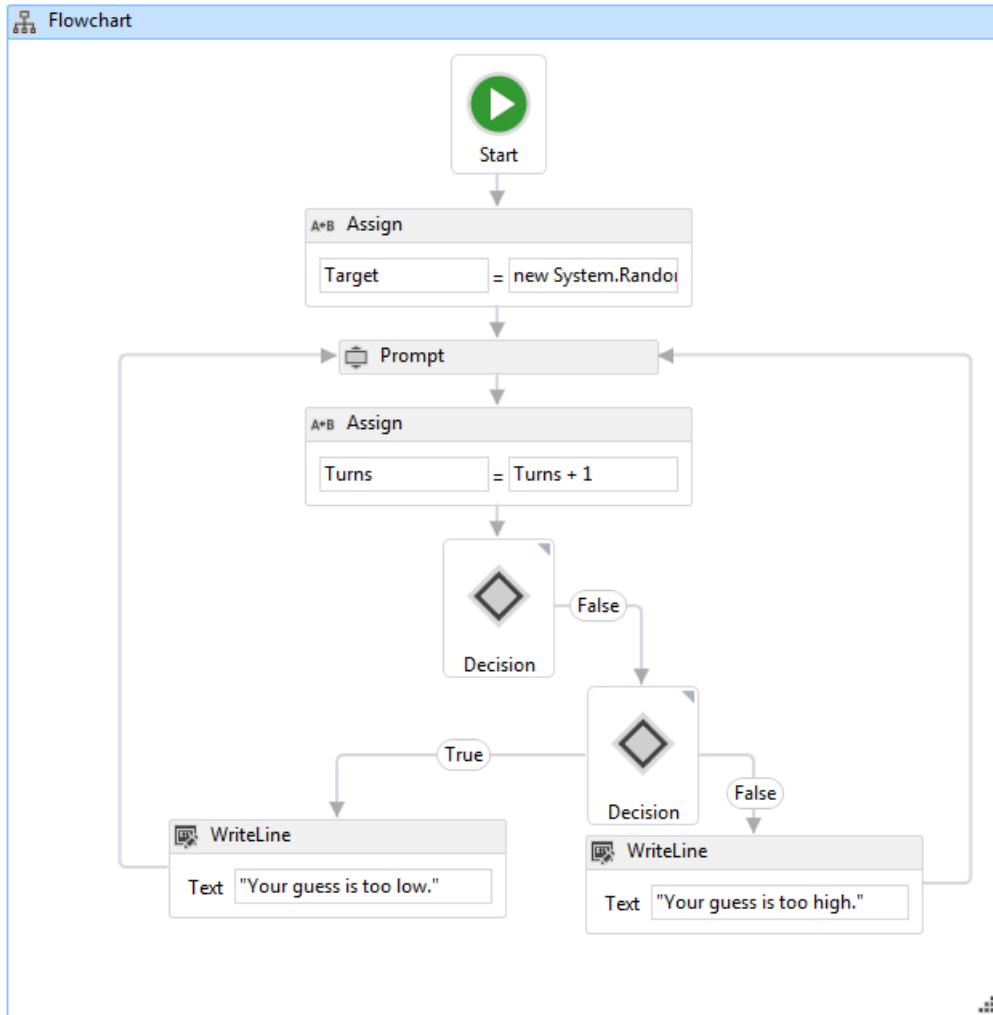
12. Connect the **WriteLine** to the left side of the **Prompt** activity that is above it.

13. Click the rightmost **WriteLine** activity to select it, and type the following expression into the **Text** property value box in the **Properties Window**.

```
"Your guess is too high."
```

14. Connect the **WriteLine** activity to the right side of the **Prompt** activity above it.

The following example illustrates the completed workflow.



### To build the workflow

1. Press CTRL+SHIFT+B to build the solution.

For instructions on how to run the workflow, please see the next topic, [How to: Run a Workflow](#). If you have already completed the [How to: Run a Workflow](#) step with a different style of workflow and wish to run it using the flowchart workflow from this step, skip ahead to the [To build and run the application](#) section of [How to: Run a Workflow](#).

### See also

- [Flowchart](#)
- [FlowDecision](#)
- [Windows Workflow Foundation Programming](#)
- [Designing Workflows](#)
- [Getting Started Tutorial](#)
- [How to: Create an Activity](#)
- [How to: Run a Workflow](#)

# How to: Create a Sequential Workflow

3/26/2019 • 4 minutes to read • [Edit Online](#)

Workflows can be constructed from built-in activities as well as from custom activities. This topic steps through creating a workflow that uses both built-in activities such as the **Sequence** activity, and the custom activities from the previous [How to: Create an Activity](#) topic. The workflow models a number guessing game.

## NOTE

Each topic in the Getting Started tutorial depends on the previous topics. To complete this topic, you must first complete [How to: Create an Activity](#).

## NOTE

To download a completed version of the tutorial, see [Windows Workflow Foundation \(WF45\) - Getting Started Tutorial](#).

## To create the workflow

1. Right-click **NumberGuessWorkflowActivities** in **Solution Explorer** and select **Add, New Item**.
2. In the **Installed, Common Items** node, select **Workflow**. Select **Activity** from the **Workflow** list.
3. Type `SequentialNumberGuessWorkflow` into the **Name** box and click **Add**.
4. Drag a **Sequence** activity from the **Control Flow** section of the **Toolbox** and drop it onto the **Drop activity here** label on the workflow design surface.

## To create the workflow variables and arguments

1. Double-click **SequentialNumberGuessWorkflow.xaml** in **Solution Explorer** to display the workflow in the designer, if it is not already displayed.
2. Click **Arguments** in the lower-left side of the workflow designer to display the **Arguments** pane.
3. Click **Create Argument**.
4. Type `MaxNumber` into the **Name** box, select **In** from the **Direction** drop-down list, select **Int32** from the **Argument type** drop-down list, and then press ENTER to save the argument.
5. Click **Create Argument**.
6. Type `Turns` into the **Name** box that is below the newly added `MaxNumber` argument, select **Out** from the **Direction** drop-down list, select **Int32** from the **Argument type** drop-down list, and then press ENTER.
7. Click **Arguments** in the lower-left side of the activity designer to close the **Arguments** pane.
8. Click **Variables** in the lower-left side of the workflow designer to display the **Variables** pane.
9. Click **Create Variable**.

**TIP**

If no **Create Variable** box is displayed, click the **Sequence** activity on the workflow designer surface to select it.

10. Type `Guess` into the **Name** box, select **Int32** from the **Variable type** drop-down list, and then press **ENTER** to save the variable.
11. Click **Create Variable**.
12. Type `Target` into the **Name** box, select **Int32** from the **Variable type** drop-down list, and then press **ENTER** to save the variable.
13. Click **Variables** in the lower-left side of the activity designer to close the **Variables** pane.

## To add the workflow activities

1. Drag an **Assign** activity from the **Primitives** section of the **Toolbox** and drop it onto the **Sequence** activity. Type `Target` into the **To** box and the following expression into the **Enter a C# expression** or **Enter a VB expression** box.

```
New System.Random().Next(1, MaxNumber + 1)
```

```
new System.Random().Next(1, MaxNumber + 1)
```

**TIP**

If the **Toolbox** window is not displayed, select **Toolbox** from the **View** menu.

2. Drag a **DoWhile** activity from the **Control Flow** section of the **Toolbox** and drop it on the workflow so that it is below the **Assign** activity.
3. Type the following expression into the **DoWhile** activity's **Condition** property value box.

```
Guess <> Target
```

```
Guess != Target
```

A **DoWhile** activity executes its child activities and then evaluates its **Condition**. If the **Condition** evaluates to `True`, then the activities in the **DoWhile** execute again. In this example, the user's guess is evaluated and the **DoWhile** continues until the guess is correct.

4. Drag a **Prompt** activity from the **NumberGuessWorkflowActivities** section of the **Toolbox** and drop it in the **DoWhile** activity from the previous step.
5. In the **Properties Window**, type `"EnterGuess"` including the quotes into the **BookmarkName** property value box for the **Prompt** activity. Type `Guess` into the **Result** property value box, and type the following expression into the **Text** property box.

```
"Please enter a number between 1 and " & MaxNumber
```

```
"Please enter a number between 1 and " + MaxNumber
```

**TIP**

If the **Properties Window** is not displayed, select **Properties Window** from the **View** menu.

6. Drag an **Assign** activity from the **Primitives** section of the **Toolbox** and drop it in the **DoWhile** activity so that it follows the **Prompt** activity.

**NOTE**

When you drop the **Assign** activity, note how the workflow designer automatically adds a **Sequence** activity to contain both the **Prompt** activity and the newly added **Assign** activity.

7. Type `Turns` into the **To** box and `Turns + 1` into the **Enter a C# expression** or **Enter a VB expression** box.
8. Drag an **If** activity from the **Control Flow** section of the **Toolbox** and drop it in the **Sequence** activity so that it follows the newly added **Assign** activity.
9. Type the following expression into the **If** activity's **Condition** property value box.

```
Guess <> Target
```

```
Guess != Target
```

10. Drag another **If** activity from the **Control Flow** section of the **Toolbox** and drop it in the **Then** section of the first **If** activity.
11. Type the following expression into the newly added **If** activity's **Condition** property value box.

```
Guess < Target
```

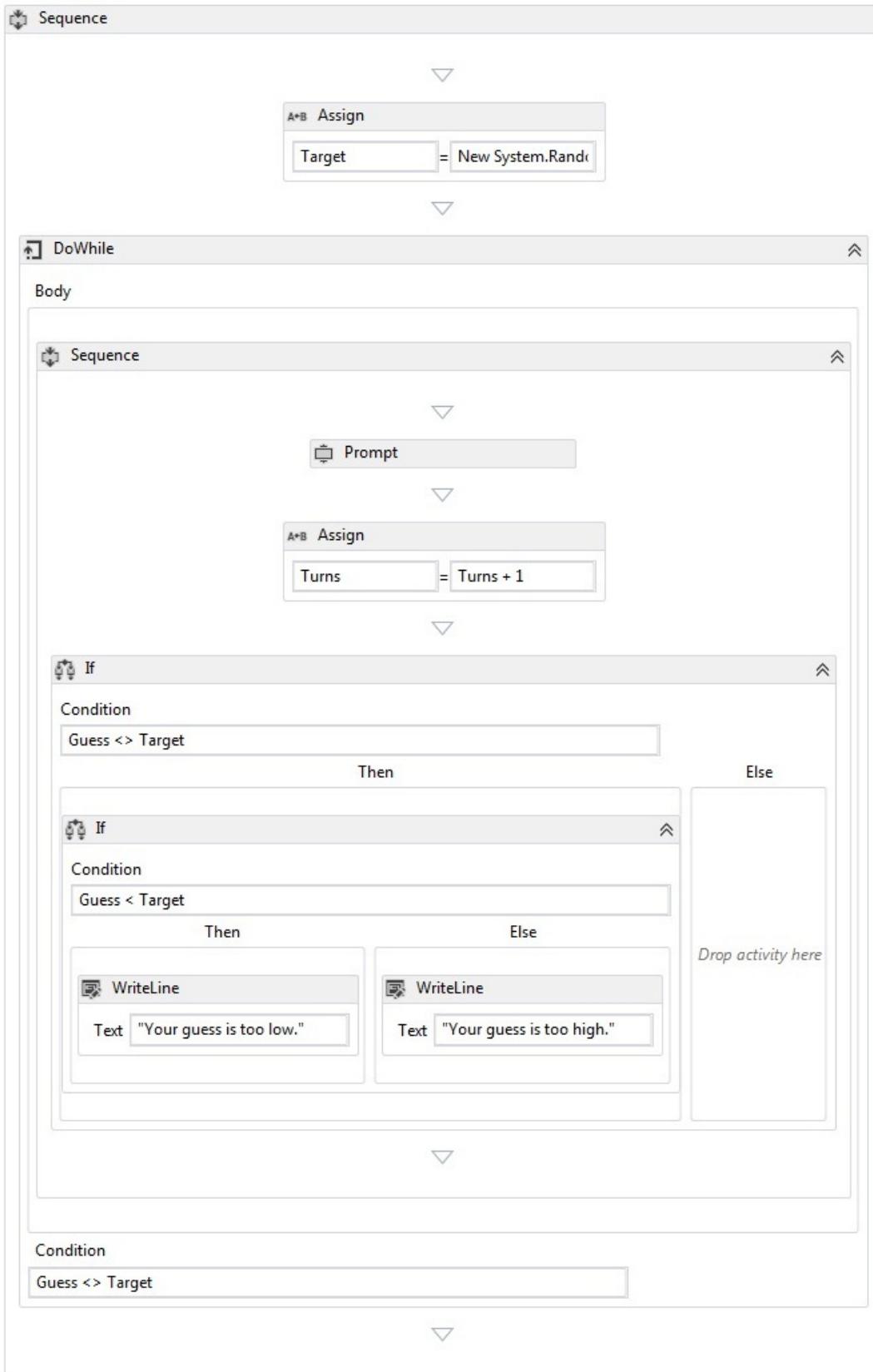
12. Drag two **WriteLine** activities from the **Primitives** section of the **Toolbox** and drop them so that one is in the **Then** section of the newly added **If** activity, and one is in the **Else** section.
13. Click the **WriteLine** activity in the **Then** section to select it, and type the following expression into the **Text** property value box.

```
"Your guess is too low."
```

14. Click the **WriteLine** activity in the **Else** section to select it, and type the following expression into the **Text** property value box.

```
"Your guess is too high."
```

The following example illustrates the completed workflow:



## To build the workflow

1. Press CTRL+SHIFT+B to build the solution.

For instructions on how to run the workflow, please see the next topic, [How to: Run a Workflow](#). If you have already completed the [How to: Run a Workflow](#) step with a different style of workflow and wish to run it using the sequential workflow from this step, skip ahead to the [To build and run the application](#) section of [How to: Run a Workflow](#).

## See also

- [Flowchart](#)
- [FlowDecision](#)
- [Windows Workflow Foundation Programming](#)
- [Designing Workflows](#)
- [Getting Started Tutorial](#)
- [How to: Create an Activity](#)
- [How to: Run a Workflow](#)

# How to: Create a State Machine Workflow

3/9/2019 • 7 minutes to read • [Edit Online](#)

Workflows can be constructed from built-in activities as well as from custom activities. This topic steps through creating a workflow that uses both built-in activities such as the **StateMachine** activity, and the custom activities from the previous [How to: Create an Activity](#) topic. The workflow models a number guessing game.

## NOTE

Each topic in the Getting Started tutorial depends on the previous topics. To complete this topic, you must first complete [How to: Create an Activity](#).

## NOTE

To download a completed version of the tutorial, see [Windows Workflow Foundation \(WF45\) - Getting Started Tutorial](#).

## To create the workflow

1. Right-click **NumberGuessWorkflowActivities** in **Solution Explorer** and select **Add, New Item**.
2. In the **Installed, Common Items** node, select **Workflow**. Select **Activity** from the **Workflow** list.
3. Type `StateMachineNumberGuessWorkflow` into the **Name** box and click **Add**.
4. Drag a **StateMachine** activity from the **State Machine** section of the **Toolbox** and drop it onto the **Drop activity here** label on the workflow design surface.

## To create the workflow variables and arguments

1. Double-click **StateMachineNumberGuessWorkflow.xaml** in **Solution Explorer** to display the workflow in the designer, if it is not already displayed.
2. Click **Arguments** in the lower-left side of the workflow designer to display the **Arguments** pane.
3. Click **Create Argument**.
4. Type `MaxNumber` into the **Name** box, select **In** from the **Direction** drop-down list, select **Int32** from the **Argument type** drop-down list, and then press ENTER to save the argument.
5. Click **Create Argument**.
6. Type `Turns` into the **Name** box that is below the newly added `MaxNumber` argument, select **Out** from the **Direction** drop-down list, select **Int32** from the **Argument type** drop-down list, and then press ENTER.
7. Click **Arguments** in the lower-left side of the activity designer to close the **Arguments** pane.
8. Click **Variables** in the lower-left side of the workflow designer to display the **Variables** pane.
9. Click **Create Variable**.

## TIP

If no **Create Variable** box is displayed, click the **StateMachine** activity on the workflow designer surface to select it.

10. Type `Guess` into the **Name** box, select **Int32** from the **Variable type** drop-down list, and then press **ENTER** to save the variable.
11. Click **Create Variable**.
12. Type `Target` into the **Name** box, select **Int32** from the **Variable type** drop-down list, and then press **ENTER** to save the variable.
13. Click **Variables** in the lower-left side of the activity designer to close the **Variables** pane.

#### To add the workflow activities

1. Click **State1** to select it. In the **Properties Window**, change the **DisplayName** to `Initialize Target`.

##### TIP

If the **Properties Window** is not displayed, select **Properties Window** from the **View** menu.

2. Double-click the newly renamed **Initialize Target** state in the workflow designer to expand it.
3. Drag an **Assign** activity from the **Primitives** section of the **Toolbox** and drop it onto the **Entry** section of the state. Type `Target` into the **To** box and the following expression into the **Enter a C# expression** or **Enter a VB expression** box.

```
New System.Random().Next(1, MaxNumber + 1)
```

```
new System.Random().Next(1, MaxNumber + 1)
```

##### TIP

If the **Toolbox** window is not displayed, select **Toolbox** from the **View** menu.

4. Return to the overall state machine view in the workflow designer by clicking **StateMachine** in the breadcrumb display at the top of the workflow designer.
5. Drag a **State** activity from the **State Machine** section of the **Toolbox** onto the workflow designer and hover it over the **Initialize Target** state. Note that four triangles will appear around the **Initialize Target** state when the new state is over it. Drop the new state on the triangle that is immediately below the **Initialize Target** state. This places the new state onto the workflow and creates a transition from the **Initialize Target** state to the new state.
6. Click **State1** to select it, change the **DisplayName** to `Enter Guess`, and then double-click the state in the workflow designer to expand it.
7. Drag a **WriteLine** activity from the **Primitives** section of the **Toolbox** and drop it onto the **Entry** section of the state.
8. Type the following expression into the **Text** property box of the **WriteLine**.

```
"Please enter a number between 1 and " & MaxNumber
```

```
"Please enter a number between 1 and " + MaxNumber
```

9. Drag an **Assign** activity from the **Primitives** section of the **Toolbox** and drop onto the **Exit** section of the state.
10. Type `Turns` into the **To** box and `Turns + 1` into the **Enter a C# expression or Enter a VB expression** box.
11. Return to the overall state machine view in the workflow designer by clicking **StateMachine** in the breadcrumb display at the top of the workflow designer.
12. Drag a **FinalState** activity from the **State Machine** section of the **Toolbox**, hover it over the **Enter Guess** state, and drop it onto the triangle that appears to the right of the **Enter Guess** state so that a transition is created between **Enter Guess** and **FinalState**.
13. The default name of the transition is **T2**. Click the transition in the workflow designer to select it, and set its **DisplayName** to **Guess Correct**. Then click and select the **FinalState**, and drag it to the right so that there is room for the full transition name to be displayed without overlaying either of the two states. This will make it easier to complete the remaining steps in the tutorial.
14. Double-click the newly renamed **Guess Correct** transition in the workflow designer to expand it.
15. Drag a **ReadInt** activity from the **NumberGuessWorkflowActivities** section of the **Toolbox** and drop it in the **Trigger** section of the transition.
16. In the **Properties Window** for the **ReadInt** activity, type `"EnterGuess"` including the quotes into the **BookmarkName** property value box, and type `Guess` into the **Result** property value box
17. Type the following expression into the **Guess Correct** transition's **Condition** property value box.

`Guess = Target`

`Guess == Target`

18. Return to the overall state machine view in the workflow designer by clicking **StateMachine** in the breadcrumb display at the top of the workflow designer.

#### NOTE

A transition occurs when the trigger event is received and the **Condition**, if present, evaluates to `True`. For this transition, if the user's `Guess` matches the randomly generated `Target`, then control passes to the **FinalState** and the workflow completes.

19. Depending on whether the guess is correct, the workflow should transition either to the **FinalState** or back to the **Enter Guess** state for another try. Both transitions share the same trigger of waiting for the user's guess to be received via the **ReadInt** activity. This is called a shared transition. To create a shared transition, click the circle that indicates the start of the **Guess Correct** transition and drag it to the desired state. In this case the transition is a self-transition, so drag the start point of the **Guess Correct** transition and drop it back onto the bottom of the **Enter Guess** state. After creating the transition, select it in the workflow designer and set its **DisplayName** property to **Guess Incorrect**.

#### NOTE

Shared transitions can also be created from within the transition designer by clicking **Add shared trigger transition** at the bottom of the transition designer, and then selecting the desired target state from the **Available states to connect** drop-down.

#### NOTE

Note that if the **Condition** of a transition evaluates to `false` (or all of the conditions of a shared trigger transition evaluate to `false`), the transition will not occur and all triggers for all the transitions from the state will be rescheduled. In this tutorial, this situation cannot happen because of the way the conditions are configured (we have specific actions for whether the guess is correct or incorrect).

20. Double-click the **Guess Incorrect** transition in the workflow designer to expand it. Note that the **Trigger** is already set to the same **ReadInt** activity that was used by the **Guess Correct** transition.
21. Type the following expression into the **Condition** property value box.

```
Guess <> Target
```

```
Guess != Target
```

22. Drag an **If** activity from the **Control Flow** section of the **Toolbox** and drop it in the **Action** section of the transition.
23. Type the following expression into the **If** activity's **Condition** property value box.

```
Guess < Target
```

24. Drag two **WriteLine** activities from the **Primitives** section of the **Toolbox** and drop them so that one is in the **Then** section of the **If** activity, and one is in the **Else** section.
25. Click the **WriteLine** activity in the **Then** section to select it, and type the following expression into the **Text** property value box.

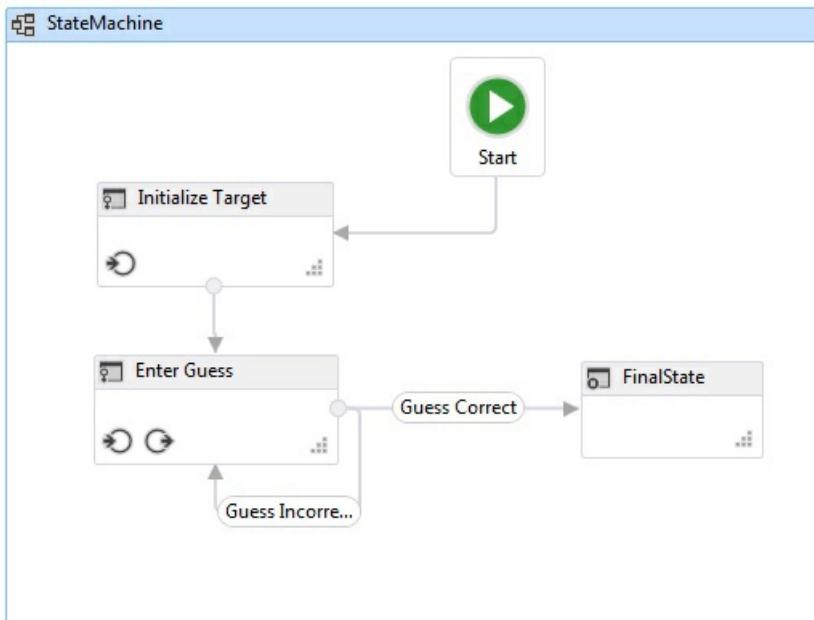
```
"Your guess is too low."
```

26. Click the **WriteLine** activity in the **Else** section to select it, and type the following expression into the **Text** property value box.

```
"Your guess is too high."
```

27. Return to the overall state machine view in the workflow designer by clicking **StateMachine** in the breadcrumb display at the top of the workflow designer.

The following example illustrates the completed workflow.



### To build the workflow

1. Press CTRL+SHIFT+B to build the solution.

For instructions on how to run the workflow, please see the next topic, [How to: Run a Workflow](#). If you have already completed the [How to: Run a Workflow](#) step with a different style of workflow and wish to run it using the state machine workflow from this step, skip ahead to the [To build and run the application](#) section of [How to: Run a Workflow](#).

### See also

- [Flowchart](#)
- [FlowDecision](#)
- [Windows Workflow Foundation Programming](#)
- [Designing Workflows](#)
- [Getting Started Tutorial](#)
- [How to: Create an Activity](#)
- [How to: Run a Workflow](#)

# How to: Run a Workflow

3/9/2019 • 8 minutes to read • [Edit Online](#)

This topic is a continuation of the Windows Workflow Foundation Getting Started tutorial and discusses how to create a workflow host and run the workflow defined in the previous [How to: Create a Workflow](#) topic.

## NOTE

Each topic in the Getting Started tutorial depends on the previous topics. To complete this topic you must first complete [How to: Create an Activity](#) and [How to: Create a Workflow](#).

## NOTE

To download a completed version of the tutorial, see [Windows Workflow Foundation \(WF45\) - Getting Started Tutorial](#).

## To create the workflow host project

1. Open the solution from the previous [How to: Create an Activity](#) topic by using Visual Studio 2012.
2. Right-click the **WF45GettingStartedTutorial** solution in **Solution Explorer** and select **Add, New Project**.

## TIP

If the **Solution Explorer** window is not displayed, select **Solution Explorer** from the **View** menu.

3. In the **Installed** node, select **Visual C#, Workflow** (or **Visual Basic, Workflow**).

## NOTE

Depending on which programming language is configured as the primary language in Visual Studio, the **Visual C#** or **Visual Basic** node may be under the **Other Languages** node in the **Installed** node.

Ensure that **.NET Framework 4.5** is selected in the .NET Framework version drop-down list. Select **Workflow Console Application** from the **Workflow** list. Type `NumberGuessWorkflowHost` into the **Name** box and click **OK**. This creates a starter workflow application with basic workflow hosting support. This basic hosting code is modified and used to run the workflow application.

4. Right-click the newly added **NumberGuessWorkflowHost** project in **Solution Explorer** and select **Add Reference**. Select **Solution** from the **Add Reference** list, check the checkbox beside **NumberGuessWorkflowActivities**, and then click **OK**.
5. Right-click **Workflow1.xaml** in **Solution Explorer** and choose **Delete**. Click **OK** to confirm.

## To modify the workflow hosting code

1. Double-click **Program.cs** or **Module1.vb** in **Solution Explorer** to display the code.

**TIP**

If the **Solution Explorer** window is not displayed, select **Solution Explorer** from the **View** menu.

Because this project was created by using the **Workflow Console Application** template, **Program.cs** or **Module1.vb** contains the following basic workflow hosting code.

```
' Create and cache the work-flow definition
Activity workflow1 = new Workflow1()
WorkflowInvoker.Invoke(workflow1)
```

```
// Create and cache the workflow definition
Activity workflow1 = new Workflow1();
WorkflowInvoker.Invoke(workflow1);
```

This generated hosting code uses [WorkflowInvoker](#). [WorkflowInvoker](#) provides a simple way for invoking a workflow as if it were a method call and can be used only for workflows that do not use persistence. [WorkflowApplication](#) provides a richer model for executing workflows that includes notification of life-cycle events, execution control, bookmark resumption, and persistence. This example uses bookmarks and [WorkflowApplication](#) is used for hosting the workflow. Add the following [using](#) or **Imports** statement at the top of **Program.cs** or **Module1.vb** below the existing [using](#) or **Imports** statements.

```
Imports NumberGuessWorkflowActivities
Imports System.Threading
```

```
using NumberGuessWorkflowActivities;
using System.Threading;
```

Replace the lines of code that use [WorkflowInvoker](#) with the following basic [WorkflowApplication](#) hosting code. This sample hosting code demonstrates the basic steps for hosting and invoking a workflow, but does not yet contain the functionality to successfully run the workflow from this topic. In the following steps, this basic code is modified and additional features are added until the application is complete.

**NOTE**

Please replace `Workflow1` in these examples with `FlowchartNumberGuessWorkflow`, `SequentialNumberGuessWorkflow`, or `StateMachineNumberGuessWorkflow`, depending on which workflow you completed in the previous [How to: Create a Workflow](#) step. If you do not replace `Workflow1` then you will get build errors when you try and build or run the workflow.

```

AutoResetEvent syncEvent = new AutoResetEvent(false);

WorkflowApplication wfApp =
    new WorkflowApplication(new Workflow1());

wfApp.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
{
    syncEvent.Set();
};

wfApp.Aborted = delegate(WorkflowApplicationAbortedEventArgs e)
{
    Console.WriteLine(e.Reason);
    syncEvent.Set();
};

wfApp.OnUnhandledException = delegate(WorkflowApplicationUnhandledExceptionEventArgs e)
{
    Console.WriteLine(e.UnhandledException.ToString());
    return UnhandledExceptionAction.Terminate;
};

wfApp.Run();

syncEvent.WaitOne();

```

```

Dim syncEvent As New AutoResetEvent(False)

Dim wfApp As New WorkflowApplication(New Workflow1())

wfApp.Completed = _
    Sub(e As WorkflowApplicationCompletedEventArgs)
        syncEvent.Set()
    End Sub

wfApp.Aborted = _
    Sub(e As WorkflowApplicationAbortedEventArgs)
        Console.WriteLine(e.Reason)
        syncEvent.Set()
    End Sub

wfApp.OnUnhandledException = _
    Function(e As WorkflowApplicationUnhandledExceptionEventArgs)
        Console.WriteLine(e.UnhandledException)
        Return UnhandledExceptionAction.Terminate
    End Function

wfApp.Run()

syncEvent.WaitOne()

```

This code creates a [WorkflowApplication](#), subscribes to three workflow life-cycle events, starts the workflow with a call to [Run](#), and then waits for the workflow to complete. When the workflow completes, the [AutoResetEvent](#) is set and the host application completes.

### To set input arguments of a workflow

1. Add the following statement at the top of **Program.cs** or **Module1.vb** below the existing `using` or `Imports` statements.

```
using System.Collections.Generic;
```

```
Imports System.Collections.Generic
```

- Replace the line of code that creates the new [WorkflowApplication](#) with the following code that creates and passes a dictionary of parameters to the workflow when it is created.

#### NOTE

Please replace `Workflow1` in these examples with `FlowchartNumberGuessWorkflow`, `SequentialNumberGuessWorkflow`, or `StateMachineNumberGuessWorkflow`, depending on which workflow you completed in the previous [How to: Create a Workflow](#) step. If you do not replace `Workflow1` then you will get build errors when you try and build or run the workflow.

```
var inputs = new Dictionary<string, object>() { { "MaxNumber", 100 } };

WorkflowApplication wfApp =
    new WorkflowApplication(new Workflow1(), inputs);
```

```
Dim inputs As New Dictionary(Of String, Object)
inputs.Add("MaxNumber", 100)

Dim wfApp As New WorkflowApplication(New Workflow1(), inputs)
```

This dictionary contains one element with a key of `MaxNumber`. Keys in the input dictionary correspond to input arguments on the root activity of the workflow. `MaxNumber` is used by the workflow to determine the upper bound for the randomly generated number.

#### To retrieve output arguments of a workflow

- Modify the [Completed](#) handler to retrieve and display the number of turns used by the workflow.

```
wfApp.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
{
    int Turns = Convert.ToInt32(e.Outputs["Turns"]);
    Console.WriteLine("Congratulations, you guessed the number in {0} turns.", Turns);

    syncEvent.Set();
};
```

```
wfApp.Completed = _
Sub(e As WorkflowApplicationCompletedEventArgs)
    Dim Turns As Integer = Convert.ToInt32(e.Outputs("Turns"))
    Console.WriteLine("Congratulations, you guessed the number in {0} turns.", Turns)

    syncEvent.Set()
End Sub
```

#### To resume a bookmark

- Add the following code at the top of the `Main` method just after the existing [AutoResetEvent](#) declaration.

```
AutoResetEvent idleEvent = new AutoResetEvent(false);
```

```
Dim idleEvent As New AutoResetEvent(False)
```

2. Add the following `Idle` handler just below the existing three workflow life-cycle handlers in `Main`.

```
wfApp.Idle = delegate(WorkflowApplicationIdleEventArgs e)
{
    idleEvent.Set();
};
```

```
wfApp.Idle = _
Sub(e As WorkflowApplicationIdleEventArgs)
    idleEvent.Set()
End Sub
```

Each time the workflow becomes idle waiting for the next guess, this handler is called and the `idleAction` `AutoResetEvent` is set. The code in the following step uses `idleEvent` and `syncEvent` to determine whether the workflow is waiting for the next guess or is complete.

#### NOTE

In this example, the host application uses auto-reset events in the `Completed` and `Idle` handlers to synchronize the host application with the progress of the workflow. It is not necessary to block and wait for the workflow to become idle before resuming a bookmark, but in this example the synchronization events are required so the host knows whether the workflow is complete or whether it is waiting on more user input by using the [Bookmark](#). For more information, see [Bookmarks](#).

3. Remove the call to `WaitOne`, and replace it with code to gather input from the user and resume the [Bookmark](#).

Remove the following line of code.

```
syncEvent.WaitOne();
```

```
syncEvent.WaitOne()
```

Replace it with the following example.

```

// Loop until the workflow completes.
WaitHandle[] handles = new WaitHandle[] { syncEvent, idleEvent };
while (WaitHandle.WaitAny(handles) != 0)
{
    // Gather the user input and resume the bookmark.
    bool validEntry = false;
    while (!validEntry)
    {
        int Guess;
        if (!Int32.TryParse(Console.ReadLine(), out Guess))
        {
            Console.WriteLine("Please enter an integer.");
        }
        else
        {
            validEntry = true;
            wfApp.ResumeBookmark("EnterGuess", Guess);
        }
    }
}

```

```

' Loop until the workflow completes.
Dim waitHandles As WaitHandle() = New WaitHandle() {syncEvent, idleEvent}
Do While WaitHandle.WaitAny(waitHandles) <> 0
    'Gather the user input and resume the bookmark.
    Dim validEntry As Boolean = False
    Do While validEntry = False
        Dim Guess As Integer
        If Int32.TryParse(Console.ReadLine(), Guess) = False Then
            Console.WriteLine("Please enter an integer.")
        Else
            validEntry = True
            wfApp.ResumeBookmark("EnterGuess", Guess)
        End If
    Loop
Loop

```

## To build and run the application

1. Right-click **NumberGuessWorkflowHost** in **Solution Explorer** and select **Set as StartUp Project**.
2. Press **CTRL+F5** to build and run the application. Try to guess the number in as few turns as possible.

To try the application with one of the other styles of workflow, replace `Workflow1` in the code that creates the `WorkflowApplication` with `FlowchartNumberGuessWorkflow`, `SequentialNumberGuessWorkflow`, or `StateMachineNumberGuessWorkflow`, depending on which workflow style you desire.

```

var inputs = new Dictionary<string, object>() { { "MaxNumber", 100 } };

WorkflowApplication wfApp =
    new WorkflowApplication(new Workflow1(), inputs);

```

```

Dim inputs As New Dictionary(Of String, Object)
inputs.Add("MaxNumber", 100)

Dim wfApp As New WorkflowApplication(New Workflow1(), inputs)

```

For instructions about how to add persistence to a workflow application, see the next topic, [How to: Create and Run a Long Running Workflow](#).

## Example

The following example is the complete code listing for the `Main` method.

### NOTE

Please replace `Workflow1` in these examples with `FlowchartNumberGuessWorkflow`, `SequentialNumberGuessWorkflow`, or `StateMachineNumberGuessWorkflow`, depending on which workflow you completed in the previous [How to: Create a Workflow](#) step. If you do not replace `Workflow1` then you will get build errors when you try and build or run the workflow.

```

static void Main(string[] args)
{
    AutoResetEvent syncEvent = new AutoResetEvent(false);
    AutoResetEvent idleEvent = new AutoResetEvent(false);

    var inputs = new Dictionary<string, object>() { { "MaxNumber", 100 } };

    WorkflowApplication wfApp =
        new WorkflowApplication(new Workflow1(), inputs);

    wfApp.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
    {
        int Turns = Convert.ToInt32(e.Outputs["Turns"]);
        Console.WriteLine("Congratulations, you guessed the number in {0} turns.", Turns);

        syncEvent.Set();
    };

    wfApp.Aborted = delegate(WorkflowApplicationAbortedEventArgs e)
    {
        Console.WriteLine(e.Reason);
        syncEvent.Set();
    };

    wfApp.OnUnhandledException = delegate(WorkflowApplicationUnhandledExceptionEventArgs e)
    {
        Console.WriteLine(e.UnhandledException.ToString());
        return UnhandledExceptionAction.Terminate;
    };

    wfApp.Idle = delegate(WorkflowApplicationIdleEventArgs e)
    {
        idleEvent.Set();
    };

    wfApp.Run();

    // Loop until the workflow completes.
    WaitHandle[] handles = new WaitHandle[] { syncEvent, idleEvent };
    while (WaitHandle.WaitAny(handles) != 0)
    {
        // Gather the user input and resume the bookmark.
        bool validEntry = false;
        while (!validEntry)
        {
            int Guess;
            if (!Int32.TryParse(Console.ReadLine(), out Guess))
            {
                Console.WriteLine("Please enter an integer.");
            }
            else
            {
                validEntry = true;
                wfApp.ResumeBookmark("EnterGuess", Guess);
            }
        }
    }
}

```

```

Sub Main()
    Dim syncEvent As New AutoResetEvent(False)
    Dim idleEvent As New AutoResetEvent(False)

    Dim inputs As New Dictionary(Of String, Object)
    inputs.Add("MaxNumber", 100)

    Dim wfApp As New WorkflowApplication(New Workflow1(), inputs)

    wfApp.Completed = _
        Sub(e As WorkflowApplicationCompletedEventArgs)
            Dim Turns As Integer = Convert.ToInt32(e.Outputs("Turns"))
            Console.WriteLine("Congratulations, you guessed the number in {0} turns.", Turns)

            syncEvent.Set()
        End Sub

    wfApp.Aborted = _
        Sub(e As WorkflowApplicationAbortedEventArgs)
            Console.WriteLine(e.Reason)
            syncEvent.Set()
        End Sub

    wfApp.OnUnhandledException = _
        Function(e As WorkflowApplicationUnhandledExceptionEventArgs)
            Console.WriteLine(e.UnhandledException)
            Return UnhandledExceptionAction.Terminate
        End Function

    wfApp.Idle = _
        Sub(e As WorkflowApplicationIdleEventArgs)
            idleEvent.Set()
        End Sub

    wfApp.Run()

    ' Loop until the workflow completes.
    Dim waitHandles As WaitHandle() = New WaitHandle() {syncEvent, idleEvent}
    Do While WaitHandle.WaitAny(waitHandles) <> 0
        'Gather the user input and resume the bookmark.
        Dim validEntry As Boolean = False
        Do While validEntry = False
            Dim Guess As Integer
            If Int32.TryParse(Console.ReadLine(), Guess) = False Then
                Console.WriteLine("Please enter an integer.")
            Else
                validEntry = True
                wfApp.ResumeBookmark("EnterGuess", Guess)
            End If
        Loop
    Loop
End Sub

```

## See also

- [WorkflowApplication](#)
- [Bookmark](#)
- [Windows Workflow Foundation Programming](#)
- [Getting Started Tutorial](#)
- [How to: Create a Workflow](#)
- [How to: Create and Run a Long Running Workflow](#)
- [Waiting for Input in a Workflow](#)

- Hosting Workflows

# How to: Create and Run a Long Running Workflow

3/9/2019 • 24 minutes to read • [Edit Online](#)

One of the central features of Windows Workflow Foundation (WF) is the runtime's ability to persist and unload idle workflows to a database. The steps in [How to: Run a Workflow](#) demonstrated the basics of workflow hosting using a console application. Examples were shown of starting workflows, workflow lifecycle handlers, and resuming bookmarks. In order to demonstrate workflow persistence effectively, a more complex workflow host is required that supports starting and resuming multiple workflow instances. This step in the tutorial demonstrates how to create a Windows form host application that supports starting and resuming multiple workflow instances, workflow persistence, and provides a basis for the advanced features such as tracking and versioning that are demonstrated in subsequent tutorial steps.

## NOTE

This tutorial step and the subsequent steps use all three workflow types from [How to: Create a Workflow](#). If you did not complete all three types you can download a completed version of the steps from [Windows Workflow Foundation \(WF45\) - Getting Started Tutorial](#).

## NOTE

To download a completed version or view a video walkthrough of the tutorial, see [Windows Workflow Foundation \(WF45\) - Getting Started Tutorial](#).

## In this topic

- [To create the persistence database](#)
- [To add the reference to the DurableInstancing assemblies](#)
- [To create the workflow host form](#)
- [To add the properties and helper methods of the form](#)
- [To configure the instance store, workflow lifecycle handlers, and extensions](#)
- [To enable starting and resuming multiple workflow types](#)
- [To start a new workflow](#)
- [To resume a workflow](#)
- [To terminate a workflow](#)
- [To build and run the application](#)

### To create the persistence database

1. Open SQL Server Management Studio and connect to the local server, for example **\SQLEXPRESS**. Right-click the **Databases** node on the local server, and select **New Database**. Name the new database **WF45GettingStartedTutorial**, accept all other values, and select **OK**.

#### NOTE

Ensure that you have **Create Database** permission on the local server before creating the database.

2. Choose **Open, File** from the **File** menu. Browse to the following folder:

C:\Windows\Microsoft.NET\Framework\v4.0.30319\sql\en

Select the following two files and click **Open**.

- SqlWorkflowInstanceStoreLogic.sql
- SqlWorkflowInstanceStoreSchema.sql

3. Choose **SqlWorkflowInstanceStoreSchema.sql** from the **Window** menu. Ensure that **WF45GettingStartedTutorial** is selected in the **Available Databases** drop-down and choose **Execute** from the **Query** menu.
4. Choose **SqlWorkflowInstanceStoreLogic.sql** from the **Window** menu. Ensure that **WF45GettingStartedTutorial** is selected in the **Available Databases** drop-down and choose **Execute** from the **Query** menu.

#### WARNING

It is important to perform the previous two steps in the correct order. If the queries are executed out of order, errors occur and the persistence database is not configured correctly.

### To add the reference to the DurableInstancing assemblies

1. Right-click **NumberGuessWorkflowHost** in **Solution Explorer** and select **Add Reference**.
2. Select **Assemblies** from the **Add Reference** list, and type **DurableInstancing** into the **Search Assemblies** box. This filters the assemblies and makes the desired references easier to select.
3. Check the checkbox beside **System.Activities.DurableInstancing** and **System.Runtime.DurableInstancing** from the **Search Results** list, and click **OK**.

### To create the workflow host form

#### NOTE

The steps in this procedure describe how to add and configure the form manually. If desired, you can download the solution files for the tutorial and add the completed form to the project. To download the tutorial files, see [Windows Workflow Foundation \(WF45\) - Getting Started Tutorial](#). Once the files are downloaded, right-click **NumberGuessWorkflowHost** and choose **Add Reference**. Add a reference to **System.Windows.Forms** and **System.Drawing**. These references are added automatically if you add a new form from the **Add, New Item** menu, but must be added manually when importing a form. Once the references are added, right-click **NumberGuessWorkflowHost** in **Solution Explorer** and choose **Add, Existing Item**. Browse to the **Form** folder in the project files, select **WorkflowHostForm.cs** (or **WorkflowHostForm.vb**), and click **Add**. If you choose to import the form, then you can skip down to the next section, [To add the properties and helper methods of the form](#).

1. Right-click **NumberGuessWorkflowHost** in **Solution Explorer** and choose **Add, New Item**.
2. In the **Installed** templates list, choose **Windows Form**, type **WorkflowHostForm** in the **Name** box, and click **Add**.
3. Configure the following properties on the form.

| PROPERTY        | VALUE       |
|-----------------|-------------|
| FormBorderStyle | FixedSingle |
| MaximizeBox     | False       |
| Size            | 400, 420    |

4. Add the following controls to the form in the order specified and configure the properties as directed.

| CONTROL         | PROPERTY: VALUE   |
|-----------------|---|
| <b>Button</b>   | Name: NewGame<br><br>Location: 13, 13<br><br>Size: 75, 23<br><br>Text: New Game   |
| <b>Label</b>    | Location: 94, 18<br><br>Text: Guess a number from 1 to  |
| <b>ComboBox</b> | Name: NumberRange<br><br>DropDownStyle: DropDownList<br><br>Items: 10, 100, 1000<br><br>Location: 228, 12<br><br>Size: 143, 21  |
| <b>Label</b>    | Location: 13, 43<br><br>Text: Workflow type   |
| <b>ComboBox</b> | Name: WorkflowType<br><br>DropDownStyle: DropDownList<br><br>Items: StateMachineNumberGuessWorkflow,<br>FlowchartNumberGuessWorkflow,<br>SequentialNumberGuessWorkflow<br><br>Location: 94, 40<br><br>Size: 277, 21 |
| <b>Label</b>    | Name: WorkflowVersion<br><br>Location: 13, 362<br><br>Text: Workflow version  |

| CONTROL         | PROPERTY: VALUE                                  |
|-----------------|--|
| <b>GroupBox</b> | Location: 13, 67<br>Size: 358, 287<br>Text: Game |

#### NOTE

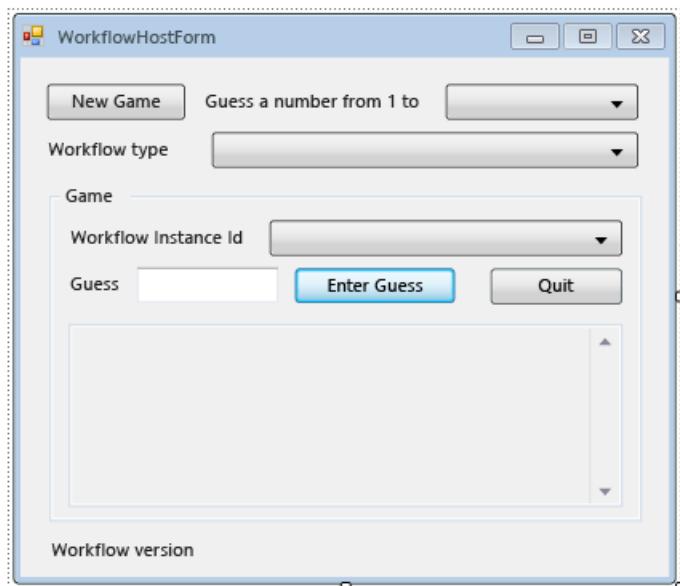
When adding the following controls, put them into the GroupBox.

| CONTROL         | PROPERTY: VALUE   |
|-----------------|---|
| <b>Label</b>    | Location: 7, 20<br>Text: Workflow Instance Id   |
| <b>ComboBox</b> | Name: InstanceId<br>DropDownStyle: DropDownList<br>Location: 121, 17<br>Size: 227, 21 |
| <b>Label</b>    | Location: 7, 47<br>Text: Guess  |
| <b>TextBox</b>  | Name: Guess<br>Location: 50, 44<br>Size: 65, 20                                       |
| <b>Button</b>   | Name: EnterGuess<br>Location: 121, 42<br>Size: 75, 23<br>Text: Enter Guess            |
| <b>Button</b>   | Name: QuitGame<br>Location: 274, 42<br>Size: 75, 23<br>Text: Quit                     |

| CONTROL | PROPERTY: VALUE   |
|---------|---|
| TextBox | Name: WorkflowStatus<br>Location: 10, 73<br>Multiline: True<br>ReadOnly: True<br>ScrollBars: Vertical<br>Size: 338, 208 |

5. Set the **AcceptButton** property of the form to **EnterGuess**.

The following example illustrates the completed form.



#### To add the properties and helper methods of the form

The steps in this section add properties and helper methods to the form class that configure the UI of the form to support running and resuming number guess workflows.

1. Right-click **WorkflowHostForm** in **Solution Explorer** and choose **View Code**.
2. Add the following `using` (or `Imports`) statements at the top of the file with the other `using` (or `Imports`) statements.

```
Imports System.Windows.Forms
Imports System.Activities.DurableInstancing
Imports System.Activities
Imports System.Data.SqlClient
Imports System.IO
```

```
using System.Windows.Forms;
using System.Activities.DurableInstancing;
using System.Activities;
using System.Data.SqlClient;
using System.IO;
```

3. Add the following member declarations to the **WorkflowHostForm** class.

```
Const connectionString = "Server=.\SQLEXPRESS;Initial Catalog=WF45GettingStartedTutorial;Integrated Security=SSPI"
Dim store As SqlWorkflowInstanceStore
Dim WorkflowStarting As Boolean
```

```
const string connectionString = "Server=.\SQLEXPRESS;Initial Catalog=WF45GettingStartedTutorial;Integrated Security=SSPI";
SqlWorkflowInstanceStore store;
bool WorkflowStarting;
```

#### NOTE

If your connection string is different, update `connectionString` to refer to your database.

4. Add a `WorkflowInstanceId` property to the `WorkflowFormHost` class.

```
Public ReadOnly Property WorkflowInstanceId() As Guid
    Get
        If InstanceId.SelectedIndex = -1 Then
            Return Guid.Empty
        Else
            Return New Guid(InstanceId.SelectedItem.ToString())
        End If
    End Get
End Property
```

```
public Guid WorkflowInstanceId
{
    get
    {
        return InstanceId.SelectedIndex == -1 ? Guid.Empty : (Guid)InstanceId.SelectedItem;
    }
}
```

The `InstanceId` combo box displays a list of persisted workflow instance ids, and the `WorkflowInstanceId` property returns the currently selected workflow.

5. Add a handler for the form `Load` event. To add the handler, switch to **Design View** for the form, click the **Events** icon at the top of the **Properties** window, and double-click **Load**.

```
Private Sub WorkflowHostForm_Load(sender As Object, e As EventArgs) Handles Me.Load
    End Sub
```

```
private void WorkflowHostForm_Load(object sender, EventArgs e)
{}
```

6. Add the following code to `WorkflowHostForm_Load`.

```

'Initialize the store and configure it so that it can be used for
'multiple WorkflowApplication instances.
store = New SqlWorkflowInstanceStore(connectionString)
WorkflowApplication.CreateDefaultInstanceOwner(store, Nothing, WorkflowIdentityFilter.Any)

'Set default ComboBox selections.
NumberRange.SelectedIndex = 0
WorkflowType.SelectedIndex = 0

ListPersistedWorkflows()

```

```

// Initialize the store and configure it so that it can be used for
// multiple WorkflowApplication instances.
store = new SqlWorkflowInstanceStore(connectionString);
WorkflowApplication.CreateDefaultInstanceOwner(store, null, WorkflowIdentityFilter.Any);

// Set default ComboBox selections.
NumberRange.SelectedIndex = 0;
WorkflowType.SelectedIndex = 0;

ListPersistedWorkflows();

```

When the form loads, the `SqlWorkflowInstanceStore` is configured, the range and workflow type combo boxes are set to default values, and the persisted workflow instances are added to the `InstanceId` combo box.

7. Add a `SelectedIndexChanged` handler for `InstanceId`. To add the handler, switch to **Design View** for the form, select the `InstanceId` combo box, click the **Events** icon at the top of the **Properties** window, and double-click **SelectedIndexChanged**.

```

Private Sub InstanceId_SelectedIndexChanged(sender As Object, e As EventArgs) Handles
InstanceId.SelectedIndexChanged

End Sub

```

```

private void InstanceId_SelectedIndexChanged(object sender, EventArgs e)
{
}

```

8. Add the following code to `InstanceId_SelectedIndexChanged`. Whenever the user selects a workflow by using the combo box this handler updates the status window.

```

If InstanceId.SelectedIndex = -1 Then
    Return
End If

'Clear the status window.
WorkflowStatus.Clear()

'Get the workflow version and display it.
'If the workflow is just starting then this info will not
'be available in the persistence store so do not try and retrieve it.
If Not WorkflowStarting Then
    Dim instance As WorkflowApplicationInstance =
        WorkflowApplication.GetInstance(WorkflowInstanceId, store)

    WorkflowVersion.Text =
        WorkflowVersionMap.GetIdentityDescription(instance.DefinitionIdentity)

    'Unload the instance.
    instance.Abandon()
End If

```

```

if (InstanceId.SelectedIndex == -1)
{
    return;
}

// Clear the status window.
WorkflowStatus.Clear();

// Get the workflow version and display it.
// If the workflow is just starting then this info will not
// be available in the persistence store so do not try and retrieve it.
if (!WorkflowStarting)
{
    WorkflowApplicationInstance instance =
        WorkflowApplication.GetInstance(this.WorkflowInstanceId, store);

    WorkflowVersion.Text =
        WorkflowVersionMap.GetIdentityDescription(instance.DefinitionIdentity);

    // Unload the instance.
    instance.Abandon();
}

```

9. Add the following `ListPersistedWorkflows` method to the form class.

```

Private Sub ListPersistedWorkflows()
    Using localCon As New SqlConnection(connectionString)
        Dim localCmd As String = _
            "Select [InstanceId] from [System.Activities.DurableInstancing].[Instances] Order By
[CreationTime]"

        Dim cmd As SqlCommand = localCon.CreateCommand()
        cmd.CommandText = localCmd
        localCon.Open()
        Using reader As SqlDataReader = cmd.ExecuteReader(CommandBehavior.CloseConnection)

            While (reader.Read())
                'Get the InstanceId of the persisted Workflow.
                Dim id As Guid = Guid.Parse(reader(0).ToString())
                InstanceId.Items.Add(id)
            End While
        End Using
    End Using
End Sub

```

```

using (SqlConnection localCon = new SqlConnection(connectionString))
{
    string localCmd =
        "Select [InstanceId] from [System.Activities.DurableInstancing].[Instances] Order By
[CreationTime]";

    SqlCommand cmd = localCon.CreateCommand();
    cmd.CommandText = localCmd;
    localCon.Open();
    using (SqlDataReader reader = cmd.ExecuteReader(CommandBehavior.CloseConnection))
    {
        while (reader.Read())
        {
            // Get the InstanceId of the persisted Workflow
            Guid id = Guid.Parse(reader[0].ToString());
            InstanceId.Items.Add(id);
        }
    }
}

```

`ListPersistedWorkflows` queries the instance store for persisted workflow instances, and adds the instance ids to the `cboInstanceId` combo box.

10. Add the following `UpdateStatus` method and corresponding delegate to the form class. This method updates the status window on the form with the status of the currently running workflow.

```

Private Delegate Sub UpdateStatusDelegate(msg As String)
Public Sub UpdateStatus(msg As String)
    'We may be on a different thread so we need to
    'make this call using BeginInvoke.
    If InvokeRequired Then
        BeginInvoke(New UpdateStatusDelegate(AddressOf UpdateStatus), msg)
    Else
        If Not msg.EndsWith(vbCrLf) Then
            msg = msg & vbCrLf
        End If

        WorkflowStatus.AppendText(msg)

        'Ensure that the newly added status is visible.
        WorkflowStatus.SelectionStart = WorkflowStatus.Text.Length
        WorkflowStatus.ScrollToCaret()
    End If
End Sub

```

```

private delegate void UpdateStatusDelegate(string msg);
public void UpdateStatus(string msg)
{
    // We may be on a different thread so we need to
    // make this call using BeginInvoke.
    if (InvokeRequired)
    {
        BeginInvoke(new UpdateStatusDelegate(UpdateStatus), msg);
    }
    else
    {
        if (!msg.EndsWith("\r\n"))
        {
            msg += "\r\n";
        }
        WorkflowStatus.AppendText(msg);

        WorkflowStatus.SelectionStart = WorkflowStatus.Text.Length;
        WorkflowStatus.ScrollToCaret();
    }
}

```

11. Add the following `GameOver` method and corresponding delegate to the form class. When a workflow completes, this method updates the form UI by removing the instance id of the completed workflow from the **InstanceId** combo box.

```

Private Delegate Sub GameOverDelegate()
Private Sub GameOver()
    If InvokeRequired Then
        BeginInvoke(New GameOverDelegate(AddressOf GameOver))
    Else
        'Remove this instance from the InstanceId combo box.
        InstanceId.Items.Remove(InstanceId.SelectedItem)
        InstanceId.SelectedIndex = -1
    End If
End Sub

```

```

private delegate void GameOverDelegate();
private void GameOver()
{
    if (InvokeRequired)
    {
        BeginInvoke(new GameOverDelegate(GameOver));
    }
    else
    {
        // Remove this instance from the combo box
        InstanceId.Items.Remove(InstanceId.SelectedItem);
        InstanceId.SelectedIndex = -1;
    }
}

```

## To configure the instance store, workflow lifecycle handlers, and extensions

1. Add a `ConfigureWorkflowApplication` method to the form class.

```

Private Sub ConfigureWorkflowApplication(wfApp As WorkflowApplication)

End Sub

```

```

private void ConfigureWorkflowApplication(WorkflowApplication wfApp)
{
}

```

This method configures the `WorkflowApplication`, adds the desired extensions, and adds handlers for the workflow lifecycle events.

2. In `ConfigureWorkflowApplication`, specify the `SqlWorkflowInstanceStore` for the `WorkflowApplication`.

```

'Configure the persistence store.
wfApp.InstanceStore = store

```

```

// Configure the persistence store.
wfApp.InstanceStore = store;

```

3. Next, create a `StringWriter` instance and add it to the `Extensions` collection of the `WorkflowApplication`.

When a `StringWriter` is added to the extensions it captures all `WriteLine` activity output. When the workflow becomes idle, the `WriteLine` output can be extracted from the `StringWriter` and displayed on the form.

```

'Add a StringWriter to the extensions. This captures the output
'from the WriteLine activities so we can display it in the form.
Dim sw As New StringWriter()
wfApp.Extensions.Add(sw)

```

```

// Add a StringWriter to the extensions. This captures the output
// from the WriteLine activities so we can display it in the form.
StringWriter sw = new StringWriter();
wfApp.Extensions.Add(sw);

```

4. Add the following handler for the `Completed` event. When a workflow successfully completes, the number

of turns taken to guess the number is displayed to the status window. If the workflow terminates, the exception information that caused the termination is displayed. At the end of the handler the `GameOver` method is called, which removes the completed workflow from the workflow list.

```
wfApp.Completed = _
Sub(e As WorkflowApplicationCompletedEventArgs)
    If e.CompletionState = ActivityInstanceState.Faulted Then
        UpdateStatus(String.Format("Workflow Terminated. Exception: {0}" & vbCrLf & "{1}", _
            e.TerminationException.GetType().FullName, _
            e.TerminationException.Message))
    ElseIf e.CompletionState = ActivityInstanceState.Canceled Then
        UpdateStatus("Workflow Canceled.")
    Else
        Dim Turns As Integer = Convert.ToInt32(e.Outputs("Turns"))
        UpdateStatus(String.Format("Congratulations, you guessed the number in {0} turns.", Turns))
    End If
    GameOver()
End Sub
```

```
wfApp.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
{
    if (e.CompletionState == ActivityInstanceState.Faulted)
    {
        UpdateStatus(string.Format("Workflow Terminated. Exception: {0}\r\n{1}",
            e.TerminationException.GetType().FullName,
            e.TerminationException.Message));
    }
    else if (e.CompletionState == ActivityInstanceState.Canceled)
    {
        UpdateStatus("Workflow Canceled.");
    }
    else
    {
        int Turns = Convert.ToInt32(e.Outputs["Turns"]);
        UpdateStatus(string.Format("Congratulations, you guessed the number in {0} turns.", Turns));
    }
    GameOver();
};
```

5. Add the following `Aborted` and `OnUnhandledException` handlers. The `GameOver` method is not called from the `Aborted` handler because when a workflow instance is aborted, it does not terminate, and it is possible to resume the instance at a later time.

```
wfApp.Aborted = _
Sub(e As WorkflowApplicationAbortedEventArgs)
    UpdateStatus(String.Format("Workflow Aborted. Exception: {0}" & vbCrLf & "{1}", _
        e.Reason.GetType().FullName, _
        e.Reason.Message))
End Sub

wfApp.OnUnhandledException = _
Function(e As WorkflowApplicationUnhandledExceptionEventArgs)
    UpdateStatus(String.Format("Unhandled Exception: {0}" & vbCrLf & "{1}", _
        e.UnhandledException.GetType().FullName, _
        e.UnhandledException.Message))
    GameOver()
    Return UnhandledExceptionAction.Terminate
End Function
```

```

wfApp.Aborted = delegate(WorkflowApplicationAbortedEventArgs e)
{
    UpdateStatus(string.Format("Workflow Aborted. Exception: {0}\r\n{1}",
        e.Reason.GetType().FullName,
        e.Reason.Message));
};

wfApp.OnUnhandledException = delegate(WorkflowApplicationUnhandledExceptionEventArgs e)
{
    UpdateStatus(string.Format("Unhandled Exception: {0}\r\n{1}",
        e.UnhandledException.GetType().FullName,
        e.UnhandledException.Message));
    GameOver();
    return UnhandledExceptionAction.Terminate;
};

```

6. Add the following `PersistableIdle` handler. This handler retrieves the `StringWriter` extension that was added, extracts the output from the `WriteLine` activities, and displays it in the status window.

```

wfApp.PersistableIdle = _
Function(e As WorkflowApplicationIdleEventArgs)
    'Send the current WriteLine outputs to the status window.
    Dim writers = e.GetInstanceExtensions(Of StringWriter)()
    For Each writer In writers
        UpdateStatus(writer.ToString())
    Next
    Return PersistableIdleAction.Unload
End Function

```

```

wfApp.PersistableIdle = delegate(WorkflowApplicationIdleEventArgs e)
{
    // Send the current WriteLine outputs to the status window.
    var writers = e.GetInstanceExtensions<StringWriter>();
    foreach (var writer in writers)
    {
        UpdateStatus(writer.ToString());
    }
    return PersistableIdleAction.Unload;
};

```

The `PersistableIdleAction` enumeration has three values: `None`, `Persist`, and `Unload`. `Persist` causes the workflow to persist but it does not cause the workflow to unload. `Unload` causes the workflow to persist and be unloaded.

The following example is the completed `ConfigureWorkflowApplication` method.

```

Private Sub ConfigureWorkflowApplication(wfApp As WorkflowApplication)
    'Configure the persistence store.
    wfApp.InstanceStore = store

    'Add a StringWriter to the extensions. This captures the output
    'from the WriteLine activities so we can display it in the form.
    Dim sw As New StringWriter()
    wfApp.Extensions.Add(sw)

    wfApp.Completed = _
        Sub(e As WorkflowApplicationCompletedEventArgs)
            If e.CompletionState = ActivityInstanceState.Faulted Then
                UpdateStatus(String.Format("Workflow Terminated. Exception: {0}" & vbCrLf & "{1}", _
                    e.TerminationException.GetType().FullName, _
                    e.TerminationException.Message))
            ElseIf e.CompletionState = ActivityInstanceState.Canceled Then
                UpdateStatus("Workflow Canceled.")
            Else
                Dim Turns As Integer = Convert.ToInt32(e.Outputs("Turns"))
                UpdateStatus(String.Format("Congratulations, you guessed the number in {0} turns.", _
                    Turns))
            End If
            GameOver()
        End Sub

    wfApp.Aborted = _
        Sub(e As WorkflowApplicationAbortedEventArgs)
            UpdateStatus(String.Format("Workflow Aborted. Exception: {0}" & vbCrLf & "{1}", _
                e.Reason.GetType().FullName, _
                e.Reason.Message))
        End Sub

    wfApp.OnUnhandledException = _
        Function(e As WorkflowApplicationUnhandledExceptionEventArgs)
            UpdateStatus(String.Format("Unhandled Exception: {0}" & vbCrLf & "{1}", _
                e.UnhandledException.GetType().FullName, _
                e.UnhandledException.Message))
            GameOver()
            Return UnhandledExceptionAction.Terminate
        End Function

    wfApp.PersistableIdle = _
        Function(e As WorkflowApplicationIdleEventArgs)
            'Send the current WriteLine outputs to the status window.
            Dim writers = e.GetInstanceExtensions(Of StringWriter)()
            For Each writer In writers
                UpdateStatus(writer.ToString())
            Next
            Return PersistableIdleAction.Unload
        End Function
End Sub

```

```

private void ConfigureWorkflowApplication(WorkflowApplication wfApp)
{
    // Configure the persistence store.
    wfApp.InstanceStore = store;

    // Add a StringWriter to the extensions. This captures the output
    // from the WriteLine activities so we can display it in the form.
    StringWriter sw = new StringWriter();
    wfApp.Extensions.Add(sw);

    wfApp.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
    {
        if (e.CompletionState == ActivityInstanceState.Faulted)
        {
            UpdateStatus(string.Format("Workflow Terminated. Exception: {0}\r\n{1}",
                e.TerminationException.GetType().FullName,
                e.TerminationException.Message));
        }
        else if (e.CompletionState == ActivityInstanceState.Canceled)
        {
            UpdateStatus("Workflow Canceled.");
        }
        else
        {
            int Turns = Convert.ToInt32(e.Outputs["Turns"]);
            UpdateStatus(string.Format("Congratulations, you guessed the number in {0} turns.",
                Turns));
        }
        GameOver();
    };

    wfApp.Aborted = delegate(WorkflowApplicationAbortedEventArgs e)
    {
        UpdateStatus(string.Format("Workflow Aborted. Exception: {0}\r\n{1}",
            e.Reason.GetType().FullName,
            e.Reason.Message));
    };

    wfApp.OnUnhandledException = delegate(WorkflowApplicationUnhandledExceptionEventArgs e)
    {
        UpdateStatus(string.Format("Unhandled Exception: {0}\r\n{1}",
            e.UnhandledException.GetType().FullName,
            e.UnhandledException.Message));
        GameOver();
        return UnhandledExceptionAction.Terminate;
    };

    wfApp.PersistableIdle = delegate(WorkflowApplicationIdleEventArgs e)
    {
        // Send the current WriteLine outputs to the status window.
        var writers = e.GetInstanceExtensions<StringWriter>();
        foreach (var writer in writers)
        {
            UpdateStatus(writer.ToString());
        }
        return PersistableIdleAction.Unload;
    };
}

```

## To enable starting and resuming multiple workflow types

In order to resume a workflow instance, the host has to provide the workflow definition. In this tutorial there are three workflow types, and subsequent tutorial steps introduce multiple versions of these types. [WorkflowIdentity](#) provides a way for a host application to associate identifying information with a persisted workflow instance. The steps in this section demonstrate how to create a utility class to assist with mapping the workflow identity from a

persisted workflow instance to the corresponding workflow definition. For more information about `WorkflowIdentity` and versioning, see [Using WorkflowIdentity and Versioning](#).

1. Right-click **NumberGuessWorkflowHost** in **Solution Explorer** and choose **Add, Class**. Type `WorkflowVersionMap` into the **Name** box and click **Add**.

2. Add the following `using` or `Imports` statements at the top of the file with the other `using` or `Imports` statements.

```
Imports NumberGuessWorkflowActivities  
Imports System.Activities
```

```
using NumberGuessWorkflowActivities;  
using System.Activities;
```

3. Replace the `WorkflowVersionMap` class declaration with the following declaration.

```
Public Module WorkflowVersionMap  
    Dim map As Dictionary(Of WorkflowIdentity, Activity)  
  
    'Current version identities.  
    Public StateMachineNumberGuessIdentity As WorkflowIdentity  
    Public FlowchartNumberGuessIdentity As WorkflowIdentity  
    Public SequentialNumberGuessIdentity As WorkflowIdentity  
  
    Sub New()  
        map = New Dictionary(Of WorkflowIdentity, Activity)  
  
        'Add the current workflow version identities.  
        StateMachineNumberGuessIdentity = New WorkflowIdentity With  
        {  
            .Name = "StateMachineNumberGuessWorkflow",  
            .Version = New Version(1, 0, 0, 0)  
        }  
  
        FlowchartNumberGuessIdentity = New WorkflowIdentity With  
        {  
            .Name = "FlowchartNumberGuessWorkflow",  
            .Version = New Version(1, 0, 0, 0)  
        }  
  
        SequentialNumberGuessIdentity = New WorkflowIdentity With  
        {  
            .Name = "SequentialNumberGuessWorkflow",  
            .Version = New Version(1, 0, 0, 0)  
        }  
  
        map.Add(StateMachineNumberGuessIdentity, New StateMachineNumberGuessWorkflow())  
        map.Add(FlowchartNumberGuessIdentity, New FlowchartNumberGuessWorkflow())  
        map.Add(SequentialNumberGuessIdentity, New SequentialNumberGuessWorkflow())  
    End Sub  
  
    Public Function GetWorkflowDefinition(identity As WorkflowIdentity) As Activity  
        Return map(identity)  
    End Function  
  
    Public Function GetIdentityDescription(identity As WorkflowIdentity) As String  
        Return identity.ToString()  
    End Function  
End Module
```

```

public static class WorkflowVersionMap
{
    static Dictionary<WorkflowIdentity, Activity> map;

    // Current version identities.
    static public WorkflowIdentity StateMachineNumberGuessIdentity;
    static public WorkflowIdentity FlowchartNumberGuessIdentity;
    static public WorkflowIdentity SequentialNumberGuessIdentity;

    static WorkflowVersionMap()
    {
        map = new Dictionary<WorkflowIdentity, Activity>();

        // Add the current workflow version identities.
        StateMachineNumberGuessIdentity = new WorkflowIdentity
        {
            Name = "StateMachineNumberGuessWorkflow",
            Version = new Version(1, 0, 0, 0)
        };

        FlowchartNumberGuessIdentity = new WorkflowIdentity
        {
            Name = "FlowchartNumberGuessWorkflow",
            Version = new Version(1, 0, 0, 0)
        };

        SequentialNumberGuessIdentity = new WorkflowIdentity
        {
            Name = "SequentialNumberGuessWorkflow",
            Version = new Version(1, 0, 0, 0)
        };

        map.Add(StateMachineNumberGuessIdentity, new StateMachineNumberGuessWorkflow());
        map.Add(FlowchartNumberGuessIdentity, new FlowchartNumberGuessWorkflow());
        map.Add(SequentialNumberGuessIdentity, new SequentialNumberGuessWorkflow());
    }

    public static Activity GetWorkflowDefinition(WorkflowIdentity identity)
    {
        return map[identity];
    }

    public static string GetIdentityDescription(WorkflowIdentity identity)
    {
        return identity.ToString();
    }
}

```

`WorkflowVersionMap` contains three workflow identities that map to the three workflow definitions from this tutorial and is used in the following sections when workflows are started and resumed.

## To start a new workflow

1. Add a `Click` handler for `NewGame`. To add the handler, switch to **Design View** for the form, and double-click `NewGame`. A `NewGame_Click` handler is added and the view switches to code view for the form. Whenever the user clicks this button a new workflow is started.

```

Private Sub NewGame_Click(sender As Object, e As EventArgs) Handles NewGame.Click
    End Sub

```

```

private void NewGame_Click(object sender, EventArgs e)
{
}

```

2. Add the following code to the click handler. This code creates a dictionary of input arguments for the workflow, keyed by argument name. This dictionary has one entry that contains the range of the randomly generated number retrieved from the range combo box.

```

Dim inputs As New Dictionary(Of String, Object)()
inputs.Add("MaxNumber", Convert.ToInt32(NumberRange.SelectedItem))

```

```

var inputs = new Dictionary<string, object>();
inputs.Add("MaxNumber", Convert.ToInt32(NumberRange.SelectedItem));

```

3. Next, add the following code that starts the workflow. The `WorkflowIdentity` and workflow definition corresponding to the type of workflow selected are retrieved using the `WorkflowVersionMap` helper class. Next, a new `WorkflowApplication` instance is created using the workflow definition, `WorkflowIdentity`, and dictionary of input arguments.

```

Dim identity As WorkflowIdentity = Nothing
Select Case WorkflowType.SelectedItem.ToString()
    Case "SequentialNumberGuessWorkflow"
        identity = WorkflowVersionMap.SequentialNumberGuessIdentity

    Case "StateMachineNumberGuessWorkflow"
        identity = WorkflowVersionMap.StateMachineNumberGuessIdentity

    Case "FlowchartNumberGuessWorkflow"
        identity = WorkflowVersionMap.FlowchartNumberGuessIdentity
End Select

Dim wf As Activity = WorkflowVersionMap.GetWorkflowDefinition(identity)

Dim wfApp = New WorkflowApplication(wf, inputs, identity)

```

```

WorkflowIdentity identity = null;
switch (WorkflowType.SelectedItem.ToString())
{
    case "SequentialNumberGuessWorkflow":
        identity = WorkflowVersionMap.SequentialNumberGuessIdentity;
        break;

    case "StateMachineNumberGuessWorkflow":
        identity = WorkflowVersionMap.StateMachineNumberGuessIdentity;
        break;

    case "FlowchartNumberGuessWorkflow":
        identity = WorkflowVersionMap.FlowchartNumberGuessIdentity;
        break;
};

Activity wf = WorkflowVersionMap.GetWorkflowDefinition(identity);

WorkflowApplication wfApp = new WorkflowApplication(wf, inputs, identity);

```

4. Next, add the following code which adds the workflow to the workflow list and displays the workflow's

version information on the form.

```
'Add the workflow to the list and display the version information.  
WorkflowStarting = True  
InstanceId.SelectedIndex = InstanceId.Items.Add(wfApp.Id)  
WorkflowVersion.Text = identity.ToString()  
WorkflowStarting = False
```

```
// Add the workflow to the list and display the version information.  
WorkflowStarting = true;  
InstanceId.SelectedIndex = InstanceId.Items.Add(wfApp.Id);  
WorkflowVersion.Text = identity.ToString();  
WorkflowStarting = false;
```

5. Call `ConfigureWorkflowApplication` to configure the instance store, extensions, and workflow lifecycle handlers for this `WorkflowApplication` instance.

```
'Configure the instance store, extensions, and  
'workflow lifecycle handlers.  
ConfigureWorkflowApplication(wfApp)
```

```
// Configure the instance store, extensions, and  
// workflow lifecycle handlers.  
ConfigureWorkflowApplication(wfApp);
```

6. Finally, call `Run`.

```
'Start the workflow.  
wfApp.Run()
```

```
// Start the workflow.  
wfApp.Run();
```

The following example is the completed `NewGame_Click` handler.

```
Private Sub NewGame_Click(sender As Object, e As EventArgs) Handles NewGame.Click
    'Start a new workflow.
    Dim inputs As New Dictionary(Of String, Object)()
    inputs.Add("MaxNumber", Convert.ToInt32(NumberRange.SelectedItem))

    Dim identity As WorkflowIdentity = Nothing
    Select Case WorkflowType.SelectedItem.ToString()
        Case "SequentialNumberGuessWorkflow"
            identity = WorkflowVersionMap.SequentialNumberGuessIdentity

        Case "StateMachineNumberGuessWorkflow"
            identity = WorkflowVersionMap.StateMachineNumberGuessIdentity

        Case "FlowchartNumberGuessWorkflow"
            identity = WorkflowVersionMap.FlowchartNumberGuessIdentity
    End Select

    Dim wf As Activity = WorkflowVersionMap.GetWorkflowDefinition(identity)

    Dim wfApp = New WorkflowApplication(wf, inputs, identity)

    'Add the workflow to the list and display the version information.
    WorkflowStarting = True
    InstanceId.SelectedIndex = InstanceId.Items.Add(wfApp.Id)
    WorkflowVersion.Text = identity.ToString()
    WorkflowStarting = False

    'Configure the instance store, extensions, and
    'workflow lifecycle handlers.
    ConfigureWorkflowApplication(wfApp)

    'Start the workflow.
    wfApp.Run()
End Sub
```

```

private void NewGame_Click(object sender, EventArgs e)
{
    var inputs = new Dictionary<string, object>();
    inputs.Add("MaxNumber", Convert.ToInt32(NumberRange.SelectedItem));

    WorkflowIdentity identity = null;
    switch (WorkflowType.SelectedItem.ToString())
    {
        case "SequentialNumberGuessWorkflow":
            identity = WorkflowVersionMap.SequentialNumberGuessIdentity;
            break;

        case "StateMachineNumberGuessWorkflow":
            identity = WorkflowVersionMap.StateMachineNumberGuessIdentity;
            break;

        case "FlowchartNumberGuessWorkflow":
            identity = WorkflowVersionMap.FlowchartNumberGuessIdentity;
            break;
    };

    Activity wf = WorkflowVersionMap.GetWorkflowDefinition(identity);

    WorkflowApplication wfApp = new WorkflowApplication(wf, inputs, identity);

    // Add the workflow to the list and display the version information.
    WorkflowStarting = true;
    InstanceId.SelectedIndex = InstanceId.Items.Add(wfApp.Id);
    WorkflowVersion.Text = identity.ToString();
    WorkflowStarting = false;

    // Configure the instance store, extensions, and
    // workflow lifecycle handlers.
    ConfigureWorkflowApplication(wfApp);

    // Start the workflow.
    wfApp.Run();
}

```

## To resume a workflow

1. Add a `Click` handler for `EnterGuess`. To add the handler, switch to **Design View** for the form, and double-click `EnterGuess`. Whenever the user clicks this button a workflow is resumed.

```

Private Sub EnterGuess_Click(sender As Object, e As EventArgs) Handles EnterGuess.Click
    End Sub

```

```

private void EnterGuess_Click(object sender, EventArgs e)
{
}

```

2. Add the following code to ensure that a workflow is selected in the workflow list, and that the user's guess is valid.

```

If WorkflowInstanceId = Guid.Empty Then
    MessageBox.Show("Please select a workflow.")
    Return
End If

Dim userGuess As Integer
If Not Int32.TryParse(Guess.Text, userGuess) Then
    MessageBox.Show("Please enter an integer.")
    Guess.SelectAll()
    Guess.Focus()
    Return
End If

```

```

if (WorkflowInstanceId == Guid.Empty)
{
    MessageBox.Show("Please select a workflow.");
    return;
}

int guess;
if (!Int32.TryParse(Guess.Text, out guess))
{
    MessageBox.Show("Please enter an integer.");
    Guess.SelectAll();
    Guess.Focus();
    return;
}

```

3. Next, retrieve the `WorkflowApplicationInstance` of the persisted workflow instance. A

`WorkflowApplicationInstance` represents a persisted workflow instance that has not yet been associated with a workflow definition. The `DefinitionIdentity` of the `WorkflowApplicationInstance` contains the `WorkflowIdentity` of the persisted workflow instance. In this tutorial, the `WorkflowVersionMap` utility class is used to map the `WorkflowIdentity` to the correct workflow definition. Once the workflow definition is retrieved, a `WorkflowApplication` is created, using the correct workflow definition.

```

Dim instance As WorkflowApplicationInstance = _
    WorkflowApplication.GetInstance(WorkflowInstanceId, store)

'Use the persisted WorkflowIdentity to retrieve the correct workflow
'definition from the dictionary.
Dim wf As Activity = _
    WorkflowVersionMap.GetWorkflowDefinition(instance.DefinitionIdentity)

'Associate the WorkflowApplication with the correct definition
Dim wfApp As WorkflowApplication = _
    New WorkflowApplication(wf, instance.DefinitionIdentity)

```

```

WorkflowApplicationInstance instance =
    WorkflowApplication.GetInstance(WorkflowInstanceId, store);

// Use the persisted WorkflowIdentity to retrieve the correct workflow
// definition from the dictionary.
Activity wf =
    WorkflowVersionMap.GetWorkflowDefinition(instance.DefinitionIdentity);

// Associate the WorkflowApplication with the correct definition
WorkflowApplication wfApp =
    new WorkflowApplication(wf, instance.DefinitionIdentity);

```

4. Once the `WorkflowApplication` is created, configure the instance store, workflow lifecycle handlers, and extensions by calling `ConfigureWorkflowApplication`. These steps must be done every time a new `WorkflowApplication` is created, and they must be done before the workflow instance is loaded into the `WorkflowApplication`. After the workflow is loaded, it is resumed with the user's guess.

```
'Configure the extensions and lifecycle handlers.  
'Do this before the instance is loaded. Once the instance is  
'loaded it is too late to add extensions.  
ConfigureWorkflowApplication(wfApp)  
  
'Load the workflow.  
wfApp.Load(instance)  
  
'Resume the workflow.  
wfApp.ResumeBookmark("EnterGuess", userGuess)
```

```
// Configure the extensions and lifecycle handlers.  
// Do this before the instance is loaded. Once the instance is  
// loaded it is too late to add extensions.  
ConfigureWorkflowApplication(wfApp);  
  
// Load the workflow.  
wfApp.Load(instance);  
  
// Resume the workflow.  
wfApp.ResumeBookmark("EnterGuess", guess);
```

5. Finally, clear the guess textbox and prepare the form to accept another guess.

```
'Clear the Guess textbox.  
Guess.Clear()  
Guess.Focus()
```

```
// Clear the Guess textbox.  
Guess.Clear();  
Guess.Focus();
```

The following example is the completed `EnterGuess_Click` handler.

```
Private Sub EnterGuess_Click(sender As Object, e As EventArgs) Handles EnterGuess.Click
    If WorkflowInstanceId = Guid.Empty Then
        MessageBox.Show("Please select a workflow.")
        Return
    End If

    Dim userGuess As Integer
    If Not Int32.TryParse(Guess.Text, userGuess) Then
        MessageBox.Show("Please enter an integer.")
        Guess.SelectAll()
        Guess.Focus()
        Return
    End If

    Dim instance As WorkflowApplicationInstance = _
        WorkflowApplication.GetInstance(WorkflowInstanceId, store)

    'Use the persisted WorkflowIdentity to retrieve the correct workflow
    'definition from the dictionary.
    Dim wf As Activity = _
        WorkflowVersionMap.GetWorkflowDefinition(instance.DefinitionIdentity)

    'Associate the WorkflowApplication with the correct definition
    Dim wfApp As WorkflowApplication = _
        New WorkflowApplication(wf, instance.DefinitionIdentity)

    'Configure the extensions and lifecycle handlers.
    'Do this before the instance is loaded. Once the instance is
    'loaded it is too late to add extensions.
    ConfigureWorkflowApplication(wfApp)

    'Load the workflow.
    wfApp.Load(instance)

    'Resume the workflow.
    wfApp.ResumeBookmark("EnterGuess", userGuess)

    'Clear the Guess textbox.
    Guess.Clear()
    Guess.Focus()
End Sub
```

```

private void EnterGuess_Click(object sender, EventArgs e)
{
    if (WorkflowInstanceId == Guid.Empty)
    {
        MessageBox.Show("Please select a workflow.");
        return;
    }

    int guess;
    if (!Int32.TryParse(Guess.Text, out guess))
    {
        MessageBox.Show("Please enter an integer.");
        Guess.SelectAll();
        Guess.Focus();
        return;
    }

    WorkflowApplicationInstance instance =
        WorkflowApplication.GetInstance(WorkflowInstanceId, store);

    // Use the persisted WorkflowIdentity to retrieve the correct workflow
    // definition from the dictionary.
    Activity wf =
        WorkflowVersionMap.GetWorkflowDefinition(instance.DefinitionIdentity);

    // Associate the WorkflowApplication with the correct definition
    WorkflowApplication wfApp =
        new WorkflowApplication(wf, instance.DefinitionIdentity);

    // Configure the extensions and lifecycle handlers.
    // Do this before the instance is loaded. Once the instance is
    // loaded it is too late to add extensions.
    ConfigureWorkflowApplication(wfApp);

    // Load the workflow.
    wfApp.Load(instance);

    // Resume the workflow.
    wfApp.ResumeBookmark("EnterGuess", guess);

    // Clear the Guess textbox.
    Guess.Clear();
    Guess.Focus();
}

```

## To terminate a workflow

1. Add a `Click` handler for `QuitGame`. To add the handler, switch to **Design View** for the form, and double-click `QuitGame`. Whenever the user clicks this button the currently selected workflow is terminated.

```

Private Sub QuitGame_Click(sender As Object, e As EventArgs) Handles QuitGame.Click

End Sub

```

```

private void QuitGame_Click(object sender, EventArgs e)
{
}

```

2. Add the following code to the `QuitGame_Click` handler. This code first checks to ensure that a workflow is selected in the workflow list. Then it loads the persisted instance into a `WorkflowApplicationInstance`, uses the `DefinitionIdentity` to determine the correct workflow definition, and then initializes the

`WorkflowApplication`. Next the extensions and workflow lifecycle handlers are configured with a call to `ConfigureWorkflowApplication`. Once the `WorkflowApplication` is configured, it is loaded, and then `Terminate` is called.

```
If WorkflowInstanceId = Guid.Empty Then
    MessageBox.Show("Please select a workflow.")
    Return
End If

Dim instance As WorkflowApplicationInstance = _
    WorkflowApplication.GetInstance(WorkflowInstanceId, store)

'Use the persisted WorkflowIdentity to retrieve the correct workflow
'definition from the dictionary.
Dim wf As Activity = WorkflowVersionMap.GetWorkflowDefinition(instance.DefinitionIdentity)

'Associate the WorkflowApplication with the correct definition.
Dim wfApp As WorkflowApplication = _
    New WorkflowApplication(wf, instance.DefinitionIdentity)

'Configure the extensions and lifecycle handlers.
ConfigureWorkflowApplication(wfApp)

'Load the workflow.
wfApp.Load(instance)

'Terminate the workflow.
wfApp.Terminate("User resigns.")
```

```
if (WorkflowInstanceId == Guid.Empty)
{
    MessageBox.Show("Please select a workflow.");
    return;
}

WorkflowApplicationInstance instance =
    WorkflowApplication.GetInstance(WorkflowInstanceId, store);

// Use the persisted WorkflowIdentity to retrieve the correct workflow
// definition from the dictionary.
Activity wf = WorkflowVersionMap.GetWorkflowDefinition(instance.DefinitionIdentity);

// Associate the WorkflowApplication with the correct definition
WorkflowApplication wfApp =
    new WorkflowApplication(wf, instance.DefinitionIdentity);

// Configure the extensions and lifecycle handlers
ConfigureWorkflowApplication(wfApp);

// Load the workflow.
wfApp.Load(instance);

// Terminate the workflow.
wfApp.Terminate("User resigns.");
```

## To build and run the application

1. Double-click **Program.cs** (or **Module1.vb**) in **Solution Explorer** to display the code.
2. Add the following `using` (or `Imports`) statement at the top of the file with the other `using` (or `Imports`) statements.

```
Imports System.Windows.Forms
```

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

3. Remove or comment out the existing workflow hosting code from [How to: Run a Workflow](#), and replace it with the following code.

```
Sub Main()
    Application.EnableVisualStyles()
    Application.Run(New WorkflowHostForm())
End Sub
```

```
static void Main(string[] args)
{
    Application.EnableVisualStyles();
    Application.Run(new WorkflowHostForm());
}
```

4. Right-click **NumberGuessWorkflowHost** in **Solution Explorer** and choose **Properties**. In the **Application** tab, specify **Windows Application** for the **Output type**. This step is optional, but if it is not followed the console window is displayed in addition to the form.
5. Press Ctrl+Shift+B to build the application.
6. Ensure that **NumberGuessWorkflowHost** is set as the startup application, and press Ctrl+F5 to start the application.
7. Select a range for the guessing game and the type of workflow to start, and click **New Game**. Enter a guess in the **Guess** box and click **Go** to submit your guess. Note that the output from the **WriteLine** activities is displayed on the form.
8. Start several workflows using different workflow types and number ranges, enter some guesses, and switch between the workflows by selecting from the **Workflow Instance Id** list.

Note that when you switch to a new workflow, the previous guesses and progress of the workflow are not displayed in the status window. The reason the status is not available is because it is not captured and saved anywhere. In the next step of the tutorial, [How to: Create a Custom Tracking Participant](#), you create a custom tracking participant that saves this information.

# How to: Create a Custom Tracking Participant

3/9/2019 • 8 minutes to read • [Edit Online](#)

Workflow tracking provides visibility into the status of workflow execution. The workflow runtime emits tracking records that describe workflow lifecycle events, activity lifecycle events, bookmark resumptions, and faults. These tracking records are consumed by tracking participants. Windows Workflow Foundation (WF) includes a standard tracking participant that writes tracking records as Event Tracing for Windows (ETW) events. If that does not meet your requirements, you can also write a custom tracking participant. This tutorial step describes how to create a custom tracking participant and tracking profile that capture the output of `WriteLine` activities so that they can be displayed to the user.

## NOTE

Each topic in the Getting Started tutorial depends on the previous topics. To complete this topic, you must first complete the previous topics. To download a completed version or view a video walkthrough of the tutorial, see [Windows Workflow Foundation \(WF45\) - Getting Started Tutorial](#).

## To create the custom tracking participant

1. Right-click **NumberGuessWorkflowHost** in **Solution Explorer** and choose **Add, Class**. Type `StatusTrackingParticipant` into the **Name** box, and click **Add**.
2. Add the following `using` (or `Imports`) statements at the top of the file with the other `using` (or `Imports`) statements.

```
Imports System.Activities.Tracking  
Imports System.IO
```

```
using System.Activities.Tracking;  
using System.IO;
```

3. Modify the `StatusTrackingParticipant` class so that it inherits from `TrackingParticipant`.

```
Public Class StatusTrackingParticipant  
    Inherits TrackingParticipant  
  
End Class
```

```
class StatusTrackingParticipant : TrackingParticipant  
{  
}
```

4. Add the following `Track` method override. There are several different types of tracking records. We are interested in the output of `WriteLine` activities, which are contained in activity tracking records. If the `TrackingRecord` is an `ActivityTrackingRecord` for a `WriteLine` activity, the `Text` of the `WriteLine` is appended to a file named after the `InstanceId` of the workflow. In this tutorial, the file is saved to the current folder of the host application.

```

Protected Overrides Sub Track(record As TrackingRecord, timeout As TimeSpan)
    Dim asr As ActivityStateRecord = TryCast(record, ActivityStateRecord)

    If Not asr Is Nothing Then
        If asr.State = ActivityStates.Executing And _
            asr.Activity.TypeName = "System.Activities.Statements.WriteLine" Then

            'Append the WriteLine output to the tracking
            'file for this instance.
            Using writer As StreamWriter = File.AppendText(record.InstanceId.ToString())
                writer.WriteLine(asr.Arguments("Text"))
                writer.Close()
            End Using
        End If
    End If
End Sub

```

```

protected override void Track(TrackingRecord record, TimeSpan timeout)
{
    ActivityStateRecord asr = record as ActivityStateRecord;

    if (asr != null)
    {
        if (asr.State == ActivityStates.Executing &&
            asr.Activity.TypeName == "System.Activities.Statements.WriteLine")
        {
            // Append the WriteLine output to the tracking
            // file for this instance
            using (StreamWriter writer = File.AppendText(record.InstanceId.ToString()))
            {
                writer.WriteLine(asr.Arguments["Text"]);
                writer.Close();
            }
        }
    }
}

```

When no tracking profile is specified, the default tracking profile is used. When the default tracking profile is used, tracking records are emitted for all `ActivityStates`. Because we only need to capture the text one time during the lifecycle of the `WriteLine` activity, we only extract the text from the `ActivityStates.Executing` state. In [To create the tracking profile and register the tracking participant](#), a tracking profile is created that specifies that only `WriteLine` `ActivityStates.Executing` tracking records are emitted.

## To create the tracking profile and register the tracking participant

1. Right-click **WorkflowHostForm** in **Solution Explorer** and choose **View Code**.
2. Add the following `using` (or `Imports`) statement at the top of the file with the other `using` (or `Imports`) statements.

```
Imports System.Activities.Tracking
```

```
using System.Activities.Tracking;
```

3. Add the following code to `ConfigureWorkflowApplication` just after the code that adds the `StringWriter` to the workflow extensions and before the workflow lifecycle handlers.

```

'Add the custom tracking participant with a tracking profile
'that only emits tracking records for WriteLine activities.
Dim query As New ActivityStateQuery()
query.ActivityName = "WriteLine"
query.States.Add(ActivityStates.Executing)
query.Arguments.Add("Text")

Dim profile As New TrackingProfile()
profile.Questions.Add(query)

Dim stp As New StatusTrackingParticipant()
stp.TrackingProfile = profile

wfApp.Extensions.Add(stp)

```

```

// Add the custom tracking participant with a tracking profile
// that only emits tracking records for WriteLine activities.
StatusTrackingParticipant stp = new StatusTrackingParticipant
{
    TrackingProfile = new TrackingProfile
    {
        Questions =
        {
            new ActivityStateQuery
            {
                ActivityName = "WriteLine",
                States = { ActivityStates.Executing },
                Arguments = { "Text" }
            }
        }
    }
};

wfApp.Extensions.Add(stp);

```

This tracking profile specifies that only activity state records for `WriteLine` activities in the `Executing` state are emitted to the custom tracking participant.

After adding the code, the start of `ConfigureWorkflowApplication` will look like the following example.

```

Private Sub ConfigureWorkflowApplication(wfApp As WorkflowApplication)
    'Configure the persistence store.
    wfApp.InstanceStore = store

    'Add a StringWriter to the extensions. This captures the output
    'from the WriteLine activities so we can display it in the form.
    Dim sw As New StringWriter()
    wfApp.Extensions.Add(sw)

    'Add the custom tracking participant with a tracking profile
    'that only emits tracking records for WriteLine activities.
    Dim query As New ActivityStateQuery()
    query.ActivityName = "WriteLine"
    query.States.Add(ActivityStates.Executing)
    query.Arguments.Add("Text")

    Dim profile As New TrackingProfile()
    profile.Questions.Add(query)

    Dim stp As New StatusTrackingParticipant()
    stp.TrackingProfile = profile

    wfApp.Extensions.Add(stp)

    'Workflow lifecycle handlers...

```

```

private void ConfigureWorkflowApplication(WorkflowApplication wfApp)
{
    // Configure the persistence store.
    wfApp.InstanceStore = store;

    // Add a StringWriter to the extensions. This captures the output
    // from the WriteLine activities so we can display it in the form.
    StringWriter sw = new StringWriter();
    wfApp.Extensions.Add(sw);

    // Add the custom tracking participant with a tracking profile
    // that only emits tracking records for WriteLine activities.
    StatusTrackingParticipant stp = new StatusTrackingParticipant
    {
        TrackingProfile = new TrackingProfile
        {
            Questions =
            {
                new ActivityStateQuery
                {
                    ActivityName = "WriteLine",
                    States = { ActivityStates.Executing },
                    Arguments = { "Text" }
                }
            }
        };
    };

    wfApp.Extensions.Add(stp);

    // Workflow lifecycle handlers...

```

## To display the tracking information

1. Right-click **WorkflowHostForm** in **Solution Explorer** and choose **View Code**.
2. In the `InstanceId_SelectedIndexChanged` handler, add the following code immediately after the code that

clears the status window.

```
'If there is tracking data for this workflow, display it
'in the status window.
If File.Exists(WorkflowInstanceId.ToString()) Then
    Dim status As String = File.ReadAllText(WorkflowInstanceId.ToString())
    UpdateStatus(status)
End If
```

```
// If there is tracking data for this workflow, display it
// in the status window.
if (File.Exists(WorkflowInstanceId.ToString()))
{
    string status = File.ReadAllText(WorkflowInstanceId.ToString());
    UpdateStatus(status);
}
```

When a new workflow is selected in the workflow list, the tracking records for that workflow are loaded and displayed in the status window. The following example is the completed `InstanceId_SelectedIndexChanged` handler.

```
Private Sub InstanceId_SelectedIndexChanged(sender As Object, e As EventArgs) Handles
InstanceId.SelectedIndexChanged
    If InstanceId.SelectedIndex = -1 Then
        Return
    End If

    'Clear the status window.
    WorkflowStatus.Clear()

    'If there is tracking data for this workflow, display it
    'in the status window.
    If File.Exists(WorkflowInstanceId.ToString()) Then
        Dim status As String = File.ReadAllText(WorkflowInstanceId.ToString())
        UpdateStatus(status)
    End If

    'Get the workflow version and display it.
    'If the workflow is just starting then this info will not
    'be available in the persistence store so do not try and retrieve it.
    If Not WorkflowStarting Then
        Dim instance As WorkflowApplicationInstance = _
            WorkflowApplication.GetInstance(WorkflowInstanceId, store)

        WorkflowVersion.Text = _
            WorkflowVersionMap.GetIdentityDescription(instance.DefinitionIdentity)

        'Unload the instance.
        instance.Abandon()
    End If
End Sub
```

```

private void InstanceId_SelectedIndexChanged(object sender, EventArgs e)
{
    if (InstanceId.SelectedIndex == -1)
    {
        return;
    }

    // Clear the status window.
    WorkflowStatus.Clear();

    // If there is tracking data for this workflow, display it
    // in the status window.
    if (File.Exists(WorkflowInstanceId.ToString()))
    {
        string status = File.ReadAllText(WorkflowInstanceId.ToString());
        UpdateStatus(status);
    }

    // Get the workflow version and display it.
    // If the workflow is just starting then this info will not
    // be available in the persistence store so do not try and retrieve it.
    if (!WorkflowStarting)
    {
        WorkflowApplicationInstance instance =
            WorkflowApplication.GetInstance(this.WorkflowInstanceId, store);

        WorkflowVersion.Text =
            WorkflowVersionMap.GetIdentityDescription(instance.DefinitionIdentity);

        // Unload the instance.
        instance.Abandon();
    }
}

```

## To build and run the application

1. Press Ctrl+Shift+B to build the application.
2. Press Ctrl+F5 to start the application.
3. Select a range for the guessing game and the type of workflow to start, and click **New Game**. Enter a guess in the **Guess** box and click **Go** to submit your guess. Note that the status of the workflow is displayed in the status window. This output is captured from the `WriteLine` activities. Switch to a different workflow by selecting one from the **Workflow Instance Id** combo box and note that the status of the current workflow is removed. Switch back to the previous workflow and note that the status is restored, similar to the following example.

### NOTE

If you switch to a workflow that was started before tracking was enabled no status is displayed. However if you make additional guesses, their status is saved because tracking is now enabled.

```

Please enter a number between 1 and 10
Your guess is too high.
Please enter a number between 1 and 10

```

#### NOTE

This information is useful for determining the range of the random number, but it does not contain any information about what guesses have been previously made. This information is in the next step, [How to: Host Multiple Versions of a Workflow Side-by-Side](#).

Make a note of the workflow instance id, and play the game through to its completion.

4. Open Windows Explorer and navigate to the **NumberGuessWorkflowHost\bin\debug** folder (or **bin\release** depending on your project settings). Note that in addition to the project executable files there are files with guid filenames. Identify the one that corresponds to the workflow instance id from the completed workflow in the previous step and open it in Notepad. The tracking information contains information similar to the following.

```
Please enter a number between 1 and 10
Your guess is too high.
Please enter a number between 1 and 10
Your guess is too high.
Please enter a number between 1 and 10
```

In addition to the absence of the user's guesses, this tracking data does not contain information about the final guess of the workflow. That is because the tracking information consists only of the `WriteLine` output from the workflow, and the final message that is displayed is done so from the `Completed` handler after the workflow completes. In next step of the tutorial, [How to: Host Multiple Versions of a Workflow Side-by-Side](#), the existing `WriteLine` activities are modified to display the user's guesses, and an additional `WriteLine` activity is added that displays the final results. After these changes are integrated, [How to: Host Multiple Versions of a Workflow Side-by-Side](#) demonstrates how to host multiple versions of a workflow at the same time.

# How to: Host Multiple Versions of a Workflow Side-by-Side

3/9/2019 • 13 minutes to read • [Edit Online](#)

`WorkflowIdentity` provides a way for workflow application developers to associate a name and a version with a workflow definition, and for this information to be associated with a persisted workflow instance. This identity information can be used by workflow application developers to enable scenarios such as side-by-side execution of multiple versions of a workflow definition, and provides the cornerstone for other functionality such as dynamic update. This step in the tutorial demonstrates how to use `WorkflowIdentity` to host multiple versions of a workflow at the same time.

## NOTE

To download a completed version or view a video walkthrough of the tutorial, see [Windows Workflow Foundation \(WF45\) - Getting Started Tutorial](#).

## In this topic

In this step of the tutorial, the `WriteLine` activities in the workflow are modified to provide additional information, and a new `WriteLine` activity is added. A copy of the original workflow assembly is stored, and the host application is updated so that it can run both the original and the updated workflows at the same time.

- [To make a copy of the NumberGuessWorkflowActivities project](#)
- [To update the workflows](#)
  - [To update the StateMachine workflow](#)
  - [To update the Flowchart workflow](#)
  - [To update the Sequential workflow](#)
- [To update WorkflowVersionMap to include the previous workflow versions](#)
- [To build and run the application](#)

## NOTE

Before following the steps in this topic, run the application, start several workflows of each type, and making one or two guesses for each one. These persisted workflows are used in this step and the following step, [How to: Update the Definition of a Running Workflow Instance](#).

## NOTE

Each step in the Getting Started tutorial depends on the previous steps. If you did not complete the previous steps you can download a completed version of the tutorial from [Windows Workflow Foundation \(WF45\) - Getting Started Tutorial](#).

### To make a copy of the NumberGuessWorkflowActivities project

1. Open the **WF45GettingStartedTutorial** solution in Visual Studio 2012 if it is not open.

2. Press CTRL+SHIFT+B to build the solution.
3. Close the **WF45GettingStartedTutorial** solution.
4. Open Windows Explorer and navigate to the folder where the tutorial solution file and the project folders are located.
5. Create a new folder named **PreviousVersions** in the same folder as **NumberGuessWorkflowHost** and **NumberGuessWorkflowActivities**. This folder is used to contain the assemblies that contain the different versions of the workflows used in the subsequent tutorial steps.
6. Navigate to the **NumberGuessWorkflowActivities\bin\debug** folder (or **bin\release** depending on your project settings). Copy **NumberGuessWorkflowActivities.dll** and paste it into the **PreviousVersions** folder.
7. Rename **NumberGuessWorkflowActivities.dll** in the **PreviousVersions** folder to **NumberGuessWorkflowActivities\_v1.dll**.

**NOTE**

The steps in this topic demonstrate one way to manage the assemblies used to contain multiple versions of the workflows. Other methods such as strong naming the assemblies and registering them in the global assembly cache could also be used.

8. Create a new folder named **NumberGuessWorkflowActivities\_du** in the same folder as **NumberGuessWorkflowHost**, **NumberGuessWorkflowActivities**, and the newly added **PreviousVersions** folder, and copy all of the files and subfolders from the **NumberGuessWorkflowActivities** folder into the new **NumberGuessWorkflowActivities\_du** folder. This backup copy of the project for the initial version of the activities is used in [How to: Update the Definition of a Running Workflow Instance](#).
9. Re-open the **WF45GettingStartedTutorial** solution in Visual Studio 2012.

### To update the workflows

In this section, the workflow definitions are updated. The two `WriteLine` activities that give feedback on the user's guess are updated, and a new `WriteLine` activity is added that provides additional information about the game once the number is guessed.

#### To update the StateMachine workflow

1. In **Solution Explorer**, under the **NumberGuessWorkflowActivities** project, double-click **StateMachineNumberGuessWorkflow.xaml**.
2. Double-click the **Guess Incorrect** transition on the state machine.
3. Update the `Text` of the left-most `WriteLine` in the `If` activity.

Guess & " is too low."

Guess + " is too low."

4. Update the `Text` of the right-most `WriteLine` in the `If` activity.

Guess & " is too high."

```
Guess + " is too high."
```

5. Return to the overall state machine view in the workflow designer by clicking **StateMachine** in the breadcrumb display at the top of the workflow designer.
6. Double-click the **Guess Correct** transition on the state machine.
7. Drag a **WriteLine** activity from the **Primitives** section of the **Toolbox** and drop it on the **Drop Action activity here** label of the transition.
8. Type the following expression into the **Text** property box.

```
Guess & " is correct. You guessed it in " & Turns & " turns."
```

```
Guess + " is correct. You guessed it in " + Turns + " turns."
```

#### To update the Flowchart workflow

1. In **Solution Explorer**, under the **NumberGuessWorkflowActivities** project, double-click **FlowchartNumberGuessWorkflow.xaml**.
2. Update the **Text** of the left-most **WriteLine** activity.

```
Guess & " is too low."
```

```
Guess + " is too low."
```

3. Update the **Text** of the right-most **WriteLine** activity.

```
Guess & " is too high."
```

```
Guess + " is too high."
```

4. Drag a **WriteLine** activity from the **Primitives** section of the **Toolbox** and drop it on the drop point of the **True** action of the topmost **FlowDecision**. The **WriteLine** activity is added to the flowchart and linked to the **True** action of the **FlowDecision**.
5. Type the following expression into the **Text** property box.

```
Guess & " is correct. You guessed it in " & Turns & " turns."
```

```
Guess + " is correct. You guessed it in " + Turns + " turns."
```

#### To update the Sequential workflow

1. In **Solution Explorer**, under the **NumberGuessWorkflowActivities** project, double-click **SequentialNumberGuessWorkflow.xaml**.
2. Update the **Text** of the left-most **WriteLine** in the **If** activity.

```
Guess & " is too low."
```

```
Guess + " is too low."
```

3. Update the `Text` of the right-most `WriteLine` activity in the `If` activity.

```
Guess & " is too high."
```

```
Guess + " is too high."
```

4. Drag a **WriteLine** activity from the **Primitives** section of the **Toolbox** and drop it after the **DoWhile** activity so that the **WriteLine** is the final activity in the root `Sequence` activity.  
5. Type the following expression into the `Text` property box.

```
Guess & " is correct. You guessed it in " & Turns & " turns."
```

```
Guess + " is correct. You guessed it in " + Turns + " turns."
```

#### To update WorkflowVersionMap to include the previous workflow versions

1. Double-click **WorkflowVersionMap.cs** (or **WorkflowVersionMap.vb**) under the **NumberGuessWorkflowHost** project to open it.
2. Add the following `using` (or `Imports`) statements to the top of the file with the other `using` (or `Imports`) statements.

```
Imports System.Reflection  
Imports System.IO
```

```
using System.Reflection;  
using System.IO;
```

3. Add three new workflow identities just below the three existing workflow identity declarations. These new `v1` workflow identities will be used provide the correct workflow definition to workflows started before the updates were made.

```
'Current version identities.  
Public StateMachineNumberGuessIdentity As WorkflowIdentity  
Public FlowchartNumberGuessIdentity As WorkflowIdentity  
Public SequentialNumberGuessIdentity As WorkflowIdentity  
  
'v1 Identities.  
Public StateMachineNumberGuessIdentity_v1 As WorkflowIdentity  
Public FlowchartNumberGuessIdentity_v1 As WorkflowIdentity  
Public SequentialNumberGuessIdentity_v1 As WorkflowIdentity
```

```

// Current version identities.

static public WorkflowIdentity StateMachineNumberGuessIdentity;
static public WorkflowIdentity FlowchartNumberGuessIdentity;
static public WorkflowIdentity SequentialNumberGuessIdentity;

// v1 identities.
static public WorkflowIdentity StateMachineNumberGuessIdentity_v1;
static public WorkflowIdentity FlowchartNumberGuessIdentity_v1;
static public WorkflowIdentity SequentialNumberGuessIdentity_v1;

```

4. In the `WorkflowVersionMap` constructor, update the `Version` property of the three current workflow identities to `2.0.0.0`.

```

'Add the current workflow version identities.
StateMachineNumberGuessIdentity = New WorkflowIdentity With
{
    .Name = "StateMachineNumberGuessWorkflow",
    .Version = New Version(2, 0, 0, 0)
}

FlowchartNumberGuessIdentity = New WorkflowIdentity With
{
    .Name = "FlowchartNumberGuessWorkflow",
    .Version = New Version(2, 0, 0, 0)
}

SequentialNumberGuessIdentity = New WorkflowIdentity With
{
    .Name = "SequentialNumberGuessWorkflow",
    .Version = New Version(2, 0, 0, 0)
}

map.Add(StateMachineNumberGuessIdentity, New StateMachineNumberGuessWorkflow())
map.Add(FlowchartNumberGuessIdentity, New FlowchartNumberGuessWorkflow())
map.Add(SequentialNumberGuessIdentity, New SequentialNumberGuessWorkflow())

```

```

// Add the current workflow version identities.
StateMachineNumberGuessIdentity = new WorkflowIdentity
{
    Name = "StateMachineNumberGuessWorkflow",
    // Version = new Version(1, 0, 0, 0),
    Version = new Version(2, 0, 0, 0)
};

FlowchartNumberGuessIdentity = new WorkflowIdentity
{
    Name = "FlowchartNumberGuessWorkflow",
    // Version = new Version(1, 0, 0, 0),
    Version = new Version(2, 0, 0, 0)
};

SequentialNumberGuessIdentity = new WorkflowIdentity
{
    Name = "SequentialNumberGuessWorkflow",
    // Version = new Version(1, 0, 0, 0),
    Version = new Version(2, 0, 0, 0)
};

map.Add(StateMachineNumberGuessIdentity, new StateMachineNumberGuessWorkflow());
map.Add(FlowchartNumberGuessIdentity, new FlowchartNumberGuessWorkflow());
map.Add(SequentialNumberGuessIdentity, new SequentialNumberGuessWorkflow());

```

The code in that adds the current versions of the workflows to the dictionary uses the current versions that are referenced in the project, so the code that initializes the workflow definitions does not need to be updated.

5. Add the following code in the constructor just after the code that adds the current versions to the dictionary.

```
'Initialize the previous workflow version identities.  
StateMachineNumberGuessIdentity_v1 = New WorkflowIdentity With  
{  
    .Name = "StateMachineNumberGuessWorkflow",  
    .Version = New Version(1, 0, 0, 0)  
}  
  
FlowchartNumberGuessIdentity_v1 = New WorkflowIdentity With  
{  
    .Name = "FlowchartNumberGuessWorkflow",  
    .Version = New Version(1, 0, 0, 0)  
}  
  
SequentialNumberGuessIdentity_v1 = New WorkflowIdentity With  
{  
    .Name = "SequentialNumberGuessWorkflow",  
    .Version = New Version(1, 0, 0, 0)  
}
```

```
// Initialize the previous workflow version identities.  
StateMachineNumberGuessIdentity_v1 = new WorkflowIdentity  
{  
    Name = "StateMachineNumberGuessWorkflow",  
    Version = new Version(1, 0, 0, 0)  
};  
  
FlowchartNumberGuessIdentity_v1 = new WorkflowIdentity  
{  
    Name = "FlowchartNumberGuessWorkflow",  
    Version = new Version(1, 0, 0, 0)  
};  
  
SequentialNumberGuessIdentity_v1 = new WorkflowIdentity  
{  
    Name = "SequentialNumberGuessWorkflow",  
    Version = new Version(1, 0, 0, 0)  
};
```

These workflow identities are associated with the initial versions of the corresponding workflow definitions.

6. Next, load the assembly that contains the initial version of the workflow definitions, and create and add the corresponding workflow definitions to the dictionary.

```

'Add the previous version workflow identities to the dictionary along with
'the corresponding workflow definitions loaded from the v1 assembly.
'Assembly.LoadFile requires an absolute path so convert this relative path
'to an absolute path.
Dim v1AssemblyPath As String = "..\..\..\PreviousVersions\NumberGuessWorkflowActivities_v1.dll"
v1AssemblyPath = Path.GetFullPath(v1AssemblyPath)
Dim v1Assembly As Assembly = Assembly.LoadFile(v1AssemblyPath)

map.Add(StateMachineNumberGuessIdentity_v1,
    v1Assembly.CreateInstance("NumberGuessWorkflowActivities.StateMachineNumberGuessWorkflow"))

map.Add(SequentialNumberGuessIdentity_v1,
    v1Assembly.CreateInstance("NumberGuessWorkflowActivities.SequentialNumberGuessWorkflow"))

map.Add(FlowchartNumberGuessIdentity_v1,
    v1Assembly.CreateInstance("NumberGuessWorkflowActivities.FlowchartNumberGuessWorkflow"))

```

```

// Add the previous version workflow identities to the dictionary along with
// the corresponding workflow definitions loaded from the v1 assembly.
// Assembly.LoadFile requires an absolute path so convert this relative path
// to an absolute path.
string v1AssemblyPath = @"..\..\..\PreviousVersions\NumberGuessWorkflowActivities_v1.dll";
v1AssemblyPath = Path.GetFullPath(v1AssemblyPath);
Assembly v1Assembly = Assembly.LoadFile(v1AssemblyPath);

map.Add(StateMachineNumberGuessIdentity_v1,
    v1Assembly.CreateInstance("NumberGuessWorkflowActivities.StateMachineNumberGuessWorkflow") as
Activity);

map.Add(SequentialNumberGuessIdentity_v1,
    v1Assembly.CreateInstance("NumberGuessWorkflowActivities.SequentialNumberGuessWorkflow") as
Activity);

map.Add(FlowchartNumberGuessIdentity_v1,
    v1Assembly.CreateInstance("NumberGuessWorkflowActivities.FlowchartNumberGuessWorkflow") as
Activity);

```

The following example is the complete listing for the updated `WorkflowVersionMap` class.

```

Public Module WorkflowVersionMap
    Dim map As Dictionary(Of WorkflowIdentity, Activity)

    'Current version identities.
    Public StateMachineNumberGuessIdentity As WorkflowIdentity
    Public FlowchartNumberGuessIdentity As WorkflowIdentity
    Public SequentialNumberGuessIdentity As WorkflowIdentity

    'v1 Identities.
    Public StateMachineNumberGuessIdentity_v1 As WorkflowIdentity
    Public FlowchartNumberGuessIdentity_v1 As WorkflowIdentity
    Public SequentialNumberGuessIdentity_v1 As WorkflowIdentity

    Sub New()
        map = New Dictionary(Of WorkflowIdentity, Activity)

        'Add the current workflow version identities.
        StateMachineNumberGuessIdentity = New WorkflowIdentity With
        {
            .Name = "StateMachineNumberGuessWorkflow",
            .Version = New Version(2, 0, 0, 0)
        }

        FlowchartNumberGuessIdentity = New WorkflowIdentity With
        {

```

```

        .Name = "FlowchartNumberGuessWorkflow",
        .Version = New Version(2, 0, 0, 0)
    }

    SequentialNumberGuessIdentity = New WorkflowIdentity With
    {
        .Name = "SequentialNumberGuessWorkflow",
        .Version = New Version(2, 0, 0, 0)
    }

    map.Add(StateMachineNumberGuessIdentity, New StateMachineNumberGuessWorkflow())
    map.Add(FlowchartNumberGuessIdentity, New FlowchartNumberGuessWorkflow())
    map.Add(SequentialNumberGuessIdentity, New SequentialNumberGuessWorkflow())

    'Initialize the previous workflow version identities.
    StateMachineNumberGuessIdentity_v1 = New WorkflowIdentity With
    {
        .Name = "StateMachineNumberGuessWorkflow",
        .Version = New Version(1, 0, 0, 0)
    }

    FlowchartNumberGuessIdentity_v1 = New WorkflowIdentity With
    {
        .Name = "FlowchartNumberGuessWorkflow",
        .Version = New Version(1, 0, 0, 0)
    }

    SequentialNumberGuessIdentity_v1 = New WorkflowIdentity With
    {
        .Name = "SequentialNumberGuessWorkflow",
        .Version = New Version(1, 0, 0, 0)
    }

    'Add the previous version workflow identities to the dictionary along with
    'the corresponding workflow definitions loaded from the v1 assembly.
    'Assembly.LoadFile requires an absolute path so convert this relative path
    'to an absolute path.
    Dim v1AssemblyPath As String =
    "...\\..\\..\\PreviousVersions\\NumberGuessWorkflowActivities_v1.dll"
    v1AssemblyPath = Path.GetFullPath(v1AssemblyPath)
    Dim v1Assembly As Assembly = Assembly.LoadFile(v1AssemblyPath)

    map.Add(StateMachineNumberGuessIdentity_v1,
    v1Assembly.CreateInstance("NumberGuessWorkflowActivities.StateMachineNumberGuessWorkflow"))

    map.Add(SequentialNumberGuessIdentity_v1,
    v1Assembly.CreateInstance("NumberGuessWorkflowActivities.SequentialNumberGuessWorkflow"))

    map.Add(FlowchartNumberGuessIdentity_v1,
    v1Assembly.CreateInstance("NumberGuessWorkflowActivities.FlowchartNumberGuessWorkflow"))
End Sub

Public Function GetWorkflowDefinition(identity As WorkflowIdentity) As Activity
    Return map(identity)
End Function

Public Function GetIdentityDescription(identity As WorkflowIdentity) As String
    Return identity.ToString()
End Function
End Module

```

```

public static class WorkflowVersionMap
{
    static Dictionary<WorkflowIdentity, Activity> map;

    // Current version identities.

```

```

static public WorkflowIdentity StateMachineNumberGuessIdentity;
static public WorkflowIdentity FlowchartNumberGuessIdentity;
static public WorkflowIdentity SequentialNumberGuessIdentity;

// v1 identities.
static public WorkflowIdentity StateMachineNumberGuessIdentity_v1;
static public WorkflowIdentity FlowchartNumberGuessIdentity_v1;
static public WorkflowIdentity SequentialNumberGuessIdentity_v1;

static WorkflowVersionMap()
{
    map = new Dictionary<WorkflowIdentity, Activity>();

    // Add the current workflow version identities.
    StateMachineNumberGuessIdentity = new WorkflowIdentity
    {
        Name = "StateMachineNumberGuessWorkflow",
        // Version = new Version(1, 0, 0, 0),
        Version = new Version(2, 0, 0, 0)
    };

    FlowchartNumberGuessIdentity = new WorkflowIdentity
    {
        Name = "FlowchartNumberGuessWorkflow",
        // Version = new Version(1, 0, 0, 0),
        Version = new Version(2, 0, 0, 0)
    };

    SequentialNumberGuessIdentity = new WorkflowIdentity
    {
        Name = "SequentialNumberGuessWorkflow",
        // Version = new Version(1, 0, 0, 0),
        Version = new Version(2, 0, 0, 0)
    };

    map.Add(StateMachineNumberGuessIdentity, new StateMachineNumberGuessWorkflow());
    map.Add(FlowchartNumberGuessIdentity, new FlowchartNumberGuessWorkflow());
    map.Add(SequentialNumberGuessIdentity, new SequentialNumberGuessWorkflow());

    // Initialize the previous workflow version identities.
    StateMachineNumberGuessIdentity_v1 = new WorkflowIdentity
    {
        Name = "StateMachineNumberGuessWorkflow",
        Version = new Version(1, 0, 0, 0)
    };

    FlowchartNumberGuessIdentity_v1 = new WorkflowIdentity
    {
        Name = "FlowchartNumberGuessWorkflow",
        Version = new Version(1, 0, 0, 0)
    };

    SequentialNumberGuessIdentity_v1 = new WorkflowIdentity
    {
        Name = "SequentialNumberGuessWorkflow",
        Version = new Version(1, 0, 0, 0)
    };

    // Add the previous version workflow identities to the dictionary along with
    // the corresponding workflow definitions loaded from the v1 assembly.
    // Assembly.LoadFile requires an absolute path so convert this relative path
    // to an absolute path.
    string v1AssemblyPath = @"..\..\..\PreviousVersions\NumberGuessWorkflowActivities_v1.dll";
    v1AssemblyPath = Path.GetFullPath(v1AssemblyPath);
    Assembly v1Assembly = Assembly.LoadFile(v1AssemblyPath);

    map.Add(StateMachineNumberGuessIdentity_v1,
        v1Assembly.CreateInstance("NumberGuessWorkflowActivities.StateMachineNumberGuessWorkflow")
    as Activity);
}

```

```

        map.Add(SequentialNumberGuessIdentity_v1,
            v1Assembly.CreateInstance("NumberGuessWorkflowActivities.SequentialNumberGuessWorkflow")
        as Activity);

        map.Add(FlowchartNumberGuessIdentity_v1,
            v1Assembly.CreateInstance("NumberGuessWorkflowActivities.FlowchartNumberGuessWorkflow") as
        Activity);
    }

    public static Activity GetWorkflowDefinition(WorkflowIdentity identity)
    {
        return map[identity];
    }

    public static string GetIdentityDescription(WorkflowIdentity identity)
    {
        return identity.ToString();
    }
}

```

### To build and run the application

1. Press CTRL+SHIFT+B to build the application, and then CTRL+F5 to start.
2. Start a new workflow by clicking **New Game**. The version of the workflow is displayed under the status window and reflects the updated version from the associated `WorkflowIdentity`. Make a note of the `InstanceId` so you can view the tracking file for the workflow when it completes, and then enter guesses until the game is complete. Note how the user's guess is displayed in the information displayed in the status window based on the updates to the `WriteLine` activities.

### Please enter a number between 1 and 10

**5 is too high.**

### Please enter a number between 1 and 10

**3 is too high.**

### Please enter a number between 1 and 10

**1 is too low.**

### Please enter a number between 1 and 10

**Congratulations, you guessed the number in 4 turns.**

```

> [!NOTE]
> The updated text from the `WriteLine` activities is displayed, but the output of the final `WriteLine`
activity that was added in this topic is not. That is because the status window is updated by the
`PersistableIdle` handler. Because the workflow completes and does not go idle after the final activity, the
`PersistableIdle` handler is not called. However, a similar message is displayed in the status window by the
`Completed` handler. If desired, code could be added to the `Completed` handler to extract the text from the
`StringWriter` and display it to the status window.

```

3. Open Windows Explorer and navigate to the **NumberGuessWorkflowHost\bin\debug** folder (or **bin\release** depending on your project settings) and open the tracking file using Notepad that corresponds to the completed workflow. If you did not make a note of the `InstanceId`, you can identify the correct tracking file by using the **Date modified** information in Windows Explorer.

**Please enter a number between 1 and 10 5 is too high. Please enter a number between 1 and 10 3 is too high. Please enter a number between 1 and 10 1 is too low. Please enter a number between 1 and 10 2 is correct. You guessed it in 4 turns.** The updated `WriteLine` output is contained within the tracking file, including the output of the `WriteLine` that was added in this topic.

4. Switch back to the number guessing application and select one of the workflows that was started before the updates were made. You can identify the version of the currently selected workflow by looking at the

version information that is displayed below the status window. Enter some guesses and note that the status updates match the `WriteLine` activity output from the previous version, and do not include the user's guess. That is because these workflows are using the previous workflow definition that does not have the `WriteLine` updates.

In the next step, [How to: Update the Definition of a Running Workflow Instance](#), the running `v1` workflow instances are updated so they contain the new functionality as the `v2` instances.

# How to: Update the Definition of a Running Workflow Instance

3/9/2019 • 32 minutes to read • [Edit Online](#)

Dynamic update provides a mechanism for workflow application developers to update the workflow definition of a persisted workflow instance. The required change can be to implement a bug fix, new requirements, or to accommodate unexpected changes. This step in the tutorial demonstrates how to use dynamic update to modify persisted instances of the `v1` number guessing workflow to match the new functionality introduced in [How to: Host Multiple Versions of a Workflow Side-by-Side](#).

## NOTE

To download a completed version or view a video walkthrough of the tutorial, see [Windows Workflow Foundation \(WF45\) - Getting Started Tutorial](#).

## In this topic

- [To create the CreateUpdateMaps project](#)
- [To update StateMachineNumberGuessWorkflow](#)
- [To update FlowchartNumberGuessWorkflow](#)
- [To update SequentialNumberGuessWorkflow](#)
- [To build and run the CreateUpdateMaps application](#)
- [To build the updated workflow assembly](#)
- [To update WorkflowVersionMap with the new versions](#)
- [To apply the dynamic updates](#)
- [To run the application with the updated workflows](#)
- [To enable starting previous versions of the workflows](#)

### To create the CreateUpdateMaps project

1. Right-click **WF45GettingStartedTutorial** in **Solution Explorer** and choose **Add, New Project**.
2. In the **Installed** node, select **Visual C#, Windows** (or **Visual Basic, Windows**).

## NOTE

Depending on which programming language is configured as the primary language in Visual Studio, the **Visual C#** or **Visual Basic** node may be under the **Other Languages** node in the **Installed** node.

Ensure that **.NET Framework 4.5** is selected in the .NET Framework version drop-down list. Select **Console Application** from the **Windows** list. Type **CreateUpdateMaps** into the **Name** box and click **OK**.

3. Right-click **CreateUpdateMaps** in **Solution Explorer** and choose **Add Reference**.

4. Select **Framework** from the **Assemblies** node in the **Add Reference** list. Type **System.Activities** into the **Search Assemblies** box to filter the assemblies and make the desired references easier to select.
5. Check the checkbox beside **System.Activities** from the **Search Results** list.
6. Type **Serialization** into the **Search Assemblies** box, and check the checkbox beside **System.Runtime.Serialization** from the **Search Results** list.
7. Type **System.Xaml** into the **Search Assemblies** box, and check the checkbox beside **System.Xaml** from the **Search Results** list.
8. Click **OK** to close **Reference Manager** and add the references.
9. Add the following `using` (or `Imports`) statements at the top of the file with the other `using` (or `Imports`) statements.

```
Imports System.Activities
Imports System.Activities.Statements
Imports System.Xaml
Imports System.Reflection
Imports System.IO
Imports System.Activities.XamlIntegration
Imports System.Activities.DynamicUpdate
Imports System.Runtime.Serialization
Imports Microsoft.VisualBasic.Activities
```

```
using System.Activities;
using System.Activities.Statements;
using System.IO;
using System.Xaml;
using System.Reflection;
using System.Activities.XamlIntegration;
using System.Activities.DynamicUpdate;
using System.Runtime.Serialization;
using Microsoft.CSharp.Activities;
```

10. Add the following two string members to the `Program` class (or `Module1`).

```
Const mapPath = "...\\..\\..\\"PreviousVersions"
Const definitionPath = "...\\..\\..\\"NumberGuessWorkflowActivities_du"
```

```
const string mapPath = @"...\\..\\..\\"PreviousVersions";
const string definitionPath = @"...\\..\\..\\"NumberGuessWorkflowActivities_du";
```

11. Add the following `StartUpdate` method to the `Program` class (or `Module1`). This method loads up the specified xaml workflow definition into an `ActivityBuilder`, and then calls `DynamicUpdate.PrepareForUpdate`. `PrepareForUpdate` makes a copy of the workflow definition inside the `ActivityBuilder`. After the workflow definition is modified, this copy is used along with the modified workflow definition to create the update map.

```

Private Function StartUpdate(name As String) As ActivityBuilder
    'Create the XamlXmlReaderSettings.
    Dim readerSettings As XamlReaderSettings = New XamlXmlReaderSettings()
    'In the XAML the "local" namespace refers to artifacts that come from
    'the same project as the XAML. When loading XAML if the currently executing
    'assembly is not the same assembly that was referred to as "local" in the XAML
    'LocalAssembly must be set to the assembly containing the artifacts.
    'Assembly.LoadFile requires an absolute path so convert this relative path
    'to an absolute path.
    readerSettings.LocalAssembly = Assembly.LoadFile(
        Path.GetFullPath(Path.Combine(mapPath, "NumberGuessWorkflowActivities_v1.dll")))

    Dim fullPath As String = Path.Combine(definitionPath, name)
    Dim xamlReader As XamlXmlReader = New XamlXmlReader(fullPath, readerSettings)

    'Load the workflow definition into an ActivityBuilder.
    Dim wf As ActivityBuilder = XamlServices.Load(
        ActivityXamlServices.CreateBuilderReader(xamlReader))

    'PrepareForUpdate makes a copy of the workflow definition in the
    'ActivityBuilder that is used for comparison when the update
    'map is created.
    DynamicUpdateServices.PrepareForUpdate(wf)

    Return wf
End Function

```

```

private static ActivityBuilder StartUpdate(string name)
{
    // Create the XamlXmlReaderSettings.
    XamlXmlReaderSettings readerSettings = new XamlXmlReaderSettings()
    {
        // In the XAML the "local" namespace refers to artifacts that come from
        // the same project as the XAML. When loading XAML if the currently executing
        // assembly is not the same assembly that was referred to as "local" in the XAML
        // LocalAssembly must be set to the assembly containing the artifacts.
        // Assembly.LoadFile requires an absolute path so convert this relative path
        // to an absolute path.
        LocalAssembly = Assembly.LoadFile(
            Path.GetFullPath(Path.Combine(mapPath, "NumberGuessWorkflowActivities_v1.dll")))
    };

    string path = Path.Combine(definitionPath, name);
    XamlXmlReader xamlReader = new XamlXmlReader(path, readerSettings);

    // Load the workflow definition into an ActivityBuilder.
    ActivityBuilder wf = XamlServices.Load(
        ActivityXamlServices.CreateBuilderReader(xamlReader))
        as ActivityBuilder;

    // PrepareForUpdate makes a copy of the workflow definition in the
    // ActivityBuilder that is used for comparison when the update
    // map is created.
    DynamicUpdateServices.PrepareForUpdate(wf);

    return wf;
}

```

12. Next, add the following `CreateUpdateMethod` to the `Program` class (or `Module1`). This creates a dynamic update map by calling `DynamicUpdateServices.CreateUpdateMap`, and then saves the update map using the specified name. This update map contains the information needed by the workflow runtime to update a persisted workflow instance that was started using the original workflow definition contained in the `ActivityBuilder` so that it completes using the updated workflow definition.

```

Private Sub CreateUpdateMaps(wf As ActivityBuilder, name As String)
    'Create the UpdateMap.
    Dim map As DynamicUpdateMap =
        DynamicUpdateServices.CreateUpdateMap(wf)

    'Serialize it to a file.
    Dim mapFullPath As String = Path.Combine(mapPath, name)
    Dim sz AsDataContractSerializer = New DataContractSerializer(GetType(DynamicUpdateMap))
    Using fs As FileStream = File.Open(mapFullPath, FileMode.Create)
        sz.WriteObject(fs, map)
    End Using
End Sub

```

```

private static void CreateUpdateMaps(ActivityBuilder wf, string name)
{
    // Create the UpdateMap.
    DynamicUpdateMap map =
        DynamicUpdateServices.CreateUpdateMap(wf);

    // Serialize it to a file.
    string path = Path.Combine(mapPath, name);
    DataContractSerializer sz = new DataContractSerializer(typeof(DynamicUpdateMap));
    using (FileStream fs = System.IO.File.Open(path, FileMode.Create))
    {
        sz.WriteObject(fs, map);
    }
}

```

13. Add the following `SaveUpdatedDefinition` method to the `Program` class (or `Module1`). This method saves the updated workflow definition once the update map is created.

```

Private Sub SaveUpdatedDefinition(wf As ActivityBuilder, name As String)
    Dim xamlPath As String = Path.Combine(definitionPath, name)
    Dim sw As StreamWriter = File.CreateText(xamlPath)
    Dim xw As XamlWriter = ActivityXamlServices.CreateBuilderWriter(
        New XamlXmlWriter(sw, New XamlSchemaContext()))
    XamlServices.Save(xw, wf)
    sw.Close()
End Sub

```

```

private static void SaveUpdatedDefinition(ActivityBuilder wf, string name)
{
    string xamlPath = Path.Combine(definitionPath, name);
    StreamWriter sw = File.CreateText(xamlPath);
    XamlWriter xw = ActivityXamlServices.CreateBuilderWriter(
        new XamlXmlWriter(sw, new XamlSchemaContext()));
    XamlServices.Save(xw, wf);
    sw.Close();
}

```

## To update StateMachineNumberGuessWorkflow

1. Add a `CreateStateMachineUpdateMap` to the `Program` class (or `Module1`).

```

Private Sub CreateStateMachineUpdateMap()

End Sub

```

```
private static void CreateStateMachineUpdateMap()
{
}
```

2. Make a call to `StartUpdate` and then get a reference to the root `StateMachine` activity of the workflow.

```
Dim wf As ActivityBuilder = StartUpdate("StateMachineNumberGuessWorkflow.xaml")

'Get a reference to the root StateMachine activity.
Dim sm As StateMachine = wf.Implementation
```

```
ActivityBuilder wf = StartUpdate("StateMachineNumberGuessWorkflow.xaml");

// Get a reference to the root StateMachine activity.
StateMachine sm = wf.Implementation as StateMachine;
```

3. Next, update the expressions of the two `WriteLine` activities that display whether the user's guess is too high or too low so that they match the updates made in [How to: Host Multiple Versions of a Workflow Side-by-Side](#).

```
'Update the Text of the two WriteLine activities that write the
'results of the user's guess. They are contained in the workflow as the
'Then and Else action of the If activity in sm.States[1].Transitions[1].Action.
Dim guessLow As Statements.If = sm.States(1).Transitions(1).Action

'Update the "too low" message.
Dim tooLow As WriteLine = guessLow.Then
tooLow.Text = New VisualBasicValue(Of String)("Guess.ToString() & "" is too low.""")

'Update the "too high" message.
Dim tooHigh As WriteLine = guessLow.Else
tooHigh.Text = New VisualBasicValue(Of String)("Guess.ToString() & "" is too high."")
```

```
// Update the Text of the two WriteLine activities that write the
// results of the user's guess. They are contained in the workflow as the
// Then and Else action of the If activity in sm.States[1].Transitions[1].Action.
If guessLow = sm.States[1].Transitions[1].Action as If;

    // Update the "too low" message.
    WriteLine tooLow = guessLow.Then as WriteLine;
    tooLow.Text = new CSharpValue<string>("Guess.ToString() + \" is too low.\\"");

    // Update the "too high" message.
    WriteLine tooHigh = guessLow.Else as WriteLine;
    tooHigh.Text = new CSharpValue<string>("Guess.ToString() + \" is too high.\\"");
```

4. Next, add the new `WriteLine` activity that displays the closing message.

```

'Create the new WriteLine that displays the closing message.
Dim wl As New WriteLine() With
{
    .Text = New VisualBasicValue(Of String) _
        ("Guess.ToString() + "" is correct. You guessed it in "" & Turns.ToString() & "" turns."")
}

'Add it as the Action for the Guess Correct transition. The Guess Correct
'transition is the first transition of States[1]. The transitions are listed
'at the bottom of the State activity designer.
sm.States(1).Transitions(0).Action = wl

```

```

// Create the new WriteLine that displays the closing message.
WriteLine wl = new WriteLine
{
    Text = new CSharpValue<string>("Guess.ToString() + \" is correct. You guessed it in \" +
    Turns.ToString() + \" turns.\")"
};

// Add it as the Action for the Guess Correct transition. The Guess Correct
// transition is the first transition of States[1]. The transitions are listed
// at the bottom of the State activity designer.
sm.States[1].Transitions[0].Action = wl;

```

5. After the workflow is updated, call `CreateUpdateMaps` and `SaveUpdatedDefinition`. `CreateUpdateMaps` creates and saves the `DynamicUpdateMap`, and `SaveUpdatedDefinition` saves the updated workflow definition.

```

'Create the update map.
CreateUpdateMaps(wf, "StateMachineNumberGuessWorkflow.map")

'Save the updated workflow definition.
SaveUpdatedDefinition(wf, "StateMachineNumberGuessWorkflow_du.xaml")

```

```

// Create the update map.
CreateUpdateMaps(wf, "StateMachineNumberGuessWorkflow.map");

// Save the updated workflow definition.
SaveUpdatedDefinition(wf, "StateMachineNumberGuessWorkflow_du.xaml");

```

The following example is the completed `CreateStateMachineUpdateMap` method.

```

Private Sub CreateStateMachineUpdateMap()
    Dim wf As ActivityBuilder = StartUpdate("StateMachineNumberGuessWorkflow.xaml")

    'Get a reference to the root StateMachine activity.
    Dim sm As StateMachine = wf.Implementation

    'Update the Text of the two WriteLine activities that write the
    'results of the user's guess. They are contained in the workflow as the
    'Then and Else action of the If activity in sm.States[1].Transitions[1].Action.
    Dim guessLow As Statements.If = sm.States(1).Transitions(1).Action

    'Update the "too low" message.
    Dim tooLow As WriteLine = guessLow.Then
    tooLow.Text = New VisualBasicValue(Of String)("Guess.ToString() & "" is too low.""")

    'Update the "too high" message.
    Dim tooHigh As WriteLine = guessLow.Else
    tooHigh.Text = New VisualBasicValue(Of String)("Guess.ToString() & "" is too high.""")

    'Create the new WriteLine that displays the closing message.
    Dim wl As New WriteLine() With
    {
        .Text = New VisualBasicValue(Of String) _
            ("Guess.ToString() + "" is correct. You guessed it in "" & Turns.ToString() & "" turns."""")
    }

    'Add it as the Action for the Guess Correct transition. The Guess Correct
    'transition is the first transition of States[1]. The transitions are listed
    'at the bottom of the State activity designer.
    sm.States(1).Transitions(0).Action = wl

    'Create the update map.
    CreateUpdateMaps(wf, "StateMachineNumberGuessWorkflow.map")

    'Save the updated workflow definition.
    SaveUpdatedDefinition(wf, "StateMachineNumberGuessWorkflow_du.xaml")
End Sub

```

```

private static void CreateStateMachineUpdateMap()
{
    ActivityBuilder wf = StartUpdate("StateMachineNumberGuessWorkflow.xaml");

    // Get a reference to the root StateMachine activity.
    StateMachine sm = wf.Implementation as StateMachine;

    // Update the Text of the two WriteLine activities that write the
    // results of the user's guess. They are contained in the workflow as the
    // Then and Else action of the If activity in sm.States[1].Transitions[1].Action.
    If guessLow = sm.States[1].Transitions[1].Action as If;

    // Update the "too low" message.
    WriteLine tooLow = guessLow.Then as WriteLine;
    tooLow.Text = new CSharpValue<string>("Guess.ToString() + \" is too low.\"");

    // Update the "too high" message.
    WriteLine tooHigh = guessLow.Else as WriteLine;
    tooHigh.Text = new CSharpValue<string>("Guess.ToString() + \" is too high.\"");

    // Create the new WriteLine that displays the closing message.
    WriteLine wl = new WriteLine
    {
        Text = new CSharpValue<string>("Guess.ToString() + \" is correct. You guessed it in \" +
        Turns.ToString() + \" turns.\")"
    };

    // Add it as the Action for the Guess Correct transition. The Guess Correct
    // transition is the first transition of States[1]. The transitions are listed
    // at the bottom of the State activity designer.
    sm.States[1].Transitions[0].Action = wl;

    // Create the update map.
    CreateUpdateMaps(wf, "StateMachineNumberGuessWorkflow.map");

    // Save the updated workflow definition.
    SaveUpdatedDefinition(wf, "StateMachineNumberGuessWorkflow_du.xaml");
}

```

## To update FlowchartNumberGuessWorkflow

1. Add the following `CreateFlowchartUpdateMethod` to the `Program` class (or `Module1`). This method is similar to `CreateStateMachineUpdateMap`. It starts with a call to `StartUpdate`, updates the flowchart workflow definition, and finishes by saving the update map and the updated workflow definition.

```

Private Sub CreateFlowchartUpdateMap()
    Dim wf As ActivityBuilder = StartUpdate("FlowchartNumberGuessWorkflow.xaml")

    'Get a reference to the root Flowchart activity.
    Dim fc As Flowchart = wf.Implementation

    'Update the Text of the two WriteLine activities that write the
    'results of the user's guess. They are contained in the workflow as the
    'True and False action of the "Guess < Target" FlowDecision, which is
    'Nodes[4].
    Dim guessLow As FlowDecision = fc.Nodes(4)

    'Update the "too low" message.
    Dim trueStep As FlowStep = guessLow.True
    Dim tooLow As WriteLine = trueStep.Action
    tooLow.Text = New VisualBasicValue(Of String)("Guess.ToString() & "" is too low.""")

    'Update the "too high" message.
    Dim falseStep As FlowStep = guessLow.False
    Dim tooHigh As WriteLine = falseStep.Action
    tooHigh.Text = New VisualBasicValue(Of String)("Guess.ToString() & "" is too high.""")

    'Create the new WriteLine that displays the closing message.
    Dim wl As New WriteLine() With
    {
        .Text = New VisualBasicValue(Of String) _
            ("Guess.ToString() + "" is correct. You guessed it in "" & Turns.ToString() & "" turns."""")
    }

    'Create a FlowStep to hold the WriteLine.
    Dim closingStep As New FlowStep() With
    {
        .Action = wl
    }

    'Add this new FlowStep to the True action of the
    '"Guess = Guess" FlowDecision
    Dim guessCorrect As FlowDecision = fc.Nodes(3)
    guessCorrect.True = closingStep

    'Add the new FlowStep to the Nodes collection.
    'If closingStep was replacing an existing node then
    'we would need to remove that Step from the collection.
    'In this example there was no existing True step to remove.
    fc.Nodes.Add(closingStep)

    'Create the update map.
    CreateUpdateMaps(wf, "FlowchartNumberGuessWorkflow.map")

    'Save the updated workflow definition.
    SaveUpdatedDefinition(wf, "FlowchartNumberGuessWorkflow_du.xaml")
End Sub

```

```

private static void CreateFlowchartUpdateMap()
{
    ActivityBuilder wf = StartUpdate("FlowchartNumberGuessWorkflow.xaml");

    // Get a reference to the root Flowchart activity.
    Flowchart fc = wf.Implementation as Flowchart;

    // Update the Text of the two WriteLine activities that write the
    // results of the user's guess. They are contained in the workflow as the
    // True and False action of the "Guess < Target" FlowDecision, which is
    // Nodes[4].
    FlowDecision guessLow = fc.Nodes[4] as FlowDecision;

    // Update the "too low" message.
    FlowStep trueStep = guessLow.True as FlowStep;
    WriteLine tooLow = trueStep.Action as WriteLine;
    tooLow.Text = new CSharpValue<string>("Guess.ToString() + \" is too low.\"");

    // Update the "too high" message.
    FlowStep falseStep = guessLow.False as FlowStep;
    WriteLine tooHigh = falseStep.Action as WriteLine;
    tooHigh.Text = new CSharpValue<string>("Guess.ToString() + \" is too high.\"");

    // Add the new WriteLine that displays the closing message.
    WriteLine wl = new WriteLine
    {
        Text = new CSharpValue<string>("Guess.ToString() + \" is correct. You guessed it in \" +
        Turns.ToString() + \" turns.\"")
    };

    // Create a FlowStep to hold the WriteLine.
    FlowStep closingStep = new FlowStep
    {
        Action = wl
    };

    // Add this new FlowStep to the True action of the
    // "Guess == Guess" FlowDecision
    FlowDecision guessCorrect = fc.Nodes[3] as FlowDecision;
    guessCorrect.True = closingStep;

    // Add the new FlowStep to the Nodes collection.
    // If closingStep was replacing an existing node then
    // we would need to remove that Step from the collection.
    // In this example there was no existing True step to remove.
    fc.Nodes.Add(closingStep);

    // Create the update map.
    CreateUpdateMaps(wf, "FlowchartNumberGuessWorkflow.map");

    // Save the updated workflow definition.
    SaveUpdatedDefinition(wf, "FlowchartNumberGuessWorkflow_du.xaml");
}

```

## To update SequentialNumberGuessWorkflow

1. Add the following `CreateSequentialUpdateMethod` to the `Program` class (or `Module1`). This method is similar to the other two methods. It starts with a call to `StartUpdate`, updates the sequential workflow definition, and finishes by saving the update map and the updated workflow definition.

```

Private Sub CreateSequentialUpdateMap()
    Dim wf As ActivityBuilder = StartUpdate("SequentialNumberGuessWorkflow.xaml")

    'Get a reference to the root activity in the workflow.
    Dim rootSequence As Sequence = wf.Implementation

    'Update the Text of the two WriteLine activities that write the
    'results of the user's guess. They are contained in the workflow as the
    'Then and Else action of the "Guess < Target" If activity.
    'Sequence[1]->DoWhile->Body->Sequence[2]->If->Then->If
    Dim gameLoop As Statements.DoWhile = rootSequence.Activities(1)
    Dim gameBody As Sequence = gameLoop.Body
    Dim guessCorrect As Statements.If = gameBody.Activities(2)
    Dim guessLow As Statements.If = guessCorrect.Then
    Dim tooLow As WriteLine = guessLow.Then
    tooLow.Text = New VisualBasicValue(Of String)("Guess.ToString() & "" is too low."")
    Dim tooHigh As WriteLine = guessLow.Else
    tooHigh.Text = New VisualBasicValue(Of String)("Guess.ToString() & "" is too high."")

    'Create the new WriteLine that displays the closing message.
    Dim wl As New WriteLine() With
    {
        .Text = New VisualBasicValue(Of String) _
            ("Guess.ToString() + "" is correct. You guessed it in "" & Turns.ToString() & "" turns."")
    }

    'Insert it as the third activity in the root sequence
    rootSequence.Activities.Insert(2, wl)

    'Create the update map.
    CreateUpdateMaps(wf, "SequentialNumberGuessWorkflow.map")

    'Save the updated workflow definition.
    SaveUpdatedDefinition(wf, "SequentialNumberGuessWorkflow_du.xaml")
End Sub

```

```

private static void CreateSequentialUpdateMap()
{
    ActivityBuilder wf = StartUpdate("SequentialNumberGuessWorkflow.xaml");

    // Get a reference to the root activity in the workflow.
    Sequence rootSequence = wf.Implementation as Sequence;

    // Update the Text of the two WriteLine activities that write the
    // results of the user's guess. They are contained in the workflow as the
    // Then and Else action of the "Guess < Target" If activity.
    // Sequence[1]->DoWhile->Body->Sequence[2]->If->Then->If
    DoWhile gameLoop = rootSequence.Activities[1] as DoWhile;
    Sequence gameBody = gameLoop.Body as Sequence;
    If guessCorrect = gameBody.Activities[2] as If;
    If guessLow = guessCorrect.Then as If;
    WriteLine tooLow = guessLow.Then as WriteLine;
    tooLow.Text = new CSharpValue<string>("Guess.ToString() + \" is too low.\\"");
    WriteLine tooHigh = guessLow.Else as WriteLine;
    tooHigh.Text = new CSharpValue<string>("Guess.ToString() + \" is too high.\\"");

    // Add the new WriteLine that displays the closing message.
    WriteLine wl = new WriteLine
    {
        Text = new CSharpValue<string>("Guess.ToString() + \" is correct. You guessed it in \" +
        Turns.ToString() + \" turns.\")"
    };

    // Insert it as the third activity in the root sequence
    rootSequence.Activities.Insert(2, wl);

    // Create the update map.
    CreateUpdateMaps(wf, "SequentialNumberGuessWorkflow.map");

    // Save the updated workflow definition.
    SaveUpdatedDefinition(wf, "SequentialNumberGuessWorkflow_du.xaml");
}

```

## To build and run the **CreateUpdateMaps** application

1. Update the `Main` method and add the following three method calls. These methods are added in the following sections. Each method updates the corresponding number guess workflow and creates a `DynamicUpdateMap` that describes the updates.

```

Sub Main()
    'Create the update maps for the changes needed to the v1 activities
    'so they match the v2 activities.
    CreateSequentialUpdateMap()
    CreateFlowchartUpdateMap()
    CreateStateMachineUpdateMap()
End Sub

```

```

static void Main(string[] args)
{
    // Create the update maps for the changes needed to the v1 activities
    // so they match the v2 activities.
    CreateSequentialUpdateMap();
    CreateFlowchartUpdateMap();
    CreateStateMachineUpdateMap();
}

```

2. Right-click **CreateUpdateMaps** in **Solution Explorer** and choose **Set as StartUp Project**.

3. Press CTRL+SHIFT+B to build the solution, and then CTRL+F5 to run the `CreateUpdateMaps` application.

**NOTE**

The `CreateUpdateMaps` application does not display any status information while running, but if you look in the **NumberGuessWorkflowActivities\_du** folder and the **PreviousVersions** folder you will see the updated workflow definition files and the update maps.

Once the update maps are created and the workflow definitions updated, the next step is to build an updated workflow assembly containing the updated definitions.

**To build the updated workflow assembly**

1. Open a second instance of Visual Studio 2012.
2. Choose **Open, Project/Solution** from the **File** menu.
3. Navigate to the **NumberGuessWorkflowActivities\_du** folder you created in [How to: Host Multiple Versions of a Workflow Side-by-Side](#), select **NumberGuessWorkflowActivities.csproj** (or **vbproj**), and click **Open**.
4. In **Solution Explorer**, right click **SequentialNumberGuessWorkflow.xaml** and choose **Exclude From Project**. Do the same thing for **FlowchartNumberGuessWorkflow.xaml** and **StateMachineNumberGuessWorkflow.xaml**. This step removes the previous versions of the workflow definitions from the project.
5. Choose **Add Existing Item** from the **Project** menu.
6. Navigate to the **NumberGuessWorkflowActivities\_du** folder you created in [How to: Host Multiple Versions of a Workflow Side-by-Side](#).
7. Choose **XAML Files (\*.xaml;\*.xoml)** from the **Files of type** drop-down list.
8. Select **SequentialNumberGuessWorkflow\_du.xaml**, **FlowchartNumberGuessWorkflow\_du.xaml**, and **StateMachineNumberGuessWorkflow\_du.xaml** and click **Add**.

**NOTE**

CTRL+Click to select multiple items at a time.

This step adds the updated versions of the workflow definitions to the project.

9. Press CTRL+SHIFT+B to build the project.
10. Choose **Close Solution** from the **File** menu. A solution file for the project is not required, so click **No** to close Visual Studio without saving a solution file. Choose **Exit** from the **File** menu to close Visual Studio.
11. Open Windows Explorer and navigate to the **NumberGuessWorkflowActivities\_du\bin\Debug** folder (or **bin\Release** depending on your project settings).
12. Rename **NumberGuessWorkflowActivities.dll** to **NumberGuessWorkflowActivities\_v15.dll**, and copy it to the **PreviousVersions** folder you created in [How to: Host Multiple Versions of a Workflow Side-by-Side](#).

**To update WorkflowVersionMap with the new versions**

1. Switch back to the initial instance of Visual Studio 2012.
2. Double-click **WorkflowVersionMap.cs** (or **WorkflowVersionMap.vb**) under the

**NumberGuessWorkflowHost** project to open it.

3. Add three new workflow identities just below the six existing workflow identity declarations. In this tutorial, `1.5.0.0` is used as the `WorkflowIdentity.Version` for the dynamic update identities. These new `v15` workflow identities will be used provide the correct workflow definition for the dynamically updated persisted workflow instances.

```
'Current version identities.  
Public StateMachineNumberGuessIdentity As WorkflowIdentity  
Public FlowchartNumberGuessIdentity As WorkflowIdentity  
Public SequentialNumberGuessIdentity As WorkflowIdentity  
  
'v1 identities.  
Public StateMachineNumberGuessIdentity_v1 As WorkflowIdentity  
Public FlowchartNumberGuessIdentity_v1 As WorkflowIdentity  
Public SequentialNumberGuessIdentity_v1 As WorkflowIdentity  
  
'v1.5 (Dynamic Update) identities.  
Public StateMachineNumberGuessIdentity_v15 As WorkflowIdentity  
Public FlowchartNumberGuessIdentity_v15 As WorkflowIdentity  
Public SequentialNumberGuessIdentity_v15 As WorkflowIdentity
```

```
// Current version identities.  
static public WorkflowIdentity StateMachineNumberGuessIdentity;  
static public WorkflowIdentity FlowchartNumberGuessIdentity;  
static public WorkflowIdentity SequentialNumberGuessIdentity;  
  
// v1 identities.  
static public WorkflowIdentity StateMachineNumberGuessIdentity_v1;  
static public WorkflowIdentity FlowchartNumberGuessIdentity_v1;  
static public WorkflowIdentity SequentialNumberGuessIdentity_v1;  
  
// v1.5 (Dynamic Update) identities.  
static public WorkflowIdentity StateMachineNumberGuessIdentity_v15;  
static public WorkflowIdentity FlowchartNumberGuessIdentity_v15;  
static public WorkflowIdentity SequentialNumberGuessIdentity_v15;
```

4. Add the following code at the end of the constructor. This code initializes the dynamic update workflow identities, loads the corresponding workflow definitions, and adds them to the workflow version dictionary.

```

'Initialize the dynamic update workflow identities.
StateMachineNumberGuessIdentity_v15 = New WorkflowIdentity With
{
    .Name = "StateMachineNumberGuessWorkflow",
    .Version = New Version(1, 5, 0, 0)
}

FlowchartNumberGuessIdentity_v15 = New WorkflowIdentity With
{
    .Name = "FlowchartNumberGuessWorkflow",
    .Version = New Version(1, 5, 0, 0)
}

SequentialNumberGuessIdentity_v15 = New WorkflowIdentity With
{
    .Name = "SequentialNumberGuessWorkflow",
    .Version = New Version(1, 5, 0, 0)
}

'Add the dynamic update workflow identities to the dictionary along with
'the corresponding workflow definitions loaded from the v15 assembly.
'Assembly.LoadFile requires an absolute path so convert this relative path
'to an absolute path.
Dim v15AssemblyPath As String = "..\..\..\PreviousVersions\NumberGuessWorkflowActivities_v15.dll"
v15AssemblyPath = Path.GetFullPath(v15AssemblyPath)
Dim v15Assembly As Assembly = Assembly.LoadFile(v15AssemblyPath)

map.Add(StateMachineNumberGuessIdentity_v15,
    v15Assembly.CreateInstance("NumberGuessWorkflowActivities.StateMachineNumberGuessWorkflow"))

map.Add(SequentialNumberGuessIdentity_v15,
    v15Assembly.CreateInstance("NumberGuessWorkflowActivities.SequentialNumberGuessWorkflow"))

map.Add(FlowchartNumberGuessIdentity_v15,
    v15Assembly.CreateInstance("NumberGuessWorkflowActivities.FlowchartNumberGuessWorkflow"))

```

```

// Initialize the dynamic update workflow identities.
StateMachineNumberGuessIdentity_v15 = new WorkflowIdentity
{
    Name = "StateMachineNumberGuessWorkflow",
    Version = new Version(1, 5, 0, 0)
};

FlowchartNumberGuessIdentity_v15 = new WorkflowIdentity
{
    Name = "FlowchartNumberGuessWorkflow",
    Version = new Version(1, 5, 0, 0)
};

SequentialNumberGuessIdentity_v15 = new WorkflowIdentity
{
    Name = "SequentialNumberGuessWorkflow",
    Version = new Version(1, 5, 0, 0)
};

// Add the dynamic update workflow identities to the dictionary along with
// the corresponding workflow definitions loaded from the v15 assembly.
// Assembly.LoadFile requires an absolute path so convert this relative path
// to an absolute path.
string v15AssemblyPath = @"..\..\..\..\PreviousVersions\NumberGuessWorkflowActivities_v15.dll";
v15AssemblyPath = Path.GetFullPath(v15AssemblyPath);
Assembly v15Assembly = Assembly.LoadFile(v15AssemblyPath);

map.Add(StateMachineNumberGuessIdentity_v15,
    v15Assembly.CreateInstance("NumberGuessWorkflowActivities.StateMachineNumberGuessWorkflow") as
Activity);

map.Add(SequentialNumberGuessIdentity_v15,
    v15Assembly.CreateInstance("NumberGuessWorkflowActivities.SequentialNumberGuessWorkflow") as
Activity);

map.Add(FlowchartNumberGuessIdentity_v15,
    v15Assembly.CreateInstance("NumberGuessWorkflowActivities.FlowchartNumberGuessWorkflow") as
Activity);

```

The following example is the completed `WorkflowVersionMap` class.

```

Public Module WorkflowVersionMap
    Dim map As Dictionary(Of WorkflowIdentity, Activity)

    'Current version identities.
    Public StateMachineNumberGuessIdentity As WorkflowIdentity
    Public FlowchartNumberGuessIdentity As WorkflowIdentity
    Public SequentialNumberGuessIdentity As WorkflowIdentity

    'v1 identities.
    Public StateMachineNumberGuessIdentity_v1 As WorkflowIdentity
    Public FlowchartNumberGuessIdentity_v1 As WorkflowIdentity
    Public SequentialNumberGuessIdentity_v1 As WorkflowIdentity

    'v1.5 (Dynamic Update) identities.
    Public StateMachineNumberGuessIdentity_v15 As WorkflowIdentity
    Public FlowchartNumberGuessIdentity_v15 As WorkflowIdentity
    Public SequentialNumberGuessIdentity_v15 As WorkflowIdentity

    Sub New()
        map = New Dictionary(Of WorkflowIdentity, Activity)

        'Add the current workflow version identities.
        StateMachineNumberGuessIdentity = New WorkflowIdentity With
        {
            .Name = "StateMachineNumberGuessWorkflow",

```

```

        .Version = New Version(2, 0, 0, 0)
    }

FlowchartNumberGuessIdentity = New WorkflowIdentity With
{
    .Name = "FlowchartNumberGuessWorkflow",
    .Version = New Version(2, 0, 0, 0)
}

SequentialNumberGuessIdentity = New WorkflowIdentity With
{
    .Name = "SequentialNumberGuessWorkflow",
    .Version = New Version(2, 0, 0, 0)
}

map.Add(StateMachineNumberGuessIdentity, New StateMachineNumberGuessWorkflow())
map.Add(FlowchartNumberGuessIdentity, New FlowchartNumberGuessWorkflow())
map.Add(SequentialNumberGuessIdentity, New SequentialNumberGuessWorkflow())

'Initialize the previous workflow version identities.
StateMachineNumberGuessIdentity_v1 = New WorkflowIdentity With
{
    .Name = "StateMachineNumberGuessWorkflow",
    .Version = New Version(1, 0, 0, 0)
}

FlowchartNumberGuessIdentity_v1 = New WorkflowIdentity With
{
    .Name = "FlowchartNumberGuessWorkflow",
    .Version = New Version(1, 0, 0, 0)
}

SequentialNumberGuessIdentity_v1 = New WorkflowIdentity With
{
    .Name = "SequentialNumberGuessWorkflow",
    .Version = New Version(1, 0, 0, 0)
}

'Add the previous version workflow identities to the dictionary along with
'the corresponding workflow definitions loaded from the v1 assembly.
'Assembly.LoadFile requires an absolute path so convert this relative path
'to an absolute path.
Dim v1AssemblyPath As String = "...\\..\\..\\" & PreviousVersions & "\\NumberGuessWorkflowActivities_v1.dll"
v1AssemblyPath = Path.GetFullPath(v1AssemblyPath)
Dim v1Assembly As Assembly = Assembly.LoadFile(v1AssemblyPath)

map.Add(StateMachineNumberGuessIdentity_v1,
    v1Assembly.CreateInstance("NumberGuessWorkflowActivities.StateMachineNumberGuessWorkflow"))

map.Add(SequentialNumberGuessIdentity_v1,
    v1Assembly.CreateInstance("NumberGuessWorkflowActivities.SequentialNumberGuessWorkflow"))

map.Add(FlowchartNumberGuessIdentity_v1,
    v1Assembly.CreateInstance("NumberGuessWorkflowActivities.FlowchartNumberGuessWorkflow"))

'Initialize the dynamic update workflow identities.
StateMachineNumberGuessIdentity_v15 = New WorkflowIdentity With
{
    .Name = "StateMachineNumberGuessWorkflow",
    .Version = New Version(1, 5, 0, 0)
}

FlowchartNumberGuessIdentity_v15 = New WorkflowIdentity With
{
    .Name = "FlowchartNumberGuessWorkflow",
    .Version = New Version(1, 5, 0, 0)
}

SequentialNumberGuessIdentity_v15 = New WorkflowIdentity With

```

```

    {
        .Name = "SequentialNumberGuessWorkflow",
        .Version = New Version(1, 5, 0, 0)
    }

    'Add the dynamic update workflow identities to the dictionary along with
    'the corresponding workflow definitions loaded from the v15 assembly.
    'Assembly.LoadFile requires an absolute path so convert this relative path
    'to an absolute path.
    Dim v15AssemblyPath As String =
    "...\\..\\..\\PreviousVersions\\NumberGuessWorkflowActivities_v15.dll"
    v15AssemblyPath = Path.GetFullPath(v15AssemblyPath)
    Dim v15Assembly As Assembly = Assembly.LoadFile(v15AssemblyPath)

    map.Add(StateMachineNumberGuessIdentity_v15,
    v15Assembly.CreateInstance("NumberGuessWorkflowActivities.StateMachineNumberGuessWorkflow"))

    map.Add(SequentialNumberGuessIdentity_v15,
    v15Assembly.CreateInstance("NumberGuessWorkflowActivities.SequentialNumberGuessWorkflow"))

    map.Add(FlowchartNumberGuessIdentity_v15,
    v15Assembly.CreateInstance("NumberGuessWorkflowActivities.FlowchartNumberGuessWorkflow"))
End Sub

Public Function GetWorkflowDefinition(identity As WorkflowIdentity) As Activity
    Return map(identity)
End Function

Public Function GetIdentityDescription(identity As WorkflowIdentity) As String
    Return identity.ToString()
End Function
End Module

```

```

public static class WorkflowVersionMap
{
    static Dictionary<WorkflowIdentity, Activity> map;

    // Current version identities.
    static public WorkflowIdentity StateMachineNumberGuessIdentity;
    static public WorkflowIdentity FlowchartNumberGuessIdentity;
    static public WorkflowIdentity SequentialNumberGuessIdentity;

    // v1 identities.
    static public WorkflowIdentity StateMachineNumberGuessIdentity_v1;
    static public WorkflowIdentity FlowchartNumberGuessIdentity_v1;
    static public WorkflowIdentity SequentialNumberGuessIdentity_v1;

    // v1.5 (Dynamic Update) identities.
    static public WorkflowIdentity StateMachineNumberGuessIdentity_v15;
    static public WorkflowIdentity FlowchartNumberGuessIdentity_v15;
    static public WorkflowIdentity SequentialNumberGuessIdentity_v15;

    static WorkflowVersionMap()
    {
        map = new Dictionary<WorkflowIdentity, Activity>();

        // Add the current workflow version identities.
        StateMachineNumberGuessIdentity = new WorkflowIdentity
        {
            Name = "StateMachineNumberGuessWorkflow",
            // Version = new Version(1, 0, 0, 0),
            Version = new Version(2, 0, 0, 0)
        };

        FlowchartNumberGuessIdentity = new WorkflowIdentity
        {

```

```

        Name = "FlowchartNumberGuessWorkflow",
        // Version = new Version(1, 0, 0, 0),
        Version = new Version(2, 0, 0, 0)
    };

    SequentialNumberGuessIdentity = new WorkflowIdentity
    {
        Name = "SequentialNumberGuessWorkflow",
        // Version = new Version(1, 0, 0, 0),
        Version = new Version(2, 0, 0, 0)
    };

    map.Add(StateMachineNumberGuessIdentity, new StateMachineNumberGuessWorkflow());
    map.Add(FlowchartNumberGuessIdentity, new FlowchartNumberGuessWorkflow());
    map.Add(SequentialNumberGuessIdentity, new SequentialNumberGuessWorkflow());

    // Initialize the previous workflow version identities.
    StateMachineNumberGuessIdentity_v1 = new WorkflowIdentity
    {
        Name = "StateMachineNumberGuessWorkflow",
        Version = new Version(1, 0, 0, 0)
    };

    FlowchartNumberGuessIdentity_v1 = new WorkflowIdentity
    {
        Name = "FlowchartNumberGuessWorkflow",
        Version = new Version(1, 0, 0, 0)
    };

    SequentialNumberGuessIdentity_v1 = new WorkflowIdentity
    {
        Name = "SequentialNumberGuessWorkflow",
        Version = new Version(1, 0, 0, 0)
    };

    // Add the previous version workflow identities to the dictionary along with
    // the corresponding workflow definitions loaded from the v1 assembly.
    // Assembly.LoadFile requires an absolute path so convert this relative path
    // to an absolute path.
    string v1AssemblyPath = @"..\..\..\PreviousVersions\NumberGuessWorkflowActivities_v1.dll";
    v1AssemblyPath = Path.GetFullPath(v1AssemblyPath);
    Assembly v1Assembly = Assembly.LoadFile(v1AssemblyPath);

    map.Add(StateMachineNumberGuessIdentity_v1,
        v1Assembly.CreateInstance("NumberGuessWorkflowActivities.StateMachineNumberGuessWorkflow")
    as Activity);

    map.Add(SequentialNumberGuessIdentity_v1,
        v1Assembly.CreateInstance("NumberGuessWorkflowActivities.SequentialNumberGuessWorkflow") as
    Activity);

    map.Add(FlowchartNumberGuessIdentity_v1,
        v1Assembly.CreateInstance("NumberGuessWorkflowActivities.FlowchartNumberGuessWorkflow") as
    Activity);

    // Initialize the dynamic update workflow identities.
    StateMachineNumberGuessIdentity_v15 = new WorkflowIdentity
    {
        Name = "StateMachineNumberGuessWorkflow",
        Version = new Version(1, 5, 0, 0)
    };

    FlowchartNumberGuessIdentity_v15 = new WorkflowIdentity
    {
        Name = "FlowchartNumberGuessWorkflow",
        Version = new Version(1, 5, 0, 0)
    };

    SequentialNumberGuessIdentity_v15 = new WorkflowIdentity

```

```

    {
        Name = "SequentialNumberGuessWorkflow",
        Version = new Version(1, 5, 0, 0)
    };

    // Add the dynamic update workflow identities to the dictionary along with
    // the corresponding workflow definitions loaded from the v15 assembly.
    // Assembly.LoadFile requires an absolute path so convert this relative path
    // to an absolute path.
    string v15AssemblyPath = @"..\..\..\..\PreviousVersions\NumberGuessWorkflowActivities_v15.dll";
    v15AssemblyPath = Path.GetFullPath(v15AssemblyPath);
    Assembly v15Assembly = Assembly.LoadFile(v15AssemblyPath);

    map.Add(StateMachineNumberGuessIdentity_v15,
        v15Assembly.CreateInstance("NumberGuessWorkflowActivities.StateMachineNumberGuessWorkflow")
    as Activity);

    map.Add(SequentialNumberGuessIdentity_v15,
        v15Assembly.CreateInstance("NumberGuessWorkflowActivities.SequentialNumberGuessWorkflow")
    as Activity);

    map.Add(FlowchartNumberGuessIdentity_v15,
        v15Assembly.CreateInstance("NumberGuessWorkflowActivities.FlowchartNumberGuessWorkflow") as
    Activity);
}

public static Activity GetWorkflowDefinition(WorkflowIdentity identity)
{
    return map[identity];
}

public static string GetIdentityDescription(WorkflowIdentity identity)
{
    return identity.ToString();
}
}

```

5. Press CTRL+SHIFT+B to build the project.

#### To apply the dynamic updates

1. Right-click **WF45GettingStartedTutorial** in **Solution Explorer** and choose **Add, New Project**.
2. In the **Installed** node, select **Visual C#, Windows** (or **Visual Basic, Windows**).

##### NOTE

Depending on which programming language is configured as the primary language in Visual Studio, the **Visual C#** or **Visual Basic** node may be under the **Other Languages** node in the **Installed** node.

- Ensure that **.NET Framework 4.5** is selected in the .NET Framework version drop-down list. Select **Console Application** from the **Windows** list. Type **ApplyDynamicUpdate** into the **Name** box and click **OK**.
3. Right-click **ApplyDynamicUpdate** in **Solution Explorer** and choose **Add Reference**.
  4. Click **Solution** and check the box next to **NumberGuessWorkflowHost**. This reference is needed so that **ApplyDynamicUpdate** can use the **NumberGuessWorkflowHost.WorkflowVersionMap** class.
  5. Select **Framework** from the **Assemblies** node in the **Add Reference** list. Type **System.Activities** into the **Search Assemblies** box. This will filter the assemblies and make the desired references easier to select.
  6. Check the checkbox beside **System.Activities** from the **Search Results** list.

7. Type **Serialization** into the **Search Assemblies** box, and check the checkbox beside **System.Runtime.Serialization** from the **Search Results** list.
8. Type **DurableInstancing** into the **Search Assemblies** box, and check the checkbox beside **System.Activities.DurableInstancing** and **System.Runtime.DurableInstancing** from the **Search Results** list.
9. Click **OK** to close **Reference Manager** and add the references.
10. Right-click **ApplyDynamicUpdate** in Solution Explorer and choose **Add, Class**. Type `DynamicUpdateInfo` into the **Name** box and click **Add**.
11. Add the following two members to the `DynamicUpdateInfo` class. The following example is the completed `DynamicUpdateInfo` class. This class contains information on the update map and new workflow identity used when a workflow instance is updated.

```
Public Class DynamicUpdateInfo
    Public updateMap As DynamicUpdateMap
    Public newIdentity As WorkflowIdentity
End Class
```

```
class DynamicUpdateInfo
{
    public DynamicUpdateMap updateMap;
    public WorkflowIdentity newIdentity;
}
```

12. Add the following `using` (or `Imports`) statements at the top of the file with the other `using` (or `Imports`) statements.

```
Imports System.Activities
Imports System.Activities.DynamicUpdate
```

```
using System.Activities;
using System.Activities.DynamicUpdate;
```

13. Double-click **Program.cs** (or **Module1.vb**) in Solution Explorer.

14. Add the following `using` (or `Imports`) statements at the top of the file with the other `using` (or `Imports`) statements.

```
Imports NumberGuessWorkflowHost
Imports System.Data.SqlClient
Imports System.Activities.DynamicUpdate
Imports System.IO
Imports System.Runtime.Serialization
Imports System.Activities
Imports System.Activities.DurableInstancing
```

```
using NumberGuessWorkflowHost;
using System.Data;
using System.Data.SqlClient;
using System.Activities;
using System.Activities.DynamicUpdate;
using System.IO;
using System.Runtime.Serialization;
using System.Activities.DurableInstancing;
```

15. Add the following connection string member to the `Program` class (or `Module1`).

```
Const connectionString = "Server=.\SQLEXPRESS;Initial Catalog=WF45GettingStartedTutorial;Integrated Security=SSPI"
```

```
const string connectionString = "Server=.\SQLEXPRESS;Initial Catalog=WF45GettingStartedTutorial;Integrated Security=SSPI";
```

**NOTE**

Depending on your edition of SQL Server, the connection string server name may be different.

16. Add the following `GetIDs` method to the `Program` class (or `Module1`). This method returns a list of persisted workflow instance ids.

```
Function GetIDs() As IList(Of Guid)
    Dim Ids As New List(Of Guid)
    Dim localCmd = _
        String.Format("Select [InstanceId] from [System.Activities.DurableInstancing].[Instances] Order By [CreationTime]")
    Using localCon = New SqlConnection(connectionString)
        Dim cmd As SqlCommand = localCon.CreateCommand()
        cmd.CommandText = localCmd
        localCon.Open()
        Using reader = cmd.ExecuteReader(CommandBehavior.CloseConnection)
            While reader.Read()
                'Get the InstanceId of the persisted Workflow
                Dim id As Guid = Guid.Parse(reader(0).ToString())

                'Add it to the list.
                Ids.Add(id)
            End While
        End Using
    End Using

    Return Ids
End Function
```

```

static IList<Guid> GetIds()
{
    List<Guid> Ids = new List<Guid>();
    string localCmd = string.Format("Select [InstanceId] from [System.Activities.DurableInstancing].[Instances] Order By [CreationTime]");
    using (SqlConnection localCon = new SqlConnection(connectionString))
    {
        SqlCommand cmd = localCon.CreateCommand();
        cmd.CommandText = localCmd;
        localCon.Open();
        using (SqlDataReader reader = cmd.ExecuteReader(CommandBehavior.CloseConnection))
        {
            while (reader.Read())
            {
                // Get the InstanceId of the persisted Workflow
                Guid id = Guid.Parse(reader[0].ToString());

                // Add it to the list.
                Ids.Add(id);
            }
        }
    }

    return Ids;
}

```

17. Add the following `LoadMap` method to the `Program` class (or `Module1`). This method creates a dictionary that maps `v1` workflow identities to the update maps and new workflow identities used to update the corresponding persisted workflow instances.

```

Function LoadMap(mapName As String) As DynamicUpdateMap
    Dim mapPath As String = Path.Combine("../..\..\..\PreviousVersions", mapName)

    Dim map As DynamicUpdateMap
    Using fs As FileStream = File.Open(mapPath, FileMode.Open)
        Dim serializer As DataContractSerializer = New
DataContractSerializer(GetType(DynamicUpdateMap))
        Dim updateMap = serializer.ReadObject(fs)
        If updateMap Is Nothing Then
            Throw New ApplicationException("DynamicUpdateMap is null.")
        End If

        map = updateMap
    End Using

    Return map
End Function

```

```

static DynamicUpdateMap LoadMap(string mapName)
{
    string path = Path.Combine(@"..\..\..\PreviousVersions", mapName);

    DynamicUpdateMap map;
    using (FileStream fs = File.Open(path, FileMode.Open))
    {
        DataContractSerializer serializer = new DataContractSerializer(typeof(DynamicUpdateMap));
        object updateMap = serializer.ReadObject(fs);
        if (updateMap == null)
        {
            throw new ApplicationException("DynamicUpdateMap is null.");
        }

        map = updateMap as DynamicUpdateMap;
    }

    return map;
}

```

18. Add the following `LoadMaps` method to the `Program` class (or `Module1`). This method loads the three update maps and creates a dictionary that maps `v1` workflow identities to the update maps.

```

Function LoadMaps() As IDictionary(Of WorkflowIdentity, DynamicUpdateInfo)
    'There are 3 update maps to describe the changes to update v1 workflows,
    'one for each of the 3 workflow types in the tutorial.
    Dim maps = New Dictionary(Of WorkflowIdentity, DynamicUpdateInfo)()

    Dim sequentialMap As DynamicUpdateMap = LoadMap("SequentialNumberGuessWorkflow.map")
    Dim sequentialInfo = New DynamicUpdateInfo With
    {
        .updateMap = sequentialMap,
        .newIdentity = WorkflowVersionMap.SequentialNumberGuessIdentity_v15
    }
    maps.Add(WorkflowVersionMap.SequentialNumberGuessIdentity_v1, sequentialInfo)

    Dim stateMap As DynamicUpdateMap = LoadMap("StateMachineNumberGuessWorkflow.map")
    Dim stateInfo = New DynamicUpdateInfo With
    {
        .updateMap = stateMap,
        .newIdentity = WorkflowVersionMap.StateMachineNumberGuessIdentity_v15
    }
    maps.Add(WorkflowVersionMap.StateMachineNumberGuessIdentity_v1, stateInfo)

    Dim flowchartMap As DynamicUpdateMap = LoadMap("FlowchartNumberGuessWorkflow.map")
    Dim flowchartInfo = New DynamicUpdateInfo With
    {
        .updateMap = flowchartMap,
        .newIdentity = WorkflowVersionMap.FlowchartNumberGuessIdentity_v15
    }
    maps.Add(WorkflowVersionMap.FlowchartNumberGuessIdentity_v1, flowchartInfo)

    Return maps
End Function

```

```

static IDictionary<WorkflowIdentity, DynamicUpdateInfo> LoadMaps()
{
    // There are 3 update maps to describe the changes to update v1 workflows,
    // one for each of the 3 workflow types in the tutorial.
    Dictionary<WorkflowIdentity, DynamicUpdateInfo> maps =
        new Dictionary<WorkflowIdentity, DynamicUpdateInfo>();

    DynamicUpdateMap sequentialMap = LoadMap("SequentialNumberGuessWorkflow.map");
    DynamicUpdateInfo sequentialInfo = new DynamicUpdateInfo
    {
        updateMap = sequentialMap,
        newIdentity = WorkflowVersionMap.SequentialNumberGuessIdentity_v15
    };
    maps.Add(WorkflowVersionMap.SequentialNumberGuessIdentity_v1, sequentialInfo);

    DynamicUpdateMap stateMap = LoadMap("StateMachineNumberGuessWorkflow.map");
    DynamicUpdateInfo stateInfo = new DynamicUpdateInfo
    {
        updateMap = stateMap,
        newIdentity = WorkflowVersionMap.StateMachineNumberGuessIdentity_v15
    };
    maps.Add(WorkflowVersionMap.StateMachineNumberGuessIdentity_v1, stateInfo);

    DynamicUpdateMap flowchartMap = LoadMap("FlowchartNumberGuessWorkflow.map");
    DynamicUpdateInfo flowchartInfo = new DynamicUpdateInfo
    {
        updateMap = flowchartMap,
        newIdentity = WorkflowVersionMap.FlowchartNumberGuessIdentity_v15
    };
    maps.Add(WorkflowVersionMap.FlowchartNumberGuessIdentity_v1, flowchartInfo);

    return maps;
}

```

19. Add the following code to `Main`. This code iterates the persisted workflow instances and examines each `WorkflowIdentity`. If the `WorkflowIdentity` maps to a `v1` workflow instance, a `WorkflowApplication` is configured with the updated workflow definition and an updated workflow identity. Next, `WorkflowApplication.Load` is called with the instance and the update map, which applies the dynamic update map. Once the update is applied, the updated instance is persisted with a call to `Unload`.

```

Dim store = New SqlWorkflowInstanceStore(connectionString)
WorkflowApplication.CreateDefaultInstanceOwner(store, Nothing, WorkflowIdentityFilter.Any)

Dim updateMaps As IDictionary(Of WorkflowIdentity, DynamicUpdateInfo) = LoadMaps()

For Each id As Guid In GetIds()
    'Get a proxy to the instance.
    Dim instance As WorkflowApplicationInstance = WorkflowApplication.GetInstance(id, store)

    Console.WriteLine("Inspecting: {0}", instance.DefinitionIdentity)

    'Only update v1 workflows.
    If Not instance.DefinitionIdentity Is Nothing AndAlso _
        instance.DefinitionIdentity.Version.Equals(New Version(1, 0, 0, 0)) Then

        Dim info As DynamicUpdateInfo = updateMaps(instance.DefinitionIdentity)

        'Associate the persisted WorkflowApplicationInstance with
        'a WorkflowApplication that is configured with the updated
        'definition and updated WorkflowIdentity.
        Dim wf As Activity = WorkflowVersionMap.GetWorkflowDefinition(info.newIdentity)
        Dim wfApp = New WorkflowApplication(wf, info.newIdentity)

        'Apply the Dynamic Update.
        wfApp.Load(instance, info.updateMap)

        'Persist the updated instance.
        wfApp.Unload()

        Console.WriteLine("Updated to: {0}", info.newIdentity)
    Else
        'Not updating this instance, so unload it.
        instance.Abandon()
    End If
Next

```

```

SqlWorkflowInstanceStore store = new SqlWorkflowInstanceStore(connectionString);
WorkflowApplication.CreateDefaultInstanceOwner(store, null, WorkflowIdentityFilter.Any);

IDictionary<WorkflowIdentity, DynamicUpdateInfo> updateMaps = LoadMaps();

foreach (Guid id in GetIds())
{
    // Get a proxy to the instance.
    WorkflowApplicationInstance instance =
        WorkflowApplication.GetInstance(id, store);

    Console.WriteLine("Inspecting: {0}", instance.DefinitionIdentity);

    // Only update v1 workflows.
    if (instance.DefinitionIdentity != null &&
        instance.DefinitionIdentity.Version.Equals(new Version(1, 0, 0, 0)))
    {
        DynamicUpdateInfo info = updateMaps[instance.DefinitionIdentity];

        // Associate the persisted WorkflowApplicationInstance with
        // a WorkflowApplication that is configured with the updated
        // definition and updated WorkflowIdentity.
        Activity wf = WorkflowVersionMap.GetWorkflowDefinition(info.newIdentity);
        WorkflowApplication wfApp =
            new WorkflowApplication(wf, info.newIdentity);

        // Apply the Dynamic Update.
        wfApp.Load(instance, info.updateMap);

        // Persist the updated instance.
        wfApp.Unload();

        Console.WriteLine("Updated to: {0}", info.newIdentity);
    }
    else
    {
        // Not updating this instance, so unload it.
        instance.Abandon();
    }
}

```

20. Right-click **ApplyDynamicUpdate** in **Solution Explorer** and choose **Set as StartUp Project**.
21. Press **CTRL+SHIFT+B** to build the solution, and then press **CTRL+F5** to run the **ApplyDynamicUpdate** application and update the persisted workflow instances. You should see output similar to the following. The version 1.0.0.0 workflows are updated to version 1.5.0.0, while the version 2.0.0.0 workflows are not updated.

**Inspecting: StateMachineNumberGuessWorkflow; Version=1.0.0.0**  
**Updated to: StateMachineNumberGuessWorkflow; Version=1.5.0.0**  
**Inspecting: StateMachineNumberGuessWorkflow; Version=1.0.0.0**  
**Updated to: StateMachineNumberGuessWorkflow; Version=1.5.0.0**  
**Inspecting: FlowchartNumberGuessWorkflow; Version=1.0.0.0**  
**Updated to: FlowchartNumberGuessWorkflow; Version=1.5.0.0**  
**Inspecting: FlowchartNumberGuessWorkflow; Version=1.0.0.0**  
**Updated to: FlowchartNumberGuessWorkflow; Version=1.5.0.0**  
**Inspecting: SequentialNumberGuessWorkflow; Version=1.0.0.0**  
**Updated to: SequentialNumberGuessWorkflow; Version=1.5.0.0**  
**Inspecting: SequentialNumberGuessWorkflow; Version=1.0.0.0**  
**Updated to: SequentialNumberGuessWorkflow; Version=1.5.0.0**  
**Inspecting: SequentialNumberGuessWorkflow; Version=1.0.0.0**

```
Updated to: SequentialNumberGuessWorkflow; Version=1.5.0.0
Inspecting: StateMachineNumberGuessWorkflow; Version=1.0.0.0
Updated to: StateMachineNumberGuessWorkflow; Version=1.5.0.0
Inspecting: FlowchartNumberGuessWorkflow; Version=1.0.0.0
Updated to: FlowchartNumberGuessWorkflow; Version=1.5.0.0
Inspecting: StateMachineNumberGuessWorkflow; Version=2.0.0.0
Inspecting: StateMachineNumberGuessWorkflow; Version=2.0.0.0
Inspecting: FlowchartNumberGuessWorkflow; Version=2.0.0.0
Inspecting: FlowchartNumberGuessWorkflow; Version=2.0.0.0
Inspecting: SequentialNumberGuessWorkflow; Version=2.0.0.0
Inspecting: SequentialNumberGuessWorkflow; Version=2.0.0.0
Press any key to continue ...
```

#### To run the application with the updated workflows

1. Right-click **NumberGuessWorkflowHost** in **Solution Explorer** and choose **Set as StartUp Project**.
2. Press CTRL+F5 to run the application.
3. Click **New Game** to start a new workflow and note the version information below the status window that indicates the workflow is a **v2** workflow.
4. Select one of the **v1** workflows you started at the beginning of the [How to: Host Multiple Versions of a Workflow Side-by-Side](#) topic. Note that the version information under the status window indicates that the workflow is a version **1.5.0.0** workflow. Note that there is no information indicated about previous guesses other than whether they were too high or too low.

**Please enter a number between 1 and 10**

**Your guess is too low.**

5. Make a note of the **InstanceId** and then enter guesses until the workflow completes. The status window displays information about the content of the guess because the **WriteLine** activities were updated by the dynamic update.

**Please enter a number between 1 and 10**

**Your guess is too low.**

**Please enter a number between 1 and 10**

**5 is too low.**

**Please enter a number between 1 and 10**

**7 is too high.**

**Please enter a number between 1 and 10**

**Congratulations, you guessed the number in 4 turns.**

6. Open Windows Explorer and navigate to the **NumberGuessWorkflowHost\bin\debug** folder (or **bin\release** depending on your project settings) and open the tracking file using Notepad that corresponds to the completed workflow. If you did not make a note of the **InstanceId** you may be able to identify the correct tracking file by using the **Date modified** information in Windows Explorer. The last line of the tracking information contains the output of the newly added **WriteLine** activity.

**Please enter a number between 1 and 10**

**Your guess is too low.**

**Please enter a number between 1 and 10**

**5 is too low.**

**Please enter a number between 1 and 10**

**7 is too high.**

**Please enter a number between 1 and 10**

**6 is correct. You guessed it in 4 turns.**

## To enable starting previous versions of the workflows

If you run out of workflows to update, you can modify the `NumberGuessWorkflowHost` application to enable starting previous versions of the workflows.

1. Double-click **WorkflowHostForm** in **Solution Explorer**, and select the **WorkflowType** combo box.
2. In the **Properties** window, select the **Items** property and click the ellipsis button to edit the **Items** collection.
3. Add the following three items to the collection.

```
StateMachineNumberGuessWorkflow v1
FlowchartNumberGuessWorkflow v1
SequentialNumberGuessWorkflow v1
```

The completed `Items` collection will have six items.

```
StateMachineNumberGuessWorkflow
FlowchartNumberGuessWorkflow
SequentialNumberGuessWorkflow
StateMachineNumberGuessWorkflow v1
FlowchartNumberGuessWorkflow v1
SequentialNumberGuessWorkflow v1
```

4. Double-click **WorkflowHostForm** in **Solution Explorer**, and select **View Code**.
5. Add three new cases to the `switch` (or `Select Case`) statement in the `NewGame_Click` handler to map the new items in the **WorkflowType** combo box to the matching workflow identities.

```
Case "SequentialNumberGuessWorkflow v1"
    identity = WorkflowVersionMap.SequentialNumberGuessIdentity_v1

Case "StateMachineNumberGuessWorkflow v1"
    identity = WorkflowVersionMap.StateMachineNumberGuessIdentity_v1

Case "FlowchartNumberGuessWorkflow v1"
    identity = WorkflowVersionMap.FlowchartNumberGuessIdentity_v1
```

```
case "SequentialNumberGuessWorkflow v1":
    identity = WorkflowVersionMap.SequentialNumberGuessIdentity_v1;
    break;

case "StateMachineNumberGuessWorkflow v1":
    identity = WorkflowVersionMap.StateMachineNumberGuessIdentity_v1;
    break;

case "FlowchartNumberGuessWorkflow v1":
    identity = WorkflowVersionMap.FlowchartNumberGuessIdentity_v1;
    break;
```

The following example contains the complete `switch` (or `Select Case`) statement.

```

Select Case WorkflowType.SelectedItem.ToString()
Case "SequentialNumberGuessWorkflow"
    identity = WorkflowVersionMap.SequentialNumberGuessIdentity

Case "StateMachineNumberGuessWorkflow"
    identity = WorkflowVersionMap.StateMachineNumberGuessIdentity

Case "FlowchartNumberGuessWorkflow"
    identity = WorkflowVersionMap.FlowchartNumberGuessIdentity

Case "SequentialNumberGuessWorkflow v1"
    identity = WorkflowVersionMap.SequentialNumberGuessIdentity_v1

Case "StateMachineNumberGuessWorkflow v1"
    identity = WorkflowVersionMap.StateMachineNumberGuessIdentity_v1

Case "FlowchartNumberGuessWorkflow v1"
    identity = WorkflowVersionMap.FlowchartNumberGuessIdentity_v1
End Select

```

```

switch (WorkflowType.SelectedItem.ToString())
{
    case "SequentialNumberGuessWorkflow":
        identity = WorkflowVersionMap.SequentialNumberGuessIdentity;
        break;

    case "StateMachineNumberGuessWorkflow":
        identity = WorkflowVersionMap.StateMachineNumberGuessIdentity;
        break;

    case "FlowchartNumberGuessWorkflow":
        identity = WorkflowVersionMap.FlowchartNumberGuessIdentity;
        break;

    case "SequentialNumberGuessWorkflow v1":
        identity = WorkflowVersionMap.SequentialNumberGuessIdentity_v1;
        break;

    case "StateMachineNumberGuessWorkflow v1":
        identity = WorkflowVersionMap.StateMachineNumberGuessIdentity_v1;
        break;

    case "FlowchartNumberGuessWorkflow v1":
        identity = WorkflowVersionMap.FlowchartNumberGuessIdentity_v1;
        break;
}

```

6. Press CTRL+F5 to build and run the application. You can now start the `v1` versions of the workflow as well as the current versions. To dynamically update these new instances, run the **ApplyDynamicUpdate** application.

# Windows Workflow Foundation Programming

3/9/2019 • 2 minutes to read • [Edit Online](#)

This section contains a set of primer topics that you should understand to become a proficient Windows Workflow Foundation (WF) programmer.

## In This Section

### [Designing Workflows](#)

Topics that describe the flow-control paradigms used in workflow development.

### [Using and Creating Activities](#)

Topics that describes the system-provided activities available in Windows Workflow Foundation (WF).

### [Windows Workflow Foundation Data Model](#)

Topics that describe variables, arguments and expressions in WF.

### [Waiting for Input in a Workflow](#)

Topics that describes how to use bookmarks and messaging activities.

### [Exceptions, Transactions, and Compensation](#)

Topics that describe how to use exception handlers, transactions, and compensation to handle run-time errors.

### [Hosting Workflows](#)

Topics that describe the details for writing workflow host applications.

### [Dynamic Update](#)

Describes how to use dynamic update to update the workflow definition of a persisted workflow instance.

### [Workflow Services](#)

Topics that describe the programming model that supports writing services declaratively.

### [Workflow Persistence](#)

Topics that describe the options for automatically or manually persisting workflow data and unloading workflows from memory.

### [Migration Guidance](#)

Topics that describe how to migrate workflows from previous versions of Windows Workflow Foundation (WF).

### [Workflow Tracking and Tracing](#)

Topics that describe workflow tracking and tracing and how these features are used for monitoring workflow applications.

### [Workflow Security](#)

Discusses how to keep your workflow secure when using SQL and Windows Communication Foundation (WCF).

### [Windows Workflow Foundation 4 Performance](#)

Discusses Windows Workflow Foundation 4 performance and compares it to the previous version of WF.

# Designing Workflows

3/9/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section describe the flow-control paradigms used in workflow development.

## In This Section

### [Flowchart Workflows](#)

Shows how to create workflows using the familiar flowchart paradigm.

### [Procedural Workflows](#)

Shows how to create workflows using a basic, sequential execution paradigm.

### [State Machine Workflows](#)

Discusses how to use Flowchart workflows to implement event-driven workflows in .NET Framework 4, as a replacement for the State Machine model used in previous versions.

### [Authoring Workflows, Activities, and Expressions Using Imperative Code](#)

Describes how to author a workflow with imperative code.

# Flowchart Workflows

11/16/2018 • 2 minutes to read • [Edit Online](#)

A flowchart is a well-known paradigm for designing programs. The Flowchart activity is typically used to implement non-sequential workflows, but can be used for sequential workflows if no `FlowDecision` nodes are used.

## Flowchart workflow structure

A Flowchart activity is an activity that contains a collection of activities to be executed. Flowcharts also contain flow control elements such as `FlowDecision` and `FlowSwitch<T>` that direct execution between contained activities based on the values of variables.

## Types of flow nodes

Different types of elements are used depending on the type of flow control required when the element executes.

Types of flowchart elements include:

- `FlowStep` - Models one step of execution in the flowchart.
- `FlowDecision` - Branches execution based on a Boolean condition, similar to `If`.
- `FlowSwitch` – Branches execution based on an exclusive switch, similar to `Switch<T>`.

Each link has an `Action` property that defines a `ActivityAction` that can be used to execute child activities, and one or more `Next` properties that define which element or elements to execute when the current element finishes execution.

### Creating a basic activity sequence with a `FlowStep` node

To model a basic sequence in which two activities execute in turn, the `FlowStep` element is used. In the following example, two `FlowStep` elements are used to execute two activities in sequence.

```
<Flowchart>
  <FlowStep>
    <Assign DisplayName="Get Name">
      <Assign.To>
        <OutArgument x:TypeArguments="x:String">[result]</OutArgument>
      </Assign.To>
      <Assign.Value>
        <InArgument x:TypeArguments="x:String">["User"]</InArgument>
      </Assign.Value>
    </Assign>
    <FlowStep.Next>
      <FlowStep>
        <WriteLine Text="["Hello, " & result]"/>
      </FlowStep>
    </FlowStep.Next>
  </FlowStep>
</Flowchart>
```

### Creating a conditional flowchart with a `FlowDecision` node

To model a conditional flow node in a flowchart workflow (that is, to create a link that functions as a traditional flowchart's decision symbol), a `FlowDecision` node is used. The `Condition` property of the node is set to an expression that defines the condition, and the `True` and `False` properties are set to `FlowNode` instances to be

executed if the expression evaluates to `true` or `false`. The following example shows how to define a workflow that uses a [FlowDecision](#) node.

```
<Flowchart>
<FlowStep>
  <Read Result="[s]" />
  <FlowStep.Next>
    <FlowDecision>
      <IsEmpty Input="[s]" />
      <FlowDecision.True>
        <FlowStep>
          <Write Text="Empty"/>
        </FlowStep>
      </FlowDecision.True>
      <FlowDecision.False>
        <FlowStep>
          <Write Text="Non-Empty"/>
        </FlowStep>
      </FlowDecision.False>
    </FlowDecision>
  </FlowStep.Next>
</FlowStep>
</Flowchart>
```

### Creating an exclusive switch with a [FlowSwitch](#) node

To model a flowchart in which one exclusive path is selected based on a matching value, the [FlowSwitch<T>](#) node is used. The [Expression](#) property is set to a [Activity<TResult>](#) with a type parameter of [Object](#) that defines the value to match choices against. The [Cases](#) property defines a dictionary of keys and [FlowNode](#) objects to match against the conditional expression, and a set of [FlowNode](#) objects that define how execution should flow if the given case matches the conditional expression. The [FlowSwitch<T>](#) also defines a [Default](#) property that defines how execution should flow if no cases match the condition expression. The following example demonstrates how to define a workflow that uses a [FlowSwitch<T>](#) element.

```
<Flowchart>
<FlowSwitch>
  <FlowStep x:Key="Red">
    <WriteRed/>
  </FlowStep>
  <FlowStep x:Key="Blue">
    <WriteBlue/>
  </FlowStep>
  <FlowStep x:Key="Green">
    <WriteGreen/>
  </FlowStep>
</FlowSwitch>
</Flowchart>
```

# Procedural Workflows

3/9/2019 • 2 minutes to read • [Edit Online](#)

Procedural workflows use flow-control methods similar to those found in procedural languages. These constructs include `While` and `If`. These workflows can be freely composed using other flow control activities such as [Flowchart](#) and [Sequence](#).

## Controlling Execution Flow

The workflow activity library has activities for modeling most flow-control methods used in procedural languages. These include:

- [While](#)
- [DoWhile](#)
- [ForEach<T>](#)
- [Parallel](#)
- [ParallelForEach<T>](#)
- [If](#)
- [Switch<T>](#)
- [Pick](#)

To use flow control activities, drag and drop them from the **Activity** toolbox into a composite activity inside the designer window.

### NOTE

If using the hosting features of Windows Server AppFabric to host workflows on a Web farm, AppFabric will move instances between different AppFabric servers. This requires that the resources are able to be shared between all nodes. None of the default .NET 4 workflow activities contain any operations that access local resources. Since AppFabric does not offer any mechanism to mark a workflow as immovable, a developer must not create custom activities that fail when a workflow is moved.

## See also

- [Flowchart Workflows](#)

# State Machine Workflows

3/9/2019 • 6 minutes to read • [Edit Online](#)

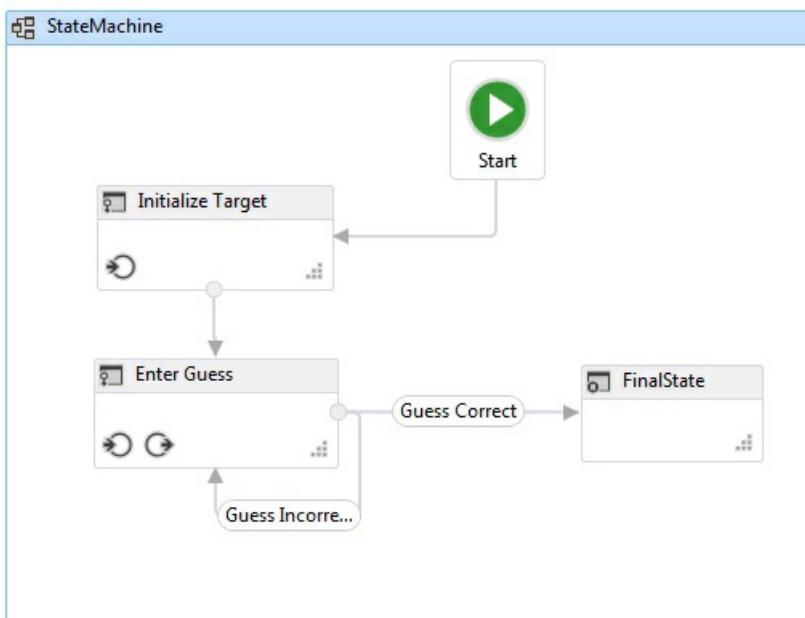
A state machine is a well-known paradigm for developing programs. The [StateMachine](#) activity, along with [State](#), [Transition](#), and other activities can be used to build state machine workflow programs. This topic provides an overview of creating state machine workflows.

## State Machine Workflow Overview

State machine workflows provide a modeling style with which you can model your workflow in an event-driven manner. A [StateMachine](#) activity contains the states and transitions that make up the logic of the state machine, and can be used anywhere an activity can be used. There are several classes in the state machine runtime:

- [StateMachine](#)
- [State](#)
- [Transition](#)

To create a state machine workflow, states are added to a [StateMachine](#) activity, and transitions are used control the flow between states. The following screenshot, from the [Getting Started Tutorial](#) step [How to: Create a State Machine Workflow](#), shows a state machine workflow with three states and three transitions. **Initialize Target** is the initial state and represents the first state in the workflow. This is designated by the line leading to it from the **Start** node. The final state in the workflow is named **FinalState**, and represents the point at which the workflow is completed.

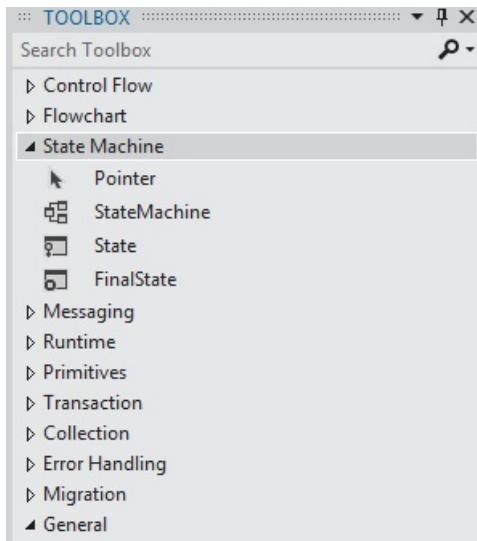


A state machine workflow must have one and only one initial state, and at least one final state. Each state that is not a final state must have at least one transition. The following sections cover creating and configuring states and transitions.

## Creating and Configuring States

A [State](#) represents a state in which a state machine can be in. To add a [State](#) to a workflow, drag the **State** activity designer from the **State Machine** section of the **Toolbox** and drop it onto a [StateMachine](#) activity on the

Windows Workflow Designer surface.



To configure a state as the **Initial State**, right-click the state and select **Set as Initial State**. Additionally, if there is no current initial state, the initial state can be designated by dragging a line from the **Start** node at the top of the workflow to the desired state. When a **StateMachine** activity is dropped onto the workflow designer, it is pre-configured with an initial state named **State1**. A state machine workflow must have one and only one initial state.

A state that represents a terminating state in a state machine is called a final state. A final state is a state that has its **IsFinal** property set to `true`, has no **Exit** activity, and no transitions originating from it. To add a final state to a workflow, drag a **FinalState** activity designer from the **State Machine** section of the **Toolbox** and drop it onto a **StateMachine** activity on the Windows Workflow Designer surface. A state machine workflow must have at least one final state.

### Configuring Entry and Exit Actions

A state can have an **Entry** and an **Exit** action. (A state configured as a final state may have only an entry action). When a workflow instance enters a state, any activities in the entry action execute. When the entry action is complete, the triggers for the state's transitions are scheduled. When a transition to another state is confirmed, the activities in the exit action are executed, even if the state transitions back to the same state. After the exit action completes, the activities in the transition's action execute, and then the new state is transitioned to, and its entry actions are scheduled.

#### NOTE

When debugging a state machine workflow, breakpoints can be placed on the root state machine activity and states within the state machine workflow. Breakpoints may not be placed directly on the transitions, but they may be placed on any activities contained within the states and transitions.

## Creating and Configuring Transitions

All states must have at least one transition, except for a final state which may not have any transitions. Transitions may be added after a state is added to a state machine workflow, or they can be created as the state is dropped.

To add a **State** and create a transition in one step, drag a **State** activity from the **State Machine** section of the **Toolbox** and hover it over another state in the workflow designer. When the dragged **State** is over another **State**, four triangles will appear around the other **State**. If the **State** is dropped onto one of the four triangles, it is added to the state machine and a transition is created from the source **State** to the dropped destination **State**. For more information, see [Transition Activity Designer](#).

To create a transition after a state is added, there are two options. The first option is to drag the state from the workflow designer surface and hover it over an existing state and drop it on one of the drop points. This is very

similar to the method described in the previous section. You can also hover the mouse over the desired source state, and drag a line to the desired destination state.

#### NOTE

A single state in a state machine can have up to 76 transitions created using the workflow designer. The limit on transitions for a state for workflows created outside the designer is limited only by system resources.

A transition may have a [Trigger](#), a [Condition](#), and an [Action](#). A transition's [Trigger](#) is scheduled when the transition's source state's [Entry](#) action is complete. Typically the [Trigger](#) is an activity that waits for some type of event to occur, but it can be any activity, or no activity at all. Once the [Trigger](#) activity is complete, the [Condition](#), if present, is evaluated. If there is no [Trigger](#) activity then the [Condition](#) is immediately evaluated. If the condition evaluates to `false`, the transition is cancelled, and the [Trigger](#) activity for all transitions from the state are rescheduled. If there are other transitions that share the same source state as the current transition, those [Trigger](#) actions are cancelled and rescheduled as well. If the [Condition](#) evaluates to `true`, or there is no condition, then the [Exit](#) action of the source state is executed, and then the [Action](#) of the transition is executed. When the [Action](#) completes, control passes to the **Target** state.

Transitions that share a common trigger are known as shared trigger transitions. Each transition in a group of shared trigger transitions has the same trigger, but a unique [Condition](#) and Action. To add additional actions to a transition and create a shared transition, click the circle that indicates the start of the desired transition and drag it to the desired state. The new transition will share a same trigger as the initial transition, but it will have a unique condition and action. Shared transitions can also be created from within the transition designer by clicking **Add shared trigger transition** at the bottom of the transition designer, and then selecting the desired target state from the **Available states to connect** drop-down.

#### NOTE

Note that if the [Condition](#) of a transition evaluates to `False` (or all of the conditions of a shared trigger transition evaluate to `False`), the transition will not occur and all triggers for all the transitions from the state will be rescheduled.

For more information on creating state machine workflows, see [How to: Create a State Machine Workflow](#), [StateMachine Activity Designer](#), [State Activity Designer](#), [FinalState Activity Designer](#), and [Transition Activity Designer](#).

## State Machine Terminology

This section defines the state machine vocabulary used throughout this topic.

### State

The basic unit that composes a state machine. A state machine can be in one state at any particular time.

### Entry Action

An activity executed when entering the state

### Exit Action

An activity executed when exiting the state

### Transition

A directed relationship between two states which represents the complete response of a state machine to an occurrence of an event of a particular type.

### Shared Transition

A transition that shares a source state and trigger with one or more transitions, but has a unique condition and

action.

#### Trigger

A triggering activity that causes a transition to occur.

#### Condition

A constraint which must evaluate to `true` after the trigger occurs in order for the transition to complete.

#### Transition Action

An activity which is executed when performing a certain transition.

#### Conditional Transition

A transition with an explicit condition.

#### Self-transition

A transition which transits from a state to itself.

#### Initial State

A state which represents the starting point of the state machine.

#### Final State

A state which represents the completion of the state machine.

## See also

- [How to: Create a State Machine Workflow](#)
- [StateMachine Activity Designer](#)
- [State Activity Designer](#)
- [FinalState Activity Designer](#)
- [Transition Activity Designer](#)

# Authoring Workflows, Activities, and Expressions Using Imperative Code

3/9/2019 • 10 minutes to read • [Edit Online](#)

A workflow definition is a tree of configured activity objects. This tree of activities can be defined many ways, including by hand-editing XAML or by using the Workflow Designer to produce XAML. Use of XAML, however, is not a requirement. Workflow definitions can also be created programmatically. This topic provides an overview of creating workflow definitions, activities, and expressions by using code. For examples of working with XAML workflows using code, see [Serializing Workflows and Activities to and from XAML](#).

## Creating Workflow Definitions

A workflow definition can be created by instantiating an instance of an activity type and configuring the activity object's properties. For activities that do not contain child activities, this can be accomplished using a few lines of code.

```
Activity wf = new WriteLine
{
    Text = "Hello World."
};

WorkflowInvoker.Invoke(wf);
```

### NOTE

The examples in this topic use [WorkflowInvoker](#) to run the sample workflows. For more information about invoking workflows, passing arguments, and the different hosting choices that are available, see [Using WorkflowInvoker and WorkflowApplication](#).

In this example, a workflow that consists of a single [WriteLine](#) activity is created. The [WriteLine](#) activity's [Text](#) argument is set, and the workflow is invoked. If an activity contains child activities, the method of construction is similar. The following example uses a [Sequence](#) activity that contains two [WriteLine](#) activities.

```
Activity wf = new Sequence
{
    Activities =
    {
        new WriteLine
        {
            Text = "Hello"
        },
        new WriteLine
        {
            Text = "World."
        }
    }
};

WorkflowInvoker.Invoke(wf);
```

### Using Object Initializers

The examples in this topic use object initialization syntax. Object initialization syntax can be a useful way to create workflow definitions in code because it provides a hierarchical view of the activities in the workflow and shows the relationship between the activities. There is no requirement to use object initialization syntax when you programmatically create workflows. The following example is functionally equivalent to the previous example.

```
WriteLine hello = new WriteLine();
hello.Text = "Hello";

WriteLine world = new WriteLine();
world.Text = "World";

Sequence wf = new Sequence();
wf.Activities.Add(hello);
wf.Activities.Add(world);

WorkflowInvoker.Invoke(wf);
```

For more information about object initializers, see [How to: Initialize Objects without Calling a Constructor \(C# Programming Guide\)](#) and [How to: Declare an Object by Using an Object Initializer](#).

### Working with Variables, Literal Values, and Expressions

When creating a workflow definition using code, be aware of what code executes as part of the creation of the workflow definition and what code executes as part of the execution of an instance of that workflow. For example, the following workflow is intended to generate a random number and write it to the console.

```
Variable<int> n = new Variable<int>
{
    Name = "n"
};

Activity wf = new Sequence
{
    Variables = { n },
    Activities =
    {
        new Assign<int>
        {
            To = n,
            Value = new Random().Next(1, 101)
        },
        new WriteLine
        {
            Text = new InArgument<string>((env) => "The number is " + n.Get(env))
        }
    }
};
```

When this workflow definition code is executed, the call to `Random.Next` is made and the result is stored in the workflow definition as a literal value. Many instances of this workflow can be invoked, and all would display the same number. To have the random number generation occur during workflow execution, an expression must be used that is evaluated each time the workflow runs. In the following example, a Visual Basic expression is used with a `VisualBasicValue<TResult>`.

```
new Assign<int>
{
    To = n,
    Value = new VisualBasicValue<int>("New Random().Next(1, 101)")
}
```

The expression in the previous example could also be implemented using a `CSharpValue<TResult>` and a C# expression.

```
new Assign<int>
{
    To = n,
    Value = new CSharpValue<int>("new Random().Next(1, 101)")
}
```

C# expressions must be compiled before the workflow containing them is invoked. If the C# expressions are not compiled, a `NotSupportedException` is thrown when the workflow is invoked with a message similar to the following: `Expression Activity type 'CSharpValue`1' requires compilation in order to run. Please ensure that the workflow has been compiled.` In most scenarios involving workflows created in Visual Studio the C# expressions are compiled automatically, but in some scenarios, such as code workflows, the C# expressions must be manually compiled. For an example of how to compile C# expressions, see the [Using C# expressions in code workflows](#) section of the [C# Expressions](#) topic.

A `VisualBasicValue<TResult>` represents an expression in Visual Basic syntax that can be used as an r-value in an expression, and a `CSharpValue<TResult>` represents an expression in C# syntax that can be used as an r-value in an expression. These expressions are evaluated each time the containing activity is executed. The result of the expression is assigned to the workflow variable `n`, and these results are used by the next activity in the workflow. To access the value of the workflow variable `n` at runtime, the `ActivityContext` is required. This can be accessed by using the following lambda expression.

#### NOTE

Note that both of these code examples are using C# as the programming language, but one uses a `VisualBasicValue<TResult>` and one uses a `CSharpValue<TResult>`. `VisualBasicValue<TResult>` and `CSharpValue<TResult>` can be used in both Visual Basic and C# projects. By default, expressions created in the workflow designer match the language of the hosting project. When creating workflows in code, the desired language is at the discretion of the workflow author.

In these examples the result of the expression is assigned to the workflow variable `n`, and these results are used by the next activity in the workflow. To access the value of the workflow variable `n` at runtime, the `ActivityContext` is required. This can be accessed by using the following lambda expression.

```
new WriteLine
{
    Text = new InArgument<string>((env) => "The number is " + n.Get(env))
}
```

For more information about lambda expressions, see [Lambda Expressions \(C# Programming Guide\)](#) or [Lambda Expressions \(Visual Basic\)](#).

Lambda expressions are not serializable to XAML format. If an attempt to serialize a workflow with lambda expressions is made, a `LambdaSerializationException` is thrown with the following message: "This workflow contains lambda expressions specified in code. These expressions are not XAML serializable. In order to make your workflow XAML-serializable, either use VisualBasicValue/VisualBasicReference or ExpressionServices.Convert(lambda). This will convert your lambda expressions into expression activities." To make this expression compatible with XAML, use `ExpressionServices` and `Convert`, as shown in the following example.

```

new WriteLine
{
    //Text = new InArgument<string>((env) => "The number is " + n.Get(env))
    Text = ExpressionServices.Convert((env) => "The number is " + n.Get(env))
}

```

A [VisualBasicValue<TResult>](#) could also be used. Note that no lambda expression is required when using a Visual Basic expression.

```

new WriteLine
{
    //Text = new InArgument<string>((env) => "The number is " + n.Get(env))
    //Text = ExpressionServices.Convert((env) => "The number is " + n.Get(env))
    Text = new VisualBasicValue<string>("\"The number is \" + n.ToString()")
}

```

At run time, Visual Basic expressions are compiled into LINQ expressions. Both of the previous examples are serializable to XAML, but if the serialized XAML is intended to be viewed and edited in the workflow designer, use [VisualBasicValue<TResult>](#) for your expressions. Serialized workflows that use [ExpressionServices.Convert](#) can be opened in the designer, but the value of the expression will be blank. For more information about serializing workflows to XAML, see [Serializing Workflows and Activities to and from XAML](#).

#### Literal Expressions and Reference Types

Literal expressions are represented in workflows by the [Literal<T>](#) activity. The following [WriteLine](#) activities are functionally equivalent.

```

new WriteLine
{
    Text = "Hello World."
},
new WriteLine
{
    Text = new Literal<string>("Hello World.")
}

```

It is invalid to initialize a literal expression with any reference type except [String](#). In the following example, an [Assign](#) activity's [Value](#) property is initialized with a literal expression using a [List<string>](#).

```

new Assign
{
    To = new OutArgument<List<string>>(items),
    Value = new InArgument<List<string>>(new List<string>())
},

```

When the workflow containing this activity is validated, the following validation error is returned: "Literal only supports value types and the immutable type System.String. The type System.Collections.Generic.List`1[System.String] cannot be used as a literal." If the workflow is invoked, an [InvalidOperationException](#) is thrown that contains the text of the validation error. This is a validation error because creating a literal expression with a reference type does not create a new instance of the reference type for each instance of the workflow. To resolve this, replace the literal expression with one that creates and returns a new instance of the reference type.

```

new Assign
{
    To = new OutArgument<List<string>>(items),
    Value = new InArgument<List<string>>(new VisualBasicValue<List<string>>("New List(Of String)"))
},

```

For more information about expressions, see [Expressions](#).

#### **Invoking Methods on Objects using Expressions and the InvokeMethod Activity**

The [InvokeMethod<TResult>](#) activity can be used to invoke static and instance methods of classes in the .NET Framework. In a previous example in this topic, a random number was generated using the [Random](#) class.

```

new Assign<int>
{
    To = n,
    Value = new VisualBasicValue<int>("New Random().Next(1, 101)")
}

```

The [InvokeMethod<TResult>](#) activity could also have been used to call the [Next](#) method of the [Random](#) class.

```

new InvokeMethod<int>
{
    TargetObject = new InArgument<Random>(new VisualBasicValue<Random>("New Random()")),
    MethodName = "Next",
    Parameters =
    {
        new InArgument<int>(1),
        new InArgument<int>(101)
    },
    Result = n
}

```

Since [Next](#) is not a static method, an instance of the [Random](#) class is supplied for the [TargetObject](#) property. In this example a new instance is created using a Visual Basic expression, but it could also have been created previously and stored in a workflow variable. In this example, it would be simpler to use the [Assign<T>](#) activity instead of the [InvokeMethod<TResult>](#) activity. If the method call ultimately invoked by either the [Assign<T>](#) or [InvokeMethod<TResult>](#) activities is long running, [InvokeMethod<TResult>](#) has an advantage since it has a [RunAsynchronously](#) property. When this property is set to `true`, the invoked method will run asynchronously with regard to the workflow. If other activities are in parallel, they will not be blocked while the method is asynchronously executing. Also, if the method to be invoked has no return value, then [InvokeMethod<TResult>](#) is the appropriate way to invoke the method.

## Arguments and Dynamic Activities

A workflow definition is created in code by assembling activities into an activity tree and configuring any properties and arguments. Existing arguments can be bound, but new arguments cannot be added to activities. This includes workflow arguments passed to the root activity. In imperative code, workflow arguments are specified as properties on a new CLR type, and in XAML they are declared by using `x:Class` and `x:Member`. Because there is no new CLR type created when a workflow definition is created as a tree of in-memory objects, arguments cannot be added. However, arguments can be added to a [DynamicActivity](#). In this example, a [DynamicActivity<TResult>](#) is created that takes two integer arguments, adds them together, and returns the result. A [DynamicActivityProperty](#) is created for each argument, and the result of the operation is assigned to the [Result](#) argument of the [DynamicActivity<TResult>](#).

```

InArgument<int> Operand1 = new InArgument<int>();
InArgument<int> Operand2 = new InArgument<int>();

DynamicActivity<int> wf = new DynamicActivity<int>
{
    Properties =
    {
        new DynamicActivityProperty
        {
            Name = "Operand1",
            Type = typeof(InArgument<int>),
            Value = Operand1
        },
        new DynamicActivityProperty
        {
            Name = "Operand2",
            Type = typeof(InArgument<int>),
            Value = Operand2
        }
    },

    Implementation = () => new Sequence
    {
        Activities =
        {
            new Assign<int>
            {
                To = new ArgumentReference<int> { ArgumentName = "Result" },
                Value = new InArgument<int>((env) => Operand1.Get(env) + Operand2.Get(env))
            }
        }
    }
};

Dictionary<string, object> wfParams = new Dictionary<string, object>
{
    { "Operand1", 25 },
    { "Operand2", 15 }
};

int result = WorkflowInvoker.Invoke(wf, wfParams);
Console.WriteLine(result);

```

For more information about dynamic activities, see [Creating an Activity at Runtime](#).

## Compiled Activities

Dynamic activities are one way to define an activity that contains arguments using code, but activities can also be created in code and compiled into types. Simple activities can be created that derive from [CodeActivity](#), and asynchronous activities that derive from [AsyncCodeActivity](#). These activities can have arguments, return values, and define their logic using imperative code. For examples of creating these types of activities, see [CodeActivity Base Class](#) and [Creating Asynchronous Activities](#).

Activities that derive from [NativeActivity](#) can define their logic using imperative code and they can also contain child activities that define the logic. They also have full access to the features of the runtime such as creating bookmarks. For examples of creating a [NativeActivity](#)-based activity, see [NativeActivity Base Class](#), [How to: Create an Activity](#), and the [Custom Composite using Native Activity](#) sample.

Activities that derive from [Activity](#) define their logic solely through the use of child activities. These activities are typically created by using the workflow designer, but can also be defined by using code. In the following example, a `Square` activity is defined that derives from `Activity<int>`. The `Square` activity has a single `InArgument<T>` named `Value`, and defines its logic by specifying a `Sequence` activity using the `Implementation` property. The

Sequence activity contains a [WriteLine](#) activity and an [Assign<T>](#) activity. Together, these three activities implement the logic of the `Square` activity.

```
class Square : Activity<int>
{
    [RequiredArgument]
    public InArgument<int> Value { get; set; }

    public Square()
    {
        this.Implementation = () => new Sequence
        {
            Activities =
            {
                new WriteLine
                {
                    Text = new InArgument<string>((env) => "Squaring the value: " + this.Value.Get(env))
                },
                new Assign<int>
                {
                    To = new OutArgument<int>((env) => this.Result.Get(env)),
                    Value = new InArgument<int>((env) => this.Value.Get(env) * this.Value.Get(env))
                }
            }
        };
    }
}
```

In the following example, a workflow definition consisting of a single `Square` activity is invoked using [WorkflowInvoker](#).

```
Dictionary<string, object> inputs = new Dictionary<string, object> {{ "Value", 5 }};
int result = WorkflowInvoker.Invoke(new Square(), inputs);
Console.WriteLine("Result: {0}", result);
```

When the workflow is invoked, the following output is displayed to the console:

**Squaring the value: 5**

**Result: 25**

# Using and Creating Activities

3/9/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section describe the system-provided activities available in Windows Workflow Foundation (WF).

## In This Section

### [Built-In Activity Library](#)

Describes the system-provided activities available in WF.

### [Designing and Implementing Custom Activities](#)

Describes how to create custom activities with the [CodeActivity](#) or the [NativeActivity](#) methods.

# .NET Framework 4.5 Built-In Activity Library

3/9/2019 • 2 minutes to read • [Edit Online](#)

.NET Framework 4.6.1 includes a new activity library with expanded functionality. The types of activities include the following:

- [Control Flow](#)
- [Flowchart](#)
- [State Machine](#)
- [Messaging Activities](#)
- [Runtime](#)
- [Primitives](#)
- [Transaction](#)
- [Collection](#)
- [Error Handling](#)
- [Migration](#)

The activities in the built-in activity library can be composed into workflows, or used alongside custom activities. The activities in the built-in activity library are sealed; they are not intended to be used to create new functionality through inheritance.

# Control Flow Activities in WF

5/4/2018 • 2 minutes to read • [Edit Online](#)

The .NET Framework 4.6.1 provides several activities for controlling flow of execution within a workflow. Some of these activities (such as `Switch` and `If`) implement flow control structures similar to those in programming environments such as Visual C#, while others (such as `Pick`) model new programming structures.

Note that while activities such as the `Parallel` and `ParallelForEach` activities schedule multiple child activities for execution simultaneously, only a single thread is used for a workflow. Each child activity of these activities executes sequentially and successive activities do not execute until previous activities either complete or go idle. As a result, these activities are most useful for applications in which several potentially blocking activities must execute in an interleaved fashion. If none of the child activities of these activities go idle, a `Parallel` activity executes just like a `Sequence` activity, and a `ParallelForEach` activity executes just like a `ForEach` activity. If, however, asynchronous activities (such as activities that derive from `AsyncCodeActivity`) or messaging activities are used, control will pass to the next branch while the child activity waits for its message to be received or its asynchronous work to be completed.

## Flow control activities

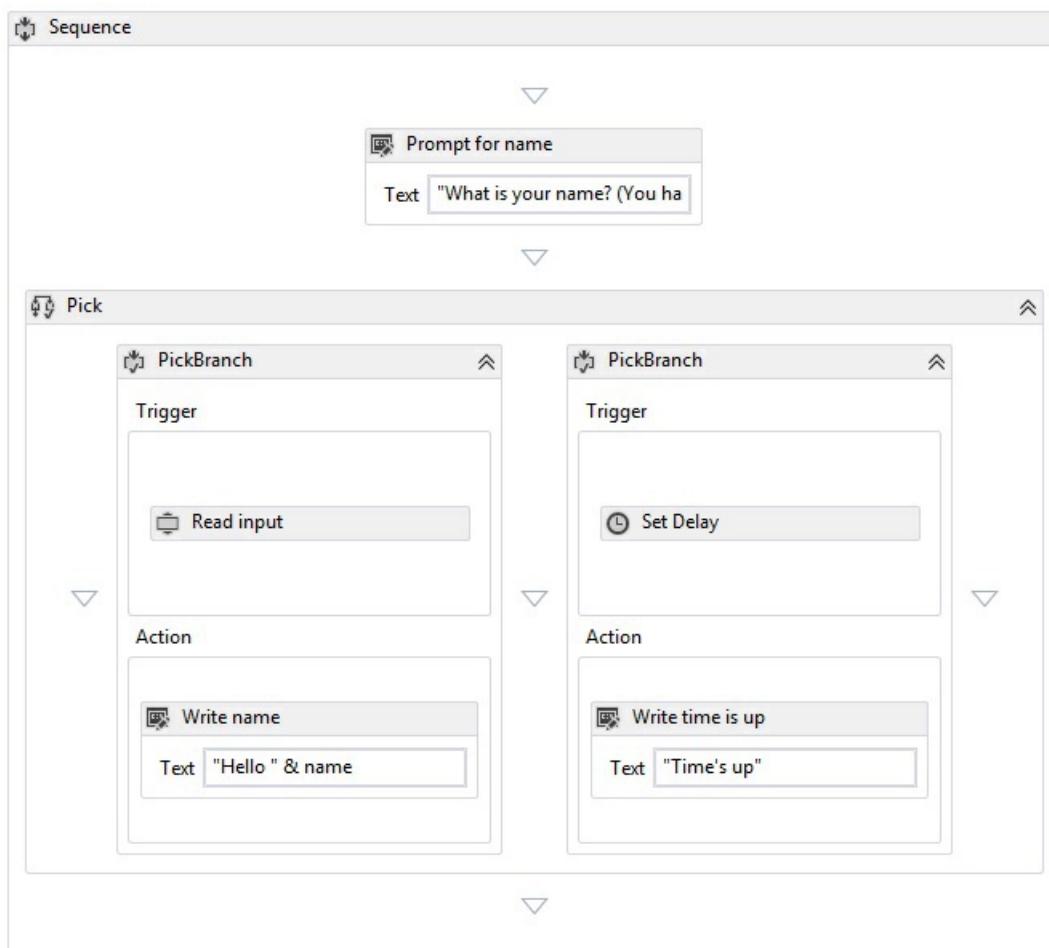
| ACTIVITY                              | DESCRIPTION   |
|---------------------------------------|---|
| <code>DoWhile</code>                  | Executes the contained activities once and continues to do so while a condition is <code>true</code> .  |
| <code>ForEach&lt;T&gt;</code>         | Executes an embedded statement in sequence for each element in a collection. <code>ForEach&lt;T&gt;</code> is similar to the keyword <code>foreach</code> , but is implemented as an activity rather than a language statement. |
| <code>If</code>                       | Executes contained activities if a condition is <code>true</code> , and can execute activities contained in the <code>Else</code> property if the condition is <code>false</code> .   |
| <code>Parallel</code>                 | Executes contained activities in parallel.  |
| <code>ParallelForEach&lt;T&gt;</code> | Executes an embedded statement in parallel for each element in a collection.  |
| <code>Pick</code>                     | Provides event-based control flow modeling.   |
| <code>PickBranch</code>               | Represents a potential path of execution in a <code>Pick</code> activity.   |
| <code>Sequence</code>                 | Executes contained activities in sequence.  |
| <code>Switch&lt;T&gt;</code>          | Selects one choice from a number of activities to execute, based on the value of a given expression.  |
| <code>While</code>                    | Executes contained activities while a condition is <code>true</code> .  |

# Pick Activity

3/25/2019 • 3 minutes to read • [Edit Online](#)

The [Pick](#) activity simplifies the modeling of a set of event triggers followed by their corresponding handlers. A [Pick](#) activity contains a collection of [PickBranch](#) activities, where each [PickBranch](#) is a pairing between a [Trigger](#) activity and an [Action](#) activity. At execution time, the triggers for all branches are executed in parallel. When one trigger completes, then its corresponding action is executed, and all other triggers are canceled. The behavior of the .NET Framework 4.6.1 [Pick](#) activity is similar to the .NET Framework 3.5 [ListenActivity](#) activity.

The following screenshot from the [Using the Pick Activity](#) SDK sample shows a Pick activity with two branches. One branch has a trigger called **Read input**, a custom activity that reads input from the command line. The second branch has a **Delay** activity trigger. If the **Read input** activity receives data before the **Delay** activity finishes, **Delay** will be canceled and a greeting will be written to the console. Otherwise, if the **Read input** activity does not receive data in the allotted time, then it will be canceled and a timeout message will be written to the console. This is a common pattern used to add a timeout to any action.



## Best practices

When using Pick, the branch that executes is the branch whose trigger completes first. Conceptually, all triggers execute in parallel, and one trigger may have executed the majority of its logic before it is canceled by the completion of another trigger. With this in mind, a general guideline to follow when using the Pick activity is to treat the trigger as representing a single event, and to put as little logic as possible into it. Ideally, the trigger should contain just enough logic to receive an event, and all the processing of that event should go into the action of the branch. This method minimizes the amount of overlap between the execution of the triggers. For example, consider

a **Pick** with two triggers, where each trigger contains a **Receive** activity followed by additional logic. If the additional logic introduces an idle point, then there is the possibility of both **Receive** activities completing successfully. One trigger will fully complete, while another will partially complete. In some scenarios, accepting a message, and then partially completing the processing of it is unacceptable. Therefore, when using WF built-in messaging activities such as **Receive** and **SendReply**, while **Receive** is commonly used in the trigger, **SendReply** and other logic should be put in the action whenever possible.

## Using the Pick activity in the designer

To use Pick in the designer, find **Pick** and **PickBranch** in the toolbox. Drag and drop **Pick** onto the canvas. By default, a new **Pick** Activity in the designer will contain two branches. To add additional branches, drag the **PickBranch** activity and drop it next to existing branches. Activities can be dropped onto the **Pick** Activity into either the **Trigger** area or the **Action** area of any **PickBranch**.

## Using the Pick Activity in code

The **Pick** activity is used by populating its **Branches** collection with **PickBranch** activities. The **PickBranch** activities each have a **Trigger** property of type **Activity**. When the specified activity completes execution, the **Action** executes.

The following code example demonstrates how to use a **Pick** activity to implement a timeout for an activity that reads a line from the console.

```
Sequence body = new Sequence()
{
    Variables = { name },
    Activities =
    {
        new System.Activities.Statements.Pick
        {
            Branches =
            {
                new PickBranch
                {
                    Trigger = new ReadLine
                    {
                        Result = name,
                        BookmarkName = "name"
                    },
                    Action = new WriteLine
                    {
                        Text = ExpressionServices.Convert<string>(ctx => "Hello " +
                            name.Get(ctx))
                    }
                },
                new PickBranch
                {
                    Trigger = new Delay
                    {
                        Duration = new TimeSpan(0, 0, 5)
                    },
                    Action = new WriteLine
                    {
                        Text = "Time is up."
                    }
                }
            }
        }
    };
}
```

```
<Sequence xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
<Sequence.Variables>
    <Variable x:TypeArguments="x:String" Name="username" />
</Sequence.Variables>
<Pick>
    <PickBranch>
        <PickBranch.Trigger>
            <ReadLine BookmarkName="name" Result="username" />
        </PickBranch.Trigger>
        <WriteLine>[String.Concat("Hello ", username)]</WriteLine>
    </PickBranch>
    <PickBranch>
        <PickBranch.Trigger>
            <Delay>00:00:05</Delay>
        </PickBranch.Trigger>
        <WriteLine>Time is up.</WriteLine>
    </PickBranch>
</Pick>
</Sequence>
```

# Flowchart Activities in WF

3/9/2019 • 2 minutes to read • [Edit Online](#)

.NET Framework 4.6.1 provides several system-provided activities for controlling execution and branching within a Flowchart.

|                                  |  |
|----------------------------------|--|
| <code>Flowchart</code>           | Executes contained activities using the familiar Flowchart paradigm.   |
| <code>FlowDecision</code>        | A specialized <code>FlowNode</code> that provides the ability to model a conditional node with two outcomes.   |
| <code>FlowSwitch&lt;T&gt;</code> | A specialized <code>FlowNode</code> that allows modeling a switch construct, with one expression of a type defined in the activity's type specifier and a single outcome for each match. |

## See also

- [Getting Started Tutorial](#)

# State Machine Activities in WF

3/9/2019 • 2 minutes to read • [Edit Online](#)

.NET Framework 4.5 provides several system-provided activities and activity designers for creating state machine workflows.

|                              |  |
|------------------------------|--|
| <a href="#">StateMachine</a> | Executes contained activities using the familiar state machine paradigm.   |
| <a href="#">State</a>        | Represents a state in a state machine.   |
| <a href="#">FinalState</a>   | Represents a terminating state in a state machine. <a href="#">FinalState</a> is an activity designer that when used creates a <a href="#">State</a> preconfigured as a terminating state. For more information, see <a href="#">FinalState Activity Designer</a> .  |
| <a href="#">Transition</a>   | Represents the transition between two states. There is no <b>Toolbox</b> item for <a href="#">Transition</a> ; transitions are created on the workflow designer by dragging and dropping a line between two states, or by dropping a state on the triangles that appear when one state is hovered over another. For more information, see <a href="#">Transition Activity Designer</a> . |

## See also

- [Getting Started Tutorial](#)

# Runtime Activities in WF

5/4/2018 • 2 minutes to read • [Edit Online](#)

.NET Framework 4.6.1 provides several system-provided activities for accessing the features of the workflow runtime, such as persistence and termination.

| ACTIVITY          | DESCRIPTION  |
|-------------------|--|
| Persist           | Explicitly requests that the workflow persist its state.             |
| TerminateWorkflow | Terminates the running workflow instance.                            |
| NoPersistScope    | A container activity that prevents child activities from persisting. |

# Primitives Activities in WF

5/4/2018 • 2 minutes to read • [Edit Online](#)

.NET Framework 4.6.1 provides several system-provided activities that provide a convenient mechanism for performing common tasks.

| ACTIVITY                       | DESCRIPTION  |
|--------------------------------|--|
| <a href="#">Assign</a>         | Assigns a value to a variable at the current scope.  |
| <a href="#">Delay</a>          | Puts one path of execution into an idle state, possibly allowing the workflow to be unloaded.        |
| <a href="#">InvokeDelegate</a> | Executes a delegate that derives from <a href="#">ActivityDelegate</a> and is exposed as a property. |
| <a href="#">InvokeMethod</a>   | Executes a public method of a CLR object.  |
| <a href="#">WriteLine</a>      | Writes a specified string to the console or a specified <a href="#">TextWriter</a> object.           |

# Transaction Activities in WF

3/9/2019 • 2 minutes to read • [Edit Online](#)

The .NET Framework 4.6.1 has several system-provided activities for modeling transactions, compensation, and cancellation. These programming models allow the workflow to continue forward progress in the event of changes in business logic and error handling. For more information about transactions, compensation, and cancellation, see [Transactions](#), [Compensation](#), and [Cancellation](#).

## Transaction Activities

|  |  |
|--|--|
| <a href="#">CancellationScope</a>      | Associates cancellation logic, in the form of an activity, with a main path of execution, also expressed as an activity.   |
| <a href="#">CompensableActivity</a>    | Supports compensation of its child activities.   |
| <a href="#">Compensate</a>             | Explicitly invokes the compensation handler of a <a href="#">CompensableActivity</a> .   |
| <a href="#">Confirm</a>                | Explicitly invokes the confirmation handler of a <a href="#">CompensableActivity</a> .   |
| <a href="#">TransactionScope</a>       | Demarcates a transaction boundary.   |
| <a href="#">TransactedReceiveScope</a> | Scopes the lifetime of a transaction that is initiated by a received message. The transaction may be flowed into the workflow on the initiating message, or created by the dispatcher when the message is received. <b>Note:</b> The <a href="#">TransactedReceiveScope</a> is located in the <b>Messaging</b> section of the <b>Toolbox</b> . |

# Collection Activities in WF

3/9/2019 • 3 minutes to read • [Edit Online](#)

Collection activities are used to work with collection objects in a workflow. .NET Framework 4.6.1 has system-provided activities for adding and removing items from a collection, testing for the existence of an item in a collection, and clearing a collection. `ExistsInCollection` and `RemoveFromCollection` have an `OutArgument<T>` of type `Boolean`, which indicates the result.

## IMPORTANT

If a collection activity is executed before setting the underlying collection object, an `InvalidOperationException` is thrown and the activity faults.

## Collection activities

|  |   |
|--|---|
| <code>AddToCollection&lt;T&gt;</code>      | Adds an item to a specified collection.   |
| <code>ClearCollection&lt;T&gt;</code>      | Clears all items from a specified collection.   |
| <code>ExistsInCollection&lt;T&gt;</code>   | Returns <code>true</code> if an item exists in a collection.  |
| <code>RemoveFromCollection&lt;T&gt;</code> | Removes an item from a specified collection and returns <code>true</code> if the item was successfully removed. |

## Using collection activities

The following code example demonstrates how to interact with a collection declared as a workflow variable. The collection used is a `List<T>` of `String` objects named `fruitList`.

```

Variable<ICollection<string>> fruitList = new Variable<ICollection<string>>
{
    Default = new VisualBasicValue<ICollection<string>>("New List(Of String) From {" &quot;Apple&quot;, &quot;Orange&quot;}"),
    Name = "FruitList"
};

Variable<bool> result = new Variable<bool>
{
    Name = "Result"
};

Activity wf = new Sequence
{
    Variables = { fruitList, result },

    Activities =
    {
        new If
        {
            Condition = new ExistsInCollection<string>
            {
                Collection = fruitList,
                Item = "Pear"
            },
            Then = new AddToCollection<string>
            {
                Collection = fruitList,
                Item = "Pear"
            },
            Else = new RemoveFromCollection<string>
            {
                Collection = fruitList,
                Item = "Apple"
            }
        },
        new RemoveFromCollection<string>
        {
            Collection = fruitList,
            Item = "Apple",
            Result = result
        },
        new If
        {
            Condition = result,
            Then = new ClearCollection<string>
            {
                Collection = fruitList,
            }
        }
    }
};

```

```

<Sequence
    xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
    xmlns:scg="clr-namespace:System.Collections.Generic;assembly=mscorlib"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Sequence.Variables>
        <x:Reference>__ReferenceID0</x:Reference>
        <x:Reference>__ReferenceID1</x:Reference>
    </Sequence.Variables>
    <If>
        <If.Condition>
            <InArgument
                x:TypeArguments="x:Boolean">
                <ExistsInCollection

```

```

        x:TypeArguments="x:String"
        Item="Pear">
    <ExistsInCollection.Result>
        <OutArgument
            x:TypeArguments="x:Boolean" />
    </ExistsInCollection.Result>
    <InArgument
        x:TypeArguments="scg:ICollection(x:String)">
        <VariableValue
            x:TypeArguments="scg:ICollection(x:String)">
            <VariableValue.Result>
                <OutArgument
                    x:TypeArguments="scg:ICollection(x:String)" />
            </VariableValue.Result>
            <VariableValue.Variable>
                <Variable
                    x:TypeArguments="scg:ICollection(x:String)"
                    x:Name="__ReferenceID0"
                    Default="[New List(Of String) From {"Apple", "Orange"}]"
                    Name="FruitList" />
            </VariableValue.Variable>
        </VariableValue>
    </InArgument>
    </ExistsInCollection>
</InArgument>
</If.Condition>
<If.Then>
    <AddToCollection
        x:TypeArguments="x:String"
        Item="Pear">
    <InArgument
        x:TypeArguments="scg:ICollection(x:String)">
        <VariableValue
            x:TypeArguments="scg:ICollection(x:String)"
            Variable="{x:Reference __ReferenceID0}">
        <VariableValue.Result>
            <OutArgument
                x:TypeArguments="scg:ICollection(x:String)" />
        </VariableValue.Result>
    </VariableValue>
    </InArgument>
    </AddToCollection>
</If.Then>
<If.Else>
    <RemoveFromCollection
        x:TypeArguments="x:String"
        Item="Apple"
        Result="{x:Null}">
    <InArgument
        x:TypeArguments="scg:ICollection(x:String)">
        <VariableValue
            x:TypeArguments="scg:ICollection(x:String)"
            Variable="{x:Reference __ReferenceID0}">
        <VariableValue.Result>
            <OutArgument
                x:TypeArguments="scg:ICollection(x:String)" />
        </VariableValue.Result>
    </VariableValue>
    </InArgument>
    </RemoveFromCollection>
</If.Else>
</If>
<RemoveFromCollection
    x:TypeArguments="x:String"
    Item="Apple">
    <RemoveFromCollection.Result>
        <OutArgument
            x:TypeArguments="x:Boolean">
            <VariableReference

```

```

        x>TypeArguments="x:Boolean">
    <VariableReference.Result>
        <OutArgument
            x>TypeArguments="Location(x:Boolean)" />
    </VariableReference.Result>
    <VariableReference.Variable>
        <Variable
            x>TypeArguments="x:Boolean"
            x>Name=__ReferenceID1"
            Name="Result" />
    </VariableReference.Variable>
    </VariableReference>
</OutArgument>
</RemoveFromCollection.Result>
<InArgument
    x>TypeArguments="scg:ICollection(x:String)">
<VariableValue
    x>TypeArguments="scg:ICollection(x:String)"
    Variable="{x:Reference __ReferenceID0}">
<VariableValue.Result>
    <OutArgument
        x>TypeArguments="scg:ICollection(x:String)" />
    </VariableValue.Result>
</VariableValue>
</InArgument>
</RemoveFromCollection>
<If>
    <If.Condition>
        <InArgument
            x>TypeArguments="x:Boolean">
        <VariableValue
            x>TypeArguments="x:Boolean"
            Variable="{x:Reference __ReferenceID1}">
        <VariableValue.Result>
            <OutArgument
                x>TypeArguments="x:Boolean" />
        </VariableValue.Result>
        </VariableValue>
        </InArgument>
    </If.Condition>
    <If.Then>
        <ClearCollection
            x>TypeArguments="x:String">
        <InArgument
            x>TypeArguments="scg:ICollection(x:String)">
        <VariableValue
            x>TypeArguments="scg:ICollection(x:String)"
            Variable="{x:Reference __ReferenceID0}">
        <VariableValue.Result>
            <OutArgument
                x>TypeArguments="scg:ICollection(x:String)" />
            </VariableValue.Result>
        </VariableValue>
        </InArgument>
    </ClearCollection>
    </If.Then>
</If>
</Sequence>

```

The above code samples can also be created using [CSharpValue<TResult>](#) instead of [VisualBasicValue<TResult>](#)

```

Variable<ICollection<string>> fruitList = new Variable<ICollection<string>>
{
    Default = new CSharpValue<ICollection<string>>("new List<String> From {\"Apple\", \"Orange\"};"),
    Name = "FruitList"
};

Variable<bool> result = new Variable<bool>
{
    Name = "Result"
};

Activity wf = new Sequence
{
    Variables = { fruitList, result },

    Activities =
    {
        new If
        {
            Condition = new ExistsInCollection<string>
            {
                Collection = fruitList,
                Item = "Pear"
            },
            Then = new AddToCollection<string>
            {
                Collection = fruitList,
                Item = "Pear"
            },
            Else = new RemoveFromCollection<string>
            {
                Collection = fruitList,
                Item = "Apple"
            }
        },
        new RemoveFromCollection<string>
        {
            Collection = fruitList,
            Item = "Apple",
            Result = result
        },
        new If
        {
            Condition = result,
            Then = new ClearCollection<string>
            {
                Collection = fruitList,
            }
        }
    }
};

```

```

<Sequence
    xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
    xmlns:scg="clr-namespace:System.Collections.Generic;assembly=mscorlib"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Sequence.Variables>
        <x:Reference>__ReferenceID0</x:Reference>
        <x:Reference>__ReferenceID1</x:Reference>
    </Sequence.Variables>
    <If>
        <If.Condition>
            <InArgument
                x:TypeArguments="x:Boolean">
                <ExistsInCollection

```

```

        x:TypeArguments="x:String"
        Item="Pear">
    <ExistsInCollection.Result>
        <OutArgument
            x:TypeArguments="x:Boolean" />
    </ExistsInCollection.Result>
    <InArgument
        x:TypeArguments="scg:ICollection(x:String)">
        <VariableValue
            x:TypeArguments="scg:ICollection(x:String)">
            <VariableValue.Result>
                <OutArgument
                    x:TypeArguments="scg:ICollection(x:String)" />
            </VariableValue.Result>
            <VariableValue.Variable>
                <Variable
                    x:TypeArguments="scg:ICollection(x:String)"
                    x:Name="__ReferenceID0"
                    Default="[new List<String> From {"Apple", "Orange"};]"
                    Name="FruitList" />
            </VariableValue.Variable>
        </VariableValue>
    </InArgument>
    </ExistsInCollection>
</InArgument>
</If.Condition>
<If.Then>
    <AddToCollection
        x:TypeArguments="x:String"
        Item="Pear">
        <InArgument
            x:TypeArguments="scg:ICollection(x:String)">
            <VariableValue
                x:TypeArguments="scg:ICollection(x:String)"
                Variable="{x:Reference __ReferenceID0}">
            <VariableValue.Result>
                <OutArgument
                    x:TypeArguments="scg:ICollection(x:String)" />
            </VariableValue.Result>
        </VariableValue>
    </InArgument>
    </AddToCollection>
</If.Then>
<If.Else>
    <RemoveFromCollection
        x:TypeArguments="x:String"
        Item="Apple"
        Result="{x:Null}">
        <InArgument
            x:TypeArguments="scg:ICollection(x:String)">
            <VariableValue
                x:TypeArguments="scg:ICollection(x:String)"
                Variable="{x:Reference __ReferenceID0}">
            <VariableValue.Result>
                <OutArgument
                    x:TypeArguments="scg:ICollection(x:String)" />
            </VariableValue.Result>
        </VariableValue>
    </InArgument>
    </RemoveFromCollection>
</If.Else>
</If>
<RemoveFromCollection
    x:TypeArguments="x:String"
    Item="Apple">
    <RemoveFromCollection.Result>
        <OutArgument
            x:TypeArguments="x:Boolean">
            <VariableReference

```

```

        x>TypeArguments="x:Boolean">
    <VariableReference.Result>
        <OutArgument
            x>TypeArguments="Location(x:Boolean)" />
    </VariableReference.Result>
    <VariableReference.Variable>
        <Variable
            x>TypeArguments="x:Boolean"
            x>Name=__ReferenceID1"
            Name="Result" />
    </VariableReference.Variable>
    </VariableReference>
</OutArgument>
</RemoveFromCollection.Result>
<InArgument
    x>TypeArguments="scg:ICollection(x:String)">
<VariableValue
    x>TypeArguments="scg:ICollection(x:String)"
    Variable="{x:Reference __ReferenceID0}">
<VariableValue.Result>
    <OutArgument
        x>TypeArguments="scg:ICollection(x:String)" />
    </VariableValue.Result>
</VariableValue>
</InArgument>
</RemoveFromCollection>
<If>
    <If.Condition>
        <InArgument
            x>TypeArguments="x:Boolean">
        <VariableValue
            x>TypeArguments="x:Boolean"
            Variable="{x:Reference __ReferenceID1}">
        <VariableValue.Result>
            <OutArgument
                x>TypeArguments="x:Boolean" />
        </VariableValue.Result>
        </VariableValue>
        </InArgument>
    </If.Condition>
    <If.Then>
        <ClearCollection
            x>TypeArguments="x:String">
        <InArgument
            x>TypeArguments="scg:ICollection(x:String)">
        <VariableValue
            x>TypeArguments="scg:ICollection(x:String)"
            Variable="{x:Reference __ReferenceID0}">
        <VariableValue.Result>
            <OutArgument
                x>TypeArguments="scg:ICollection(x:String)" />
            </VariableValue.Result>
        </VariableValue>
        </InArgument>
    </ClearCollection>
    </If.Then>
</If>
</Sequence>

```

## See also

- [Authoring Workflows, Activities, and Expressions Using Imperative Code](#)

# Error Handling Activities in WF

3/9/2019 • 2 minutes to read • [Edit Online](#)

.NET Framework 4.6.1 provides several system-provided activities for implementing error handling and recovery. For more information, see [Exceptions](#).

## Error handling activities

| <a href="#">Rethrow</a>  | Rethrows the last exception thrown from within a <code>TryCatch</code> activity. |
|--------------------------|--|
| <a href="#">Throw</a>    | Throws an exception.   |
| <a href="#">TryCatch</a> | Implements exception handling.   |

# Migration Activity in WF

3/9/2019 • 2 minutes to read • [Edit Online](#)

.NET Framework 4.6.1 provides the [Interop](#) activity for executing activities that derive from [Activity](#) within a workflow that is based on [Activity](#). For more information, see the [Migration Guidance](#) section.

## NOTE

The [Interop](#) activity does not appear in the workflow designer toolbox unless the workflow's project has its **Target Framework** setting set to **.Net Framework 4** or higher.

# Designing and Implementing Custom Activities

3/9/2019 • 2 minutes to read • [Edit Online](#)

Custom activities in .NET Framework 4.6.1 are created by either assembling system-provided activities into composite activities or by creating new types that derive from [CodeActivity](#), [AsyncCodeActivity](#), or [NativeActivity](#). This section describes how to create custom activities with either method.

## IMPORTANT

Custom activities by default display within the workflow designer as a simple rectangle with the activity's name. To provide a custom visual representation of your activity in the workflow designer you must also create a custom designer. For more information, see [Using Custom Activity Designers and Templates](#).

## In This Section

### [Activity Authoring Options](#)

Discusses the authoring styles available in .NET Framework 4.6.1.

### [Using a custom activity](#)

Describes how to add a custom activity to a workflow project.

### [Creating Asynchronous Activities](#)

Describes how to create asynchronous activities.

### [Configuring Activity Validation](#)

Describes how activity validation can be used to identify and report errors in an activity's configuration prior to its execution.

### [Creating an Activity at Runtime](#)

Discusses how to create activities at runtime using [DynamicActivity](#).

### [Workflow Execution Properties](#)

Describes how to use workflow execution properties to add context specific properties to an activity's environment

### [Using Activity Delegates](#)

Discusses how to author and use activities that contain activity delegates.

### [Using Activity Extensions](#)

Describes how to author and use activity extensions.

### [Consuming OData Feeds from a Workflow](#)

Describes several methods for calling a WCF Data Service from a workflow.

### [Activity Definition Scoping and Visibility](#)

Describes the options and rules for defining data scoping and member visibility for activities.

# Activity Authoring Options in WF

9/19/2018 • 2 minutes to read • [Edit Online](#)

.NET Framework 4.6.1 provides several options for creating custom activities. The correct method to use for authoring a given activity depends on what run-time features are required.

## Deciding Which Base Activity Class to Use for Authoring Custom Activities

The following table lists the features available in the custom activity base classes.

| BASE ACTIVITY CLASS             | FEATURES AVAILABLE  |
|---------------------------------|---|
| <a href="#">Activity</a>        | Composes groups of system-provided and custom activities into a composite activity.   |
| <a href="#">CodeActivity</a>    | Implements imperative functionality by providing an <a href="#">Execute</a> method that can be overridden. Also provides access to tracking, variables, and arguments..   |
| <a href="#">NativeActivity</a>  | Provides all of the features of <a href="#">CodeActivity</a> , plus aborting activity execution, canceling child activity execution, using bookmarks, and scheduling activities, activity actions, and functions.                         |
| <a href="#">DynamicActivity</a> | Provides a DOM-like approach to constructing activities that interfaces with the WF designer and the run-time machinery through <a href="#">ICustomTypeDescriptor</a> , allowing new activities to be created without defining new types. |

## Authoring Activities using Activity

Activities that derive from [Activity](#) compose functionality by assembling other existing activities. These activities can be existing custom activities and activities from the .NET Framework 4.6.1 activity library. Assembling these activities is the most basic way to create custom functionality. This approach is most typically taken when using a visual design environment for authoring workflows.

## Authoring Activities using CodeActivity or AsyncCodeActivity

Activities that derive from [CodeActivity](#) or [AsyncCodeActivity](#) can implement imperative functionality by overriding the [Execute](#) method with custom imperative code. The custom code is executed when the activity is executed by the runtime. While activities created in this way have access to custom functionality, they do not have access to all of the features of the runtime, such as full access to the execution environment, the ability to schedule child activities, bookmark creation, or support for a Cancel or Abort method. When a [CodeActivity](#) executes, it has access to a reduced version of the execution environment (through the [CodeActivityContext](#) or [AsyncCodeActivityContext](#) class). Activities created using [CodeActivity](#) have access to argument and variable resolution, extensions, and tracking. Asynchronous activity scheduling can be done using [AsyncCodeActivity](#).

## Authoring Activities using NativeActivity

Activities that derive from [NativeActivity](#), like those that derive from [CodeActivity](#), create imperative functionality by overriding [Execute](#), but also have access to all of the functionality of the workflow runtime through the [NativeActivityContext](#) that gets passed into the [Execute](#) method. This context has support for scheduling and canceling child activities, executing [ActivityAction](#) and [ActivityFunc<TResult>](#) objects, flowing transactions into a workflow, invoking asynchronous processes, canceling and aborting execution, access to execution properties and extensions, and bookmarks (handles for resuming paused workflows).

## Authoring Activities using DynamicActivity

Unlike the other three types of activity, new functionality is not created by deriving new types from [DynamicActivity](#) (the class is sealed), but instead, by assembling functionality into the [Properties](#) and [Implementation](#) properties using an activity document object model (DOM).

## Authoring Activities that Return a Result

Many activities must return a result after their execution. Although it is possible to always define a custom [OutArgument<T>](#) on an activity for this purpose, it is suggested to instead use [Activity<TResult>](#), or derive from [CodeActivity<TResult>](#) or [NativeActivity<TResult>](#). Each of these base classes has an [OutArgument<T>](#) named [Result](#) that your activity can use for its return value. Activities that return a result should only be used if only one result needs to be returned from an activity; if multiple results need to be returned, separate [OutArgument<T>](#) members should be used instead.

# Workflow Activity Authoring Using the Activity Class

3/9/2019 • 2 minutes to read • [Edit Online](#)

The most basic way to create an activity using Windows Workflow Foundation (WF) in .NET Framework 4.6.1 is to create a class that inherits from [Activity](#) that creates functionality by assembling custom activities or activities from the [Built-In Activity Library](#). This topic demonstrates how to create an activity that writes two messages to the console.

## To create a custom Activity using the activity designer

1. Open Visual Studio 2012.
2. Select File, New, Project. Select **Workflow 4.0** under **Visual C#** in the **Project Types** window, and select the **v2010** node. Select **Activity Library** in the **Templates** window. Name the new project HelloActivity.
3. Open the new activity. Drag a [Sequence](#) activity from the toolbox onto the designer surface.
4. Drag a [WriteLine](#) activity into the [Sequence](#) activity. Enter `"Hello World"` (with quotes) into the **Text** field.
5. Drag a second [WriteLine](#) activity into the [Sequence](#) activity, below the first one. Enter `"Goodbye"` (with quotes) into the **Text** field.

# Workflow Activity Authoring Using the CodeActivity Class

10/3/2018 • 2 minutes to read • [Edit Online](#)

Activities created by inheriting from [CodeActivity](#) can implement basic imperative behavior by overriding the [Execute](#) method.

## Using CodeActivityContext

Features of the workflow runtime can be accessed from within the [Execute](#) method by using members of the `context` parameter, of type [CodeActivityContext](#). The features available through [CodeActivityContext](#) include the following:

- Getting and setting the values of variables and arguments.
- Custom tracking features using [Track](#).
- Access to the activity's execution properties using [GetProperty](#).

### To create a custom activity that inherits from CodeActivity

1. Open Visual Studio 2010.
2. Select **File**, **New**, and then **Project**. Select **Workflow 4.0** under **Visual C#** in the **Project Types** window, and select the **v2010** node. Select **Activity Library** in the **Templates** window. Name the new project HelloActivity.
3. Right-click Activity1.xaml in the HelloActivity project and select **Delete**.
4. Right-click the HelloActivity project and select **Add**, and then **Class**. Name the new class HelloActivity.cs.
5. In the HelloActivity.cs file, add the following `using` directives.

```
using System.Activities;
using System.Activities.Statements;
```

6. Make the new class inherit from [CodeActivity](#) by adding a base class to the class declaration.

```
class HelloActivity : CodeActivity
```

7. Add functionality to the class by adding an [Execute](#) method.

```
protected override void Execute(CodeActivityContext context)
{
    Console.WriteLine("Hello World!");
}
```

8. Use the [CodeActivityContext](#) to create a tracking record.

```
protected override void Execute(CodeActivityContext context)
{
    Console.WriteLine("Hello World!");
    CustomTrackingRecord record = new CustomTrackingRecord("MyRecord");
    record.Data.Add(new KeyValuePair<String, Object>("ExecutionTime", DateTime.Now));
    context.Track(record);
}
```

# NativeActivity Base Class

10/3/2018 • 2 minutes to read • [Edit Online](#)

[NativeActivity](#) is an abstract class with a protected constructor. Like [CodeActivity](#), [NativeActivity](#) is used for writing imperative behavior by implementing an [Execute](#) method. Unlike [CodeActivity](#), [NativeActivity](#) has access to all of the exposed features of the workflow runtime through the [NativeActivityContext](#) object passed to the [Execute](#) method.

## Using NativeActivityContext

Features of the workflow runtime can be accessed from within the [Execute](#) method by using members of the [context](#) parameter, of type [NativeActivityContext](#). The features available through [NativeActivityContext](#) include the following:

- Getting and setting of arguments and variables.
- Scheduling child activities with [ScheduleActivity](#)
- Aborting activity execution using [Abort](#).
- Canceling child execution using [CancelChild](#) and [CancelChildren](#).
- Access to activity bookmarks using such methods as [CreateBookmark](#), [RemoveBookmark](#), and [ResumeBookmark](#).
- Custom tracking features using [Track](#).
- Access to the activity's execution properties and value properties using [GetProperty](#) and [GetValue](#).
- Scheduling activity actions and functions using [ScheduleAction](#) and [ScheduleFunc](#).

### To create a custom activity that inherits from NativeActivity

1. Open Visual Studio 2010.
2. Select **File**, **New**, and then **Project**. Select **Workflow 4.0** under **Visual C#** in the **Project Types** window, and select the **v2010** node. Select **Activity Library** in the **Templates** window. Name the new project HelloActivity.
3. Right-click Activity1.xaml in the HelloActivity project and select **Delete**.
4. Right-click the HelloActivity project and select **Add**, and then **Class**. Name the new class HelloActivity.cs.
5. In the HelloActivity.cs file, add the following [using](#) directives.

```
using System.Activities;
using System.Activities.Statements;
```

6. Make the new class inherit from [NativeActivity](#) by adding a base class to the class declaration.

```
class HelloActivity : NativeActivity
```

7. Add functionality to the class by adding an [Execute](#) method.

```
protected override void Execute(NativeActivityContext context)
{
    Console.WriteLine("Hello World!");
}
```

8. Override the [CacheMetadata](#) method and call the appropriate Add method to let the workflow runtime know about the custom activity's variables, arguments, children, and delegates. For more information see the [NativeActivityMetadata](#) class.
9. Use the [NativeActivityContext](#) object to schedule a bookmark. See [Bookmarks](#) for details on how to create, schedule, and resume a bookmark.

```
protected override void Execute(NativeActivityContext context)
{
    // Create a Bookmark and wait for it to be resumed.
    context.CreateBookmark(BookmarkName.Get(context),
        new BookmarkCallback(OnResumeBookmark));
}
```

# Using a custom activity

5/4/2018 • 2 minutes to read • [Edit Online](#)

Activities that derive from [Activity](#) or its subclasses can be composed into larger workflows, or created directly in code. This topic describes how to use custom activities in workflows created either in code or in the designer.

## NOTE

Custom activities can be used in the same project in which they are defined, as long as both the custom activity and the activity that uses it are compiled (i.e. loaded by an instantiating type generated by the build process). If the referencing activity is loaded dynamically (e.g. using `ActivityXAMLServices`), then the referenced assembly should be placed in a different project, or the designer-generated XAML needs to be hand-edited to enable this.

### Using a custom activity to a workflow project

1. Add a reference from the host project to the activity library project containing the custom activity.
2. Build the solution.
3. To use the custom activity in the designer, locate the custom activity in the toolbox, and drag the activity onto the designer surface.
4. To use the custom activity in code, add a `Using` statement that refers to the custom activity project, and pass a new instance of the activity to [Invoke](#).

# Creating Asynchronous Activities in WF

3/9/2019 • 6 minutes to read • [Edit Online](#)

[AsyncCodeActivity](#) provides activity authors a base class to use that enables derived activities to implement asynchronous execution logic. This is useful for custom activities that must perform asynchronous work without holding the workflow scheduler thread and blocking any activities that may be able to run in parallel. This topic provides an overview of how to create custom asynchronous activities using [AsyncCodeActivity](#).

## Using AsyncCodeActivity

[System.Activities](#) provides custom activity authors with different base classes for different activity authoring requirements. Each one carries a particular semantic and provides a workflow author (and the activity runtime) a corresponding contract. An [AsyncCodeActivity](#) based activity is an activity that performs work asynchronously relative to the scheduler thread and whose execution logic is expressed in managed code. As a result of going asynchronous, an [AsyncCodeActivity](#) may induce an idle point during execution. Due to the volatile nature of asynchronous work, an [AsyncCodeActivity](#) always creates a no persist block for the duration of the activity's execution. This prevents the workflow runtime from persisting the workflow instance in the middle of the asynchronous work, and also prevents the workflow instance from unloading while the asynchronous code is executing.

### AsyncCodeActivity Methods

Activities that derive from [AsyncCodeActivity](#) can create asynchronous execution logic by overriding the [BeginExecute](#) and [EndExecute](#) methods with custom code. When called by the runtime, these methods are passed an [AsyncCodeActivityContext](#). [AsyncCodeActivityContext](#) allows the activity author to provide shared state across [BeginExecute](#)/ [EndExecute](#) in the context's [UserState](#) property. In the following example, a `GenerateRandom` activity generates a random number asynchronously.

```

public sealed class GenerateRandom : AsyncCodeActivity<int>
{
    static Random r = new Random();

    protected override IAsyncResult BeginExecute(AsyncCodeActivityContext context, AsyncCallback callback,
object state)
    {
        // Create a delegate that references the method that implements
        // the asynchronous work. Assign the delegate to the UserState,
        // invoke the delegate, and return the resulting IAsyncResult.
        Func<int> GetRandomDelegate = new Func<int>(GetRandom);
        context.UserState = GetRandomDelegate;
        return GetRandomDelegate.BeginInvoke(callback, state);
    }

    protected override int EndExecute(AsyncCodeActivityContext context, IAsyncResult result)
    {
        // Get the delegate from the UserState and call EndInvoke
        Func<int> GetRandomDelegate = (Func<int>)context.UserState;
        return (int)GetRandomDelegate.EndInvoke(result);
    }

    int GetRandom()
    {
        // This activity simulates taking a few moments
        // to generate the random number. This code runs
        // asynchronously with respect to the workflow thread.
        Thread.Sleep(5000);

        return r.Next(1, 101);
    }
}

```

The previous example activity derives from `AsyncCodeActivity<TResult>`, and has an elevated `OutArgument<int>` named `Result`. The value returned by the `GetRandom` method is extracted and returned by the `EndExecute` override, and this value is set as the `Result` value. Asynchronous activities that do not return a result should derive from `AsyncCodeActivity`. In the following example, a `DisplayRandom` activity is defined which derives from `AsyncCodeActivity`. This activity is like the `GetRandom` activity but instead of returning a result it displays a message to the console.

```

public sealed class DisplayRandom : AsyncCodeActivity
{
    static Random r = new Random();

    protected override IAsyncResult BeginExecute(AsyncCodeActivityContext context, AsyncCallback callback,
object state)
    {
        // Create a delegate that references the method that implements
        // the asynchronous work. Assign the delegate to the UserState,
        // invoke the delegate, and return the resulting IAsyncResult.
        Action GetRandomDelegate = new Action(GetRandom);
        context.UserState = GetRandomDelegate;
        return GetRandomDelegate.BeginInvoke(callback, state);
    }

    protected override void EndExecute(AsyncCodeActivityContext context, IAsyncResult result)
    {
        // Get the delegate from the UserState and call EndInvoke
        Action GetRandomDelegate = (Action)context.UserState;
        GetRandomDelegate.EndInvoke(result);
    }

    void GetRandom()
    {
        // This activity simulates taking a few moments
        // to generate the random number. This code runs
        // asynchronously with respect to the workflow thread.
        Thread.Sleep(5000);

        Console.WriteLine("Random Number: {0}", r.Next(1, 101));
    }
}

```

Note that because there is no return value, `DisplayRandom` uses an `Action` instead of a `Func<T,TResult>` to invoke its delegate, and the delegate returns no value.

`AsyncCodeActivity` also provides a `Cancel` override. While `BeginExecute` and `EndExecute` are required overrides, `Cancel` is optional, and can be overridden so the activity can clean up its outstanding asynchronous state when it is being canceled or aborted. If clean up is possible and

`AsyncCodeActivity.ExecutingActivityInstance.IsCancellationRequested` is `true`, the activity should call `MarkCanceled`. Any exceptions thrown from this method are fatal to the workflow instance.

```

protected override void Cancel(AsyncCodeActivityContext context)
{
    // Implement any cleanup as a result of the asynchronous work
    // being canceled, and then call MarkCanceled.
    if (context.IsCancellationRequested)
    {
        context.MarkCanceled();
    }
}

```

## Invoking Asynchronous Methods on a Class

Many of the classes in the .NET Framework provide asynchronous functionality, and this functionality can be asynchronously invoked by using an `AsyncCodeActivity` based activity. In the following example, an activity is created that asynchronously creates a file by using the `FileStream` class.

```

public sealed class FileWriter : AsyncCodeActivity
{
    public FileWriter()
        : base()
    {
    }

    protected override IAsyncResult BeginExecute(AsyncCodeActivityContext context, AsyncCallback callback,
object state)
    {
        string tempFileName = Path.GetTempFileName();
        Console.WriteLine("Writing to file: " + tempFileName);

        FileStream file = File.Open(tempFileName, FileMode.Create);

        context.UserState = file;

        byte[] bytes = UnicodeEncoding.Unicode.GetBytes("123456789");
        return file.BeginWrite(bytes, 0, bytes.Length, callback, state);
    }

    protected override void EndExecute(AsyncCodeActivityContext context, IAsyncResult result)
    {
        FileStream file = (FileStream)context.UserState;

        try
        {
            file.EndWrite(result);
            file.Flush();
        }
        finally
        {
            file.Close();
        }
    }
}

```

## Sharing State Between the `BeginExecute` and `EndExecute` Methods

In the previous example, the `FileStream` object that was created in `BeginExecute` was accessed in the `EndExecute`. This is possible because the `file` variable was passed in the `AsyncCodeActivityContext.UserState` property in `BeginExecute`. This is the correct method for sharing state between `BeginExecute` and `EndExecute`. It is incorrect to use a member variable in the derived class (`FileWriter` in this case) to share state between `BeginExecute` and `EndExecute` because the activity object may be referenced by multiple activity instances. Attempting to use a member variable to share state can result in values from one `ActivityInstance` overwriting or consuming values from another `ActivityInstance`.

## Accessing Argument Values

The environment of an `AsyncCodeActivity` consists of the arguments defined on the activity. These arguments can be accessed from the `BeginExecute/EndExecute` overrides using the `AsyncCodeActivityContext` parameter. The arguments cannot be accessed in the delegate, but the argument values or any other desired data can be passed in to the delegate using its parameters. In the following example, a random number-generating activity is defined that obtains the inclusive upper bound from its `Max` argument. The value of the argument is passed in to the asynchronous code when the delegate is invoked.

```

public sealed class GenerateRandomMax : AsyncCodeActivity<int>
{
    public InArgument<int> Max { get; set; }

    static Random r = new Random();

    protected override IAsyncResult BeginExecute(AsyncCodeActivityContext context, AsyncCallback callback,
object state)
    {
        // Create a delegate that references the method that implements
        // the asynchronous work. Assign the delegate to the UserState,
        // invoke the delegate, and return the resulting IAsyncResult.
        Func<int, int> GetRandomDelegate = new Func<int, int>(GetRandom);
        context.UserState = GetRandomDelegate;
        return GetRandomDelegate.BeginInvoke(Max.Get(context), callback, state);
    }

    protected override int EndExecute(AsyncCodeActivityContext context, IAsyncResult result)
    {
        // Get the delegate from the UserState and call EndInvoke
        Func<int, int> GetRandomDelegate = (Func<int, int>)context.UserState;
        return (int)GetRandomDelegate.EndInvoke(result);
    }

    int GetRandom(int max)
    {
        // This activity simulates taking a few moments
        // to generate the random number. This code runs
        // asynchronously with respect to the workflow thread.
        Thread.Sleep(5000);

        return r.Next(1, max + 1);
    }
}

```

## Scheduling Actions or Child Activities Using `AsyncCodeActivity`

`AsyncCodeActivity` derived custom activities provide a method for performing work asynchronously with regard to the workflow thread, but do not provide the ability to schedule child activities or actions. However, asynchronous behavior can be incorporated with scheduling of child activities through composition. An asynchronous activity can be created, and then composed with an [Activity](#) or [NativeActivity](#) derived activity to provide asynchronous behavior and scheduling of child activities or actions. For example, an activity could be created that derives from [Activity](#), and has as its implementation a [Sequence](#) containing the asynchronous activity as well the other activities that implement the logic of the activity. For more examples of composing activities using [Activity](#) and [NativeActivity](#), see [How to: Create an Activity](#) and [Activity Authoring Options](#).

## See also

- [Action](#)
- [Func<T,TResult>](#)

# Error handling in asynchronous activities

5/4/2018 • 2 minutes to read • [Edit Online](#)

Providing error handling in an [AsyncCodeActivity](#) involves routing the error through the activity's callback system. This topic describes how to get an error that is thrown in an asynchronous operation back to the host, using the SendMail activity sample.

## Returning an error thrown in an asynchronous activity back to the host

Routing an error in an asynchronous operation back to the host in the SendMail activity sample involves the following steps:

- Add an Exception property to the `SendMailAsyncResult` class.
- Copy the thrown error to that property in the `SendCompleted` event handler.
- Throw the exception in the `EndExecute` event handler.

The resulting code is as follows.

```
class SendMailAsyncResult : IAsyncResult
{
    ...
    public Exception Error { get; set; }
    ...
    void SendCompleted(object sender, AsyncCompletedEventArgs e)
    {
        Error = e.Error;
        this.asyncWaitHandle.Set();
        callback(this);
    }
}

public sealed class SendMail: AsyncCodeActivity
{
    ...
    protected override void EndExecute(AsyncCodeActivityContext context, IAsyncResult result)
    {
        SendMailAsyncResult sendMailResult = result as SendMailAsyncResult;
        if (sendMailResult != null && sendMailResult.Error != null)
            throw sendMailResult.Error;
    }
}
```

# Configuring Activity Validation

3/9/2019 • 2 minutes to read • [Edit Online](#)

Activity validation enables activity authors and users to identify and report errors in an activity's configuration prior to its execution. Windows Workflow Foundation (WF) provides the following three types of activity validation:

- `RequiredArgument` and `OverloadGroup` attributes.
- Imperative code-based validation.
- Declarative constraints.

`RequiredArgument` and `OverloadGroup` attributes indicate that certain arguments on an activity are required.

Imperative code-based validation provides a simple way for an activity to provide validation about itself, and declarative constraints enable validation about the activity and its relationship with the containing workflow. If an activity is not configured properly according to the validation requirements, validation errors and warnings are returned. If the containing workflow is created using the workflow designer, any validation errors and warnings are displayed in the designer. If the workflow is created outside of the workflow designer any validation errors are returned when the workflow is invoked. Regardless of how the workflow was created, a workflow with validation errors is never allowed to execute. This section provides an overview of these types of activity validation and how activity validation is invoked.

## In This Section

### [Required Arguments and Overload Groups](#)

Describes how to use the `RequiredArgument` and `OverloadGroup` attributes to provide validation.

### [Imperative Code-Based Validation](#)

Describes how to use code-based validation for `CodeActivity` and `NativeActivity` based activities.

### [Declarative Constraints](#)

Describes how to use declarative constraints to provide complex activity validation.

### [Invoking Activity Validation](#)

Discusses when activity validation is invoked automatically and how to explicitly invoke validation.

## Reference

## Related Sections

# Required Arguments and Overload Groups

3/9/2019 • 4 minutes to read • [Edit Online](#)

Activities can be configured so that certain arguments are required to be bound for the activity to be valid for execution. The `RequiredArgument` attribute is used to indicate that certain arguments on an activity are required and the `OverloadGroup` attribute is used to group categories of required arguments together. By using the attributes, activity authors can provide simple or complex activity validation configurations.

## Using Required Arguments

To use the `RequiredArgument` attribute in an activity, indicate the desired arguments using `RequiredArgumentAttribute`. In this example, an `Add` activity is defined that has two required arguments.

```
public sealed class Add : CodeActivity<int>
{
    [RequiredArgument]
    public InArgument<int> Operand1 { get; set; }

    [RequiredArgument]
    public InArgument<int> Operand2 { get; set; }

    protected override int Execute(CodeActivityContext context)
    {
        return Operand1.Get(context) + Operand2.Get(context);
    }
}
```

In XAML, required arguments are also indicated by using `RequiredArgumentAttribute`. In this example the `Add` activity is defined by using three arguments and uses an `Assign<T>` activity to perform the add operation.

```
<Activity x:Class="ValidationDemo.Add" ...>
<x:Members>
    <x:Property Name="Operand1" Type="InArgument(x:Int32)">
        <x:Property.Attributes>
            <RequiredArgumentAttribute />
        </x:Property.Attributes>
    </x:Property>
    <x:Property Name="Operand2" Type="InArgument(x:Int32)">
        <x:Property.Attributes>
            <RequiredArgumentAttribute />
        </x:Property.Attributes>
    </x:Property>
    <x:Property Name="Result" Type="OutArgument(x:Int32)" />
</x:Members>
<Assign>
    <Assign.To>
        <OutArgument x:TypeArguments="x:Int32">[Result]</OutArgument>
    </Assign.To>
    <Assign.Value>
        <InArgument x:TypeArguments="x:Int32">[Operand1 + Operand2]</InArgument>
    </Assign.Value>
</Assign>
</Activity>
```

If the activity is used and either of the required arguments is not bound the following validation error is returned.

## Value for a required activity argument 'Operand1' was not supplied.

### NOTE

For more information about checking for and handling validation errors and warnings, see [Invoking Activity Validation](#).

## Using Overload Groups

Overload groups provide a method for indicating which combinations of arguments are valid in an activity. Arguments are grouped together by using [OverloadGroupAttribute](#). Each group is given a name that is specified by the [OverloadGroupAttribute](#). The activity is valid when only one set of arguments in an overload group are bound. In the following example, a `CreateLocation` class is defined.

```
class CreateLocation: Activity
{
    [RequiredArgument]
    public InArgument<string> Name { get; set; }

    public InArgument<string> Description { get; set; }

    [RequiredArgument]
    [OverloadGroup("G1")]
    public InArgument<int> Latitude { get; set; }

    [RequiredArgument]
    [OverloadGroup("G1")]
    public InArgument<int> Longitude { get; set; }

    [RequiredArgument]
    [OverloadGroup("G2")]
    [OverloadGroup("G3")]
    public InArgument<string> Street { get; set; }

    [RequiredArgument]
    [OverloadGroup("G2")]
    public InArgument<string> City { get; set; }

    [RequiredArgument]
    [OverloadGroup("G2")]
    public InArgument<string> State { get; set; }

    [RequiredArgument]
    [OverloadGroup("G3")]
    public InArgument<int> Zip { get; set; }
}
```

The objective of this activity is to specify a location in the US. To do this, the user of the activity can specify the location using one of three groups of arguments. To specify the valid combinations of arguments, three overload groups are defined. `G1` contains the `Latitude` and `Longitude` arguments. `G2` contains `Street`, `City`, and `State`. `G3` contains `Street` and `Zip`. `Name` is also a required argument, but it is not part of an overload group. For this activity to be valid, `Name` would have to be bound together with all of the arguments from one and only one overload group.

In the following example, taken from the [Database Access Activities](#) sample, there are two overload groups:

`ConnectionString` and `ConfigFileSectionName`. For this activity to be valid, either the `ProviderName` and `ConnectionString` arguments must be bound, or the `ConfigName` argument, but not both.

```

Public class DbUpdate: AsyncCodeActivity
{
    [RequiredArgument]
    [OverloadGroup("ConnectionString")]
    [DefaultValue(null)]
    public InArgument<string> ProviderName { get; set; }

    [RequiredArgument]
    [OverloadGroup("ConnectionString")]
    [DependsOn("ProviderName")]
    [DefaultValue(null)]
    public InArgument<string> ConnectionString { get; set; }

    [RequiredArgument]
    [OverloadGroup("ConfigFileSectionName")]
    [DefaultValue(null)]
    public InArgument<string> ConfigName { get; set; }

    [DefaultValue(null)]
    public CommandType CommandType { get; set; }

    [RequiredArgument]
    public InArgument<string> Sql { get; set; }

    [DependsOn("Sql")]
    [DefaultValue(null)]
    public IDictionary<string, Argument> Parameters { get; }

    [DependsOn("Parameters")]
    public OutArgument<int> AffectedRecords { get; set; }
}

```

When defining an overload group:

- An overload group cannot be a subset or an equivalent set of another overload group.

#### NOTE

There is one exception to this rule. If an overload group is a subset of another overload group, and the subset contains only arguments where `RequiredArgument` is `false`, then the overload group is valid.

- Overload groups can overlap but it is an error if the intersection of the groups contains all the required arguments of one or both of the overload groups. In the previous example the `G2` and `G3` overload groups overlapped, but because the intersection did not contain all the arguments of one or both of the groups this was valid.

When binding arguments in an overload group:

- An overload group is considered bound if all the `RequiredArgument` arguments in the group are bound.
- If a group has zero `RequiredArgument` arguments and at least one argument bound, then the group is considered bound.
- It is a validation error if no overload groups are bound unless one overload group has no `RequiredArgument` arguments in it.
- It is an error to have more than one overload group bound, that is, all required arguments in one overload group are bound and any argument in another overload group is also bound.

# Imperative Code-Based Validation

3/9/2019 • 2 minutes to read • [Edit Online](#)

Imperative code-based validation provides a simple way for an activity to provide validation about itself, and is available for activities that derive from [CodeActivity](#), [AsyncCodeActivity](#), and [NativeActivity](#). Validation code that determines any validation errors or warnings is added to the activity.

## Using Code-Based Validation

Code-based validation is supported by activities that derive from [CodeActivity](#), [AsyncCodeActivity](#), and [NativeActivity](#). Validation code can be placed in the [CacheMetadata](#) override, and validation errors or warnings can be added to the metadata argument. In the following example, if the `Cost` is greater than the `Price`, a validation error is added to the metadata.

### NOTE

Note that `Cost` and `Price` are not arguments to the activity, but are properties that are set at design time. That is why their values can be validated in the [CacheMetadata](#) override. The value of the data flowing through an argument cannot be validated at design time because the data does not flow until run time, but activity arguments can be validated to ensure that they are bound by using the [RequiredArgument](#) attribute and overload groups. This example code sees the [RequiredArgument](#) attribute for the `Description` argument, and if it is not bound then a validation error is generated. Required arguments are covered in [Required Arguments and Overload Groups](#).

```
public sealed class CreateProduct : CodeActivity
{
    public double Price { get; set; }
    public double Cost { get; set; }

    // [RequiredArgument] attribute will generate a validation error
    // if the Description argument is not set.
    [RequiredArgument]
    public InArgument<string> Description { get; set; }

    protected override void CacheMetadata(CodeActivityMetadata metadata)
    {
        base.CacheMetadata(metadata);
        // Determine when the activity has been configured in an invalid way.
        if (this.Cost > this.Price)
        {
            // Add a validation error with a custom message.
            metadata.AddValidationError("The Cost must be less than or equal to the Price.");
        }
    }

    protected override void Execute(CodeActivityContext context)
    {
        // Not needed for the sample.
    }
}
```

By default, a validation error is added to the metadata when [AddValidationError](#) is called. To add a validation warning, use the [AddValidationError](#) overload that takes a [ValidationWarning](#), and specify that the [ValidationWarning](#) represents a warning by setting the [IsWarning](#) property.

Validation occurs when a workflow is modified in the workflow designer and any validation errors or warnings are displayed in the workflow designer. Validation also occurs at run time when a workflow is invoked and if any validation errors occur, an [InvalidWorkflowException](#) is thrown by the default validation logic. For more information about invoking validation and accessing any validation warnings or errors, see [Invoking Activity Validation](#).

Any exceptions that are thrown from [CacheMetadata](#) are not treated as validation errors. These exceptions will escape from the call to [Validate](#) and must be handled by the caller.

Code-based validation is useful for validating the activity that contains the code, but it does not have visibility into the other activities in the workflow. Declarative constraints validation provides the ability to validate the relationships between an activity and other activities in the workflow, and is covered in the [Declarative Constraints](#) topic.

# Declarative Constraints

3/9/2019 • 4 minutes to read • [Edit Online](#)

Declarative constraints provide a powerful method of validation for an activity and its relationships with other activities. Constraints are configured for an activity during the authoring process, but additional constraints can also be specified by the workflow host. This topic provides an overview of using declarative constraints to provide activity validation.

## Using Declarative Constraints

A constraint is an activity that contains validation logic. This constraint activity can be authored in code or in XAML. After a constraint activity is created, activity authors add this constraint to the [Constraints](#) property of the activity to validate, or they use the constraint to provide additional validation by using the [AdditionalConstraints](#) property of a [ValidationSettings](#) instance. The validation logic can consist of simple validations such as validating an activity's metadata, but it can also perform validation that takes into account the relationship of the current activity to its parent, children, and sibling activities. Constraints are authored by using the [Constraint<T>](#) activity, and several additional validation activities are provided to assist with the creation of validation errors and warnings and to provide information about related activities in the workflow.

### AssertValidation and AddValidationError

The [AssertValidation](#) activity evaluates the expression referenced by its [Assertion](#) property, and if the expression evaluates to `false`, a validation error or warning is added to the [ValidationResults](#). The [Message](#) property describes the validation error and the [IsWarning](#) property indicates whether the validation failure is an error or a warning. The default value for [IsWarning](#) is `false`.

In the following example, a constraint is declared that returns a validation warning if the [DisplayName](#) of the activity being validated is two characters or less in length. The generic type parameter used for [Constraint<T>](#) specifies the type of activity that is validated by the constraint. This constraint uses [Activity](#) as the generic type and can be used to validate all types of activities.

```
public static Constraint ActivityDisplayNameIsNotSetWarning()
{
    DelegateInArgument<Activity> element = new DelegateInArgument<Activity>();

    return new Constraint<Activity>
    {
        Body = new ActivityAction<Activity, ValidationContext>
        {
            Argument1 = element,
            Handler = new AssertValidation
            {
                IsWarning = true,
                Assertion = new InArgument<bool>(env => (element.Get(env).DisplayName.Length > 2)),
                Message = new InArgument<string>("It is a best practice to have a DisplayName of more than 2
characters."),
            }
        };
    };
}
```

To specify this constraint for an activity, it is added to the [Constraints](#) of the activity, as shown in the following example code.

```

public sealed class SampleActivity : CodeActivity
{
    public SampleActivity()
    {
        base.Constraints.Add(ActivityDisplayNameIsNotSetWarning());
    }

    // Activity implementation omitted.
}

```

The host could also specify this constraint for activities in a workflow by using [AdditionalConstraints](#), which is covered in the next section.

The [AddValidationError](#) activity is used to generate a validation error or warning without requiring the evaluation of an expression. Its properties are similar to [AssertValidation](#) and it can be used in conjunction with flow control activities of a constraint such as the [If](#) activity.

## Workflow Relationship Activities

Several validation activities are available that provide information about the other activities in the workflow in relation to the activity being validated. [GetParentChain](#) returns a collection of activities that contains all of the activities between the current activity and the root activity. [GetChildSubtree](#) provides a collection of activities that contains the child activities in a recursive pattern, and [GetWorkflowTree](#) gets all the activities in the workflow.

In the following example, a [CreateState](#) activity is defined. The [CreateState](#) activity must be contained within a [CreateCountry](#) activity, and the [GetParent](#) method returns a constraint that enforces this requirement. [GetParent](#) uses the [GetParentChain](#) activity in conjunction with a [ForEach<T>](#) activity to inspect the parent activities of the [CreateState](#) activity to determine if the requirement is met.

```

public sealed class CreateState : CodeActivity
{
    public CreateState()
    {
        base.Constraints.Add(CheckParent());
        this.Cities = new List<Activity>();
    }

    public List<Activity> Cities { get; set; }

    public string Name { get; set; }

    static Constraint CheckParent()
    {
        DelegateInArgument<CreateState> element = new DelegateInArgument<CreateState>();
        DelegateInArgument<ValidationContext> context = new DelegateInArgument<ValidationContext>();
        Variable<bool> result = new Variable<bool>();
        DelegateInArgument<Activity> parent = new DelegateInArgument<Activity>();

        return new Constraint<CreateState>
        {
            Body = new ActivityAction<CreateState, ValidationContext>
            {
                Argument1 = element,
                Argument2 = context,
                Handler = new Sequence
                {
                    Variables =
                    {
                        result
                    },
                    Activities =
                    {
                        new ForEach<Activity>
                        {
                            Condition = new ExpressionActivity<bool>(
                                new AndExpressionActivity(
                                    new EqualExpressionActivity(
                                        new GetParentActivity(),
                                        element),
                                    new NotEqualExpressionActivity(
                                        new GetParentActivity(),
                                        parent)))
                        }
                    }
                }
            }
        };
    }
}

```

```

    {
        Values = new GetParentChain
        {
            ValidationContext = context
        },
        Body = new ActivityAction<Activity>
        {
            Argument = parent,
            Handler = new If()
            {
                Condition = new InArgument<bool>((env) =>
object.Equals(parent.Get(env).GetType(), typeof(CreateCountry))),
                Then = new Assign<bool>
                {
                    Value = true,
                    To = result
                }
            }
        },
        new AssertValidation
        {
            Assertion = new InArgument<bool>(result),
            Message = new InArgument<string> ("CreateState has to be inside a CreateCountry
activity"),
        }
    }
};

protected override void Execute(CodeActivityContext context)
{
    // not needed for the sample
}
}

```

## Additional Constraints

Workflow host authors can specify additional validation constraints for activities in a workflow by creating constraints and adding them to the [AdditionalConstraints](#) dictionary of a [ValidationSettings](#) instance. Each item in [AdditionalConstraints](#) contains the type of activity for which the constraints apply and a list of the additional constraints for that type of activity. When validation is invoked for the workflow, each activity of the specified type, including derived classes, evaluates the constraints. In this example, the [ActivityDisplayNameIsNotSetWarning](#) constraint from the previous section is applied to all activities in a workflow.

```

Activity wf = new Sequence
{
    // Workflow Details Omitted.
};

ValidationSettings settings = new ValidationSettings()
{

    AdditionalConstraints =
    {
        {typeof(Activity), new List<Constraint> {ActivityDisplayNameIsNotSetWarning()}},
    }
};

// Validate the workflow.
ValidationResults results = ActivityValidationServices.Validate(wf, settings);

// Evaluate the results.
if (results.Errors.Count == 0 && results.Warnings.Count == 0)
{
    Console.WriteLine("No warnings or errors");
}
else
{
    foreach (ValidationError error in results.Errors)
    {
        Console.WriteLine("Error in " + error.Source.DisplayName + ": " + error.Message);
    }
    foreach (ValidationError warning in results.Warnings)
    {
        Console.WriteLine("Warning in " + warning.Source.DisplayName + ": " + warning.Message);
    }
}

```

If the [OnlyUseAdditionalConstraints](#) property of [ValidationSettings](#) is `true`, then only the specified additional constraints are evaluated when validation is invoked by calling [Validate](#). This can be useful for inspecting workflows for specific validation configurations. Note however that when the workflow is invoked, the validation logic configured in the workflow is evaluated and must pass for the workflow to successfully begin. For more information about invoking validation, see [Invoking Activity Validation](#).

# Invoking Activity Validation

3/9/2019 • 8 minutes to read • [Edit Online](#)

Activity validation provides a method to identify and report errors in any activity's configuration prior to its execution. Validation occurs when a workflow is modified in the workflow designer and any validation errors or warnings are displayed in the workflow designer. Validation also occurs at run time when a workflow is invoked and if any validation errors occur, an `InvalidWorkflowException` is thrown by the default validation logic. Windows Workflow Foundation (WF) provides the `ActivityValidationServices` class that can be used by workflow application and tooling developers to explicitly validate an activity. This topic describes how to use `ActivityValidationServices` to perform activity validation.

## Using ActivityValidationServices

`ActivityValidationServices` has two `Validate` overloads that are used to invoke an activity's validation logic. The first overload takes the root activity to be validated and returns a collection of validation errors and warnings. In the following example, a custom `Add` activity is used that has two required arguments.

```
public sealed class Add : CodeActivity<int>
{
    [RequiredArgument]
    public InArgument<int> Operand1 { get; set; }

    [RequiredArgument]
    public InArgument<int> Operand2 { get; set; }

    protected override int Execute(CodeActivityContext context)
    {
        return Operand1.Get(context) + Operand2.Get(context);
    }
}
```

The `Add` activity is used inside a `Sequence`, but its two required arguments are not bound, as shown in the following example.

```
Variable<int> Operand1 = new Variable<int>{ Default = 10 };
Variable<int> Operand2 = new Variable<int>{ Default = 15 };
Variable<int> Result = new Variable<int>();

Activity wf = new Sequence
{
    Variables = { Operand1, Operand2, Result },
    Activities =
    {
        new Add(),
        new WriteLine
        {
            Text = new InArgument<string>(env => "The result is " + Result.Get(env))
        }
    }
};
```

This workflow can be validated by calling `Validate`. `Validate` returns a collection of any validation errors or warnings contained by the activity and any children, as shown in the following example.

```

ValidationResults results = ActivityValidationServices.Validate(wf);

if (results.Errors.Count == 0 && results.Warnings.Count == 0)
{
    Console.WriteLine("No warnings or errors");
}
else
{
    foreach (ValidationError error in results.Errors)
    {
        Console.WriteLine("Error: {0}", error.Message);
    }
    foreach (ValidationError warning in results.Warnings)
    {
        Console.WriteLine("Warning: {0}", warning.Message);
    }
}

```

When [Validate](#) is called on this sample workflow, two validation errors are returned.

**Error: Value for a required activity argument 'Operand2' was not supplied.**

**Error: Value for a required activity argument 'Operand1' was not supplied.** If this workflow was invoked, an [InvalidOperationException](#) would be thrown, as shown in the following example.

```

try
{
    WorkflowInvoker.Invoke(wf);
}
catch (Exception ex)
{
    Console.WriteLine(ex);
}

```

**System.Activities.InvalidWorkflowException:**

**The following errors were encountered while processing the workflow tree:**

**'Add': Value for a required activity argument 'Operand2' was not supplied.**

**'Add': Value for a required activity argument 'Operand1' was not supplied.** For this example workflow to be valid, the two required arguments of the `Add` activity must be bound. In the following example, the two required arguments are bound to workflow variables along with the result value. In this example the [Result](#) argument is bound along with the two required arguments. The [Result](#) argument is not required to be bound and does not cause a validation error if it is not. It is the responsibility of the workflow author to bind [Result](#) if its value is used elsewhere in the workflow.

```

new Add
{
    Operand1 = Operand1,
    Operand2 = Operand2,
    Result = Result
}

```

## Validating Required Arguments on the Root Activity

If the root activity of a workflow has arguments, these are not bound until the workflow is invoked and parameters are passed to the workflow. The following workflow passes validation, but an exception is thrown if the workflow is invoked without passing in the required arguments, as shown in the following example.

```

Activity wf = new Add();

ValidationResults results = ActivityValidationServices.Validate(wf);
// results has no errors or warnings, but when the workflow
// is invoked, an InvalidWorkflowException is thrown.
try
{
    WorkflowInvoker.Invoke(wf);
}
catch (Exception ex)
{
    Console.WriteLine(ex);
}

```

**System.ArgumentException: The root activity's argument settings are incorrect.**

**Either fix the workflow definition or supply input values to fix these errors:**

**'Add': Value for a required activity argument 'Operand2' was not supplied.**

**'Add': Value for a required activity argument 'Operand1' was not supplied.** After the correct arguments are passed, the workflow completes successfully, as shown in the following example.

```

Add wf = new Add();

ValidationResults results = ActivityValidationServices.Validate(wf);
// results has no errors or warnings, and the workflow completes
// successfully because the required arguments were passed.
try
{
    Dictionary<string, object> wfparams = new Dictionary<string, object>
    {
        { "Operand1", 10 },
        { "Operand2", 15 }
    };

    int result = WorkflowInvoker.Invoke(wf, wfparams);
    Console.WriteLine("Result: {0}", result);
}
catch (Exception ex)
{
    Console.WriteLine(ex);
}

```

#### NOTE

In this example, the root activity was declared as `Add` instead of `Activity` as in the previous example. This allows the `WorkflowInvoker.Invoke` method to return a single integer that represents the results of the `Add` activity instead of a dictionary of `out` arguments. The variable `wf` could also have been declared as `Activity<int>`.

When validating root arguments, it is the responsibility of the host application to ensure that all required arguments are passed when the workflow is invoked.

#### Invoking Imperative Code-Based Validation

Imperative code-based validation provides a simple way for an activity to provide validation about itself, and is available for activities that derive from [CodeActivity](#), [AsyncCodeActivity](#), and [NativeActivity](#). Validation code that determines any validation errors or warnings is added to the activity. When validation is invoked on the activity, these warnings or errors are contained in the collection returned by the call to [Validate](#). In the following example, a `CreateProduct` activity is defined. If the `Cost` is greater than the `Price`, a validation error is added to the metadata in the `CacheMetadata` override.

```

public sealed class CreateProduct : CodeActivity
{
    public double Price { get; set; }
    public double Cost { get; set; }

    // [RequiredArgument] attribute will generate a validation error
    // if the Description argument is not set.
    [RequiredArgument]
    public InArgument<string> Description { get; set; }

    protected override void CacheMetadata(CodeActivityMetadata metadata)
    {
        base.CacheMetadata(metadata);
        // Determine when the activity has been configured in an invalid way.
        if (this.Cost > this.Price)
        {
            // Add a validation error with a custom message.
            metadata.AddValidationError("The Cost must be less than or equal to the Price.");
        }
    }

    protected override void Execute(CodeActivityContext context)
    {
        // Not needed for the sample.
    }
}

```

In this example, a workflow is configured using the `CreateProduct` activity. In this workflow, the `Cost` is greater than the `Price`, and the required `Description` argument is not set. When validation is invoked, the following errors are returned.

```

Activity wf = new Sequence
{
    Activities =
    {
        new CreateProduct
        {
            Cost = 75.00,
            Price = 55.00
            // Cost > Price and required Description argument not set.
        },
        new WriteLine
        {
            Text = "Product added."
        }
    }
};

ValidationResults results = ActivityValidationServices.Validate(wf);

if (results.Errors.Count == 0 && results.Warnings.Count == 0)
{
    Console.WriteLine("No warnings or errors");
}
else
{
    foreach (ValidationError error in results.Errors)
    {
        Console.WriteLine("Error: {0}", error.Message);
    }
    foreach (ValidationError warning in results.Warnings)
    {
        Console.WriteLine("Warning: {0}", warning.Message);
    }
}

```

**Error: The Cost must be less than or equal to the Price.**

**Error: Value for a required activity argument 'Description' was not supplied.**

#### NOTE

Custom activity authors can provide validation logic in an activity's [CacheMetadata](#) override. Any exceptions that are thrown from [CacheMetadata](#) are not treated as validation errors. These exceptions will escape from the call to [Validate](#) and must be handled by the caller.

## Using ValidationSettings

By default, all activities in the activity tree are evaluated when validation is invoked by [ActivityValidationServices](#). [ValidationSettings](#) allows the validation to be customized in several different ways by configuring its three properties. [SingleLevel](#) specifies whether the validator should walk the entire activity tree or only apply validation logic to the supplied activity. The default for this value is `false`. [AdditionalConstraints](#) specifies additional constraint mapping from a type to a list of constraints. For the base type of each activity in the activity tree being validated there is a lookup into [AdditionalConstraints](#). If a matching constraint list is found, all constraints in the list are evaluated for the activity. [OnlyUseAdditionalConstraints](#) specifies whether the validator should evaluate all constraints or only those specified in [AdditionalConstraints](#). The default value is `false`. [AdditionalConstraints](#) and [OnlyUseAdditionalConstraints](#) are useful for workflow host authors to add additional validation for workflows, such as policy constraints for tools such as FxCop. For more information about constraints, see [Declarative Constraints](#).

To use [ValidationSettings](#), configure the desired properties, and then pass it in the call to [Validate](#). In this example,

a workflow that consists of a [Sequence](#) with a custom `Add` activity is validated. The `Add` activity has two required arguments.

```
public sealed class Add : CodeActivity<int>
{
    [RequiredArgument]
    public InArgument<int> Operand1 { get; set; }

    [RequiredArgument]
    public InArgument<int> Operand2 { get; set; }

    protected override int Execute(CodeActivityContext context)
    {
        return Operand1.Get(context) + Operand2.Get(context);
    }
}
```

The following `Add` activity is used in a [Sequence](#), but its two required arguments are not bound.

```
Variable<int> Operand1 = new Variable<int> { Default = 10 };
Variable<int> Operand2 = new Variable<int> { Default = 15 };
Variable<int> Result = new Variable<int>();

Activity wf = new Sequence
{
    Variables = { Operand1, Operand2, Result },
    Activities =
    {
        new Add(),
        new WriteLine
        {
            Text = new InArgument<string>(env => "The result is " + Result.Get(env))
        }
    }
};
```

For the following example, validation is performed with [SingleLevel](#) set to `true`, so only the root [Sequence](#) activity is validated.

```
ValidationSettings settings = new ValidationSettings
{
    SingleLevel = true
};

ValidationResults results = ActivityValidationServices.Validate(wf, settings);

if (results.Errors.Count == 0 && results.Warnings.Count == 0)
{
    Console.WriteLine("No warnings or errors");
}
else
{
    foreach (ValidationError error in results.Errors)
    {
        Console.WriteLine("Error: {0}", error.Message);
    }
    foreach (ValidationError warning in results.Warnings)
    {
        Console.WriteLine("Warning: {0}", warning.Message);
    }
}
```

This code displays the following output:

**No warnings or errors** Even though the `Add` activity has required arguments that are not bound, validation is successful because only the root activity is evaluated. This type of validation is useful for validating only specific elements in an activity tree, such as validation of a property change of a single activity in a designer. Note that if this workflow is invoked, the full validation configured in the workflow is evaluated and an `InvalidOperationException` would be thrown. `ActivityValidationServices` and `ValidationSettings` configure only validation explicitly invoked by the host, and not the validation that occurs when a workflow is invoked.

# Creating an Activity at Runtime with DynamicActivity

3/9/2019 • 2 minutes to read • [Edit Online](#)

[DynamicActivity](#) is a concrete, sealed class with a public constructor. [DynamicActivity](#) can be used to assemble activity functionality at runtime using an activity DOM.

## DynamicActivity Features

[DynamicActivity](#) has access to execution properties, arguments and variables, but no access to run-time services such as scheduling child activities or tracking.

Top-level properties can be set using workflow [Argument](#) objects. In imperative code, these arguments are created using CLR properties on a new type. In XAML, they are declared using `x:Class` and `x:Member` tags.

Activities constructed using [DynamicActivity](#) interface with the designer using [ICustomTypeDescriptor](#). Activities created in the designer can be loaded dynamically using [Load](#), as demonstrated in the following procedure.

### To create an activity at runtime using imperative code

1. Open Visual Studio 2010.
2. Select **File, New, Project**. Select **Workflow 4.0** under **Visual C#** in the **Project Types** window, and select the **v2010** node. Select **Sequential Workflow Console Application** in the **Templates** window. Name the new project DynamicActivitySample.
3. Right-click Workflow1.xaml in the HelloActivity project and select **Delete**.
4. Open Program.cs. Add the following directive to the top of the file.

```
using System.Collections.Generic;
```

5. Replace the contents of the `Main` method with the following code, which creates a [Sequence](#) activity that contains a single [WriteLine](#) activity and assigns it to the [Implementation](#) property of a new dynamic activity.

```

//Define the input argument for the activity
var textOut = new InArgument<string>();
//Create the activity, property, and implementation
Activity dynamicWorkflow = new DynamicActivity()
{
    Properties =
    {
        new DynamicActivityProperty
        {
            Name = "Text",
            Type = typeof(InArgument<String>),
            Value = textOut
        }
    },
    Implementation = () => new Sequence()
    {
        Activities =
        {
            new WriteLine()
            {
                Text = new InArgument<string>(env => textOut.Get(env))
            }
        }
    }
};
//Execute the activity with a parameter dictionary
WorkflowInvoker.Invoke(dynamicWorkflow, new Dictionary<string, object> { { "Text", "Hello
World!" } });
Console.ReadLine();

```

6. Execute the application. A console window with the text "Hello World!" displays.

#### To create an activity at runtime using XAML

1. Open Visual Studio 2010.
2. Select **File, New, Project**. Select **Workflow 4.0** under **Visual C#** in the **Project Types** window, and select the **v2010** node. Select **Workflow Console Application** in the **Templates** window. Name the new project **DynamicActivitySample**.
3. Open Workflow1.xaml in the HelloActivity project. Click the **Arguments** option at the bottom of the designer. Create a new **In** argument called **TextToWrite** of type **String**.
4. Drag a **WriteLine** activity from the **Primitives** section of the toolbox onto the designer surface. Assign the value **TextToWrite** to the **Text** property of the activity.
5. Open Program.cs. Add the following directive to the top of the file.

```
using System.Activities.XamlIntegration;
```

6. Replace the contents of the **Main** method with the following code.

```

Activity act2 = ActivityXamlServices.Load(@"Workflow1.xaml");
results = WorkflowInvoker.Invoke(act2, new Dictionary<string, object> { {
    "TextToWrite", "HelloWorld!" } });
Console.ReadLine();

```

7. Execute the application. A console window with the text "Hello World!" appears.
8. Right-click the Workflow1.xaml file in the **Solution Explorer** and select **View Code**. Note that the activity class is created with **x:Class** and the property is created with **x:Property**.

## See also

- [Authoring Workflows, Activities, and Expressions Using Imperative Code](#)

# Workflow Execution Properties

1/23/2019 • 3 minutes to read • [Edit Online](#)

Through thread local storage (TLS), the CLR maintains an execution context for each thread. This execution context governs well-known thread properties such as the thread identity, the ambient transaction, and the current permission set in addition to user-defined thread properties like named slots.

Unlike programs directly targeting the CLR, workflow programs are hierarchically scoped trees of activities that execute in a thread-agnostic environment. This implies that the standard TLS mechanisms cannot directly be used to determine what context is in scope for a given work item. For example, two parallel branches of execution might use different transactions, yet the scheduler might interleave their execution on the same CLR thread.

Workflow execution properties provide a mechanism to add context specific properties to an activity's environment. This allows an activity to declare which properties are in scope for its sub-tree and also provides hooks for setting up and tearing down TLS to properly interoperate with CLR objects.

## Creating and Using Workflow Execution Properties

Workflow execution properties usually implement the [IExecutionProperty](#) interface, though properties focused on messaging may implement [ISendMessageCallback](#) and [IReceiveMessageCallback](#) instead. To create a workflow execution property, create a class that implements the [IExecutionProperty](#) interface and implement the members [SetupWorkflowThread](#) and [CleanupWorkflowThread](#). These members provide the execution property with an opportunity to properly set up and tear down the thread local storage during each pulse of work of the activity that contains the property, including any child activities. In this example, a [ConsoleColorProperty](#) is created that sets the [Console.ForegroundColor](#).

```
class ConsoleColorProperty : IExecutionProperty
{
    public const string Name = "ConsoleColorProperty";

    ConsoleColor original;
    ConsoleColor color;

    public ConsoleColorProperty(ConsoleColor color)
    {
        this.color = color;
    }

    void IExecutionProperty.SetupWorkflowThread()
    {
        original = Console.ForegroundColor;
        Console.ForegroundColor = color;
    }

    void IExecutionProperty.CleanupWorkflowThread()
    {
        Console.ForegroundColor = original;
    }
}
```

Activity authors can use this property by registering it in the activity's execute override. In this example, a [ConsoleColorScope](#) activity is defined that registers the [ConsoleColorProperty](#) by adding it to the [Properties](#) collection of the current [NativeActivityContext](#).

```

public sealed class ConsoleColorScope : NativeActivity
{
    public ConsoleColorScope()
        : base()
    {
    }

    public ConsoleColor Color { get; set; }
    public Activity Body { get; set; }

    protected override void Execute(NativeActivityContext context)
    {
        context.Properties.Add(ConsoleColorProperty.Name, new ConsoleColorProperty(this.Color));

        if (this.Body != null)
        {
            context.ScheduleActivity(this.Body);
        }
    }
}

```

When the activity's body starts a pulse of work, the [SetupWorkflowThread](#) method of the property is called, and when the pulse of work is complete, the [CleanupWorkflowThread](#) is called. In this example, a workflow is created that uses a [Parallel](#) activity with three branches. The first two branches use the [ConsoleColorScope](#) activity and the third branch does not. All three branches contain two [WriteLine](#) activities and a [Delay](#) activity. When the [Parallel](#) activity executes, the activities that are contained in the branches execute in an interleaved manner, but as each child activity executes the correct console color is applied by the [ConsoleColorProperty](#).

```

Activity wf = new Parallel
{
    Branches =
    {
        new ConsoleColorScope
        {
            Color = ConsoleColor.Blue,
            Body = new Sequence
            {
                Activities =
                {
                    new WriteLine
                    {
                        Text = "Start blue text."
                    },
                    new Delay
                    {
                        Duration = TimeSpan.FromSeconds(1)
                    },
                    new WriteLine
                    {
                        Text = "End blue text."
                    }
                }
            }
        },
        new ConsoleColorScope
        {
            Color = ConsoleColor.Red,
            Body = new Sequence
            {
                Activities =
                {
                    new WriteLine
                    {
                        Text = "Start red text."
                    }
                }
            }
        }
    }
}

```

```

        },
        new Delay
        {
            Duration = TimeSpan.FromSeconds(1)
        },
        new WriteLine
        {
            Text = "End red text."
        }
    }
},
new Sequence
{
    Activities =
    {
        new WriteLine
        {
            Text = "Start default text."
        },
        new Delay
        {
            Duration = TimeSpan.FromSeconds(1)
        },
        new WriteLine
        {
            Text = "End default text."
        }
    }
}
};

WorkflowInvoker.Invoke(wf);

```

When the workflow is invoked, the following output is written to the console window.

```

Start blue text.
Start red text.
Start default text.
End blue text.
End red text.
End default text.

```

#### **NOTE**

Although it is not shown in the previous output, each line of text in the console window is displayed in the indicated color.

Workflow execution properties can be used by custom activity authors, and they also provide the mechanism for handle management for activities such as the [CorrelationScope](#) and [TransactionScope](#) activities.

## See also

- [IExecutionProperty](#)
- [IPropertyRegistrationCallback](#)
- [RegistrationContext](#)

# Using Activity Delegates

3/9/2019 • 5 minutes to read • [Edit Online](#)

Activity delegates enable activity authors to expose callbacks with specific signatures, for which users of the activity can provide activity-based handlers. Two types of activity delegates are available: `ActivityAction<T>` is used to define activity delegates that do not have a return value, and `ActivityFunc<TResult>` is used to define activity delegates that do have a return value.

Activity delegates are useful in scenarios where a child activity must be constrained to having a certain signature. For example, a `While` activity can contain any type of child activity with no constraints, but the body of a `ForEach<T>` activity is an `ActivityAction<T>`, and the child activity that is ultimately executed by `ForEach<T>` must have an `InArgument<T>` that is the same type of the members of the collection that the `ForEach<T>` enumerates.

## Using `ActivityAction`

Several .NET Framework 4.6.1 activities use activity actions, such as the `Catch` activity and the `ForEach<T>` activity. In each case, the activity action represents a location where the workflow author specifies an activity to provide the desired behavior when composing a workflow using one of these activities. In the following example, a `ForEach<T>` activity is used to display text to the console window. The body of the `ForEach<T>` is specified by using an `ActivityAction<T>` that matches the type of the `ForEach<T>` which is string. The `WriteLine` activity specified in the `Handler` has its `Text` argument bound to the string values in the collection that the `ForEach<T>` activity iterates.

```
DelegateInArgument<string> actionArgument = new DelegateInArgument<string>();

Activity wf = new ForEach<string>
{
    Body = new ActivityAction<string>
    {
        Argument = actionArgument,
        Handler = new WriteLine
        {
            Text = new InArgument<string>(actionArgument)
        }
    }
};

List<string> items = new List<string>();
items.Add("Hello");
items.Add("World.");

Dictionary<string, object> inputs = new Dictionary<string, object>();
inputs.Add("Values", items);

WorkflowInvoker.Invoke(wf, inputs);
```

The `actionArgument` is used to flow the individual items in the collection to the `WriteLine`. When the workflow is invoked, the following output is displayed to the console.

```
HelloWorld.
```

The examples in this topic use object initialization syntax. Object initialization syntax can be a useful way to create workflow definitions in code because it provides a hierarchical view of the activities in the workflow and shows the

relationship between the activities. There is no requirement to use object initialization syntax when you programmatically create workflows. The following example is functionally equivalent to the previous example.

```
DelegateInArgument<string> actionArgument = new DelegateInArgument<string>();

WriteLine output = new WriteLine();
output.Text = new InArgument<string>(actionArgument);

ActivityAction<string> body = new ActivityAction<string>();
body.Argument = actionArgument;
body.Handler = output;

ForEach<string> wf = new ForEach<string>();
wf.Body = body;

List<string> items = new List<string>();
items.Add("Hello");
items.Add("World.");

Dictionary<string, object> inputs = new Dictionary<string, object>();
inputs.Add("Values", items);

WorkflowInvoker.Invoke(wf, inputs);
```

For more information about object initializers, see [How to: Initialize Objects without Calling a Constructor \(C# Programming Guide\)](#) and [How to: Declare an Object by Using an Object Initializer](#).

In the following example, a `TryCatch` activity is used in a workflow. An `ApplicationException` is thrown by the workflow, and is handled by a `Catch<TException>` activity. The handler for the `Catch<TException>` activity's activity action is a `WriteLine` activity, and the exception detail is flowed through to it using the `ex` `DelegateInArgument<T>`.

```
DelegateInArgument<ApplicationException> ex = new DelegateInArgument<ApplicationException>()
{
    Name = "ex"
};

Activity wf = new TryCatch
{
    Try = new Throw()
    {
        Exception = new InArgument<Exception>((env) => new ApplicationException("An ApplicationException was thrown."))
    },
    Catches =
    {
        new Catch<ApplicationException>
        {
            Action = new ActivityAction<ApplicationException>
            {
                Argument = ex,
                Handler = new WriteLine()
                {
                    Text = new InArgument<string>((env) => ex.Get(env).Message)
                }
            }
        },
        Finally = new WriteLine()
        {
            Text = "Executing in Finally."
        }
    };
}
```

When creating a custom activity that defines an `ActivityAction<T>`, use an `InvokeAction<T>` to model the invocation of that `ActivityAction<T>`. In this example, a custom `WriteLineWithNotification` activity is defined. This activity is composed of a `Sequence` that contains a `WriteLine` activity followed by an `InvokeAction<T>` that invokes an `ActivityAction<T>` that takes one string argument.

```
public class WriteLineWithNotification : Activity
{
    public InArgument<string> Text { get; set; }
    public ActivityAction<string> TextProcessedAction { get; set; }

    public WriteLineWithNotification()
    {
        this.Implementation = () => new Sequence
        {
            Activities =
            {
                new WriteLine
                {
                    Text = new InArgument<string>((env) => Text.Get(env))
                },
                new InvokeAction<string>
                {
                    Action = TextProcessedAction,
                    Argument = new InArgument<string>((env) => Text.Get(env))
                }
            }
        };
    }
}
```

When a workflow is created by using the `WriteLineWithNotification` activity, the workflow author specifies the desired custom logic in the activity action's `Handler`. In this example, a workflow is created that uses the `WriteLineWithNotification` activity, and a `WriteLine` activity is used as the `Handler`.

```

// Create and invoke the workflow without specifying any activity action
// for TextProcessed.
Activity wf = new WriteLineWithNotification
{
    Text = "Hello World."
};

WorkflowInvoker.Invoke(wf);

// Output:
// Hello World.

// Create and Invoke the workflow with specifying an activity action
// for TextProcessed.
DelegateInArgument<string> msg = new DelegateInArgument<string>();
Activity wf2 = new WriteLineWithNotification
{
    Text = "Hello World with activity action.",
    TextProcessedAction = new ActivityAction<string>
    {
        Argument = msg,
        Handler = new WriteLine
        {
            Text = new InArgument<string>((env) => "Handler of: " + msg.Get(env))
        }
    }
};

// Invoke the workflow with an activity action specified
WorkflowInvoker.Invoke(wf2);

// Output:
// Hello World with activity action.
// Handler of: Hello World with activity action.

```

There are multiple generic versions of [InvokeAction<T>](#) and [ActivityAction<T>](#) provided for passing one or more arguments.

## Using ActivityFunc

[ActivityAction<T>](#) is useful when there is no result value from the activity, and [ActivityFunc<TResult>](#) is used when a result value is returned. When creating a custom activity that defines an [ActivityFunc<TResult>](#), use an [InvokeFunc<TResult>](#) to model the invocation of that [ActivityFunc<TResult>](#). In the following example, a `WriteFillerText` activity is defined. To supply the filler text, an [InvokeFunc<TResult>](#) is specified that takes an integer argument and has a string result. Once the filler text is retrieved, it is displayed to the console using a [WriteLine](#) activity.

```

public class WriteFillerText : Activity
{
    public ActivityFunc<int, string> GetText { get; set; }
    public InArgument<int> Quantity { get; set; }

    Variable<string> text = new Variable<string>
    {
        Name = "Text"
    };

    public WriteFillerText()
    {
        this.Implementation = () => new Sequence
        {
            Variables =
            {
                text
            },
            Activities =
            {
                new InvokeFunc<int, string>
                {
                    Func = GetText,
                    Argument = new InArgument<int>((env) => Quantity.Get(env)),
                    Result = new OutArgument<string>(text)
                },
                new WriteLine
                {
                    Text = new InArgument<string>(text)
                }
            }
        };
    }
}

```

To supply the text, an activity must be used that takes one `int` argument and has a string result. This example shows a `TextGenerator` activity that meets these requirements.

```

public class TextGenerator : CodeActivity<string>
{
    public InArgument<int> Quantity { get; set; }
    public InArgument<string> Text { get; set; }

    protected override string Execute(CodeActivityContext context)
    {
        // Provide a quantity of Random Text
        int q = Quantity.Get(context);
        if (q < 1)
        {
            q = 1;
        }

        string text = Text.Get(context) + " ";
        StringBuilder sb = new StringBuilder(text.Length * q);
        for (int i = 0; i < q; i++)
        {
            sb.Append(text);
        }

        return sb.ToString();
    }
}

```

To use the `TextGenerator` activity with the `WriteFillerText` activity, specify it as the [Handler](#).

```
DelegateInArgument<int> actionArgument = new DelegateInArgument<int>();

Activity wf = new WriteFillerText
{
    Quantity = 5,
    GetText = new ActivityFunc<int, string>
    {
        Argument = actionArgument,
        Handler = new TextGenerator
        {
            Quantity = new InArgument<int>(actionArgument),
            Text = "Hello World."
        }
    }
};

WorkflowInvoker.Invoke(wf);
```

# Using Activity Extensions

10/3/2018 • 2 minutes to read • [Edit Online](#)

Activities can interact with workflow application extensions that allow the host to provide additional functionality that is not explicitly modeled in the workflow. This topic describes how to create and use an extension to count the number of times the activity executes.

## To use an activity extension to count executions

1. Open Visual Studio 2010. Select **New, Project**. Under the **Visual C# node**, select **Workflow**. Select **Workflow Console Application** from the list of templates. Name the project **Extensions**. Click **OK** to create the project.
2. Add a `using` statement in the Program.cs file for the **System.Collections.Generic** namespace.

```
using System.Collections.Generic;
```

3. In the Program.cs file, create a new class named **ExecutionCountExtension**. The following code creates a workflow extension that tracks instance IDs when its **Register** method is called.

```
// This extension collects a list of workflow IDs
public class ExecutionCountExtension
{
    IList<Guid> instances = new List<Guid>();

    public int ExecutionCount
    {
        get
        {
            return this.instances.Count;
        }
    }

    public IEnumerable<Guid> InstanceIds
    {
        get
        {
            return this.instances;
        }
    }

    public void Register(Guid activityInstanceId)
    {
        if (!this.instances.Contains<Guid>(activityInstanceId))
        {
            instances.Add(activityInstanceId);
        }
    }
}
```

4. Create an activity that consumes the **ExecutionCountExtension**. The following code defines an activity that retrieves the **ExecutionCountExtension** object from the runtime and calls its **Register** method when the activity executes.

```
// Activity that consumes an extension provided by the host. If the extension is available
// in the context, it will invoke (in this case, registers the Id of the executing workflow)
public class MyActivity: CodeActivity
{
    protected override void Execute(CodeActivityContext context)
    {
        ExecutionCountExtension ext = context.GetExtension<ExecutionCountExtension>();
        if (ext != null)
        {
            ext.Register(context.WorkflowInstanceId);
        }
    }
}
```

5. Implement the activity in the **Main** method of the program.cs file. The following code contains methods to generate two different workflows, execute each workflow several times, and display the resulting data that is contained in the extension.

```

class Program
{
    // Creates a workflow that uses the activity that consumes the extension
    static Activity CreateWorkflow1()
    {
        return new Sequence
        {
            Activities =
            {
                new MyActivity()
            }
        };
    }

    // Creates a workflow that uses two instances of the activity that consumes the extension
    static Activity CreateWorkflow2()
    {
        return new Sequence
        {
            Activities =
            {
                new MyActivity(),
                new MyActivity()
            }
        };
    }

    static void Main(string[] args)
    {
        // create the extension
        ExecutionCountExtension executionCountExt = new ExecutionCountExtension();

        // configure the first invoker and execute 3 times
        WorkflowInvoker invoker = new WorkflowInvoker(CreateWorkflow1());
        invoker.Extensions.Add(executionCountExt);
        invoker.Invoke();
        invoker.Invoke();
        invoker.Invoke();

        // configure the second invoker and execute 2 times
        WorkflowInvoker invoker2 = new WorkflowInvoker(CreateWorkflow2());
        invoker2.Extensions.Add(executionCountExt);
        invoker2.Invoke();
        invoker2.Invoke();

        // show the data in the extension
        Console.WriteLine("Executed {0} times", executionCountExt.ExecutionCount);
        executionCountExt.InstanceIds.ToList().ForEach(i => Console.WriteLine("...{0}", i));
    }
}

```

# Consuming OData feeds from a workflow

3/25/2019 • 12 minutes to read • [Edit Online](#)

WCF Data Services is a component of the .NET Framework that enables you to create services that use the Open Data Protocol (OData) to expose and consume data over the Web or intranet by using the semantics of representational state transfer (REST). OData exposes data as resources that are addressable by URIs. Any application can interact with an OData-based data service if it can send an HTTP request and process the OData feed that a data service returns. In addition, WCF Data Services includes client libraries that provide a richer programming experience when you consume OData feeds from .NET Framework applications. This topic provides an overview of consuming an OData feed in a workflow with and without using the client libraries.

## Using the sample Northwind OData service

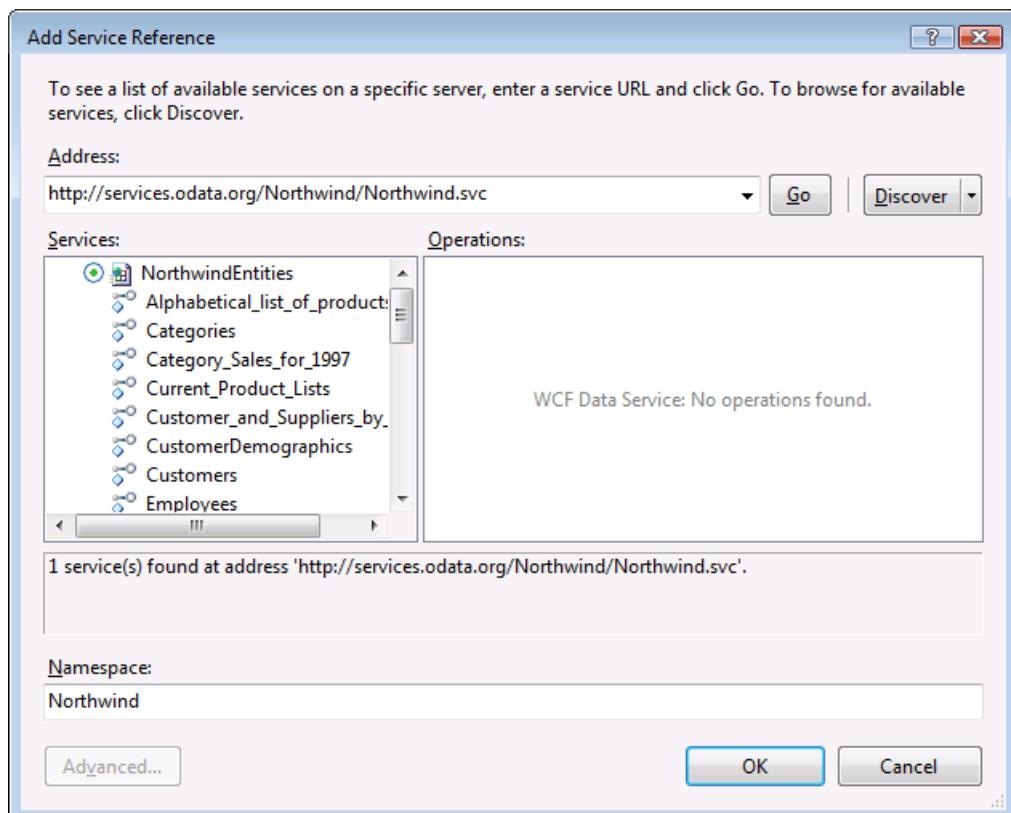
The examples in this topic use the sample Northwind data service located at <http://services.odata.org/Northwind/Northwind.svc/>. This service is provided as part of the [OData SDK](#) and provides read-only access to the sample Northwind database. If write access is desired, or if a local WCF Data Service is desired, you can follow the steps of the [WCF Data Services Quickstart](#) to create a local OData service that provides access to the Northwind database. If you follow the quickstart, substitute the local URI for the one provided in the example code in this topic.

## Consuming an OData feed using the client libraries

WCF Data Services includes client libraries that enable you to more easily consume an OData feed from .NET Framework and client applications. These libraries simplify sending and receiving HTTP messages. They also translate the message payload into CLR objects that represent entity data. The client libraries feature the two core classes `DataContext` and `DataServiceQuery<TElement>`. These classes enable you to query a data service and then work with the returned entity data as CLR objects. This section covers two approaches to creating activities that use the client libraries.

### Adding a service reference to the WCF Data service

To generate the Northwind client libraries, you can use the **Add Service Reference** dialog box in Visual Studio 2012 to add a reference to the Northwind OData service.



Note that there are no service operations exposed by the service, and in the **Services** list there are items representing the entities exposed by the Northwind data service. When the service reference is added, classes will be generated for these entities and they can be used in the client code. The examples in this topic use these classes and the `NorthwindEntities` class to perform the queries.

#### NOTE

For more information, see [Generating the Data Service Client Library \(WCF Data Services\)](#).

## Using asynchronous methods

To address possible latency issues that may occur when accessing resources over the Web, we recommend accessing WCF Data Services asynchronously. The WCF Data Services client libraries include asynchronous methods for invoking queries, and Windows Workflow Foundation (WF) provides the `AsyncCodeActivity` class for authoring asynchronous activities. `AsyncCodeActivity` derived activities can be written to take advantage of .NET Framework classes that have asynchronous methods, or the code to be executed asynchronously can be put into a method and invoked by using a delegate. This section provides two examples of an `AsyncCodeActivity` derived activity; one that uses the asynchronous methods of the WCF Data Services client libraries and one that uses a delegate.

#### NOTE

For more information, see [Asynchronous Operations \(WCF Data Services\)](#) and [Creating Asynchronous Activities](#).

## Using client library asynchronous methods

The `DataServiceQuery<TElement>` class provides `BeginExecute` and `EndExecute` methods for querying an OData service asynchronously. These methods can be called from the `BeginExecute` and `EndExecute` overrides of an `AsyncCodeActivity` derived class. When the `AsyncCodeActivity BeginExecute` override returns, the workflow can go idle (but not persist), and when the asynchronous work is completed, `EndExecute` is invoked by the runtime.

In the following example, an `OrdersByCustomer` activity is defined that has two input arguments. The `CustomerId`

argument represents the customer who identifies which orders to return, and the `ServiceUri` argument represents the URI of the OData service to be queried. Because the activity derives from

`AsyncCodeActivity<IEnumerable<Order>>` there is also a `Result` output argument that is used to return the results of the query. The `BeginExecute` override creates a LINQ query that selects all orders of the specified customer. This query is specified as the `UserState` of the passed `AsyncCodeActivityContext`, and then the query's `BeginExecute` method is called. Note that the callback and state that are passed into the query's `BeginExecute` are the ones that are passed in to the activity's `BeginExecute` method. When the query has finished executing, the activity's `EndExecute` method is invoked. The query is retrieved from the `UserState`, and then the query's `EndExecute` method is called. This method returns an `IEnumerable<T>` of the specified entity type; in this case `Order`. Since `IEnumerable<Order>` is the generic type of the `AsyncCodeActivity<TResult>`, this `IEnumerable` is set as the `Result OutArgument<T>` of the activity.

```
class OrdersByCustomer : AsyncCodeActivity<IEnumerable<Order>>
{
    [RequiredArgument]
    public InArgument<string> CustomerId { get; set; }

    [RequiredArgument]
    public InArgument<string> ServiceUri { get; set; }

    protected override IAsyncResult BeginExecute(AsyncCodeActivityContext context, AsyncCallback callback,
object state)
    {
        NorthwindEntities dataContext = new NorthwindEntities(new Uri(ServiceUri.Get(context)));

        // Define a LINQ query that returns Orders and
        // Order_Details for a specific customer.
        DataServiceQuery<Order> ordersQuery = (DataServiceQuery<Order>)
            from o in dataContext.Orders.Expand("Order_Details")
            where o.Customer.CustomerID == CustomerId.Get(context)
            select o;

        // Specify the query as the UserState for the AsyncCodeActivityContext
        context.UserState = ordersQuery;

        // The callback and state used here are the ones passed into
        // the BeginExecute of this activity.
        return ordersQuery.BeginExecute(callback, state);
    }

    protected override IEnumerable<Order> EndExecute(AsyncCodeActivityContext context, IAsyncResult result)
    {
        // Get the DataServiceQuery from the context.UserState
        DataServiceQuery<Order> ordersQuery = context.UserState as DataServiceQuery<Order>;

        // Return an IEnumerable of the query results.
        return ordersQuery.EndExecute(result);
    }
}
```

In the following example, the `OrdersByCustomer` activity retrieves a list of orders for the specified customer, and then a `ForEach<T>` activity enumerates the returned orders and writes the date of each order to the console.

```

Variable<IEnumerable<Order>> orders = new Variable<IEnumerable<Order>>();
DelegateInArgument<Order> order = new DelegateInArgument<Order>();

Activity wf = new Sequence
{
    Variables = { orders },
    Activities =
    {
        new WriteLine
        {
            Text = "Calling WCF Data Service..."
        },
        new OrdersByCustomer
        {
            ServiceUri = "http://services.odata.org/Northwind/Northwind.svc/",
            CustomerId = "ALFKI",
            Result = orders
        },
        new ForEach<Order>
        {
            Values = orders,
            Body = new ActivityAction<Order>
            {
                Argument = order,
                Handler = new WriteLine
                {
                    Text = new InArgument<string>((env) => string.Format("{0:d}", order.Get(env).OrderDate))
                }
            }
        }
    }
};

WorkflowInvoker.Invoke(wf);

```

When this workflow is invoked, the following data is written to the console:

```

Calling WCF Data Service...
8/25/1997
10/3/1997
10/13/1997
1/15/1998
3/16/1998
4/9/1998

```

#### **NOTE**

If a connection to the OData server cannot be established, you will get an exception similar to the following exception:

Unhandled Exception: System.InvalidOperationException: An error occurred while processing this request. --->  
 System.Net.WebException: Unable to connect to the remote server ---> System.Net.Sockets.SocketException: A connection attempt failed because the connected party did not properly respond after a period of time, or established connection failed because connected host has failed to respond.

If any additional processing of the data returned by the query is required, it can be done in the activity's [EndExecute](#) override. Both [BeginExecute](#) and [EndExecute](#) are invoked by using the workflow thread, and any code in these overrides does not run asynchronously. If the additional processing is extensive or long-running, or the query results are paged, you should consider the approach discussed in the next section, which uses a delegate to execute the query and perform additional processing asynchronously.

#### **Using a delegate**

In addition to invoking the asynchronous method of a .NET Framework class, an `AsyncCodeActivity`-based activity can also define the asynchronous logic in one of its methods. This method is specified by using a delegate in the activity's `BeginExecute` override. When the method returns, the runtime invokes the activity's `EndExecute` override. When calling an OData service from a workflow, this method can be used to query the service and provide any additional processing.

In the following example, a `ListCustomers` activity is defined. This activity queries the sample Northwind data service and returns a `List<Customer>` that contains all of the customers in the Northwind database. The asynchronous work is performed by the `GetCustomers` method. This method queries the service for all customers, and then copies them into a `List<Customer>`. It then checks to see if the results are paged. If so, it queries the service for the next page of results, adds them to the list, and continues until all of the customer data has been retrieved.

#### NOTE

For more information about paging in WCF Data Services, see [How to: Load Paged Results \(WCF Data Services\)](#).

Once all customers are added, the list is returned. The `GetCustomers` method is specified in the activity's `BeginExecute` override. Since the method has a return value, a `Func<string, List<Customer>>` is created to specify the method.

#### NOTE

If the method that performs the asynchronous work does not have a return value, an `Action` is used instead of a `Func<TResult>`. For examples of creating an asynchronous example using both approaches, see [Creating Asynchronous Activities](#).

This `Func<TResult>` is assigned to the `UserState`, and then `BeginInvoke` is called. Since the method to be invoked does not have access to the activity's environment of arguments, the value of the `ServiceUri` argument is passed as the first parameter, together with the callback and state that were passed into `BeginExecute`. When `GetCustomers` returns, the runtime invokes `EndExecute`. The code in `EndExecute` retrieves the delegate from the `UserState`, calls `EndInvoke`, and returns the result, which is the list of customers returned from the `GetCustomers` method.

```

class ListCustomers : AsyncCodeActivity<List<Customer>>
{
    [RequiredArgument]
    public InArgument<string> ServiceUri { get; set; }

    protected override IAsyncResult BeginExecute(AsyncCodeActivityContext context, AsyncCallback callback,
object state)
    {
        // Create a delegate that references the method that implements
        // the asynchronous work. Assign the delegate to the UserState,
        // invoke the delegate, and return the resulting IAsyncResult.
        Func<string, List<Customer>> GetCustomersDelegate = new Func<string, List<Customer>>(GetCustomers);
        context.UserState = GetCustomersDelegate;
        return GetCustomersDelegate.BeginInvoke(ServiceUri.Get(context), callback, state);
    }

    protected override List<Customer> EndExecute(AsyncCodeActivityContext context, IAsyncResult result)
    {
        // Get the delegate from the UserState and call EndInvoke
        Func<string, List<Customer>> GetCustomersDelegate = (Func<string, List<Customer>>)context.UserState;
        return (List<Customer>)GetCustomersDelegate.EndInvoke(result);
    }

    List<Customer> GetCustomers(string serviceUri)
    {
        // Get all customers here and add them to a list. This method doesn't have access to the
        // activity's environment of arguments, so the Service Uri is passed in.

        // Create the DataServiceContext using the service URI.
        NorthwindEntities context = new NorthwindEntities(new Uri(serviceUri));

        // Return all customers.
        QueryOperationResponse<Customer> response =
            context.Customers.Execute() as QueryOperationResponse<Customer>;

        // Add them to the list.
        List<Customer> customers = new List<Customer>(response);

        // Is this the complete list or are the results paged?
        DataServiceQueryContinuation<Customer> token;
        while ((token = response.GetContinuation()) != null)
        {
            // Load the next page of results.
            response = context.Execute<Customer>(token) as QueryOperationResponse<Customer>;

            // Add the next page of customers to the list.
            customers.AddRange(response);
        }

        // Return the list of customers
        return customers;
    }
}

```

In the following example, the `ListCustomers` activity retrieves a list of customers, and then a `ForEach<T>` activity enumerates them and writes the company name and contact name of each customer to the console.

```

Variable<List<Customer>> customers = new Variable<List<Customer>>();
DelegateInArgument<Customer> customer = new DelegateInArgument<Customer>();

Activity wf = new Sequence
{
    Variables = { customers },
    Activities =
    {
        new WriteLine
        {
            Text = "Calling WCF Data Service..."
        },
        new ListCustomers
        {
            ServiceUri = "http://services.odata.org/Northwind/Northwind.svc/",
            Result = customers
        },
        new ForEach<Customer>
        {
            Values = customers,
            Body = new ActivityAction<Customer>
            {
                Argument = customer,
                Handler = new WriteLine
                {
                    Text = new InArgument<string>((env) => string.Format("{0}, Contact: {1}",
                        customer.Get(env).CompanyName, customer.Get(env).ContactName))
                }
            }
        }
    }
};

WorkflowInvoker.Invoke(wf);

```

When this workflow is invoked, the following data is written to the console. Since this query returns many customers, only part of the output is displayed here.

```

Calling WCF Data Service...
Alfreds Futterkiste, Contact: Maria Anders
Ana Trujillo Emparedados y helados, Contact: Ana Trujillo
Antonio Moreno Taquería, Contact: Antonio Moreno
Around the Horn, Contact: Thomas Hardy
Berglunds snabbköp, Contact: Christina Berglund
...

```

## Consuming an OData feed without using the client libraries

OData exposes data as resources that are addressable by URIs. When you use the client libraries these URIs are created for you, but you do not have to use the client libraries. If desired, OData services can be accessed directly without using the client libraries. When not using the client libraries the location of the service and the desired data are specified by the URI and the results are returned in the response to the HTTP request. This raw data can then be processed or manipulated in the desired manner. One way to retrieve the results of an OData query is by using the [WebClient](#) class. In this example, the contact name for the customer represented by the key ALFKI is retrieved.

```

string uri = "http://services.odata.org/Northwind/Northwind.svc/Customers('ALFKI')/ContactName";
 WebClient client = new WebClient();
 string data = client.DownloadString(uri);
 Console.WriteLine("Raw data returned:\n{0}", data);

```

When this code is run, the following output is displayed to the console:

```
Raw data returned:  
<?xml version="1.0" encoding="utf-8" standalone="yes"?>  
<ContactName xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices">Maria Anders</ContactName>
```

In a workflow, the code from this example could be incorporated into the [Execute](#) override of a [CodeActivity](#)-based custom activity, but the same functionality can also be accomplished by using the [InvokeMethod<TResult>](#) activity. The [InvokeMethod<TResult>](#) activity enables workflow authors to invoke static and instance methods of a class, and also has an option to invoke the specified method asynchronously. In the following example, an [InvokeMethod<TResult>](#) activity is configured to call the [DownloadString](#) method of the [WebClient](#) class and return a list of customers.

```
new InvokeMethod<string>
{
    TargetObject = new InArgument<WebClient>(new VisualBasicValue<WebClient>("New System.Net.WebClient()")),
    MethodName = "DownloadString",
    Parameters =
    {
        new InArgument<string>("http://services.odata.org/Northwind/Northwind.svc/Customers('ALFKI')/Orders")
    },
    Result = data,
    RunAsynchronously = true
},
```

[InvokeMethod<TResult>](#) can call both static and instance methods of a class. Since [DownloadString](#) is an instance method of the [WebClient](#) class, a new instance of the [WebClient](#) class is specified for the [TargetObject](#).

[DownloadString](#) is specified as the [MethodName](#), the URI that contains the query is specified in the [Parameters](#) collection, and the return value is assigned to the [Result](#) value. The [RunAsynchronously](#) value is set to [true](#), which means that the method invocation will run asynchronously with regard to the workflow. In the following example, a workflow is constructed that uses the [InvokeMethod<TResult>](#) activity to query the sample Northwind data service for a list of orders for a specific customer, and then the returned data is written to the console.

```

Variable<string> data = new Variable<string>();

Activity wf = new Sequence
{
    Variables = { data },
    Activities =
    {
        new WriteLine
        {
            Text = "Calling WCF Data Service..."
        },
        new InvokeMethod<string>
        {
            TargetObject = new InArgument<WebClient>(new VisualBasicValue<WebClient>("New
System.Net.WebClient()")),
            MethodName = "DownloadString",
            Parameters =
            {
                new InArgument<string>
                ("http://services.odata.org/Northwind/Northwind.svc/Customers('ALFKI')/Orders")
            },
            Result = data,
            RunAsynchronously = true
        },
        new WriteLine
        {
            Text = new InArgument<string>(env => string.Format("Raw data returned:\n{0}", data.Get(env)))
        }
    }
};

WorkflowInvoker.Invoke(wf);

```

When this workflow is invoked, the following output is displayed to the console. Since this query returns several orders, only part of the output is displayed here.

```

Calling WCF Data Service...
Raw data returned:

<?xml version="1.0" encoding="utf-8" standalone="yes"?>*
<feed
  xml:base="http://services.odata.org/Northwind/Northwind.svc/"
  xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
  xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
  xmlns="http://www.w3.org/2005/Atom">
  <title type="text">Orders</title>
  <id>http://services.odata.org/Northwind/Northwind.svc/Customers('ALFKI')/Orders</id>
  <updated>2010-05-19T19:37:07Z</updated>
  <link rel="self" title="Orders" href="Orders" />
  <entry>
    <id>http://services.odata.org/Northwind/Northwind.svc/Orders(10643)</id>
    <title type="text"></title>
    <updated>2010-05-19T19:37:07Z</updated>
    <author>
      <name />
    </author>
    <link rel="edit" title="Order" href="Orders(10643)" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Customer"
      type="application/atom+xml;type=entry" title="Customer" href="Orders(10643)/Customer" />
    ...
  </entry>
</feed>

```

This example provides one method that workflow application authors can use to consume the raw data returned from an OData service. For more information about accessing WCF Data Services using URLs, see [Accessing Data Service Resources \(WCF Data Services\)](#) and [OData: URI Conventions](#).



# Activity Definition Scoping and Visibility

3/9/2019 • 4 minutes to read • [Edit Online](#)

Activity definition scoping and visibility, just like scoping and visibility of an object, is the ability of other objects or activities to access members of the activity. Activity definition is performed by the following implementations:

1. Determining the members ([Argument](#), [Variable](#), and [ActivityDelegate](#) objects, and child activities) an activity exposes to its users.
2. Implementing the execution logic of the activity

The implementation may involve members that are not exposed to consumers of the activity, but are rather details of the implementation. Similar to type definition, the activity model allows an author to qualify the visibility of an activity member regarding the definition of the activity being defined. This visibility governs aspects of member usage, such as data scoping.

## Scope

In addition to data scoping, activity model visibility can restrict access to other aspects of the activity, such as validation, debugging, tracking, or tracing. Execution properties use visibility and scoping for constraining execution characteristics to a particular scope of definition. Secondary roots use visibility and scoping to constrain the state captured by a [CompensableActivity](#) to the scope of definition in which the compensable activities are used.

## Definition and usage

A workflow is written by authoring new activities by inheriting from base activity classes, and by using activities from the [Built-In Activity Library](#). In order to use an activity, the activity author must configure the visibility of each component of its definition.

### Activity Members

The activity model defines the arguments, variables, delegates, and child activities that the activity makes available to consumers. Each of these members can be declared as `public` or `private`. Public members are configured by the consumer of the activity, whereas `private` members use an implementation fixed by the author of the activity. The visibility rules for data scoping are as follows:

1. Public members and the public members of public child activities can reference public variables.
2. Private members and the public members of public child activities can reference arguments and private variables.

A member that can be set by the consumer of an activity should never be made private.

### Authoring Models

Custom activities are defined by using [NativeActivity](#), [Activity](#), [CodeActivity](#), or [AsyncCodeActivity](#). Activities that derive from these classes can expose different member types with different visibilities.

#### NativeActivity

Activities that derive from [NativeActivity](#) have behavior that is written in imperative code, and can optionally be defined by using existing activities. Deriving activities from [NativeActivity](#) grants access to all of the features exposed by the runtime. Any member of such an activity can be defined by using either public or private visibility, except arguments, which can only be declared as `public`.

Members of classes derived from [NativeActivity](#) are declared to the runtime using the [NativeActivityMetadata](#) struct passed to the [CacheMetadata](#) method.

## Activity

Activities created by using [Activity](#) have behavior that is designed strictly through composing other activities. The [Activity](#) class has one implementation child activity, obtained by the runtime using [Implementation](#). An activity deriving from [Activity](#) can define public arguments, public variables, imported ActivityDelegates, and imported Activities.

Imported ActivityDelegates and Activities are declared as public children of the activity, but cannot be directly scheduled by the activity. This information is used during validation to avoid running parent-facing validations at locations where the activity will never execute. Also, imported children, just like public children, can be referenced and scheduled by the activity's implementation. This means that an activity which imports an activity called Activity1 can contain a [Sequence](#) in its implementation which schedules Activity1.

## CodeActivity/ AsyncCodeActivity

This base class is used for authoring behavior in imperative code. Activities that derive from this class only have access to the arguments they expose. This means that the only members that these activities can expose are public arguments. No other members or visibilities apply to these activities.

## Summary of visibilities

The following table summarizes the information earlier in this section.

| MEMBER TYPE       | NATIVEACTIVITY  | ACTIVITY   | CODEACTIVITY/<br>ASYNCCODEACTIVITY |
|-------------------|-----------------|--|------------------------------------|
| Arguments         | Public/ Private | Public   | not applicable                     |
| Variables         | Public/ Private | Public   | not applicable                     |
| Child Activities  | Public/ Private | Public, one fixed private child defined in Implementation. | not applicable                     |
| ActivityDelegates | Public/ Private | Public   | not applicable                     |

In general, a member that cannot be set by the consumer of an activity should not be made public.

## Execution Properties

In some scenarios, it is useful to scope a particular execution property to the public children of an activity. The [ExecutionProperties](#) collection provides this capability with the [Add](#) method. This method has a Boolean parameter indicating whether a particular property is scoped to all children, or just those that are public. If this parameter is set to `true`, the property will only be visible to public members and the public members of their public children.

## Secondary Roots

A secondary root is the runtime's internal mechanism for managing state for compensation activities. When a [CompensableActivity](#) has finished running, its state is not cleaned up immediately. Instead, the state is maintained by the runtime in a secondary root until the compensation episode has completed. The location environments captured with the secondary root correspond to the scope of definition in which the Compensable activity is used.

# Creating custom flow control activities

3/14/2019 • 2 minutes to read • [Edit Online](#)

The .NET Framework contains a variety of flow-control activities that function similarly to abstract programming structures (such as [Flowchart](#)) or to standard programming statements (such as [If](#)). This topic discusses the architecture of one of the sample projects, [Non-Generic ForEach](#).

## Creating the custom class

Since the Non-Generic ForEach class will need to schedule child activities, it will need to derive from [NativeActivity](#), since activities that derive from [CodeActivity](#) do not have this functionality.

```
public sealed class ForEach : NativeActivity
{

```

The custom class requires several members to store data being used by the activity, and to provide functionality to execute the activity's child activities. These members include:

- `valueEnumerator` : The non-public [Variable<T>](#) of type [IEnumerator](#) used to iterate over the collection of items. This member is of type [Variable<T>](#) because it is used internally in the activity, rather than an argument or public property, which would be used if this object were to have an origin outside the activity.
- `OnChildComplete` : The public [CompletionCallback](#) property that executes when each child completes execution. This member is defined as a CLR property, since its value will not change for different instances of the activity.
- `Values` : The collection of inputs used for the iterations of the child activity. This member is of type [InArgument<T>](#), since the origin of the data is outside the activity, but the contents of the collection is not expected to change during the execution of the activity. If the activity needed the functionality to change the contents of this collection while the activity was executing (to add or remove activities, for instance), this member would have been defined as an [ActivityAction](#), which then would have been evaluated every time it was accessed, so that changes would be available to the activity.
- `Body` : This member defines the activity to be executed for each item in the `Values` collection. This member is defined as an [ActivityAction](#) so that it is evaluated every time it is accessed.
- `Execute` : This method uses the `InternalExecute`, `OnForEachComplete`, and `GetStateAndExecute` non-public members to schedule the execution and assign the completion handler of the child activity defined in the `Body` member.
- `CacheMetadata` : This member provides the runtime with the information it needs to execute the activity. By default, an activity's `CacheMetadata` method will inform the runtime of all public members of the activity, but since this activity uses private members for some functionality, it needs to inform the runtime of their existence so that the runtime can be aware of them. In this case, the `CacheMetadata` function is overridden so that the private `valueEnumerator` member can be accessed. This member also creates an argument for the values for the activity so that they can be passed in to the activity during execution.

# Windows Workflow Foundation Data Model

3/9/2019 • 2 minutes to read • [Edit Online](#)

The Windows Workflow Foundation data model is composed of three concepts: variables, arguments, and expressions. Variables represent the storage of data and arguments represent the flow of data into and out of an activity. Arguments are bound (assigned a value) using expressions that may reference variables.

## In This Section

### [Variables and Arguments](#)

Describes the concepts of variables and arguments and how they are used.

### [Expressions](#)

Describes expressions and how they are used in workflow development.

### [C# Expressions](#)

Describes C# expressions in workflows, introduced with .NET Framework 4.5.

### [Properties vs. Arguments](#)

Describes how to select which type to use for activity input.

### [Exposing data with CacheMetadata](#)

Describes how to expose a custom set of metadata about an executing activity's members.

# Variables and Arguments

3/9/2019 • 5 minutes to read • [Edit Online](#)

In Windows Workflow Foundation (WF), variables represent the storage of data and arguments represent the flow of data into and out of an activity. An activity has a set of arguments and they make up the signature of the activity. Additionally, an activity can maintain a list of variables to which a developer can add or remove variables during the design of a workflow. An argument is bound using an expression that returns a value.

## Variables

Variables are storage locations for data. Variables are declared as part of the definition of a workflow. Variables take on values at runtime and these values are stored as part of the state of a workflow instance. A variable definition specifies the type of the variable and optionally, the name. The following code shows how to declare a variable, assign a value to it using an [Assign<T>](#) activity, and then display its value to the console using a [WriteLine](#) activity.

```
// Define a variable named "str" of type string.  
Variable<string> var = new Variable<string>  
{  
    Name = "str"  
};  
  
// Declare the variable within a Sequence, assign  
// a value to it, and then display it.  
Activity wf = new Sequence()  
{  
    Variables = { var },  
    Activities =  
    {  
        new Assign<string>  
        {  
            To = var,  
            Value = "Hello World."  
        },  
        new WriteLine  
        {  
            Text = var  
        }  
    }  
};  
  
WorkflowInvoker.Invoke(wf);
```

A default value expression can optionally be specified as part of a variable declaration. Variables also can have modifiers. For example, if a variable is read-only then the read-only [VariableModifiers](#) modifier can be applied. In the following example, a read-only variable is created that has a default value assigned.

```
// Define a read-only variable with a default value.  
Variable<string> var = new Variable<string>  
{  
    Default = "Hello World.",  
    Modifiers = VariableModifiers.ReadOnly  
};
```

# Variable Scoping

The lifetime of a variable at runtime is equal to the lifetime of the activity that declares it. When an activity completes, its variables are cleaned up and can no longer be referenced.

## Arguments

Activity authors use arguments to define the way data flows into and out of an activity. Each argument has a specified direction: [In](#), [Out](#), or [InOut](#).

The workflow runtime makes the following guarantees about the timing of data movement into and out of activities:

1. When an activity starts executing, the values of all of its input and input/output arguments are calculated.  
For example, regardless of when [Get](#) is called, the value returned is the one calculated by the runtime prior to its invocation of [Execute](#).
2. When [Set](#) is called, the runtime sets the value immediately.
3. Arguments can optionally have their [EvaluationOrder](#) specified. [EvaluationOrder](#) is a zero-based value that specifies the order in which the argument is evaluated. By default, the evaluation order of the argument is unspecified and is equal to the [UnspecifiedEvaluationOrder](#) value. Set [EvaluationOrder](#) to a value greater or equal to zero to specify an evaluation order for this argument. Windows Workflow Foundation evaluates arguments with a specified evaluation order in ascending order. Note that arguments with an unspecified evaluation order are evaluated before those with a specified evaluation order.

An activity author can use a strongly-typed mechanism for exposing its arguments. This is accomplished by declaring properties of type [InArgument<T>](#), [OutArgument<T>](#), and [InOutArgument<T>](#). This allows an activity author to establish a specific contract about the data going into and out of an activity.

### Defining the Arguments on an Activity

Arguments can be defined on an activity by specifying properties of type [InArgument<T>](#), [OutArgument<T>](#), and [InOutArgument<T>](#). The following code shows how to define the arguments for a [Prompt](#) activity that takes in a string to display to the user and returns a string that contains the user's response.

```
public class Prompt : Activity
{
    public InArgument<string> Text { get; set; }
    public OutArgument<string> Response { get; set; }
    // Rest of activity definition omitted.
}
```

#### NOTE

Activities that return a single value can derive from [Activity<TResult>](#), [NativeActivity<TResult>](#), or [CodeActivity<TResult>](#). These activities have a well-defined [OutArgument<T>](#) named [Result](#) that contains the return value of the activity.

## Using Variables and Arguments in Workflows

The following example shows how variables and arguments are used in a workflow. The workflow is a sequence that declares three variables: [var1](#), [var2](#), and [var3](#). The first activity in the workflow is an [Assign](#) activity that assigns the value of variable [var1](#) to the variable [var2](#). This is followed by a [WriteLine](#) activity that prints the value of the [var2](#) variable. Next is another [Assign](#) activity that assigns the value of variable [var2](#) to the variable [var3](#). Finally there is another [WriteLine](#) activity that prints the value of the [var3](#) variable. The first [Assign](#) activity uses [InArgument<string>](#) and [OutArgument<string>](#) objects that explicitly represent the bindings for the

activity's arguments. `InArgument<string>` is used for `Value` because the value is flowing into the `Assign<T>` activity through its `Value` argument, and `OutArgument<string>` is used for `To` because the value is flowing out of the `To` argument into the variable. The second `Assign` activity accomplishes the same thing with more compact but equivalent syntax that uses implicit casts. The `WriteLine` activities also use the compact syntax.

```
// Declare three variables; the first one is given an initial value.
Variable<string> var1 = new Variable<string>()
{
    Default = "one"
};
Variable<string> var2 = new Variable<string>();
Variable<string> var3 = new Variable<string>();

// Define the workflow
Activity wf = new Sequence
{
    Variables = { var1, var2, var3 },
    Activities =
    {
        new Assign<string>()
        {
            Value = new InArgument<string>(var1),
            To = new OutArgument<string>(var2)
        },
        new WriteLine() { Text = var2 },
        new Assign<string>()
        {
            Value = var2,
            To = var3
        },
        new WriteLine() { Text = var3 }
    }
};

WorkflowInvoker.Invoke(wf);
```

## Using Variables and Arguments in Code-Based Activities

The previous examples show how to use arguments and variables in workflows and declarative activities. Arguments and variables are also used in code-based activities. Conceptually the usage is very similar. Variables represent data storage within the activity, and arguments represent the flow of data into or out of the activity, and are bound by the workflow author to other variables or arguments in the workflow that represent where the data flows to or from. To get or set the value of a variable or argument in an activity, an activity context must be used that represents the current execution environment of the activity. This is passed into the `Execute` method of the activity by the workflow runtime. In this example, a custom `Add` activity is defined that has two `In` arguments. To access the value of the arguments, the `Get` method is used and the context that was passed in by the workflow runtime is used.

```
public sealed class Add : CodeActivity<int>
{
    [RequiredArgument]
    public InArgument<int> Operand1 { get; set; }

    [RequiredArgument]
    public InArgument<int> Operand2 { get; set; }

    protected override int Execute(CodeActivityContext context)
    {
        return Operand1.Get(context) + Operand2.Get(context);
    }
}
```

For more information about working with arguments, variables, and expressions in code, see [Authoring Workflows, Activities, and Expressions Using Imperative Code](#) and [Required Arguments and Overload Groups](#).

# Expressions

3/9/2019 • 3 minutes to read • [Edit Online](#)

A Windows Workflow Foundation (WF) expression is any activity that returns a result. All expression activities derive indirectly from `Activity<TResult>`, which contains an `OutArgument` property named `Result` as the activity's return value. WF ships with a wide range of expression activities from simple ones like `VariableValue<T>` and `VariableReference<T>`, which provide access to single workflow variable through operator activities, to complex activities such as `VisualBasicReference<TResult>` and `VisualBasicValue<TResult>` that offer access to the full breadth of Visual Basic language to produce the result. Additional expression activities can be created by deriving from `CodeActivity<TResult>` or `NativeActivity<TResult>`.

## Using Expressions

Workflow designer uses `VisualBasicValue<TResult>` and `VisualBasicReference<TResult>` for all expressions in Visual Basic projects, and `CSharpValue<TResult>` and `CSharpReference<TResult>` for expressions in C# workflow projects.

### NOTE

Support for C# expressions in workflow projects was introduced in .NET Framework 4.5. For more information, see [C# Expressions](#).

Workflows produced by designer are saved in XAML, where expressions appear enclosed in square brackets, as in the following example.

```
<Sequence xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Sequence.Variables>
    <Variable x:TypeArguments="x:Int32" Default="1" Name="a" />
    <Variable x:TypeArguments="x:Int32" Default="2" Name="b" />
    <Variable x:TypeArguments="x:Int32" Default="3" Name="c" />
    <Variable x:TypeArguments="x:Int32" Default="0" Name="r" />
  </Sequence.Variables>
  <Assign>
    <Assign.To>
      <OutArgument x:TypeArguments="x:Int32">[r]</OutArgument>
    </Assign.To>
    <Assign.Value>
      <InArgument x:TypeArguments="x:Int32">[a + b + c]</InArgument>
    </Assign.Value>
  </Assign>
</Sequence>
```

When defining a workflow in code, any expression activities can be used. The following example shows the usage of a composition of operator activities to add three numbers.

```

Variable<int> a = new Variable<int>("a", 1);
Variable<int> b = new Variable<int>("b", 2);
Variable<int> c = new Variable<int>("c", 3);
Variable<int> r = new Variable<int>("r", 0);

Sequence w = new Sequence
{
    Variables = { a, b, c, r },
    Activities =
    {
        new Assign {
            To = new OutArgument<int>(r),
            Value = new InArgument<int> {
                Expression = new Add<int, int, int> {
                    Left = new Add<int, int, int> {
                        Left = new InArgument<int>(a),
                        Right = new InArgument<int>(b)
                    },
                    Right = new InArgument<int>(c)
                }
            }
        }
    }
};

```

The same workflow can be expressed more compactly by using C# lambda expressions, as shown in the following example.

```

Variable<int> a = new Variable<int>("a", 1);
Variable<int> b = new Variable<int>("b", 2);
Variable<int> c = new Variable<int>("c", 3);
Variable<int> r = new Variable<int>("r", 0);

Sequence w = new Sequence
{
    Variables = { a, b, c, r },
    Activities =
    {
        new Assign {
            To = new OutArgument<int>(r),
            Value = new InArgument<int>((ctx) => a.Get(ctx) + b.Get(ctx) + c.Get(ctx))
        }
    }
};

```

The workflow can also be expressed by using Visual Basic expression activities, as shown in the following example.

```

Variable<int> a = new Variable<int>("a", 1);
Variable<int> b = new Variable<int>("b", 2);
Variable<int> c = new Variable<int>("c", 3);
Variable<int> r = new Variable<int>("r", 0);

Sequence w = new Sequence
{
    Variables = { a, b, c, r },
    Activities =
    {
        new Assign {
            To = new OutArgument<int>(r),
            Value = new InArgument<int>(new VisualBasicValue<int>("a + b + c"))
        }
    }
};

```

## Extending Available Expressions with Custom Expression Activities

Expressions in .NET Framework 4.6.1 are extensible allowing for additional expression activities to be created. The following example shows an activity that returns a sum of three integer values.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Activities;

namespace ExpressionsDemo
{
    public sealed class AddThreeValues : CodeActivity<int>
    {
        public InArgument<int> Value1 { get; set; }
        public InArgument<int> Value2 { get; set; }
        public InArgument<int> Value3 { get; set; }

        protected override int Execute(CodeActivityContext context)
        {
            return Value1.Get(context) +
                Value2.Get(context) +
                Value3.Get(context);
        }
    }
}

```

With this new activity you can rewrite the previous workflow that added three values as shown in the following example.

```
Variable<int> a = new Variable<int>("a", 1);
Variable<int> b = new Variable<int>("b", 2);
Variable<int> c = new Variable<int>("c", 3);
Variable<int> r = new Variable<int>("r", 0);

Sequence w = new Sequence
{
    Variables = { a, b, c, r },
    Activities =
    {
        new Assign {
            To = new OutArgument<int>(r),
            Value = new InArgument<int> {
                Expression = new AddThreeValues() {
                    Value1 = new InArgument<int>(a),
                    Value2 = new InArgument<int>(b),
                    Value3 = new InArgument<int>(c)
                }
            }
        }
    }
};
```

For more information about using expressions in code, see [Authoring Workflows, Activities, and Expressions Using Imperative Code](#).

# C# Expressions

3/9/2019 • 8 minutes to read • [Edit Online](#)

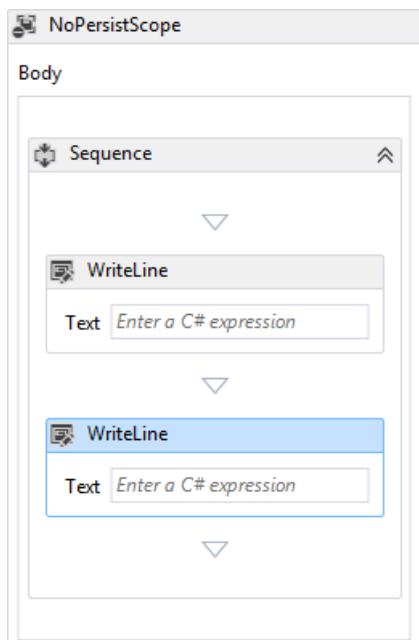
Starting with .NET Framework 4.5, C# expressions are supported in Windows Workflow Foundation (WF). New C# workflow projects created in Visual Studio 2012 that target .NET Framework 4.5 use C# expressions, and Visual Basic workflow projects use Visual Basic expressions. Existing .NET Framework 4 workflow projects that use Visual Basic expressions can be migrated to .NET Framework 4.6.1 regardless of the project language and are supported. This topic provides an overview of C# expressions in WF.

## Using C# expressions in workflows

- [Using C# expressions in the Workflow Designer](#)
  - [Backwards compatibility](#)
- [Using C# expressions in code workflows](#)
- [Using C# expressions in XAML workflows](#)
  - [Compiled Xaml](#)
  - [Loose Xaml](#)
- [Using C# expressions in XAMLX workflow services](#)

### Using C# expressions in the Workflow Designer

Starting with .NET Framework 4.5, C# expressions are supported in Windows Workflow Foundation (WF). C# workflow projects created in Visual Studio 2012 that target .NET Framework 4.5 use C# expressions, while Visual Basic workflow projects use Visual Basic expressions. To specify the desired C# expression, type it into the box labeled **Enter a C# expression**. This label is displayed in the properties window when the activity is selected in the designer, or on the activity in the workflow designer. In the following example, two `WriteLine` activities are contained within a `Sequence` inside a `NoPersistScope`.



## NOTE

C# expressions are supported only in Visual Studio, and are not supported in the re-hosted workflow designer. For more information about new WF45 features supported in the re-hosted designer, see [Support for New Workflow Foundation 4.5 Features in the Rehosted Workflow Designer](#).

## Backwards compatibility

Visual Basic expressions in existing .NET Framework 4 C# workflow projects that have been migrated to .NET Framework 4.6.1 are supported. When the Visual Basic expressions are viewed in the workflow designer, the text of the existing Visual Basic expression is replaced with **Value was set in XAML**, unless the Visual Basic expression is valid C# syntax. If the Visual Basic expression is valid C# syntax, then the expression is displayed. To update the Visual Basic expressions to C#, you can edit them in the workflow designer and specify the equivalent C# expression. It is not required to update the Visual Basic expressions to C#, but once the expressions are updated in the workflow designer they are converted to C# and may not be reverted to Visual Basic.

## Using C# expressions in code workflows

C# expressions are supported in .NET Framework 4.6.1 code based workflows, but before the workflow can be invoked the C# expressions must be compiled using `TextExpressionCompiler.Compile`. Workflow authors can use `CSharpValue` to represent the r-value of an expression, and `CSharpReference` to represent the l-value of an expression. In the following example, a workflow is created with an `Assign` activity and a `WriteLine` activity contained in a `Sequence` activity. A `CSharpReference` is specified for the `To` argument of the `Assign`, and represents the l-value of the expression. A `CSharpValue` is specified for the `Value` argument of the `Assign`, and for the `Text` argument of the `WriteLine`, and represents the r-value for those two expressions.

```
Variable<int> n = new Variable<int>
{
    Name = "n"
};

Activity wf = new Sequence
{
    Variables = { n },
    Activities =
    {
        new Assign<int>
        {
            To = new CSharpReference<int>("n"),
            Value = new CSharpValue<int>("new Random().Next(1, 101)")
        },
        new WriteLine
        {
            Text = new CSharpValue<string>("\"The number is \" + n")
        }
    }
};

CompileExpressions(wf);

WorkflowInvoker.Invoke(wf);
```

After the workflow is constructed, the C# expressions are compiled by calling the `CompileExpressions` helper method and then the workflow is invoked. The following example is the `CompileExpressions` method.

```

static void CompileExpressions(Activity activity)
{
    // activityName is the Namespace.Type of the activity that contains the
    // C# expressions.
    string activityName = activity.GetType().ToString();

    // Split activityName into Namespace and Type.Append _CompiledExpressionRoot to the type name
    // to represent the new type that represents the compiled expressions.
    // Take everything after the last . for the type name.
    string activityType = activityName.Split('.').Last() + "_CompiledExpressionRoot";
    // Take everything before the last . for the namespace.
    string activityNamespace = string.Join(".", activityName.Split('.').Reverse().Skip(1).Reverse());

    // Create a TextExpressionCompilerSettings.
    TextExpressionCompilerSettings settings = new TextExpressionCompilerSettings
    {
        Activity = activity,
        Language = "C#",
        ActivityName = activityType,
        ActivityNamespace = activityNamespace,
        RootNamespace = null,
        GenerateAsPartialClass = false,
        AlwaysGenerateSource = true,
        ForImplementation = false
    };

    // Compile the C# expression.
    TextExpressionCompilerResults results =
        new TextExpressionCompiler(settings).Compile();

    // Any compilation errors are contained in the CompilerMessages.
    if (results.HasErrors)
    {
        throw new Exception("Compilation failed.");
    }

    // Create an instance of the new compiled expression type.
    ICompiledExpressionRoot compiledExpressionRoot =
        Activator.CreateInstance(results.ResultType,
            new object[] { activity }) as ICompiledExpressionRoot;

    // Attach it to the activity.
    CompiledExpressionInvoker.SetCompiledExpressionRoot(
        activity, compiledExpressionRoot);
}

```

#### NOTE

If the C# expressions are not compiled, a [NotSupportedException](#) is thrown when the workflow is invoked with a message similar to the following: `Expression Activity type 'CSharpValue' 1' requires compilation in order to run. Please ensure that the workflow has been compiled.'`

If your custom code based workflow uses `DynamicActivity`, then some changes to the `CompileExpressions` method are required, as demonstrated in the following code example.

```

static void CompileExpressions(DynamicActivity dynamicActivity)
{
    // activityName is the Namespace.Type of the activity that contains the
    // C# expressions. For Dynamic Activities this can be retrieved using the
    // name property , which must be in the form Namespace.Type.
    string activityName = dynamicActivity.Name;

    // Split activityName into Namespace and Type.Append _CompiledExpressionRoot to the type name
    // to represent the new type that represents the compiled expressions.
    // Take everything after the last . for the type name.
    string activityType = activityName.Split('.').Last() + "_CompiledExpressionRoot";
    // Take everything before the last . for the namespace.
    string activityNamespace = string.Join(".", activityName.Split('.').Reverse().Skip(1).Reverse());

    // Create a TextExpressionCompilerSettings.
    TextExpressionCompilerSettings settings = new TextExpressionCompilerSettings
    {
        Activity = dynamicActivity,
        Language = "C#",
        ActivityName = activityType,
        ActivityNamespace = activityNamespace,
        RootNamespace = null,
        GenerateAsPartialClass = false,
        AlwaysGenerateSource = true,
        ForImplementation = true
    };

    // Compile the C# expression.
    TextExpressionCompilerResults results =
        new TextExpressionCompiler(settings).Compile();

    // Any compilation errors are contained in the CompilerMessages.
    if (results.HasErrors)
    {
        throw new Exception("Compilation failed.");
    }

    // Create an instance of the new compiled expression type.
    ICompiledExpressionRoot compiledExpressionRoot =
        Activator.CreateInstance(results.ResultType,
            new object[] { dynamicActivity }) as ICompiledExpressionRoot;

    // Attach it to the activity.
    CompiledExpressionInvoker.SetCompiledExpressionRootForImplementation(
        dynamicActivity, compiledExpressionRoot);
}

```

There are several differences in the `CompileExpressions` overload that compiles the C# expressions in a dynamic activity.

- The parameter to `CompileExpressions` is a `DynamicActivity`.
- The type name and namespace are retrieved using the `DynamicActivity.Name` property.
- `TextExpressionCompilerSettings.ForImplementation` is set to `true`.
- `CompiledExpressionInvoker.SetCompiledExpressionRootForImplementation` is called instead of `CompiledExpressionInvoker.SetCompiledExpressionRoot`.

For more information about working with expressions in code, see [Authoring Workflows, Activities, and Expressions Using Imperative Code](#).

## Using C# expressions in XAML workflows

C# expressions are supported in XAML workflows. Compiled XAML workflows are compiled into a type, and

loose XAML workflows are loaded by the runtime and compiled into an activity tree when the workflow is executed.

- [Compiled Xaml](#)
- [Loose Xaml](#)

#### **Compiled Xaml**

C# expressions are supported in compiled XAML workflows that are compiled to a type as part of a C# workflow project that targets .NET Framework 4.6.1. Compiled XAML is the default type of workflow authoring in Visual Studio, and C# workflow projects created in Visual Studio that target .NET Framework 4.6.1 use C# expressions.

#### **Loose Xaml**

C# expressions are supported in loose XAML workflows. The workflow host program that loads and invokes the loose XAML workflow must target .NET Framework 4.6.1, and [CompileExpressions](#) must be set to `true` (the default is `false`). To set [CompileExpressions](#) to `true`, create an [ActivityXamlServicesSettings](#) instance with its [CompileExpressions](#) property set to `true`, and pass it as a parameter to [ActivityXamlServices.Load](#). If [CompileExpressions](#) is not set to `true`, a [NotSupportedException](#) will be thrown with a message similar to the following: `Expression Activity type 'CSharpValue' 1' requires compilation in order to run. Please ensure that the workflow has been compiled.`

```
ActivityXamlServicesSettings settings = new ActivityXamlServicesSettings
{
    CompileExpressions = true
};

DynamicActivity<int> wf = ActivityXamlServices.Load(new StringReader(serializedAB), settings) as
DynamicActivity<int>;
```

For more information about working with XAML workflows, see [Serializing Workflows and Activities to and from XAML](#).

#### **Using C# expressions in XAMLX workflow services**

C# expressions are supported in XAMLX workflow services. When a workflow service is hosted in IIS or WAS then no additional steps are required, but if the XAML workflow service is self-hosted, then the C# expressions must be compiled. To compile the C# expressions in a self-hosted XAMLX workflow service, first load the XAMLX file into a [WorkflowService](#), and then pass the [Body](#) of the [WorkflowService](#) to the [CompileExpressions](#) method described in the previous [Using C# expressions in code workflows](#) section. In the following example, a XAMLX workflow service is loaded, the C# expressions are compiled, and then the workflow service is opened and waits for requests.

```
// Load the XAMLX workflow service.  
WorkflowService workflow1 =  
    (WorkflowService)XamlServices.Load(xamlxPath);  
  
// Compile the C# expressions in the workflow by passing the Body to CompileExpressions.  
CompileExpressions(workflow1.Body);  
  
// Initialize the WorkflowServiceHost.  
var host = new WorkflowServiceHost(workflow1, new Uri("http://localhost:8293/Service1.xamlx"));  
  
// Enable Metadata publishing/  
ServiceMetadataBehavior smb = new ServiceMetadataBehavior();  
smb.HttpGetEnabled = true;  
smb.MetadataExporter.PolicyVersion = PolicyVersion.Policy15;  
host.Description.Behaviors.Add(smb);  
  
// Open the WorkflowServiceHost and wait for requests.  
host.Open();  
Console.WriteLine("Press enter to quit");  
Console.ReadLine();
```

If the C# expressions are not compiled, the `Open` operation succeeds but the workflow will fail when it is invoked.

The following `CompileExpressions` method is the same as the method from the previous [Using C# expressions in code workflows](#) section.

```

static void CompileExpressions(Activity activity)
{
    // activityName is the Namespace.Type of the activity that contains the
    // C# expressions.
    string activityName = activity.GetType().ToString();

    // Split activityName into Namespace and Type.Append _CompiledExpressionRoot to the type name
    // to represent the new type that represents the compiled expressions.
    // Take everything after the last . for the type name.
    string activityType = activityName.Split('.').Last() + "_CompiledExpressionRoot";
    // Take everything before the last . for the namespace.
    string activityNamespace = string.Join(".", activityName.Split('.').Reverse().Skip(1).Reverse());

    // Create a TextExpressionCompilerSettings.
    TextExpressionCompilerSettings settings = new TextExpressionCompilerSettings
    {
        Activity = activity,
        Language = "C#",
        ActivityName = activityType,
        ActivityNamespace = activityNamespace,
        RootNamespace = null,
        GenerateAsPartialClass = false,
        AlwaysGenerateSource = true,
        ForImplementation = false
    };

    // Compile the C# expression.
    TextExpressionCompilerResults results =
        new TextExpressionCompiler(settings).Compile();

    // Any compilation errors are contained in the CompilerMessages.
    if (results.HasErrors)
    {
        throw new Exception("Compilation failed.");
    }

    // Create an instance of the new compiled expression type.
    ICompiledExpressionRoot compiledExpressionRoot =
        Activator.CreateInstance(results.ResultType,
            new object[] { activity }) as ICompiledExpressionRoot;

    // Attach it to the activity.
    CompiledExpressionInvoker.SetCompiledExpressionRoot(
        activity, compiledExpressionRoot);
}

```

# Properties vs. Arguments

5/4/2018 • 2 minutes to read • [Edit Online](#)

There are several options available for passing data into an activity. In addition to using [InArgument](#), activities can also be developed that receive data using either standard CLR Properties or public [ActivityAction](#) properties. This topic discusses how to select the appropriate method type.

## Using CLR Properties

When passing data into an activity, CLR properties (that is, public methods that use Get and Set routines to expose data) are the option that has the most restrictions. The value of a parameter passed into a CLR property must be known when the solution is compiled; this value will be the same for every instance of the workflow. In this way, a value passed into a CLR property is similar to a constant defined in code; this value can't change for the life of the activity, and can't be changed for different instances of the activity. [MethodName](#) is an example of a CLR property exposed by an activity; the method name that the activity calls can't be changed based on runtime conditions, and will be the same for every instance of the activity.

## Using Arguments

Arguments should be used when data is only evaluated once during the lifetime of the activity; that is, its value will not change during the lifetime of the activity, but the value can be different for different instances of the activity. [Condition](#) is an example of a value that gets evaluated once; therefore it is defined as an argument. [Text](#) is another example of a method that should be defined as an argument, since it is only evaluated once during the activity's execution, but it can be different for different instances of the activity.

## Using ActivityAction

When data needs to be evaluated multiple times during the lifetime of an activity's execution, an [ActivityAction](#) should be used. For example, the [Condition](#) property is evaluated for each iteration of the [While](#) loop. If an [InArgument](#) were used for this purpose, the loop would never exit, since the argument would not be reevaluated for each iteration, and would always return the same result.

# Exposing data with CacheMetadata

3/6/2019 • 2 minutes to read • [Edit Online](#)

Before executing an activity, the workflow runtime obtains all of the information about the activity that it needs in order to maintain its execution. The workflow runtime gets this information during the execution of the [CacheMetadata](#) method. The default implementation of this method provides the runtime with all of the public arguments, variables, and child activities exposed by the activity at the time it is executed; if the activity needs to give more information to the runtime than this (such as private members, or activities to be scheduled by the activity), this method can be overridden to provide it.

## Default CacheMetadata behavior

The default implementation of [CacheMetadata](#) for activities that derive from [NativeActivity](#) processes the following method types in the following ways:

- [InArgument<T>](#), [OutArgument<T>](#), or [InOutArgument<T>](#) (generic arguments): These arguments are exposed to the runtime as arguments with a name and type equal to the exposed property name and type, the appropriate argument direction, and some validation data.
- [Variable](#) or any subclass thereof: These members are exposed to the runtime as public variables.
- [Activity](#) or any subclass thereof: These members are exposed to the runtime as public child activities. The default behavior can be implemented explicitly by calling [AddImportedChild](#), passing in the child activity.
- [ActivityDelegate](#) or any subclass thereof: These members are exposed to the runtime as public delegates.
- [ICollection](#) of type [Variable](#): All elements in the collection are exposed to the runtime as public variables.
- [ICollection](#) of type [Activity](#): All elements in the collection are exposed to the runtime as public children.
- [ICollection](#) of type [ActivityDelegate](#): All elements in the collection are exposed to the runtime as public delegates.

The [CacheMetadata](#) for activities that derive from [Activity](#), [CodeActivity](#), and [AsyncCodeActivity](#) also function as above, except for the following differences:

- Classes that derive from [Activity](#) cannot schedule child activities or delegates, so such members are exposed as imported children and delegates; the
- Classes that derive from [CodeActivity](#) and [AsyncCodeActivity](#) do not support variables, children, or delegates, so only arguments will be exposed.

## Overriding CacheMetadata to provide information to the runtime

The following code snippet demonstrates how to add information about members to an activity's metadata during the execution of the [CacheMetadata](#) method. Note that the base of the method is called to cache all public data about the activity.

```

protected override void CacheMetadata(NativeActivityMetadata metadata)
{
    base.CacheMetadata(metadata);
    metadata.AddImplementationChild(this._writeLine);
    metadata.AddVariable(this._myVariable);
    metadata.AddImplementationVariable(this._myImplementationVariable);

    RuntimeArgument argument = new RuntimeArgument("MyArgument", ArgumentDirection.In, typeof(SomeType));
    metadata.Bind(argument, this.SomeName);
    metadata.AddArgument(argument);
}

```

## Using CacheMetadata to expose implementation children

In order to pass data to child activities that are to be scheduled by an activity using variables, it is necessary to add the variables as implementation variables; public variables cannot have their values set this way. The reason for this is that activities are intended to be executed more as implementations of functions (which have parameters), rather than encapsulated classes (which have properties). However, there are situations in which the arguments must be explicitly set, such as when using [ScheduleActivity](#), since the scheduled activity doesn't have access to the parent activity's arguments in the way a child activity would.

The following code snippet demonstrates how to pass an argument from a native activity into a scheduled activity using [CacheMetadata](#).

```

public sealed class ChildActivity : NativeActivity
{
    public WriteLine _writeLine;
    public InArgument<string> Message { get; set; }
    private Variable<string> MessageVariable { get; set; }
    public ChildActivity()
    {
        MessageVariable = new Variable<string>();
        _writeLine = new WriteLine
        {
            Text = new InArgument<string>(MessageVariable),
        };
    }
    protected override void CacheMetadata(NativeActivityMetadata metadata)
    {
        base.CacheMetadata(metadata);
        metadata.AddImplementationVariable(this.MessageVariable);
        metadata.AddImplementationChild(this._writeLine);
    }
    protected override void Execute(NativeActivityContext context)
    {
        string configuredMessage = context.GetValue(Message);
        context.SetValue(MessageVariable, configuredMessage);
        context.ScheduleActivity(this._writeLine);
    }
}

```

# Waiting for Input in a Workflow

3/9/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section discuss how to use bookmarks and messaging activities.

## In This Section

### [Bookmarks](#)

Describes how to use bookmarks.

# Bookmarks

5/4/2018 • 3 minutes to read • [Edit Online](#)

Bookmarks are the mechanism that enables an activity to passively wait for input without holding onto a workflow thread. When an activity signals that it is waiting for stimulus, it can create a bookmark. This indicates to the runtime that the activity's execution should not be considered complete even when the currently executing method (which created the [Bookmark](#)) returns.

## Bookmark Basics

A [Bookmark](#) represents a point at which execution can be resumed (and through which input can be delivered) within a workflow instance. Typically, a [Bookmark](#) is given a name and external (host or extension) code is responsible for resuming the bookmark with relevant data. When a [Bookmark](#) is resumed, the workflow runtime schedules the [BookmarkCallback](#) delegate that was associated with that [Bookmark](#) at the time of its creation.

## Bookmark Options

The [BookmarkOptions](#) class specifies the type of [Bookmark](#) being created. The possible non mutually-exclusive values are [None](#), [MultipleResume](#), and [NonBlocking](#). Use [None](#), the default, when creating a [Bookmark](#) that is expected to be resumed exactly once. Use [MultipleResume](#) when creating a [Bookmark](#) that can be resumed multiple times. Use [NonBlocking](#) when creating a [Bookmark](#) that might never be resumed. Unlike bookmarks created using the default [BookmarkOptions](#), [NonBlocking](#) bookmarks do not prevent an activity from completing.

## Bookmark Resumption

Bookmarks can be resumed by code outside of a workflow using one of the [ResumeBookmark](#) overloads. In this example, a `ReadLine` activity is created. When executed, the `ReadLine` activity creates a [Bookmark](#), registers a callback, and then waits for the [Bookmark](#) to be resumed. When it is resumed, the `ReadLine` activity assigns the data that was passed with the [Bookmark](#) to its [Result](#) argument.

```

public sealed class ReadLine : NativeActivity<string>
{
    [RequiredArgument]
    public InArgument<string> BookmarkName { get; set; }

    protected override void Execute(NativeActivityContext context)
    {
        // Create a Bookmark and wait for it to be resumed.
        context.CreateBookmark(BookmarkName.Get(context),
            new BookmarkCallback(OnResumeBookmark));
    }

    // NativeActivity derived activities that do asynchronous operations by calling
    // one of the CreateBookmark overloads defined on System.Activities.NativeActivityContext
    // must override the CanInduceIdle property and return true.
    protected override bool CanInduceIdle
    {
        get { return true; }
    }

    public void OnResumeBookmark(NativeActivityContext context, Bookmark bookmark, object obj)
    {
        // When the Bookmark is resumed, assign its value to
        // the Result argument.
        Result.Set(context, (string)obj);
    }
}

```

In this example, a workflow is created that uses the `ReadLine` activity to gather the user's name and display it to the console window. The host application performs the actual work of gathering the input and passes it to the workflow by resuming the `Bookmark`.

```

Variable<string> name = new Variable<string>
{
    Name = "name"
};

Activity wf = new Sequence
{
    Variables =
    {
        name
    },
    Activities =
    {
        new WriteLine()
        {
            Text = "What is your name?"
        },
        new ReadLine()
        {
            BookmarkName = "UserName",
            Result = name
        },
        new WriteLine()
        {
            Text = new InArgument<string>((env) => "Hello, " + name.Get(env))
        }
    }
};

AutoResetEvent syncEvent = new AutoResetEvent(false);

// Create the WorkflowApplication using the desired
// workflow definition.
WorkflowApplication wfApp = new WorkflowApplication(wf);

// Handle the desired lifecycle events.
wfApp.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
{
    // Signal the host that the workflow is complete.
    syncEvent.Set();
};

// Start the workflow.
wfApp.Run();

// Collect the user's name and resume the bookmark.
// Bookmark resumption only occurs when the workflow
// is idle. If a call to ResumeBookmark is made and the workflow
// is not idle, ResumeBookmark blocks until the workflow becomes
// idle before resuming the bookmark.
wfApp.ResumeBookmark("UserName", Console.ReadLine());

// Wait for Completed to arrive and signal that
// the workflow is complete.
syncEvent.WaitOne();

```

When the `ReadLine` activity is executed, it creates a `Bookmark` named `UserName` and then waits for the bookmark to be resumed. The host collects the desired data and then resumes the `Bookmark`. The workflow resumes, displays the name, and then completes. Note that no synchronization code is required with regard to resuming the bookmark. A `Bookmark` can only be resumed when the workflow is idle, and if the workflow is not idle, the call to `ResumeBookmark` blocks until the workflow becomes idle.

## Bookmark Resumption Result

`ResumeBookmark` returns a `BookmarkResumptionResult` enumeration value to indicate the results of the bookmark resumption request. The possible return values are `Success`, `NotReady`, and `NotFound`. Hosts and extensions can use this value to determine how to proceed.

# Exceptions, Transactions, and Compensation

3/9/2019 • 2 minutes to read • [Edit Online](#)

WF provides several different mechanisms for handling run-time error conditions in workflows. Workflows can use a combination of exception handlers, transactions, cancellation, and compensation to handle and recover gracefully from error conditions.

## In This Section

### [Exceptions](#)

Demonstrates how to use the [TryCatch](#) activity to handle exceptions in a workflow.

### [Transactions](#)

Demonstrates how to use the [TransactionScope](#) activity to use transactions in a workflow.

### [Compensation](#)

Describes compensation in workflows and demonstrates how to use compensation activities such as [CompensableActivity](#), [Compensate](#), and [Confirm](#).

### [Cancellation](#)

Describes how to perform cancellation handling in workflows using built-in activities as well as custom activities.

### [Debugging Workflows](#)

Describes how to debug workflows.

# Exceptions

3/9/2019 • 4 minutes to read • [Edit Online](#)

Workflows can use the [TryCatch](#) activity to handle exceptions that are raised during the execution of a workflow. These exceptions can be handled or they can be re-thrown using the [Rethrow](#) activity. Activities in the [Finally](#) section are executed when either the [Try](#) section or the [Catches](#) section completes. Workflows hosted by a [WorkflowApplication](#) instance can also use the [OnUnhandledException](#) event handler to handle exceptions that are not handled by a [TryCatch](#) activity.

## Causes of Exceptions

In a workflow, exceptions can be generated in the following ways:

- A time-out of transactions in [TransactionScope](#).
- An explicit exception thrown by the workflow using the [Throw](#) activity.
- A .NET Framework 4.6.1 exception thrown from an activity.
- An exception thrown from external code, such as libraries, components, or services that are used in the workflow.

## Handling Exceptions

If an exception is thrown by an activity and is unhandled, the default behavior is to terminate the workflow instance. If a custom [OnUnhandledException](#) handler is present, it can override this default behavior. This handler gives the workflow host author an opportunity to provide the appropriate handling, such as custom logging, aborting the workflow, canceling the workflow, or terminating the workflow. If a workflow raises an exception that is not handled, the [OnUnhandledException](#) handler is invoked. There are three possible actions returned from [OnUnhandledException](#) which determine the final outcome of the workflow.

- **Cancel** - A cancelled workflow instance is a graceful exit of a branch execution. You can model cancellation behavior (for example, by using a [CancellationScope](#) activity). The [Completed](#) handler is invoked when the cancellation process completes. A cancelled workflow is in the [Cancelled](#) state.
- **Terminate** - A terminated workflow instance cannot be resumed or restarted. This triggers the [Completed](#) event in which you can provide an exception as the reason it was terminated. The [Terminated](#) handler is invoked when the termination process completes. A terminated workflow is in the [Faulted](#) state.
- **Abort** - An aborted workflow instances can be resumed only if it has been configured to be persistent. Without persistence, a workflow cannot be resumed. At the point a workflow is aborted, any work done (in memory) since the last persistence point will be lost. For an aborted workflow, the [Aborted](#) handler is invoked using the exception as the reason when the abort process completes. However, unlike [Cancelled](#) and [Terminated](#), the [Completed](#) handler is not invoked. An aborted workflow is in an [Aborted](#) state.

The following example invokes a workflow that throws an exception. The exception is unhandled by the workflow and the [OnUnhandledException](#) handler is invoked. The [WorkflowApplicationUnhandledEventArgs](#) are inspected to provide information about the exception, and the workflow is terminated.

```

Activity wf = new Sequence
{
    Activities =
    {
        new WriteLine
        {
            Text = "Starting the workflow."
        },
        new Throw
        {
            Exception = new InArgument<Exception>((env) =>
                new ApplicationException("Something unexpected happened."))
        },
        new WriteLine
        {
            Text = "Ending the workflow."
        }
    }
};

WorkflowApplication wfApp = new WorkflowApplication(wf);

wfApp.OnUnhandledException = delegate(WorkflowApplicationUnhandledEventArgs e)
{
    // Display the unhandled exception.
    Console.WriteLine("OnUnhandledException in Workflow {0}\n{1}",
        e.InstanceId, e.UnhandledException.Message);

    Console.WriteLine("ExceptionSource: {0} - {1}",
        e.ExceptionSource.DisplayName, e.ExceptionSourceInstanceId);

    // Instruct the runtime to terminate the workflow.
    return UnhandledExceptionAction.Terminate;

    // Other choices are UnhandledExceptionAction.Abort and
    // UnhandledExceptionAction.Cancel
};

wfApp.Run();

```

## Handling Exceptions with the TryCatch Activity

Handling exceptions inside a workflow is performed with the [TryCatch](#) activity. The [TryCatch](#) activity has a [Catches](#) collection of [Catch](#) activities that are each associated with a specific [Exception](#) type. If the exception thrown by an activity that is contained in the [Try](#) section of a [TryCatch](#) activity matches the exception of a [Catch<TException>](#) activity in the [Catches](#) collection, then the exception is handled. If the exception is explicitly re-thrown or a new exception is thrown then this exception passes up to the parent activity. The following code example shows a [TryCatch](#) activity that handles an [ApplicationException](#) that is thrown in the [Try](#) section by a [Throw](#) activity. The exception's message is written to the console by the [Catch<TException>](#) activity, and then a message is written to the console in the [Finally](#) section.

```

DelegateInArgument<ApplicationException> ex = new DelegateInArgument<ApplicationException>()
{
    Name = "ex"
};

Activity wf = new TryCatch
{
    Try = new Throw()
    {
        Exception = new InArgument<Exception>((env) => new ApplicationException("An ApplicationException was thrown."))
    },
    Catches =
    {
        new Catch<ApplicationException>
        {
            Action = new ActivityAction<ApplicationException>
            {
                Argument = ex,
                Handler = new WriteLine()
                {
                    Text = new InArgument<string>((env) => ex.Get(env).Message)
                }
            }
        },
        Finally = new WriteLine()
        {
            Text = "Executing in Finally."
        }
    }
};

```

The activities in the **Finally** section are executed when either the **Try** section or the **Catches** section successfully completes. The **Try** section successfully completes if no exceptions are thrown from it, and the **Catches** section successfully completes if no exceptions are thrown or rethrown from it. If an exception is thrown in the **Try** section of a **TryCatch** and is either not handled by a **Catch<TException>** in the **Catches** section, or is rethrown from the **Catches**, the activities in the **Finally** will not be executed unless the one of the following occurs.

- The exception is caught by a higher level **TryCatch** activity in the workflow, regardless of whether it is rethrown from that higher level **TryCatch**.
- The exception is not handled by a higher level **TryCatch**, escapes the root of the workflow, and the workflow is configured to cancel instead of terminate or abort. Workflows hosted using **WorkflowApplication** can configure this by handling **OnUnhandledException** and returning **Cancel**. An example of handling **OnUnhandledException** is provided previously in this topic. Workflow services can configure this by using **WorkflowUnhandledExceptionBehavior** and specifying **Cancel**. For an example of configuring **WorkflowUnhandledExceptionBehavior**, see [Workflow Service Host Extensibility](#).

## Exception Handling versus Compensation

The difference between exception handling and compensation is that exception handling occurs during the execution of an activity. Compensation occurs after an activity has successfully completed. Exception handling provides an opportunity to clean up after the activity raises the exception, whereas compensation provides a mechanism by which the successfully completed work of a previously completed activity can be undone. For more information, see [Compensation](#).

## See also

- [TryCatch](#)

- [OnUnhandledException](#)
- [CompensableActivity](#)

# Workflow Transactions

3/9/2019 • 2 minutes to read • [Edit Online](#)

WF provides support for participating in [System.Transactions](#) transactions by using the [TransactionScope](#) activity to scope a transacted unit of work. While the [System.Transactions.TransactionScope](#) must be explicitly completed the [System.Activities.Statements.TransactionScope](#) activity implicitly calls complete on the transaction upon successful completion. Any activities that are contained in the [Body](#) of the [TransactionScope](#) activity participate in the transaction. WF can flow transactions into a workflow through the use of the [TransactedReceiveScope](#) activity. Like the [TransactionScope](#) activity, any activity contained in the [Body](#) participates in the transaction. WF ensures that activities dependent on [Transaction.Current](#) works with both [TransactionScope](#) and [TransactedReceiveScope](#). If the system-provided activities do not address all requirements, custom activities can be built using the [RuntimeTransactionHandle](#) to enable advanced flow and transaction control scenarios.

In the following example, a workflow is constructed consisting of a [Sequence](#) activity that contains child activities including a [TransactionScope](#) activity. The [Body](#) activities of the [TransactionScope](#) execute under the transaction initialized by the [TransactionScope](#) activity.

```
static Activity ScenarioOne()
{
    return new Sequence
    {
        Activities =
        {
            new WriteLine { Text = "    Begin workflow" },

            new TransactionScope
            {
                Body = new Sequence
                {
                    Activities =
                    {
                        new WriteLine { Text = "    Begin TransactionScope" },
                        new PrintTransactionId(),
                        new TransactionScopeTest(),
                        new WriteLine { Text = "    End TransactionScope" },
                    },
                },
            },
            new WriteLine { Text = "    End workflow" },
        };
    };
}
```

For more information, see about using [TransactedReceiveScope](#), see [Flowing Transactions into and out of Workflow Services](#).

## See also

- [TransactionScope](#)
- [TransactedReceiveScope](#)
- [Transaction.Current](#)



# Compensation

3/9/2019 • 9 minutes to read • [Edit Online](#)

Compensation in Windows Workflow Foundation (WF) is the mechanism by which previously completed work can be undone or compensated (following the logic defined by the application) when a subsequent failure occurs. This section describes how to use compensation in workflows.

## Compensation vs. Transactions

A transaction allows you to combine multiple operations into a single unit of work. Using a transaction gives your application the ability to abort (roll back) all changes executed from within the transaction if any errors occur during any part of the transaction process. However, using transactions may not be appropriate if the work is long running. For example, a travel planning application is implemented as a workflow. The steps of the workflow may consist of booking a flight, waiting for manager approval, and then paying for the flight. This process could take many days and it is not practical for the steps of booking and paying for the flight to participate in the same transaction. In a scenario such as this, compensation could be used to undo the booking step of the workflow if there is a failure later in the processing.

### NOTE

This topic covers compensation in workflows. For more information about transactions in workflows, see [Transactions](#) and [TransactionScope](#). For more information about transactions, see [System.Transactions](#) and [System.Transactions.Transaction](#).

## Using CompensableActivity

[CompensableActivity](#) is the core compensation activity in WF. Any activities that perform work that may need to be compensated are placed into the [Body](#) of a [CompensableActivity](#). In this example, the reservation step of purchasing a flight is placed into the [Body](#) of a [CompensableActivity](#) and the cancellation of the reservation is placed into the [CompensationHandler](#). Immediately following the [CompensableActivity](#) in the workflow are two activities that wait for manager approval and then complete the purchasing step of the flight. If an error condition causes the workflow to be canceled after the [CompensableActivity](#) has successfully completed, then the activities in the [CompensationHandler](#) handler are scheduled and the flight is canceled.

```
Activity wf = new Sequence()
{
    Activities =
    {
        new CompensableActivity
        {
            Body = new ReserveFlight(),
            CompensationHandler = new CancelFlight()
        },
        new ManagerApproval(),
        new PurchaseFlight()
    }
};
```

The following example is the workflow in XAML.

```

<Sequence
    xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
    xmlns:c="clr-namespace:CompensationExample;assembly=CompensationExample"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <CompensableActivity>
        <CompensableActivity.Result>
            <OutArgument
                x:TypeArguments="CompensationToken" />
        </CompensableActivity.Result>
        <CompensableActivity.CompensationHandler>
            <c:CancelFlight />
        </CompensableActivity.CompensationHandler>
        <c:ReserveFlight />
    </CompensableActivity>
    <c:ManagerApproval />
    <c:PurchaseFlight />
</Sequence>

```

When the workflow is invoked, the following output is displayed to the console.

**ReserveFlight:** Ticket is reserved.

**ManagerApproval:** Manager approval received.

**PurchaseFlight:** Ticket is purchased.

**Workflow completed successfully with status: Closed.**

#### NOTE

The sample activities in this topic such as `ReserveFlight` display their name and purpose to the console to help illustrate the order in which the activities are executed when compensation occurs.

### Default Workflow Compensation

By default, if the workflow is canceled, the compensation logic is run for any compensable activity that has successfully completed and has not already been confirmed or compensated.

#### NOTE

When a `CompensableActivity` is *confirmed*, compensation for the activity can no longer be invoked. The confirmation process is described later in this section.

In this example, an exception is thrown after the flight is reserved but before the manager approval step.

```

Activity wf = new Sequence()
{
    Activities =
    {
        new CompensableActivity
        {
            Body = new ReserveFlight(),
            CompensationHandler = new CancelFlight()
        },
        new SimulatedErrorCondition(),
        new ManagerApproval(),
        new PurchaseFlight()
    }
};

```

This example is the workflow in XAML.

```

<Sequence
    xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
    xmlns:c="clr-namespace:CompensationExample;assembly=CompensationExample"
    xmlns:x="http://schemas.microsoft.com/wifx/2006/xaml">
    <CompensableActivity>
        <CompensableActivity.Result>
            <OutArgument
                x:TypeArguments="CompensationToken" />
        </CompensableActivity.Result>
        <CompensableActivity.CompensationHandler>
            <c:CancelFlight />
        </CompensableActivity.CompensationHandler>
        <c:ReserveFlight />
    </CompensableActivity>
    <c:SimulatedErrorCondition />
    <c:ManagerApproval />
    <c:PurchaseFlight />
</Sequence>

```

```

AutoResetEvent syncEvent = new AutoResetEvent(false);
WorkflowApplication wfApp = new WorkflowApplication(wf);

wfApp.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
{
    if (e.TerminationException != null)
    {
        Console.WriteLine("Workflow terminated with exception:\n{0}: {1}",
            e.TerminationException.GetType().FullName,
            e.TerminationException.Message);
    }
    else
    {
        Console.WriteLine("Workflow completed successfully with status: {0}.",
            e.CompletionState);
    }

    syncEvent.Set();
};

wfApp.OnUnhandledException = delegate(WorkflowApplicationUnhandledExceptionEventArgs e)
{
    Console.WriteLine("Workflow Unhandled Exception:\n{0}: {1}",
        e.UnhandledException.GetType().FullName,
        e.UnhandledException.Message);

    return UnhandledExceptionAction.Cancel;
};

wfApp.Run();
syncEvent.WaitOne();

```

When the workflow is invoked, the simulated error condition exception is handled by the host application in [OnUnhandledException](#), the workflow is canceled, and the compensation logic is invoked.

**ReserveFlight: Ticket is reserved.**

**SimulatedErrorCondition: Throwing an ApplicationException.**

**Workflow Unhandled Exception:**

**System.ApplicationException: Simulated error condition in the workflow.**

**CancelFlight: Ticket is canceled.**

**Workflow completed successfully with status: Canceled.**

**Cancellation and CompensableActivity**

If the activities in the [Body](#) of a [CompensableActivity](#) have not completed and the activity is canceled, the activities in the [CancellationHandler](#) are executed.

#### NOTE

The [CancellationHandler](#) is only invoked if the activities in the [Body](#) of the [CompensableActivity](#) have not completed and the activity is canceled. The [CompensationHandler](#) is only executed if the activities in the [Body](#) of the [CompensableActivity](#) have successfully completed and compensation is subsequently invoked on the activity.

The [CancellationHandler](#) gives workflow authors the opportunity to provide any appropriate cancellation logic. In the following example, an exception is thrown during the execution of the [Body](#), and then the [CancellationHandler](#) is invoked.

```
Activity wf = new Sequence()
{
    Activities =
    {
        new CompensableActivity
        {
            Body = new Sequence
            {
                Activities =
                {
                    new ChargeCreditCard(),
                    new SimulatedErrorCondition(),
                    new ReserveFlight()
                }
            },
            CompensationHandler = new CancelFlight(),
            CancellationHandler = new CancelCreditCard()
        },
        new ManagerApproval(),
        new PurchaseFlight()
    }
};
```

This example is the workflow in XAML

```

<Sequence
    xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
    xmlns:c="clr-namespace:CompensationExample;assembly=CompensationExample"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <CompensableActivity>
        <CompensableActivity.Result>
            <OutArgument
                x:TypeArguments="CompensationToken" />
        </CompensableActivity.Result>
        <Sequence>
            <c:ChargeCreditCard />
            <c:SimulatedErrorCondition />
            <c:ReserveFlight />
        </Sequence>
        <CompensableActivity.CancellationHandler>
            <c:CancelCreditCard />
        </CompensableActivity.CancellationHandler>
        <CompensableActivity.CompensationHandler>
            <c:CancelFlight />
        </CompensableActivity.CompensationHandler>
    </CompensableActivity>
    <c:ManagerApproval />
    <c:PurchaseFlight />
</Sequence>

```

When the workflow is invoked, the simulated error condition exception is handled by the host application in [OnUnhandledException](#), the workflow is canceled, and the cancellation logic of the [CompensableActivity](#) is invoked. In this example, the compensation logic and the cancellation logic have different goals. If the [Body](#) completed successfully, then this means the credit card was charged and the flight booked, so the compensation should undo both steps. (In this example, canceling the flight automatically cancels the credit card charges.) However, if the [CompensableActivity](#) is canceled, this means the [Body](#) did not complete and so the logic of the [CancellationHandler](#) needs to be able to determine how to best handle the cancellation. In this example, the [CancellationHandler](#) cancels the credit card charge, but since [ReserveFlight](#) was the last activity in the [Body](#), it does not attempt to cancel the flight. Since [ReserveFlight](#) was the last activity in the [Body](#), if it had successfully completed then the [Body](#) would have completed and no cancellation would be possible.

**ChargeCreditCard:** Charge credit card for flight.

**SimulatedErrorCondition:** Throwing an [ApplicationException](#).

**Workflow Unhandled Exception:**

**System.ApplicationException:** Simulated error condition in the workflow.

**CancelCreditCard:** Cancel credit card charges.

**Workflow completed successfully with status: Canceled.** For more information about cancellation, see [Cancellation](#).

## Explicit Compensation Using the Compensate Activity

In the previous section, implicit compensation was covered. Implicit compensation can be appropriate for simple scenarios, but if more explicit control is required over the scheduling of compensation handling then the [Compensate](#) activity can be used. To initiate the compensation process with the [Compensate](#) activity, the [CompensationToken](#) of the [CompensableActivity](#) for which compensation is desired is used. The [Compensate](#) activity can be used to initiate compensation on any completed [CompensableActivity](#) that has not been confirmed or compensated. For example, a [Compensate](#) activity could be used in the [Catches](#) section of a [TryCatch](#) activity, or any time after the [CompensableActivity](#) has completed. In this example, the [Compensate](#) activity is used in the [Catches](#) section of a [TryCatch](#) activity to reverse the action of the [CompensableActivity](#).

```

Variable<CompensationToken> token1 = new Variable<CompensationToken>
{
    Name = "token1",
};

Activity wf = new TryCatch()
{
    Variables =
    {
        token1
    },
    Try = new Sequence
    {
        Activities =
        {
            new CompensableActivity
            {
                Body = new ReserveFlight(),
                CompensationHandler = new CancelFlight(),
                ConfirmationHandler = new ConfirmFlight(),
                Result = token1
            },
            new SimulatedErrorCondition(),
            new ManagerApproval(),
            new PurchaseFlight()
        }
    },
    Catches =
    {
        new Catch<ApplicationException>()
        {
            Action = new ActivityAction<ApplicationException>()
            {
                Handler = new Compensate()
                {
                    Target = token1
                }
            }
        }
    }
};

```

This example is the workflow in XAML.

```

<TryCatch
    xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
    xmlns:c="clr-namespace:CompensationExample;assembly=CompensationExample"
    xmlns:s="clr-namespace:System;assembly=mscorlib"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <TryCatch.Variables>
        <Variable
            x:TypeArguments="CompensationToken"
            x:Name="__ReferenceID0"
            Name="token1" />
    </TryCatch.Variables>
    <TryCatch.Try>
        <Sequence>
            <CompensableActivity>
                <CompensableActivity.Result>
                    <OutArgument
                        x:TypeArguments="CompensationToken">
                        <VariableReference
                            x:TypeArguments="CompensationToken"
                            Variable="{x:Reference __ReferenceID0}">
                            <VariableReference.Result>
                                <OutArgument
                                    x:TypeArguments="Location(CompensationToken)" />
                            </VariableReference.Result>
                        </VariableReference>
                    </OutArgument>
                </CompensableActivity.Result>
                <CompensableActivity.CompensationHandler>
                    <c:CancelFlight />
                </CompensableActivity.CompensationHandler>
                <CompensableActivity.ConfirmationHandler>
                    <c:ConfirmFlight />
                </CompensableActivity.ConfirmationHandler>
                <c:ReserveFlight />
            </CompensableActivity>
            <c:SimulatedErrorCondition />
            <c:ManagerApproval />
            <c:PurchaseFlight />
        </Sequence>
    </TryCatch.Try>
    <TryCatch.Catches>
        <Catch
            x:TypeArguments="s:ApplicationException">
            <ActivityAction
                x:TypeArguments="s:ApplicationException">
                <Compensate>
                    <Compensate.Target>
                        <InArgument
                            x:TypeArguments="CompensationToken">
                            <VariableValue
                                x:TypeArguments="CompensationToken"
                                Variable="{x:Reference __ReferenceID0}">
                                <VariableValue.Result>
                                    <OutArgument
                                        x:TypeArguments="CompensationToken" />
                                </VariableValue.Result>
                            </VariableValue>
                        </InArgument>
                    </Compensate.Target>
                </Compensate>
            </ActivityAction>
        </Catch>
    </TryCatch.Catches>
</TryCatch>

```

When the workflow is invoked, the following output is displayed to the console.

**ReserveFlight:** Ticket is reserved.

**SimulatedErrorCondition:** Throwing an ApplicationException.

**CancelFlight:** Ticket is canceled.

**Workflow completed successfully with status: Closed.**

### Confirming Compensation

By default, compensable activities can be compensated any time after they have completed. In some scenarios this may not be appropriate. In the previous example the compensation to reserving the ticket was to cancel the reservation. However, after the flight has been completed this compensation step is no longer valid. Confirming the compensable activity invokes the activity specified by the [ConfirmationHandler](#). One possible use for this is to allow any resources that are necessary to perform the compensation to be released. After a compensable activity is confirmed it is not possible for it to be compensated, and if this is attempted an [InvalidOperationException](#) exception is thrown. When a workflow completes successfully, all non-confirmed and non-compensated compensable activities that completed successfully are confirmed in reverse order of completion. In this example the flight is reserved, purchased, and completed, and then the compensable activity is confirmed. To confirm a [CompensableActivity](#), use the [Confirm](#) activity and specify the [CompensationToken](#) of the [CompensableActivity](#) to confirm.

```
Variable<CompensationToken> token1 = new Variable<CompensationToken>
{
    Name = "token1",
};

Activity wf = new Sequence()
{
    Variables =
    {
        token1
    },
    Activities =
    {
        new CompensableActivity
        {
            Body = new ReserveFlight(),
            CompensationHandler = new CancelFlight(),
            ConfirmationHandler = new ConfirmFlight(),
            Result = token1
        },
        new ManagerApproval(),
        new PurchaseFlight(),
        new TakeFlight(),
        new Confirm()
        {
            Target = token1
        }
    }
};
```

This example is the workflow in XAML.

```

<Sequence
    xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
    xmlns:c="clr-namespace:CompensationExample;assembly=CompensationExample"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Sequence.Variables>
        <x:Reference>__ReferenceID0</x:Reference>
    </Sequence.Variables>
    <CompensableActivity>
        <CompensableActivity.Result>
            <OutArgument
                x:TypeArguments="CompensationToken">
                <VariableReference
                    x:TypeArguments="CompensationToken">
                    <VariableReference.Result>
                        <OutArgument
                            x:TypeArguments="Location(CompensationToken)" />
                    </VariableReference.Result>
                    <VariableReference.Variable>
                        <Variable
                            x:TypeArguments="CompensationToken"
                            x:Name="__ReferenceID0"
                            Name="token1" />
                    </VariableReference.Variable>
                </VariableReference>
            </OutArgument>
        </CompensableActivity.Result>
        <CompensableActivity.CompensationHandler>
            <c:CancelFlight />
        </CompensableActivity.CompensationHandler>
        <CompensableActivity.ConfirmationHandler>
            <c:ConfirmFlight />
        </CompensableActivity.ConfirmationHandler>
        <c:ReserveFlight />
    </CompensableActivity>
    <c:ManagerApproval />
    <c:PurchaseFlight />
    <c:TakeFlight />
    <Confirm>
        <Confirm.Target>
            <InArgument
                x:TypeArguments="CompensationToken">
                <VariableValue
                    x:TypeArguments="CompensationToken"
                    Variable="{x:Reference __ReferenceID0}">
                    <VariableValue.Result>
                        <OutArgument
                            x:TypeArguments="CompensationToken" />
                    </VariableValue.Result>
                </VariableValue>
            </InArgument>
        </Confirm.Target>
    </Confirm>
</Sequence>

```

When the workflow is invoked, the following output is displayed to the console.

**ReserveFlight: Ticket is reserved.**

**ManagerApproval: Manager approval received.**

**PurchaseFlight: Ticket is purchased.**

**TakeFlight: Flight is completed.**

**ConfirmFlight: Flight has been taken, no compensation possible.**

**Workflow completed successfully with status: Closed.**

## Nesting Compensation Activities

A [CompensableActivity](#) can be placed into the [Body](#) section of another [CompensableActivity](#). A [CompensableActivity](#) may not be placed into a handler of another [CompensableActivity](#). It is the responsibility of a parent [CompensableActivity](#) to ensure that when it is canceled, confirmed, or compensated, all child compensable activities that have completed successfully and have not already been confirmed or compensated must be confirmed or compensated before the parent completes cancellation, confirmation, or compensation. If this is not modeled explicitly the parent [CompensableActivity](#) will implicitly compensate child compensable activities if the parent received the cancel or compensate signal. If the parent received the confirm signal the parent will implicitly confirm child compensable activities. If the logic to handle cancellation, confirmation, or compensation is explicitly modeled in the handler of the parent [CompensableActivity](#), any child not explicitly handled will be implicitly confirmed.

## See also

- [CompensableActivity](#)
- [Compensate](#)
- [Confirm](#)
- [CompensationToken](#)

# Modeling Cancellation Behavior in Workflows

3/9/2019 • 9 minutes to read • [Edit Online](#)

Activities can be canceled inside a workflow, for example by a [Parallel](#) activity canceling incomplete branches when its [CompletionCondition](#) evaluates to `true`, or from outside the workflow, if the host calls [Cancel](#). To provide cancellation handling, workflow authors can use the [CancellationScope](#) activity, the [CompensableActivity](#) activity, or create custom activities that provide cancellation logic. This topic provides an overview of cancellation in workflows.

## Cancellation, Compensation, and Transactions

Transactions give your application the ability to abort (roll back) all changes executed within the transaction if any errors occur during any part of the transaction process. However, not all work that may need to be canceled or undone is appropriate for transactions, such as long-running work or work that does not involve transactional resources. Compensation provides a model for undoing previously completed non-transactional work if there is a subsequent failure in the workflow. Cancellation provides a model for workflow and activity authors to handle non-transactional work that was not completed. If an activity has not completed its execution and it is canceled, its cancellation logic will be invoked if it is available.

### NOTE

For more information about transactions and compensation, see [Transactions](#) and [Compensation](#).

## Using CancellationScope

The [CancellationScope](#) activity has two sections that can contain child activities: [Body](#) and [CancellationHandler](#). The [Body](#) is where the activities that make up the logic of the activity are placed, and the [CancellationHandler](#) is where the activities that provide cancellation logic for the activity are placed. An activity can be canceled only if it has not completed. In the case of the [CancellationScope](#) activity, completion refers to the completion of the activities in the [Body](#). If a cancellation request is scheduled and the activities in the [Body](#) have not completed, then the [CancellationScope](#) will be marked as [Canceled](#) and the [CancellationHandler](#) activities will be executed.

### Cancelling a Workflow from the Host

A host can cancel a workflow by calling the [Cancel](#) method of the [WorkflowApplication](#) instance that is hosting the workflow. In the following example a workflow is created that has a [CancellationScope](#). The workflow is invoked, and then the host makes a call to [Cancel](#). The main execution of the workflow is stopped, the [CancellationHandler](#) of the [CancellationScope](#) is invoked, and then the workflow completes with a status of [Canceled](#).

```

Activity wf = new CancellationScope
{
    Body = new Sequence
    {
        Activities =
        {
            new WriteLine
            {
                Text = "Starting the workflow."
            },
            new Delay
            {
                Duration = TimeSpan.FromSeconds(5)
            },
            new WriteLine
            {
                Text = "Ending the workflow."
            }
        }
    },
    CancellationHandler = new WriteLine
    {
        Text = "CancellationHandler invoked."
    }
};

// Create a WorkflowApplication instance.
WorkflowApplication wfApp = new WorkflowApplication(wf);

// Subscribe to any desired workflow lifecycle events.
wfApp.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
{
    if (e.CompletionState == ActivityInstanceState.Faulted)
    {
        Console.WriteLine("Workflow {0} Terminated.", e.InstanceId);
        Console.WriteLine("Exception: {0}\n{1}",
            e.TerminationException.GetType().FullName,
            e.TerminationException.Message);
    }
    else if (e.CompletionState == ActivityInstanceState.Canceled)
    {
        Console.WriteLine("Workflow {0} Canceled.", e.InstanceId);
    }
    else
    {
        Console.WriteLine("Workflow {0} Completed.", e.InstanceId);
    }
};

// Run the workflow.
wfApp.Run();

Thread.Sleep(TimeSpan.FromSeconds(1));

wfApp.Cancel();

```

When this workflow is invoked, the following output is displayed to the console.

**Starting the workflow.**

**CancellationHandler invoked.**

**Workflow b30ebb30-df46-4d90-a211-e31c38d8db3c Canceled.**

**NOTE**

When a [CancellationScope](#) activity is canceled and the [CancellationHandler](#) invoked, it is the responsibility of the workflow author to determine the progress that the canceled activity made before it was canceled in order to provide the appropriate cancellation logic. The [CancellationHandler](#) does not provide any information about the progress of the canceled activity.

A workflow can also be canceled from the host if an unhandled exception bubbles up past the root of the workflow and the [OnUnhandledException](#) handler returns [Cancel](#). In this example the workflow starts and then throws an [ApplicationException](#). This exception is unhandled by the workflow and so the [OnUnhandledException](#) handler is invoked. The handler instructs the runtime to cancel the workflow, and the [CancellationHandler](#) of the currently executing [CancellationScope](#) activity is invoked.

```

Activity wf = new CancellationScope
{
    Body = new Sequence
    {
        Activities =
        {
            new WriteLine
            {
                Text = "Starting the workflow."
            },
            new Throw
            {
                Exception = new InArgument<Exception>((env) =>
                    new ApplicationException("An ApplicationException was thrown."))
            },
            new WriteLine
            {
                Text = "Ending the workflow."
            }
        }
    },
    CancellationHandler = new WriteLine
    {
        Text = "CancellationHandler invoked."
    }
};

// Create a WorkflowApplication instance.
WorkflowApplication wfApp = new WorkflowApplication(wf);

// Subscribe to any desired workflow lifecycle events.
wfApp.OnUnhandledException = delegate(WorkflowApplicationUnhandledEventArgs e)
{
    // Display the unhandled exception.
    Console.WriteLine("OnUnhandledException in Workflow {0}\n{1}",
        e.InstanceId, e.UnhandledException.Message);

    // Instruct the runtime to cancel the workflow.
    return UnhandledExceptionAction.Cancel;
};

wfApp.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
{
    if (e.CompletionState == ActivityInstanceState.Faulted)
    {
        Console.WriteLine("Workflow {0} Terminated.", e.InstanceId);
        Console.WriteLine("Exception: {0}\n{1}",
            e.TerminationException.GetType().FullName,
            e.TerminationException.Message);
    }
    else if (e.CompletionState == ActivityInstanceState.Canceled)
    {
        Console.WriteLine("Workflow {0} Canceled.", e.InstanceId);
    }
    else
    {
        Console.WriteLine("Workflow {0} Completed.", e.InstanceId);
    }
};

// Run the workflow.
wfApp.Run();

```

When this workflow is invoked, the following output is displayed to the console.

**Starting the workflow.**

**OnUnhandledException in Workflow 6bb2d5d6-f49a-4c6d-a988-478afb86dbe9**

**An ApplicationException was thrown.**

**CancellationHandler invoked.**

**Workflow 6bb2d5d6-f49a-4c6d-a988-478afb86dbe9 Canceled.**

**Canceling an Activity from Inside a Workflow**

An activity can also be canceled by its parent. For example, if a [Parallel](#) activity has multiple executing branches and its [CompletionCondition](#) evaluates to `true` then its incomplete branches will be canceled. In this example a [Parallel](#) activity is created that has two branches. Its [CompletionCondition](#) is set to `true` so the [Parallel](#) completes as soon as any one of its branches is completed. In this example branch 2 completes, and so branch 1 is canceled.

```

Activity wf = new Parallel
{
    CompletionCondition = true,
    Branches =
    {
        new CancellationScope
        {
            Body = new Sequence
            {
                Activities =
                {
                    new WriteLine
                    {
                        Text = "Branch 1 starting."
                    },
                    new Delay
                    {
                        Duration = TimeSpan.FromSeconds(2)
                    },
                    new WriteLine
                    {
                        Text = "Branch 1 complete."
                    }
                }
            },
            CancellationHandler = new WriteLine
            {
                Text = "Branch 1 canceled."
            }
        },
        new WriteLine
        {
            Text = "Branch 2 complete."
        }
    }
};

// Create a WorkflowApplication instance.
WorkflowApplication wfApp = new WorkflowApplication(wf);

wfApp.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
{
    if (e.CompletionState == ActivityInstanceState.Faulted)
    {
        Console.WriteLine("Workflow {0} Terminated.", e.InstanceId);
        Console.WriteLine("Exception: {0}\n{1}",
            e.TerminationException.GetType().FullName,
            e.TerminationException.Message);
    }
    else if (e.CompletionState == ActivityInstanceState.Canceled)
    {
        Console.WriteLine("Workflow {0} Canceled.", e.InstanceId);
    }
    else
    {
        Console.WriteLine("Workflow {0} Completed.", e.InstanceId);
    }
};

// Run the workflow.
wfApp.Run();

```

When this workflow is invoked, the following output is displayed to the console.

**Branch 1 starting.**

**Branch 2 complete.**

**Branch 1 canceled.**

**Workflow e0685e24-18ef-4a47-acf3-5c638732f3be Completed.** Activities are also canceled if an exception bubbles up past the root of the activity but is handled at a higher level in the workflow. In this example, the main logic of the workflow consists of a [Sequence](#) activity. The [Sequence](#) is specified as the [Body](#) of a [CancellationScope](#) activity which is contained by a [TryCatch](#) activity. An exception is thrown from the body of the [Sequence](#), is handled by the parent [TryCatch](#) activity, and the [Sequence](#) is canceled.

```
Activity wf = new TryCatch
{
    Try = new CancellationScope
    {
        Body = new Sequence
        {
            Activities =
            {
                new WriteLine
                {
                    Text = "Sequence starting."
                },
                new Throw
                {
                    Exception = new InArgument<Exception>((env) =>
                        new ApplicationException("An ApplicationException was thrown."))
                },
                new WriteLine
                {
                    Text = "Sequence complete."
                }
            }
        },
        CancellationHandler = new WriteLine
        {
            Text = "Sequence canceled."
        }
    },
    Catches =
    {
        new Catch<ApplicationException>
        {
            Action = new ActivityAction<ApplicationException>
            {
                Handler = new WriteLine
                {
                    Text = "Exception caught."
                }
            }
        }
    }
};

// Create a WorkflowApplication instance.
WorkflowApplication wfApp = new WorkflowApplication(wf);

wfApp.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
{
    if (e.CompletionState == ActivityInstanceState.Faulted)
    {
        Console.WriteLine("Workflow {0} Terminated.", e.InstanceId);
        Console.WriteLine("Exception: {0}\n{1}",
            e.TerminationException.GetType().FullName,
            e.TerminationException.Message);
    }
}
```

```

        }
        else if (e.CompletionState == ActivityInstanceState.Canceled)
        {
            Console.WriteLine("Workflow {0} Canceled.", e.InstanceId);
        }
        else
        {
            Console.WriteLine("Workflow {0} Completed.", e.InstanceId);
        }
    };

    // Run the workflow.
    wfApp.Run();
}

```

When this workflow is invoked, the following output is displayed to the console.

**Sequence starting.**

**Sequence canceled.**

**Exception caught.**

**Workflow e3c18939-121e-4c43-af1c-ba1ce977ce55 Completed.**

### Throwing Exceptions from a CancellationHandler

Any exceptions thrown from the [CancellationHandler](#) of a [CancellationScope](#) are fatal to the workflow. If there is a possibility of exceptions escaping from a [CancellationHandler](#), use a [TryCatch](#) in the [CancellationHandler](#) to catch and handle these exceptions.

### Cancellation using CompensableActivity

Like the [CancellationScope](#) activity, the [CompensableActivity](#) has a [CancellationHandler](#). If a [CompensableActivity](#) is canceled, any activities in its [CancellationHandler](#) are invoked. This can be useful for undoing partially completed compensable work. For information about how to use [CompensableActivity](#) for compensation and cancellation, see [Compensation](#).

## Cancellation using Custom Activities

Custom activity authors can implement cancellation logic into their custom activities in several different ways. Custom activities that derive from [Activity](#) can implement cancellation logic by placing a [CancellationScope](#) or other custom activity that contains cancellation logic in the body of the activity. [AsyncCodeActivity](#) and [NativeActivity](#) derived activities can override their respective [Cancel](#) method and provide cancellation logic there. [CodeActivity](#) derived activities do not provide any provision for cancellation because all their work is performed in a single burst of execution when the runtime calls the [Execute](#) method. If the execute method has not yet been called and a [CodeActivity](#) based activity is canceled, the activity closes with a status of [Canceled](#) and the [Execute](#) method is not called.

### Cancellation using NativeActivity

[NativeActivity](#) derived activities can override the [Cancel](#) method to provide custom cancellation logic. If this method is not overridden, then the default workflow cancellation logic is applied. Default cancellation is the process that occurs for a [NativeActivity](#) that does not override the [Cancel](#) method or whose [Cancel](#) method calls the base [NativeActivity Cancel](#) method. When an activity is canceled, the runtime flags the activity for cancellation and automatically handles certain cleanup. If the activity only has outstanding bookmarks, the bookmarks will be removed and the activity will be marked as [Canceled](#). Any outstanding child activities of the canceled activity will in turn be canceled. Any attempt to schedule additional child activities will result in the attempt being ignored and the activity will be marked as [Canceled](#). If any outstanding child activity completes in the [Canceled](#) or [Faulted](#) state, then the activity will be marked as [Canceled](#). Note that a cancellation request can be ignored. If an activity does not have any outstanding bookmarks or executing child activities and does not schedule any additional work items after being flagged for cancellation, it will complete successfully. This default cancellation suffices for many scenarios, but if additional cancellation logic is needed, then the built-in cancellation activities or custom activities can be used.

In the following example, the `Cancel` override of a [NativeActivity](#) based custom `ParallelForEach` activity is defined. When the activity is canceled, this override handles the cancellation logic for the activity. This example is part of the [Non-Generic ParallelForEach](#) sample.

```
protected override void Cancel(NativeActivityContext context)
{
    // If we do not have a completion condition then we can just
    // use default logic.
    if (this.CompletionCondition == null)
    {
        base.Cancel(context);
    }
    else
    {
        context.CancelChildren();
    }
}
```

[NativeActivity](#) derived activities can determine if cancellation has been requested by inspecting the `IsCancellationRequested` property, and mark themselves as canceled by calling the `MarkCanceled` method. Calling `MarkCanceled` does not immediately complete the activity. As usual, the runtime will complete the activity when it has no more outstanding work, but if `MarkCanceled` is called the final state will be `Canceled` instead of `Closed`.

### Cancellation using AsyncCodeActivity

[AsyncCodeActivity](#) based activities can also provide custom cancellation logic by overriding the `Cancel` method. If this method is not overridden, then no cancellation handling is performed if the activity is canceled. In the following example, the `Cancel` override of an [AsyncCodeActivity](#) based custom `ExecutePowerShell` activity is defined. When the activity is canceled, it performs the desired cancellation behavior.

```
// Called by the runtime to cancel the execution of this asynchronous activity.
protected override void Cancel(AsyncCodeActivityContext context)
{
    Pipeline pipeline = context.UserState as Pipeline;
    if (pipeline != null)
    {
        pipeline.Stop();
        DisposePipeline(pipeline);
    }
    base.Cancel(context);
}
```

[AsyncCodeActivity](#) derived activities can determine if cancellation has been requested by inspecting the `IsCancellationRequested` property, and mark themselves as canceled by calling the `MarkCanceled` method.

# Debugging Workflows

8/31/2018 • 2 minutes to read • [Edit Online](#)

.NET Framework 4.6.1 offers several options for debugging running workflows from the development environment. Workflows can be debugged in the designer, in XAML, and in code.

## Debugging in the Workflow Designer

Breakpoints can be set on activities in the workflow designer by either highlighting the activity and pressing **F9** or by using the activity's context menu. Execution of the workflow then breaks when the workflow host is run in debug mode. In the following screenshot, workflow execution is paused at a breakpoint. For more information, see [Debugging Workflows with the Workflow Designer](#).

## Debugging in XAML

If a workflow has paused at a breakpoint in the designer, the workflow can also be debugged in XAML. To view the point of execution in XAML, select **XAML View** in the workflow designer when workflow execution is paused. Debugging can be switched back to the designer by re-opening the workflow in the designer from the solution explorer. For more information, see [How to: Debug XAML with the Workflow Designer](#).

## Debugging in Code

Code breakpoints can be used in .NET Framework 4.6.1 in the same way that they can be used in other imperative applications. Click the left margin of the code pane to create a code breakpoint, or press **F9** to place a breakpoint at the cursor location.

## Attaching to a Workflow Process

Workflow debugging also supports using Visual Studio's infrastructure to attach to a process. This enables the workflow author to debug a workflow running in a different host environment such as Internet Information Services (IIS) 7.0.

## Remote Debugging

Windows Workflow Foundation (WF) remote debugging functions the same as remote debugging for other Visual Studio components. For information on using remote debugging, see [How to: Enable Remote Debugging](#).

### NOTE

If the workflow application targets the x86 architecture and is hosted on a computer running a 64 bit operating system, then remote debugging will not work unless Visual Studio is installed on the remote computer or the target for the workflow application is changed to **Any CPU**.

## Extending the Workflow Debugging Service

The workflow debugger service is now public and can be used to create custom applications such as monitoring, simulation, and debugging in a re-hosted designer. For more information, see the [DebuggerService](#) topic.

# Hosting Workflows

3/9/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section discuss hosting workflows.

## In This Section

### [Workflow Hosting Options](#)

Describes how to select the appropriate host for your workflows.

### [Using WorkflowInvoker and WorkflowApplication](#)

Describes how to use [WorkflowInvoker](#) and [WorkflowApplication](#) to invoke workflows.

### [Activity Tree Inspection](#)

Describes how workflow host authors can inspect a workflow using [WorkflowInspectionServices](#).

### [Serializing Workflows and Activities to and from XAML](#)

Describes how to serialize workflows to XAML and work with serialized workflow definitions.

### [Using WorkflowIdentity and Versioning](#)

Describes how to use [WorkflowIdentity](#) to host multiple versions of a workflow side-by-side.

## See also

- [Windows Workflow Foundation Programming](#)
- [Designing Workflows](#)
- [Windows Workflow Foundation Data Model](#)

# Workflow Hosting Options

3/9/2019 • 2 minutes to read • [Edit Online](#)

Most of the Windows Workflow Foundation (WF) samples use workflows that are hosted in a console application, but this isn't a realistic scenario for real-world workflows. Workflows in actual business applications will be hosted in persistent processes- either a Windows service authored by the developer, or a server application such as IIS 7.0 or AppFabric. The differences between these approaches are as follows.

## Hosting workflows in IIS with Windows AppFabric

Using IIS with AppFabric is the preferred host for workflows. The host application for workflows using AppFabric is Windows Activation Service, which removes the dependency on HTTP over IIS alone.

## Hosting workflows in IIS alone

Using IIS 7.0 alone is not recommended, as there are management and monitoring tools available with AppFabric that facilitate maintenance of running applications. Workflows should only be hosted in IIS 7.0 alone if there are infrastructure concerns with moving to AppFabric.

### WARNING

IIS 7.0 recycles application pools periodically for various reasons. When an application pool is recycled, IIS stops accepting messages to the old pool, and instantiates a new application pool to accept new requests. If a workflow continues working after sending a response, IIS 7.0 will not be aware of the work being done, and may recycle the hosting application pool. If this happens, the workflow will abort, and tracking services will record a [1004 - WorkflowInstanceAborted](#) message with an empty Reason field.

If persistence is used, the host must explicitly restart aborted instances from the last persistence point.

If AppFabric is used, the workflow management service will eventually resume the workflow from the last successful persistence point if persistence is used. If no persistence is used, and the workflow performs operations outside a Request/Response pattern, data will be lost when the workflow aborts.

## Hosting a workflow in a custom Windows Service

Creating a custom workflow service to host the workflow will require the developer to duplicate a lot of the functionality provided out-of-box by AppFabric, but will allow for more flexibility with custom functionality. This option should only be considered if AppFabric proves to not be an option.

# Using WorkflowInvoker and WorkflowApplication

3/9/2019 • 12 minutes to read • [Edit Online](#)

Windows Workflow Foundation (WF) provides several methods of hosting workflows. [WorkflowInvoker](#) provides a simple way for invoking a workflow as if it were a method call and can be used only for workflows that do not use persistence. [WorkflowApplication](#) provides a richer model for executing workflows that includes notification of lifecycle events, execution control, bookmark resumption, and persistence. [WorkflowServiceHost](#) provides support for messaging activities and is primarily used with workflow services. This topic introduces you to workflow hosting with [WorkflowInvoker](#) and [WorkflowApplication](#). For more information about hosting workflows with [WorkflowServiceHost](#), see [Workflow Services](#) and [Hosting Workflow Services Overview](#).

## Using WorkflowInvoker

[WorkflowInvoker](#) provides a model for executing a workflow as if it were a method call. To invoke a workflow using [WorkflowInvoker](#), call the [Invoke](#) method and pass in the workflow definition of the workflow to invoke. In this example, a [WriteLine](#) activity is invoked using [WorkflowInvoker](#).

```
Activity wf = new WriteLine
{
    Text = "Hello World."
};

WorkflowInvoker.Invoke(wf);
```

When a workflow is invoked using [WorkflowInvoker](#), the workflow executes on the calling thread and the [Invoke](#) method blocks until the workflow is complete, including any idle time. To configure a time-out interval in which the workflow must complete, use one of the [Invoke](#) overloads that takes a [TimeSpan](#) parameter. In this example, a workflow is invoked twice with two different time-out intervals. The first workflow completes, but the second does not.

```

Activity wf = new Sequence()
{
    Activities =
    {
        new WriteLine()
        {
            Text = "Before the 1 minute delay."
        },
        new Delay()
        {
            Duration = TimeSpan.FromMinutes(1)
        },
        new WriteLine()
        {
            Text = "After the 1 minute delay."
        }
    }
};

// This workflow completes successfully.
WorkflowInvoker.Invoke(wf, TimeSpan.FromMinutes(2));

// This workflow does not complete and a TimeoutException
// is thrown.
try
{
    WorkflowInvoker.Invoke(wf, TimeSpan.FromSeconds(30));
}
catch (TimeoutException ex)
{
    Console.WriteLine(ex.Message);
}

```

#### **NOTE**

The [TimeoutException](#) is only thrown if the time-out interval elapses and the workflow becomes idle during execution. A workflow that takes longer than the specified time-out interval to complete completes successfully if the workflow does not become idle.

[WorkflowInvoker](#) also provides asynchronous versions of the `invoke` method. For more information, see [InvokeAsync](#) and [BeginInvoke](#).

#### **Setting Input Arguments of a Workflow**

Data can be passed into a workflow using a dictionary of input parameters, keyed by argument name, that map to the input arguments of the workflow. In this example, a [WriteLine](#) is invoked and the value for its [Text](#) argument is specified using the dictionary of input parameters.

```

Activity wf = new WriteLine();

Dictionary<string, object> inputs = new Dictionary<string, object>();
inputs.Add("Text", "Hello World.");

WorkflowInvoker.Invoke(wf, inputs);

```

#### **Retrieving Output Arguments of a Workflow**

The output parameters of a workflow can be obtained using the outputs dictionary that is returned from the call to [Invoke](#). The following example invokes a workflow consisting of a single [Divide](#) activity that has two input arguments and two output arguments. When the workflow is invoked, the [arguments](#) dictionary is passed which contains the values for each input argument, keyed by argument name. When the call to [Invoke](#) returns, each

output argument is returned in the `outputs` dictionary, also keyed by argument name.

```
public sealed class Divide : CodeActivity
{
    [RequiredArgument]
    public InArgument<int> Dividend { get; set; }

    [RequiredArgument]
    public InArgument<int> Divisor { get; set; }

    public OutArgument<int> Remainder { get; set; }
    public OutArgument<int> Result { get; set; }

    protected override void Execute(CodeActivityContext context)
    {
        int quotient = Dividend.Get(context) / Divisor.Get(context);
        int remainder = Dividend.Get(context) % Divisor.Get(context);

        Result.Set(context, quotient);
        Remainder.Set(context, remainder);
    }
}
```

```
int dividend = 500;
int divisor = 36;

Dictionary<string, object> arguments = new Dictionary<string, object>();
arguments.Add("Dividend", dividend);
arguments.Add("Divisor", divisor);

IDictionary<string, object> outputs =
    WorkflowInvoker.Invoke(new Divide(), arguments);

Console.WriteLine("{0} / {1} = {2} Remainder {3}",
    dividend, divisor, outputs["Result"], outputs["Remainder"]);
```

If the workflow derives from [ActivityWithResult](#), such as `CodeActivity<TResult>` or `Activity<TResult>`, and there are output arguments in addition to the well-defined `Result` output argument, a non-generic overload of `Invoke` must be used in order to retrieve the additional arguments. To do this, the workflow definition passed into `Invoke` must be of type [Activity](#). In this example the `Divide` activity derives from `CodeActivity<int>`, but is declared as [Activity](#) so that a non-generic overload of `Invoke` is used which returns a dictionary of arguments instead of a single return value.

```
public sealed class Divide : CodeActivity<int>
{
    public InArgument<int> Dividend { get; set; }
    public InArgument<int> Divisor { get; set; }
    public OutArgument<int> Remainder { get; set; }

    protected override int Execute(CodeActivityContext context)
    {
        int quotient = Dividend.Get(context) / Divisor.Get(context);
        int remainder = Dividend.Get(context) % Divisor.Get(context);

        Remainder.Set(context, remainder);

        return quotient;
    }
}
```

```

int dividend = 500;
int divisor = 36;

Dictionary<string, object> arguments = new Dictionary<string, object>();
arguments.Add("Dividend", dividend);
arguments.Add("Divisor", divisor);

Activity wf = new Divide();

IDictionary<string, object> outputs =
    WorkflowInvoker.Invoke(wf, arguments);

Console.WriteLine("{0} / {1} = {2} Remainder {3}",
    dividend, divisor, outputs["Result"], outputs["Remainder"]);

```

## Using WorkflowApplication

[WorkflowApplication](#) provides a rich set of features for workflow instance management. [WorkflowApplication](#) acts as a thread safe proxy to the actual [WorkflowInstance](#), which encapsulates the runtime, and provides methods for creating and loading workflow instances, pausing and resuming, terminating, and notification of lifecycle events. To run a workflow using [WorkflowApplication](#) you create the [WorkflowApplication](#), subscribe to any desired lifecycle events, start the workflow, and then wait for it to finish. In this example, a workflow definition that consists of a [WriteLine](#) activity is created and a [WorkflowApplication](#) is created using the specified workflow definition. [Completed](#) is handled so the host is notified when the workflow completes, the workflow is started with a call to [Run](#), and then the host waits for the workflow to complete. When the workflow completes, the [AutoResetEvent](#) is set and the host application can resume execution, as shown in the following example.

```

AutoResetEvent syncEvent = new AutoResetEvent(false);

Activity wf = new WriteLine
{
    Text = "Hello World."
};

// Create the WorkflowApplication using the desired
// workflow definition.
WorkflowApplication wfApp = new WorkflowApplication(wf);

// Handle the desired lifecycle events.
wfApp.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
{
    syncEvent.Set();
};

// Start the workflow.
wfApp.Run();

// Wait for Completed to arrive and signal that
// the workflow is complete.
syncEvent.WaitOne();

```

### WorkflowApplication Lifecycle Events

In addition to [Completed](#), host authors can be notified when a workflow is unloaded ([Unloaded](#)), aborted ([Aborted](#)), becomes idle ([Idle](#) and [PersistableIdle](#)), or an unhandled exception occurs ([OnUnhandledException](#)). Workflow application developers can handle these notifications and take appropriate action, as shown in the following example.

```

wfApp.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
{

```

```

        if (e.CompletionState == ActivityInstanceState.Faulted)
    {
        Console.WriteLine("Workflow {0} Terminated.", e.InstanceId);
        Console.WriteLine("Exception: {0}\n{1}",
            e.TerminationException.GetType().FullName,
            e.TerminationException.Message);
    }
    else if (e.CompletionState == ActivityInstanceState.Canceled)
    {
        Console.WriteLine("Workflow {0} Canceled.", e.InstanceId);
    }
    else
    {
        Console.WriteLine("Workflow {0} Completed.", e.InstanceId);

        // Outputs can be retrieved from the Outputs dictionary,
        // keyed by argument name.
        // Console.WriteLine("The winner is {0}.", e.Outputs["Winner"]);
    }
};

wfApp.Aborted = delegate(WorkflowApplicationAbortedEventArgs e)
{
    // Display the exception that caused the workflow
    // to abort.
    Console.WriteLine("Workflow {0} Aborted.", e.InstanceId);
    Console.WriteLine("Exception: {0}\n{1}",
        e.Reason.GetType().FullName,
        e.Reason.Message);
};

wfApp.Idle = delegate(WorkflowApplicationIdleEventArgs e)
{
    // Perform any processing that should occur
    // when a workflow goes idle. If the workflow can persist,
    // both Idle and PersistableIdle are called in that order.
    Console.WriteLine("Workflow {0} Idle.", e.InstanceId);
};

wfApp.PersistableIdle = delegate(WorkflowApplicationIdleEventArgs e)
{
    // Instruct the runtime to persist and unload the workflow.
    // Choices are None, Persist, and Unload.
    return PersistableIdleAction.Unload;
};

wfApp.Unloaded = delegate(WorkflowApplicationEventArgs e)
{
    Console.WriteLine("Workflow {0} Unloaded.", e.InstanceId);
};

wfApp.OnUnhandledException = delegate(WorkflowApplicationUnhandledExceptionEventArgs e)
{
    // Display the unhandled exception.
    Console.WriteLine("OnUnhandledException in Workflow {0}\n{1}",
        e.InstanceId, e.UnhandledException.Message);

    Console.WriteLine("ExceptionSource: {0} - {1}",
        e.ExceptionSource.DisplayName, e.ExceptionSourceInstanceId);

    // Instruct the runtime to terminate the workflow.
    // Other choices are Abort and Cancel. Terminate
    // is the default if no OnUnhandledException handler
    // is present.
    return UnhandledExceptionAction.Terminate;
};

```

## Setting Input Arguments of a Workflow

Data can be passed into a workflow as it is started using a dictionary of parameters, similar to the way data is passed in when using [WorkflowInvoker](#). Each item in the dictionary maps to an input argument of the specified workflow. In this example, a workflow that consists of a [WriteLine](#) activity is invoked and its [Text](#) argument is specified using the dictionary of input parameters.

```
AutoResetEvent syncEvent = new AutoResetEvent(false);

Activity wf = new WriteLine();

// Create the dictionary of input parameters.
Dictionary<string, object> inputs = new Dictionary<string, object>();
inputs.Add("Text", "Hello World!");

// Create the WorkflowApplication using the desired
// workflow definition and dictionary of input parameters.
WorkflowApplication wfApp = new WorkflowApplication(wf, inputs);

// Handle the desired lifecycle events.
wfApp.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
{
    syncEvent.Set();
};

// Start the workflow.
wfApp.Run();

// Wait for Completed to arrive and signal that
// the workflow is complete.
syncEvent.WaitOne();
```

## Retrieving Output Arguments of a Workflow

When a workflow completes, any output arguments can be retrieved in the [Completed](#) handler by accessing the [WorkflowApplicationCompletedEventArgs.Outputs](#) dictionary. The following example hosts a workflow using [WorkflowApplication](#). A [WorkflowApplication](#) instance is constructed using a workflow definition consisting of a single `DiceRoll` activity. The `DiceRoll` activity has two output arguments that represent the results of the dice roll operation. When the workflow completes, the outputs are retrieved in the [Completed](#) handler.

```
public sealed class DiceRoll : CodeActivity
{
    public OutArgument<int> D1 { get; set; }
    public OutArgument<int> D2 { get; set; }

    static Random r = new Random();

    protected override void Execute(CodeActivityContext context)
    {
        D1.Set(context, r.Next(1, 7));
        D2.Set(context, r.Next(1, 7));
    }
}
```

```

// Create a WorkflowApplication instance.
WorkflowApplication wfApp = new WorkflowApplication(new DiceRoll());

// Subscribe to any desired workflow lifecycle events.
wfApp.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
{
    if (e.CompletionState == ActivityInstanceState.Faulted)
    {
        Console.WriteLine("Workflow {0} Terminated.", e.InstanceId);
        Console.WriteLine("Exception: {0}\n{1}",
            e.TerminationException.GetType().FullName,
            e.TerminationException.Message);
    }
    else if (e.CompletionState == ActivityInstanceState.Canceled)
    {
        Console.WriteLine("Workflow {0} Canceled.", e.InstanceId);
    }
    else
    {
        Console.WriteLine("Workflow {0} Completed.", e.InstanceId);

        // Outputs can be retrieved from the Outputs dictionary,
        // keyed by argument name.
        Console.WriteLine("The two dice are {0} and {1}.",
            e.Outputs["D1"], e.Outputs["D2"]);
    }
};

// Run the workflow.
wfApp.Run();

```

#### NOTE

[WorkflowApplication](#) and [WorkflowInvoker](#) take a dictionary of input arguments and return a dictionary of `out` arguments. These dictionary parameters, properties, and return values are of type `IDictionary<string, object>`. The actual instance of the dictionary class that is passed in can be any class that implements `IDictionary<string, object>`. In these examples, `Dictionary<string, object>` is used. For more information about dictionaries, see [IDictionary<TKey, TValue>](#) and [Dictionary<TKey, TValue>](#).

## Passing Data into a Running Workflow Using Bookmarks

Bookmarks are the mechanism by which an activity can passively wait to be resumed and are a mechanism for passing data into a running workflow instance. If an activity is waiting for data, it can create a [Bookmark](#) and register a callback method to be called when the [Bookmark](#) is resumed, as shown in the following example.

```
public sealed class ReadLine : NativeActivity<string>
{
    [RequiredArgument]
    public InArgument<string> BookmarkName { get; set; }

    protected override void Execute(NativeActivityContext context)
    {
        // Create a Bookmark and wait for it to be resumed.
        context.CreateBookmark(BookmarkName.Get(context),
            new BookmarkCallback(OnResumeBookmark));
    }

    // NativeActivity derived activities that do asynchronous operations by calling
    // one of the CreateBookmark overloads defined on System.Activities.NativeActivityContext
    // must override the CanInduceIdle property and return true.
    protected override bool CanInduceIdle
    {
        get { return true; }
    }

    public void OnResumeBookmark(NativeActivityContext context, Bookmark bookmark, object obj)
    {
        // When the Bookmark is resumed, assign its value to
        // the Result argument.
        Result.Set(context, (string)obj);
    }
}
```

When executed, the `ReadLine` activity creates a `Bookmark`, registers a callback, and then waits for the `Bookmark` to be resumed. When it is resumed, the `ReadLine` activity assigns the data that was passed with the `Bookmark` to its `Result` argument. In this example, a workflow is created that uses the `ReadLine` activity to gather the user's name and display it to the console window.

```

Variable<string> name = new Variable<string>();

Activity wf = new Sequence
{
    Variables = { name },
    Activities =
    {
        new WriteLine
        {
            Text = "What is your name?"
        },
        new ReadLine
        {
            BookmarkName = "UserName",
            Result = new OutArgument<string>(name)
        },
        new WriteLine
        {
            Text = new InArgument<string>((env) =>
                ("Hello, " + name.Get(env)))
        }
    }
};

// Create a WorkflowApplication instance.
WorkflowApplication wfApp = new WorkflowApplication(wf);

// Workflow lifecycle events omitted except idle.
AutoResetEvent idleEvent = new AutoResetEvent(false);

wfApp.Idle = delegate(WorkflowApplicationIdleEventArgs e)
{
    idleEvent.Set();
};

// Run the workflow.
wfApp.Run();

// Wait for the workflow to go idle before gathering
// the user's input.
idleEvent.WaitOne();

// Gather the user's input and resume the bookmark.
// Bookmark resumption only occurs when the workflow
// is idle. If a call to ResumeBookmark is made and the workflow
// is not idle, ResumeBookmark blocks until the workflow becomes
// idle before resuming the bookmark.
BookmarkResumptionResult result = wfApp.ResumeBookmark("UserName",
    Console.ReadLine());

// Possible BookmarkResumptionResult values:
// Success, NotFound, or NotReady
Console.WriteLine("BookmarkResumptionResult: {0}", result);

```

When the `ReadLine` activity is executed, it creates a `Bookmark` named `UserName` and then waits for the bookmark to be resumed. The host collects the desired data and then resumes the `Bookmark`. The workflow resumes, displays the name, and then completes.

The host application can inspect the workflow to determine if there are any active bookmarks. It can do this by calling the `GetBookmarks` method of a `WorkflowApplication` instance, or by inspecting the `WorkflowApplicationIdleEventArgs` in the `Idle` handler.

The following code example is like the previous example except that the active bookmarks are enumerated before the bookmark is resumed. The workflow is started, and once the `Bookmark` is created and the workflow goes idle,

`GetBookmarks` is called. When the workflow is completed, the following output is displayed to the console.

**What is your name?**

**BookmarkName: UserName - OwnerDisplayName: ReadLine**

**Steve**

**Hello, Steve**

```
Variable<string> name = new Variable<string>();

Activity wf = new Sequence
{
    Variables = { name },
    Activities =
    {
        new WriteLine
        {
            Text = "What is your name?"
        },
        new ReadLine
        {
            BookmarkName = "UserName",
            Result = new OutArgument<string>(name)
        },
        new WriteLine
        {
            Text = new InArgument<string>((env) =>
                ("Hello, " + name.Get(env)))
        }
    }
};

// Create a WorkflowApplication instance.
WorkflowApplication wfApp = new WorkflowApplication(wf);

// Workflow lifecycle events omitted except idle.
AutoResetEvent idleEvent = new AutoResetEvent(false);

wfApp.Idle = delegate(WorkflowApplicationIdleEventArgs e)
{
    // You can also inspect the bookmarks from the Idle handler
    // using e.Bookmarks

    idleEvent.Set();
};

// Run the workflow.
wfApp.Run();

// Wait for the workflow to go idle and give it a chance
// to create the Bookmark.
idleEvent.WaitOne();

// Inspect the bookmarks
foreach (BookmarkInfo info in wfApp.GetBookmarks())
{
    Console.WriteLine("BookmarkName: {0} - OwnerDisplayName: {1}",
        info.BookmarkName, info.OwnerDisplayName);
}

// Gather the user's input and resume the bookmark.
wfApp.ResumeBookmark("UserName", Console.ReadLine());
```

The following code example inspects the `WorkflowApplicationIdleEventArgs` passed into the `Idle` handler of a `WorkflowApplication` instance. In this example the workflow going idle has one `Bookmark` with a name of

`EnterGuess`, owned by an activity named `ReadInt`. This code example is based off of [How to: Run a Workflow](#), which is part of the [Getting Started Tutorial](#). If the `Idle` handler in that step is modified to contain the code from this example, the following output is displayed.

### **BookmarkName: EnterGuess - OwnerDisplayName: ReadInt**

```
wfApp.Idle = delegate(WorkflowApplicationIdleEventArgs e)
{
    foreach (BookmarkInfo info in e.Bookmarks)
    {
        Console.WriteLine("BookmarkName: {0} - OwnerDisplayName: {1}",
            info.BookmarkName, info.OwnerDisplayName);
    }

    idleEvent.Set();
};
```

## Summary

`WorkflowInvoker` provides a lightweight way to invoke workflows, and although it provides methods for passing data in at the start of a workflow and extracting data from a completed workflow, it does not provide for more complex scenarios which is where `WorkflowApplication` can be used.

# Activity Tree Inspection

3/9/2019 • 2 minutes to read • [Edit Online](#)

Activity tree inspection is used by workflow application authors to inspect the workflows hosted by the application. By using [WorkflowInspectionServices](#), workflows can be searched for specific child activities, individual activities and their properties can be enumerated, and runtime metadata of the activities can be cached at a specific time. This topic provides an overview of [WorkflowInspectionServices](#) and how to use it to inspect an activity tree.

## Using WorkflowInspectionServices

The [GetActivities](#) method is used to enumerate all of the activities in the specified activity tree. [GetActivities](#) returns an enumerable that touches all activities within the tree including children, delegate handlers, variable defaults, and argument expressions. In the following example, a workflow definition is created by using a [Sequence](#), [While](#), [ForEach<T>](#), [WriteLine](#), and expressions. After the workflow definition is created, it is invoked and then the [InspectActivity](#) method is called.

```

Variable<List<string>> items = new Variable<List<string>>
{
    Default = new VisualBasicValue<List<string>>("New List(Of String)()")
};

DelegateInArgument<string> item = new DelegateInArgument<string>();

Activity wf = new Sequence
{
    Variables = { items },
    Activities =
    {
        new While((env) => items.Get(env).Count < 5)
        {
            Body = new AddToCollection<string>
            {
                Collection = new InArgument<ICollection<string>>(items),
                Item = new InArgument<string>((env) => "List Item " + (items.Get(env).Count + 1))
            }
        },
        new ForEach<string>
        {
            Values = new InArgument<IEnumerable<string>>(items),
            Body = new ActivityAction<string>
            {
                Argument = item,
                Handler = new WriteLine
                {
                    Text = item
                }
            }
        },
        new Sequence
        {
            Activities =
            {
                new WriteLine
                {
                    Text = "Items added to collection."
                }
            }
        }
    }
};

WorkflowInvoker.Invoke(wf);

InspectActivity(wf, 0);

```

To enumerate the activities, the [GetActivities](#) is called on the root activity, and again recursively on each returned activity. In the following example, the [DisplayName](#) of each activity and expression in the activity tree is written to the console.

```

static void InspectActivity(Activity root, int indent)
{
    // Inspect the activity tree using WorkflowInspectionServices.
    IEnumarator<Activity> activities =
        WorkflowInspectionServices.GetActivities(root).GetEnumarator();

    Console.WriteLine("{0}{1}", new string(' ', indent), root.DisplayName);

    while (activities.MoveNext())
    {
        InspectActivity(activities.Current, indent + 2);
    }
}

```

This sample code provides the following output.

**List Item 1**

**List Item 2**

**List Item 3**

**List Item 4**

**List Item 5**

**Items added to collection.**

**Sequence**

**Literal<List<String>>**

**While**

**AddToCollection<String>**

**VariableValue<ICollection<String>>**

**LambdaValue<String>**

**LocationReferenceValue<List<String>>**

**LambdaValue<Boolean>**

**LocationReferenceValue<List<String>>**

**ForEach<String>**

**VariableValue<IEnumerable<String>>**

**WriteLine**

**DelegateArgumentValue<String>**

**Sequence**

**WriteLine**

**Literal<String>** To retrieve a specific activity instead of enumerating all of the activities, [Resolve](#) is used. Both [Resolve](#) and [GetActivities](#) perform metadata caching if `WorkflowInspectionServices.CacheMetadata` has not been previously called. If `CacheMetadata` has been called then [GetActivities](#) is based on the existing metadata. Therefore, if tree changes have been made since the last call to `CacheMetadata`, [GetActivities](#) might give unexpected results. If changes have been made to the workflow after calling [GetActivities](#), metadata can be re-cached by calling the [ActivityValidationServices Validate](#) method. Caching metadata is discussed in the next section.

## Caching Metadata

Caching the metadata for an activity builds and validates a description of the activity's arguments, variables, child activities, and activity delegates. Metadata, by default, is cached by the runtime when an activity is prepared for execution. If a workflow host author wants to cache the metadata for an activity or activity tree before this, for example to take all of the cost upfront, then [CacheMetadata](#) can be used to cache the metadata at the desired time.

# Serializing Workflows and Activities to and from XAML

3/9/2019 • 4 minutes to read • [Edit Online](#)

In addition to being compiled into types that are contained in assemblies, workflow definitions can also be serialized to XAML. These serialized definitions can be reloaded for editing or inspection, passed to a build system for compilation, or loaded and invoked. This topic provides an overview of serializing workflow definitions and working with XAML workflow definitions.

## Working with XAML Workflow Definitions

To create a workflow definition for serialization, the [ActivityBuilder](#) class is used. Creating an [ActivityBuilder](#) is very similar to creating a [DynamicActivity](#). Any desired arguments are specified, and the activities that constitute the behavior are configured. In the following example, an [Add](#) activity is created that takes two input arguments, adds them together, and returns the result. Because this activity returns a result, the generic [ActivityBuilder<TResult>](#) class is used.

```
ActivityBuilder<int> ab = new ActivityBuilder<int>();
ab.Name = "Add";
ab.Properties.Add(new DynamicActivityProperty { Name = "Operand1", Type = typeof(InArgument<int>) });
ab.Properties.Add(new DynamicActivityProperty { Name = "Operand2", Type = typeof(InArgument<int>) });
ab.Implementation = new Sequence
{
    Activities =
    {
        new WriteLine
        {
            Text = new VisualBasicValue<string>("Operand1.ToString() + \" + " + Operand2.ToString()")
        },
        new Assign<int>
        {
            To = new ArgumentReference<int> { ArgumentName = "Result" },
            Value = new VisualBasicValue<int>("Operand1 + Operand2")
        }
    }
};
```

Each of the [DynamicActivityProperty](#) instances represents one of the input arguments to the workflow, and the [Implementation](#) contains the activities that make up the logic of the workflow. Note that the r-value expressions in this example are Visual Basic expressions. Lambda expressions are not serializable to XAML unless [Convert](#) is used. If the serialized workflows are intended to be opened or edited in the workflow designer then Visual Basic expressions should be used. For more information, see [Authoring Workflows, Activities, and Expressions Using Imperative Code](#).

To serialize the workflow definition represented by the [ActivityBuilder](#) instance to XAML, use [ActivityXamlServices](#) to create a [XamlWriter](#), and then use [XamlServices](#) to serialize the workflow definition by using the [XamlWriter](#). [ActivityXamlServices](#) has methods to map [ActivityBuilder](#) instances to and from XAML, and to load XAML workflows and return a [DynamicActivity](#) that can be invoked. In the following example, the [ActivityBuilder](#) instance from the previous example is serialized to a string, and also saved to a file.

```

// Serialize the workflow to XAML and store it in a string.
StringBuilder sb = new StringBuilder();
StringWriter tw = new StringWriter(sb);
XamlWriter xw = ActivityXamlServices.CreateBuilderWriter(new XamlXmlWriter(tw, new XamlSchemaContext()));
XamlServices.Save(xw, ab);
string serializedAB = sb.ToString();

// Display the XAML to the console.
Console.WriteLine(serializedAB);

// Serialize the workflow to XAML and save it to a file.
StreamWriter sw = File.CreateText(@"C:\Workflows\add.xaml");
XamlWriter xw2 = ActivityXamlServices.CreateBuilderWriter(new XamlXmlWriter(sw, new XamlSchemaContext()));
XamlServices.Save(xw2, ab);
sw.Close();

```

The following example represents the serialized workflow.

```

<Activity
  x:TypeArguments="x:Int32"
  x:Class="Add"
  xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <x:Members>
    <x:Property Name="Operand1" Type="InArgument(x:Int32)" />
    <x:Property Name="Operand2" Type="InArgument(x:Int32)" />
  </x:Members>
  <Sequence>
    <WriteLine Text="[Operand1.ToString() + " + " + Operand2.ToString()]" />
    <Assign x:TypeArguments="x:Int32" Value="[Operand1 + Operand2]">
      <Assign.To>
        <OutArgument x:TypeArguments="x:Int32">
          <ArgumentReference x:TypeArguments="x:Int32" ArgumentName="Result" />
        </OutArgument>
      </Assign.To>
    </Assign>
  </Sequence>
</Activity>

```

To load a serialized workflow, the [ActivityXamlServices Load](#) method is used. This takes the serialized workflow definition and returns a [DynamicActivity](#) that represents the workflow definition. Note that the XAML is not deserialized until [CacheMetadata](#) is called on the body of the [DynamicActivity](#) during the validation process. If validation is not explicitly called then it is performed when the workflow is invoked. If the XAML workflow definition is invalid, then an [ArgumentException](#) exception is thrown. Any exceptions thrown from [CacheMetadata](#) escape from the call to [Validate](#) and must be handled by the caller. In the following example, the serialized workflow from the previous example is loaded and invoked by using [WorkflowInvoker](#).

```

// Load the workflow definition from XAML and invoke it.
DynamicActivity<int> wf = ActivityXamlServices.Load(new StringReader(serializedAB)) as DynamicActivity<int>;
Dictionary<string, object> wfParams = new Dictionary<string, object>
{
  { "Operand1", 25 },
  { "Operand2", 15 }
};

int result = WorkflowInvoker.Invoke(wf, wfParams);
Console.WriteLine(result);

```

When this workflow is invoked, the following output is displayed to the console.

**NOTE**

For more information about invoking workflows with input and output arguments, see [Using WorkflowInvoker](#) and [WorkflowApplication](#) and [Invoke](#).

If the serialized workflow contains C# expressions, then an [ActivityXamlServicesSettings](#) instance with its [CompileExpressions](#) property set to `true` must be passed as a parameter to [ActivityXamlServices.Load](#), otherwise a [NotSupportedException](#) will be thrown with a message similar to the following:

`Expression Activity type 'CSharpValue' 1' requires compilation in order to run. Please ensure that the workflow has been compiled.'`

```
ActivityXamlServicesSettings settings = new ActivityXamlServicesSettings
{
    CompileExpressions = true
};

DynamicActivity<int> wf = ActivityXamlServices.Load(new StringReader(serializedAB), settings) as
DynamicActivity<int>;
```

For more information, see [C# Expressions](#).

A serialized workflow definition can also be loaded into an [ActivityBuilder](#) instance by using the [ActivityXamlServices CreateBuilderReader](#) method. After a serialized workflow is loaded into an [ActivityBuilder](#) instance it can be inspected and modified. This is useful for custom workflow designer authors and provides a mechanism for saving and reloading workflow definitions during the design process. In the following example, the serialized workflow definition from the previous example is loaded and its properties are inspected.

```
// Create a new ActivityBuilder and initialize it using the serialized
// workflow definition.
ActivityBuilder<int> ab2 = XamlServices.Load(
    ActivityXamlServices.CreateBuilderReader(
        new XamlXmlReader(new StringReader(serializedAB)))) as ActivityBuilder<int>;

// Now you can continue working with the ActivityBuilder, inspect
// properties, etc...
Console.WriteLine("There are {0} arguments in the activity builder.", ab2.Properties.Count);
foreach (var prop in ab2.Properties)
{
    Console.WriteLine("Name: {0}, Type: {1}", prop.Name, prop.Type);
}
```

# Using WorkflowIdentity and Versioning

3/9/2019 • 6 minutes to read • [Edit Online](#)

[WorkflowIdentity](#) provides a way for workflow application developers to associate a name and a [Version](#) with a workflow definition, and for this information to be associated with a persisted workflow instance. This identity information can be used by workflow application developers to enable scenarios such as side-by-side execution of multiple versions of a workflow definition, and provides the cornerstone for other functionality such as dynamic update. This topic provides an overview of using [WorkflowIdentity](#) with [WorkflowApplication](#) hosting. For information on side-by-side execution of workflow definitions in a workflow service, see [Side by Side Versioning in WorkflowServiceHost](#). For information on dynamic update, see [Dynamic Update](#).

## In this topic

- [Using WorkflowIdentity](#)
  - [Side-by-side Execution using WorkflowIdentity](#)
- [Upgrading .NET Framework 4 Persistence Databases to Support Workflow Versioning](#)
  - [To upgrade the database schema](#)

## Using WorkflowIdentity

To use [WorkflowIdentity](#), create an instance, configure it, and associate it with a [WorkflowApplication](#) instance. A [WorkflowIdentity](#) instance contains three identifying pieces of information. [Name](#) and [Version](#) contain a name and a [Version](#) and are required, and [Package](#) is optional and can be used to specify an additional string containing information such as assembly name or other desired information. A [WorkflowIdentity](#) is unique if any of its three properties are different from another [WorkflowIdentity](#).

### IMPORTANT

A [WorkflowIdentity](#) should not contain any personally identifiable information (PII). Information about the [WorkflowIdentity](#) used to create an instance is emitted to any configured tracking services at several different points of the activity life-cycle by the runtime. WF Tracking does not have any mechanism to hide PII (sensitive user data). Therefore, a [WorkflowIdentity](#) instance should not contain any PII data as it will be emitted by the runtime in tracking records and may be visible to anyone with access to view the tracking records.

In the following example, a [WorkflowIdentity](#) is created and associated with an instance of a workflow created using a [MortgageWorkflow](#) workflow definition.

```
WorkflowIdentity identityV1 = new WorkflowIdentity
{
    Name = "MortgageWorkflow v1",
    Version = new Version(1, 0, 0, 0)
};

WorkflowApplication wfApp = new WorkflowApplication(new MortgageWorkflow(), identity);

// Configure the WorkflowApplication with persistence and desired workflow event handlers.
ConfigureWorkflowApplication(wfApp);

// Run the workflow.
wfApp.Run();
```

When reloading and resuming a workflow, a [WorkflowIdentity](#) that is configured to match the [WorkflowIdentity](#) of the persisted workflow instance must be used.

```
WorkflowApplication wfApp = new WorkflowApplication(new MortgageWorkflow(), identityV1);

// Configure the WorkflowApplication with persistence and desired workflow event handlers.
ConfigureWorkflowApplication(wfApp);

// Load the workflow.
wfApp.Load(instanceId);

// Resume the workflow...
```

If the [WorkflowIdentity](#) used when reloading the workflow instance does not match the persisted [WorkflowIdentity](#), a [VersionMismatchException](#) is thrown. In the following example a load attempt is made on the [MortgageWorkflow](#) instance that was persisted in the previous example. This load attempt is made using a [WorkflowIdentity](#) configured for a newer version of the mortgage workflow that does not match the persisted instance.

```
WorkflowApplication wfApp = new WorkflowApplication(new MortgageWorkflow_v2(), identityV2);

// Configure the WorkflowApplication with persistence and desired workflow event handlers.
ConfigureWorkflowApplication(wfApp);

// Attempt to load the workflow instance.
wfApp.Load(instanceId);

// Resume the workflow...
```

When the previous code is executed, the following [VersionMismatchException](#) is thrown.

**The WorkflowIdentity ('MortgageWorkflow v1; Version=1.0.0.0') of the loaded instance does not match the WorkflowIdentity ('MortgageWorkflow v2; Version=2.0.0.0') of the provided workflow definition. The instance can be loaded using a different definition, or updated using Dynamic Update.**

#### Side-by-side Execution using WorkflowIdentity

[WorkflowIdentity](#) can be used to facilitate the execution of multiple versions of a workflow side-by-side. One common scenario is changing business requirements on a long-running workflow. Many instances of a workflow could be running when an updated version is deployed. The host application can be configured to use the updated workflow definition when starting new instances, and it is the responsibility of the host application to provide the correct workflow definition when resuming instances. [WorkflowIdentity](#) can be used to identify and supply the matching workflow definition when resuming workflow instances.

To retrieve the [WorkflowIdentity](#) of a persisted workflow instance, the [GetInstance](#) method is used. The [GetInstance](#) method takes the [Id](#) of the persisted workflow instance and the [SqlWorkflowInstanceStore](#) that contains the persisted instance and returns a [WorkflowApplicationInstance](#). A [WorkflowApplicationInstance](#) contains information about a persisted workflow instance, including its associated [WorkflowIdentity](#). This associated [WorkflowIdentity](#) can be used by the host to supply the correct workflow definition when loading and resuming the workflow instance.

#### NOTE

A null [WorkflowIdentity](#) is valid, and can be used by the host to map instances that were persisted with no associated [WorkflowIdentity](#) to the proper workflow definition. This scenario can occur when a workflow application was not initially written with workflow versioning, or when an application is upgraded from .NET Framework 4. For more information, see [Upgrading .NET Framework 4 Persistence Databases to Support Workflow Versioning](#).

In the following example a `Dictionary<WorkflowIdentity, Activity>` is used to associate `WorkflowIdentity` instances with their matching workflow definitions, and a workflow is started using the `MortgageWorkflow` workflow definition, which is associated with the `identityV1 WorkflowIdentity`.

```
WorkflowIdentity identityV1 = new WorkflowIdentity
{
    Name = "MortgageWorkflow v1",
    Version = new Version(1, 0, 0, 0)
};

WorkflowIdentity identityV2 = new WorkflowIdentity
{
    Name = "MortgageWorkflow v2",
    Version = new Version(2, 0, 0, 0)
};

Dictionary<WorkflowIdentity, Activity> WorkflowVersionMap = new Dictionary<WorkflowIdentity, Activity>();
WorkflowVersionMap.Add(identityV1, new MortgageWorkflow());
WorkflowVersionMap.Add(identityV2, new MortgageWorkflow_v2());

WorkflowApplication wfApp = new WorkflowApplication(new MortgageWorkflow(), identityV1);

// Configure the WorkflowApplication with persistence and desired workflow event handlers.
ConfigureWorkflowApplication(wfApp);

// Run the workflow.
wfApp.Run();
```

In the following example, information about the persisted workflow instance from the previous example is retrieved by calling `GetInstance`, and the persisted `WorkflowIdentity` information is used to retrieve the matching workflow definition. This information is used to configure the `WorkflowApplication`, and then the workflow is loaded. Note that because the `Load` overload that takes the `WorkflowApplicationInstance` is used, the `SqlWorkflowInstanceStore` that was configured on the `WorkflowApplicationInstance` is used by the `WorkflowApplication` and therefore its `InstanceStore` property does not need to be configured.

#### NOTE

If the `InstanceStore` property is set, it must be set with the same `SqlWorkflowInstanceStore` instance used by the `WorkflowApplicationInstance` or else an `ArgumentException` will be thrown with the following message:

```
The instance is configured with a different InstanceStore than this WorkflowApplication. .
```

```
// Get the WorkflowApplicationInstance of the desired workflow from the specified
// SqlWorkflowInstanceStore.
WorkflowApplicationInstance instance = WorkflowApplication.GetInstance(instanceId, store);

// Use the persisted WorkflowIdentity to retrieve the correct workflow
// definition from the dictionary.
Activity definition = WorkflowVersionMap[instance.DefinitionIdentity];

WorkflowApplication wfApp = new WorkflowApplication(definition, instance.DefinitionIdentity);

// Configure the WorkflowApplication with persistence and desired workflow event handlers.
ConfigureWorkflowApplication(wfApp);

// Load the persisted workflow instance.
wfApp.Load(instance);

// Resume the workflow...
```

# Upgrading .NET Framework 4 Persistence Databases to Support Workflow Versioning

A `SqlWorkflowInstanceStoreSchemaUpgrade.sql` database script is provided to upgrade persistence databases created using the .NET Framework 4 database scripts. This script updates the databases to support the new versioning capabilities introduced in .NET Framework 4.5. Any persisted workflow instances in the databases are given default versioning values, and can then participate in side-by-side execution and dynamic update.

If a .NET Framework 4.5 workflow application attempts any persistence operations that use the new versioning features on a persistence database which has not been upgraded using the provided script, an `InstancePersistenceCommandException` is thrown with a message similar to the following message.

**The `SqlWorkflowInstanceStore` has a database version of '4.0.0.0'. `InstancePersistenceCommand` '`System.Activities.DurableInstancing.CreateWorkflowOwnerWithIdentityCommand`' cannot be run against this database version. Please upgrade the database to '4.5.0.0'.**

## To upgrade the database schema

1. Open SQL Server Management Studio and connect to the persistence database server, for example `.SQLEXPRESS`.
2. Choose **Open, File** from the **File** menu. Browse to the following folder:  
`C:\Windows\Microsoft.NET\Framework\4.0.30319\sql\en`
3. Select **SqlWorkflowInstanceStoreSchemaUpgrade.sql** and click **Open**.
4. Select the name of the persistence database in the **Available Databases** drop-down.
5. Choose **Execute** from the **Query** menu.

When the query completes, the database schema is upgraded, and if desired, you can view the default workflow identity that was assigned to the persisted workflow instances. Expand your persistence database in the **Databases** node of the **Object Explorer**, and then expand the **Views** node. Right-click **System.Activities.DurableInstancing.Instances** and choose **Select Top 1000 Rows**. Scroll to end of the columns and note that there are six additional columns added to the view: **IdentityName**, **IdentityPackage**, **Build**, **Major**, **Minor**, and **Revision**. Any persisted workflows will have a value of **NULL** for these fields, representing a null workflow identity.

# Dynamic Update

3/9/2019 • 6 minutes to read • [Edit Online](#)

Dynamic update provides a mechanism for workflow application developers to update the workflow definition of a persisted workflow instance. This can be to implement a bug fix, new requirements, or to accommodate unexpected changes. This topic provides an overview of the dynamic update functionality introduced in .NET Framework 4.5.

## Dynamic Update

To apply dynamic updates to a persisted workflow instance, a [DynamicUpdateMap](#) is created that contains instructions for the runtime that describe how to modify the persisted workflow instance to reflect the desired changes. Once the update map is created, it is applied to the desired persisted workflow instances. Once the dynamic update is applied, the workflow instance may be resumed using the new updated workflow definition. There are four steps required to create and apply an update map.

1. [Prepare the workflow definition for dynamic update](#)
2. [Update the workflow definition to reflect the desired changes](#)
3. [Create the update map](#)
4. [Apply the update map to the desired persisted workflow instances](#)

### NOTE

Note that steps 1 through 3, which cover the creation of the update map, may be performed independently of applying the update. A common scenario that the workflow developer will create the update map offline, and then an administrator will apply the update at a later time.

This topic provides an overview of the dynamic update process of adding a new activity to a persisted instance of a compiled Xaml workflow.

### Prepare the workflow definition for dynamic update

The first step in the dynamic update process is to prepare the desired workflow definition for update. This is done by calling the [DynamicUpdateServices.PrepareForUpdate](#) method and passing in the workflow definition to modify. This method validates and then walks the workflow tree to identify all of the objects such as public activities and variables that need to be tagged so they can be compared later with the modified workflow definition. When this is complete, the workflow tree is cloned and attached to the original workflow definition. When the update map is created, the updated version of the workflow definition is compared with the original workflow definition and the update map is generated based on the differences.

To prepare a Xaml workflow for dynamic update it may be loaded into an [ActivityBuilder](#), and then the [ActivityBuilder](#) is passed into [DynamicUpdateServices.PrepareForUpdate](#).

### NOTE

For more information about working with serialized workflows and [ActivityBuilder](#), see [Serializing Workflows and Activities to and from XAML](#).

In the following example, a `MortgageWorkflow` definition (that consists of a `Sequence` with several child activities) is

loaded into an [ActivityBuilder](#), and then prepared for dynamic update. After the method returns, the [ActivityBuilder](#) contains the original workflow definition as well as a copy.

```
// Load the MortgageWorkflow definition from Xaml into
// an ActivityBuilder.
XamlXmlReaderSettings readerSettings = new XamlXmlReaderSettings()
{
    LocalAssembly = Assembly.GetExecutingAssembly()
};

XamlXmlReader xamlReader = new XamlXmlReader(@"C:\WorkflowDefinitions\MortgageWorkflow.xaml",
    readerSettings);

ActivityBuilder ab = XamlServices.Load(
    ActivityXamlServices.CreateBuilderReader(xamlReader)) as ActivityBuilder;

// Prepare the workflow definition for dynamic update.
DynamicUpdateServices.PrepareForUpdate(ab);
```

#### NOTE

To download the sample code that accompanies this topic, see [Dynamic Update sample code](#).

### Update the workflow definition to reflect the desired changes

Once the workflow definition has been prepared for updating, the desired changes can be made. You can add or remove activities, add, move or delete public variables, add or remove arguments, and make changes to the signature of activity delegates. You cannot remove a running activity or change the signature of a running delegate. These changes may be made using code, or in a re-hosted workflow designer. In the following example, a custom `VerifyAppraisal` activity is added to the Sequence that makes up the body of the `MortgageWorkflow` from the previous example.

```
// Make desired changes to the definition. In this example, we are
// inserting a new VerifyAppraisal activity as the 3rd child of the root Sequence.
VerifyAppraisal va = new VerifyAppraisal
{
    Result = new VisualBasicReference<bool>("LoanCriteria")
};

// Get the Sequence that makes up the body of the workflow.
Sequence s = ab.Implementation as Sequence;

// Insert the new activity into the Sequence.
s.Activities.Insert(2, va);
```

### Create the update map

Once the workflow definition that was prepared for update has been modified, the update map can be created. To create a dynamic update map, the [DynamicUpdateServices.CreateUpdateMap](#) method is invoked. This returns a [DynamicUpdateMap](#) that contains the information the runtime needs to modify a persisted workflow instance so that it may be loaded and resumed with the new workflow definition. In the following example, a dynamic map is created for the modified `MortgageWorkflow` definition from the previous example.

```
// Create the update map.
DynamicUpdateMap map = DynamicUpdateServices.CreateUpdateMap(ab);
```

This update map can immediately be used to modify persisted workflow instances, or more typically it can be saved and the updates applied later. One way to save the update map is to serialize it to a file, as shown in the

following example.

```
// Serialize the update map to a file.  
DataContractSerializer serializer = new DataContractSerializer(typeof(DynamicUpdateMap));  
using (FileStream fs = System.IO.File.Open(@"C:\WorkflowDefinitions\MortgageWorkflow.map", FileMode.Create))  
{  
    serializer.WriteObject(fs, map);  
}
```

When [DynamicUpdateServices.CreateUpdateMap](#) returns, the cloned workflow definition and other dynamic update information that was added in the call to [DynamicUpdateServices.PrepareForUpdate](#) is removed, and the modified workflow definition is ready to be saved so that it can be used later when resuming updated workflow instances. In the following example, the modified workflow definition is saved to [MortgageWorkflow\\_v2.xaml](#).

```
// Save the modified workflow definition.  
StreamWriter sw = File.CreateText(@"C:\WorkflowDefinitions\MortgageWorkflow_v1.1.xaml");  
XamlWriter xw = ActivityXamlServices.CreateBuilderWriter(new XamlXmlWriter(sw, new XamlSchemaContext()));  
XamlServices.Save(xw, ab);  
sw.Close();
```

### Apply the update map to the desired persisted workflow instances

Applying the update map can be done at any time after creating it. It can be done right away using the [DynamicUpdateMap](#) instance that was returned by [DynamicUpdateServices.CreateUpdateMap](#), or it can be done later using a saved copy of the update map. To update a workflow instance, load it into a [WorkflowApplicationInstance](#) using [WorkflowApplication.GetInstance](#). Next, create a [WorkflowApplication](#) using the updated workflow definition, and the desired [WorkflowIdentity](#). This [WorkflowIdentity](#) may be different than the one that was used to persist the original workflow, and typically is in order to reflect that the persisted instance has been modified. Once the [WorkflowApplication](#) is created, it is loaded using the overload of [WorkflowApplication.Load](#) that takes a [DynamicUpdateMap](#), and then unloaded with a call to [WorkflowApplication.Unload](#). This applies the dynamic update and persists the updated workflow instance.

```

// Load the serialized update map.
DynamicUpdateMap map;
using (FileStream fs = File.Open(@"C:\WorkflowDefinitions\MortgageWorkflow.map", FileMode.Open))
{
   DataContractSerializer serializer = new DataContractSerializer(typeof(DynamicUpdateMap));
    object updateMap = serializer.ReadObject(fs);
    if (updateMap == null)
    {
        throw new ApplicationException("DynamicUpdateMap is null.");
    }

    map = (DynamicUpdateMap)updateMap;
}

// Retrieve a list of workflow instance ids that corresponds to the
// workflow instances to update. This step is the responsibility of
// the application developer.
List<Guid> ids = GetPersistedWorkflowIds();
foreach (Guid id in ids)
{
    // Get a proxy to the persisted workflow instance.
    SqlWorkflowInstanceStore store = new SqlWorkflowInstanceStore(connectionString);
    WorkflowApplicationInstance instance = WorkflowApplication.GetInstance(id, store);

    // If desired, you can inspect the WorkflowIdentity of the instance
    // using the DefinitionIdentity property to determine whether to apply
    // the update.
    Console.WriteLine(instance.DefinitionIdentity);

    // Create a workflow application. You must specify the updated workflow definition, and
    // you may provide an updated WorkflowIdentity if desired to reflect the update.
    WorkflowIdentity identity = new WorkflowIdentity
    {
        Name = "MortgageWorkflow v1.1",
        Version = new Version(1, 1, 0, 0)
    };

    // Load the persisted workflow instance using the updated workflow definition
    // and with an updated WorkflowIdentity. In this example the MortgageWorkflow class
    // contains the updated definition.
    WorkflowApplication wfApp = new WorkflowApplication(new MortgageWorkflow(), identity);

    // Apply the dynamic update on the loaded instance.
    wfApp.Load(instance, map);

    // Unload the updated instance.
    wfApp.Unload();
}

```

## Resuming an Updated Workflow Instance

Once dynamic update has been applied, the workflow instance may be resumed. Note that the new updated definition and [WorkflowIdentity](#) must be used.

### NOTE

For more information about working with [WorkflowApplication](#) and [WorkflowIdentity](#), see [Using WorkflowIdentity and Versioning](#).

In the following example, the `MortgageWorkflow_v1.1.xaml` workflow from the previous example has been compiled, and is loaded and resumed using the updated workflow definition.

```
// Load the persisted workflow instance using the updated workflow definition
// and updated WorkflowIdentity.
WorkflowIdentity identity = new WorkflowIdentity
{
    Name = "MortgageWorkflow v1.1",
    Version = new Version(1, 1, 0, 0)
};

WorkflowApplication wfApp = new WorkflowApplication(new MortgageWorkflow(), identity);

// Configure persistence and desired workflow event handlers.
// (Omitted for brevity.)
ConfigureWorkflowApplication(wfApp);

// Load the persisted workflow instance.
wfApp.Load(InstanceId);

// Resume the workflow.
// wfApp.ResumeBookmark(...);
```

**NOTE**

To download the sample code that accompanies this topic, see [Dynamic Update sample code](#).

# Contract First Workflow Service Development

3/9/2019 • 5 minutes to read • [Edit Online](#)

Starting with .NET Framework 4.5, Windows Workflow Foundation (WF) features better integration between web services and workflows in the form of contract-first workflow development. The contract-first workflow development tool allows you to design the contract in code first. The tool then automatically generates an activity template in the toolbox for the operations in the contract. This topic provides an overview of how the activities and properties in a workflow service map to the attributes of a service contract. For a step-by-step example of creating a contract-first workflow service, see [How to: Create a workflow service that consumes an existing service contract](#).

## In this topic

- [Mapping service contract attributes to workflow attributes](#)
  - [Service Contract Attributes](#)
  - [Operation Contract Attributes](#)
  - [Message Contract Attributes](#)
  - [Data Contract Attributes](#)
  - [Fault Contract Attributes](#)
- [Additional Support and Implementation Information](#)
  - [Unsupported service contract features](#)
  - [Generation of configured messaging activities](#)

## Mapping service contract attributes to workflow attributes

The tables in the following sections specify the different WCF attributes and properties and how they are mapped to the messaging activities and properties in a contract-first workflow.

- [Service Contract Attributes](#)
- [Operation Contract Attributes](#)
- [Message Contract Attributes](#)
- [Data Contract Attributes](#)
- [Fault Contract Attributes](#)

### Service Contract Attributes

| PROPERTY NAME     | SUPPORTED | DESCRIPTION  | WF VALIDATION |
|-------------------|-----------|--|---------------|
| CallbackContract  | No        | Gets or sets the type of callback contract when the contract is a duplex contract.     | (N/A)         |
| ConfigurationName | No        | Gets or sets the name used to locate the service in an application configuration file. | (N/A)         |

| PROPERTY NAME      | SUPPORTED | DESCRIPTION   | WF VALIDATION                                       |
|--------------------|-----------|---|---|
| HasProtectionLevel | Yes       | Gets a value that indicates whether the member has a protection level assigned.                               | Receive.ProtectionLevel should not be null.         |
| Name               | Yes       | Gets or sets the name for the <portType> element in Web Services Description Language (WSDL).                 | Receive.ServiceContractName.LocalName should match. |
| Namespace          | Yes       | Gets or sets the namespace of the <portType> element in Web Services Description Language (WSDL).             | Receive.ServiceContractName.NameSpace should match  |
| ProtectionLevel    | Yes       | Specifies whether the binding for the contract must support the value of the ProtectionLevel property.        | Receive.ProtectionLevel should match.               |
| SessionMode        | No        | Gets or sets whether sessions are allowed, not allowed or required.   | (N/A)   |
| TypeId             | No        | When implemented in a derived class, gets a unique identifier for this Attribute. (Inherited from Attribute.) | (N/A)   |

Insert subsection body here.

### Operation Contract Attributes

| PROPERTY NAME      | SUPPORTED | DESCRIPTION  | WF VALIDATION                               |
|--------------------|-----------|--|---|
| Action             | Yes       | Gets or sets the WS-Addressing action of the request message.  | Receive.Action should match.                |
| AsyncPattern       | No        | Indicates that an operation is implemented asynchronously using a Begin<methodName> and End<methodName> method pair in a service contract. | (N/A)                                       |
| HasProtectionLevel | Yes       | Gets a value that indicates whether the messages for this operation must be encrypted, signed, or both.                                    | Receive.ProtectionLevel should not be null. |

| PROPERTY NAME   | SUPPORTED | DESCRIPTION   | WF VALIDATION   |
|-----------------|-----------|---|---|
| IsInitiating    | No        | Gets or sets a value that indicates whether the method implements an operation that can initiate a session on the server(if such a session exists). | (N/A)   |
| IsOneWay        | Yes       | Gets or sets a value that indicates whether an operation returns a reply message.   | (No SendReply for this Receive OR no ReceiveReply for this Send). |
| IsTerminating   | No        | Gets or sets a value that indicates whether the service operation causes the server to close the session after the reply message, if any, is sent.  | (N/A)   |
| Name            | Yes       | Gets or sets the name of the operation.   | Receive.OperationName should match.                               |
| ProtectionLevel | Yes       | Gets or sets a value that specifies whether the messages of an operation must be encrypted, signed, or both.  | Receive.ProtectionLevel should match.                             |
| ReplyAction     | Yes       | Gets or sets the value of the SOAP action for the reply message of the operation.   | SendReply.Action should match.                                    |
| TypeId          | No        | When implemented in a derived class, gets a unique identifier for this Attribute. (Inherited from Attribute.)                                       | (N/A)   |

### Message Contract Attributes

| PROPERTY NAME      | SUPPORTED | DESCRIPTION   | WF VALIDATION  |
|--------------------|-----------|---|--|
| HasProtectionLevel | Yes       | Gets a value that indicates whether the message has a protection level.                     | No validation<br>(Receive.Content and SendReply.Content must match the message contract type). |
| IsWrapped          | Yes       | Gets or sets a value that specifies whether the message body has a wrapper element.         | No validation<br>(Receive.Content and SendReply.Content must match the message contract type). |
| ProtectionLevel    | No        | Gets or sets a value that specified whether the message must be encrypted, signed, or both. | (N/A)  |

| PROPERTY NAME    | SUPPORTED | DESCRIPTION   | WF VALIDATION  |
|------------------|-----------|---|--|
| TypeId           | Yes       | When implemented in a derived class, gets a unique identifier for this Attribute. (Inherited from Attribute.) | No validation<br>(Receive.Content and SendReply.Content must match the message contract type). |
| WrapperName      | Yes       | Gets or sets the name of the wrapper element of the message body.   | No validation<br>(Receive.Content and SendReply.Content must match the message contract type). |
| WrapperNamespace | No        | Gets or sets the namespace of the message body wrapper element.   | (N/A)  |

### Data Contract Attributes

| PROPERTY NAME | SUPPORTED | DESCRIPTION   | WF VALIDATION  |
|---------------|-----------|---|--|
| IsReference   | No        | Gets or sets a value that indicates whether to preserve object reference data.                                | (N/A)  |
| Name          | Yes       | Gets or sets the name of the data contract for the type.  | No validation<br>(Receive.Content and SendReply.Content must match the message contract type). |
| Namespace     | Yes       | Gets or sets the namespace for the data contract for the type.  | No validation<br>(Receive.Content and SendReply.Content must match the message contract type). |
| TypeId        | No        | When implemented in a derived class, gets a unique identifier for this Attribute. (Inherited from Attribute.) | (N/A)  |

### Fault Contract Attributes

| PROPERTY NAME | SUPPORTED | DESCRIPTION  | WF VALIDATION                           |
|---------------|-----------|--|---|
| Action        | Yes       | Gets or sets the action of the SOAP fault message that is specified as part of the operation contract. | SendReply.Action should match.          |
| DetailType    | Yes       | Gets the type of a serializable object that contains error information.                                | SendReply.Content should match the type |

| PROPERTY NAME      | SUPPORTED | DESCRIPTION   | WF VALIDATION |
|--------------------|-----------|---|---------------|
| HasProtectionLevel | No        | Gets a value that indicates whether the SOAP fault message has a protection level assigned.                   | (N/A)         |
| Name               | No        | Gets or sets the name of the fault message in Web Services Description Language (WSDL).                       | (N/A)         |
| Namespace          | No        | Gets or sets the namespace of the SOAP fault.   | (N/A)         |
| ProtectionLevel    | No        | Specifies the level of protection the SOAP fault requires from the binding.                                   | (N/A)         |
| TypeId             | No        | When implemented in a derived class, gets a unique identifier for this Attribute. (Inherited from Attribute.) | (N/A)         |

## Additional Support and Implementation Information

- [Unsupported service contract features](#)
- [Generation of configured messaging activities](#)

### Unsupported service contract features

- Use of TPL (Task Parallel Library) Tasks in contracts is not supported.
- Inheritance in Service Contracts is not supported.

### Generation of configured messaging activities

Two public static methods are added to the [Receive](#) and [SendReply](#) activities to support the generation of pre-configured message activities when using contract-first workflow services.

- [Receive.FromOperationDescription](#)
- [SendReply.FromOperationDescription](#)

The activity generated by these methods should pass contract validation, and therefore these methods are used internally as part of the validation logic for [Receive](#) and [SendReply](#). The [OperationName](#), [ServiceContractName](#), [Action](#), [SerializerOption](#), [ProtectionLevel](#), and [KnownTypes](#) are all pre-configured to match the imported contract. In the content properties page for the activities in the workflow designer, the **Message** or **Parameters** sections are also pre-configured to match the contract.

WCF fault contracts are also handled by returning a separate set of configured [SendReply](#) activities for each of the faults that show up in the [Faults FaultDescriptionCollection](#).

For other parts of [OperationDescription](#) that are unsupported by WF services today (e.g. WebGet/WebInvoke behaviors, or custom operation behaviors), the API will ignore those values as part of the generation and configuration. No exceptions will be thrown.

# How to: Create a workflow service that consumes an existing service contract

3/14/2019 • 2 minutes to read • [Edit Online](#)

.NET Framework 4.5 features better integration between web services and workflows in the form of contract-first workflow development. The contract-first workflow development tool allows you to design the contract in code first. The tool then automatically generates an activity template in the toolbox for the operations in the contract.

## NOTE

This topic provides step-by-step guidance on creating a contract-first workflow service. For more information about contract-first workflow service development, see [Contract First Workflow Service Development](#).

## Creating the workflow project

1. In Visual Studio, select **File, New Project**. Select the **WCF** node under the **C#** node in the **Templates** tree, and select the **WCF Workflow Service Application** template.
2. Name the new project `ContractFirst` and click **Ok**.

## Creating the service contract

1. Right-click the project in **Solution Explorer** and select **Add, New Item...**. Select the **Code** node on the left, and the **Class** template on the right. Name the new class `IBookService` and click **Ok**.
2. In the top of the code window that appears, add a Using statement to `System.ServiceModel`.

```
using System.ServiceModel;
```

3. Change the sample class definition to the following interface definition.

```
[ServiceContract]
public interface IBookService
{
    [OperationContract]
    void Buy(string bookName);

    [OperationContract(IsOneWay=true)]
    void Checkout();
}
```

4. Build the project by pressing **Ctrl+Shift+B**.

## Importing the service contract

1. Right-click the project in **Solution Explorer** and select **Import Service Contract**. Under **<Current Project>**, open all sub-nodes and select **IBookService**. Click **OK**.
2. A dialog will open, alerting you that the operation completed successfully, and that the generated activities will appear in the toolbox after you build the project. Click **OK**.
3. Build the project by pressing **Ctrl+Shift+B**, so that the imported activities will appear in the toolbox.
4. In **Solution Explorer**, open Service1.xamlx. The workflow service will appear in the designer.

5. Select the **Sequence** activity. In the Properties window, click the... button in the **ImplementedContract** property. In the **Type Collection Editor** window that appears, click the **Type** dropdown, and select the **Browse for Types...** entry. In the **Browse and Select a .NET Type** dialog, under <Current Project>, open all sub-nodes and select **IBookService**. Click **OK**. In the **Type Collection Editor** dialog, click **OK**.
6. Select and delete the **ReceiveRequest** and **SendResponse** activities.
7. From the toolbox, drag a **Buy\_ReceiveAndSendReply** and a **Checkout\_Receive** activity onto the **Sequential Service** activity.

# Workflow Persistence

3/9/2019 • 2 minutes to read • [Edit Online](#)

Workflow persistence is the durable capture of a workflow instance's state, independent of process or computer information. This is done to provide a well-known point of recovery for the workflow instance in the event of system failure, or to preserve memory by unloading workflow instances that are not actively doing work, or to move the state of the workflow instance from one node to another node in a server farm.

Persistence enables process agility, scalability, recovery in the face of failure, and the ability to manage memory more efficiently. The persistence process includes the identification of a persistence point, the gathering of the data to be saved, and finally the delegation of the actual storage of the data to a persistence provider.

To enable persistence for a workflow, you need to associate an instance store with the **WorkflowApplication** or **WorkflowServiceHost** as mentioned in [How to: Enable Persistence for Workflows and Workflow Services](#). The **WorkflowApplication** and **WorkflowServiceHost** use the instance store associated with them to enable persisting workflow instances into a persistence store and loading workflow instances into memory based on the workflow instance data stored in the persistence store.

The .NET Framework 4.6.1 ships with the **SqlWorkflowInstanceStore** class, which allows persistence of data and metadata about workflow instances into a SQL Server 2005 or SQL Server 2008 database. See [SQL Workflow Instance Store](#) for more details.

To store and load your application-specific data along with the workflow instance-related information, you can create persistence participants that extend the **PersistenceParticipant** class. A persistence participant participates in the persistence process to save custom serializable data into the persistence store, to load the data from the instance store into memory, and to perform any additional logic under a persistence transaction. For more information, see [Persistence Participants](#).

Windows Server App Fabric simplifies the process of configuring persistence. For more information, see [Persistence Concepts with Windows Server App Fabric](#)

## Implicit Persistence Points

The following list contains examples of the conditions upon which a workflow is persisted when an instance store is associated with a workflow.

- When a **TransactionScope** activity completes or a **TransactedReceiveScope** activity completes.
- When a workflow instance becomes idle and the **WorkflowIdleBehavior** is set on workflow host. This occurs, for example, when you use messaging activities or a **Delay** activity.
- When a WorkflowApplication becomes idle and the **PersistableIdle** property of the application is set to **PersistableIdleAction.Persist**.
- When a host application is instructed to persist or unload a workflow instance.
- When a workflow instance is terminated or finishes.
- When a **Persist** activity executes.
- When an instance of a workflow developed using a previous version of Windows Workflow Foundation encounters a persistence point during interoperable execution.

## In This Section

- [SQL Workflow Instance Store](#)
- [Instance Stores](#)
- [Persistence Participants](#)
- [Persistence Best Practices](#)
- [Non-Persisted Workflow Instances](#)
- [Pausing and Resuming a Workflow](#)

# SQL Workflow Instance Store

3/9/2019 • 2 minutes to read • [Edit Online](#)

The .NET Framework 4.6.1 ships with the SQL Workflow Instance Store, which allows workflows to persist state information about workflow instances in a SQL Server 2005 or SQL Server 2008 database. This feature is primarily implemented in the form of the [SqlWorkflowInstanceStore](#) class, which derives from the abstract [InstanceStore](#) class of the persistence framework. The SQL Workflow Instance Store feature constitutes a SQL persistence provider, which is a concrete implementation of the persistence API that a host uses to send persistence commands to the store.

The SQL Workflow Instance Store supports both self-hosted workflows or workflow services that use [WorkflowApplication](#) or [WorkflowServiceHost](#) as well as services hosted in WAS using [WorkflowServiceHost](#). You can configure the SQL Workflow Instance Store feature for self-hosted services programmatically by using the object model exposed by the feature. You can configure this feature for services hosted by [WorkflowServiceHost](#) both programmatically by using the object model and also by using an XML configuration file.

The SQL Workflow Instance Store feature (**SqlWorkflowInstanceStore** class) does not implement [PersistenceProviderFactory](#) and hence does not offer persistence support for durable non-workflow WCF services. It also does not implement [WorkflowPersistenceService](#) and hence does not offer persistence support for 3.x workflows. The feature supports persistence for only WF 4.0 (and later) workflows and workflow services. The feature also does not support any databases other than SQL Server 2005 and SQL Server 2008.

The topics in this section describe properties and features of the SQL Workflow Instance Store and provide you with details on configuring the store.

Windows Server App Fabric provides its own instance store and tooling to simplify the configuration and use of the instance store. For more information, see [Windows Server App Fabric Instance Store](#). For more information about the App Fabric SQL Server Persistence Database see [App Fabric SQL Server Persistence Database](#)

## In This Section

- [Properties of SQL Workflow Instance Store](#)
- [How to: Enable SQL Persistence for Workflows and Workflow Services](#)
- [Instance Activation](#)
- [Support for Queries](#)
- [Store Extensibility](#)
- [Security](#)
- [SQL Server Persistence Database](#)

## See also

- [Persistence Samples](#)

# Properties of SQL Workflow Instance Store

3/9/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section provide details about properties of the SQL Workflow Instance Store.

## In This Section

- [Instance Encoding Option](#)
- [Instance Completion Action](#)
- [Instance Locked Exception Action](#)
- [Host Lock Renewal Period](#)
- [Runnable Instances Detection Period](#)
- [Connection String and Connection String Name](#)

## See also

- [Windows Server App Fabric Instance Store](#)
- [App Fabric SQL Server Persistence Database](#)

# Instance Encoding Option

5/4/2018 • 2 minutes to read • [Edit Online](#)

The **Instance Encoding Option** property of the SQL Workflow Instance Store lets you specify whether the SQL persistence provider should compress the workflow instance state information using the GZip algorithm before saving the information into the persistence database. The allowed values for this property are: GZip and None. The default value is None. The following list describes these options.

1. **GZip**. The persistence provider encodes the state information using the GZip algorithm before persisting the state information in the persistence database.
2. **None**. The persistence provider does not encode the state information before saving the information into the persistence database.

Encoding workflow instance state information using the GZip reduces memory consumption in the SQL database and also reduces network consumption if the database resides on a different computer on the network from the computer on which the workflow service host is running. A general guidance is to set the **Instance Encoding Option** property to **None** if the workflow instance state is small.

# Instance Completion Action

5/4/2018 • 2 minutes to read • [Edit Online](#)

The **Instance Completion Action** property of the SQL Workflow Instance Store lets you specify whether the data and metadata of workflow instances is deleted from the persistence database after the instances are completed. The allowed values for this property are **DeleteAll** and **DeleteNothing**. The following list describes these options:

- **DeleteAll (default).** If the value of the property is set to DeleteAll, the data and metadata of workflow instances is deleted from the persistence database after the instances are completed.
- **DeleteNothing.** If the value of the property is set to DeleteNothing, the data and metadata of workflow instances is kept in the persistence database even after the instances are completed.

**Caution**

Keeping instance state information after the instances are completed causes the persistence database to grow in size. As the size of the database grows the database operations that the persistence subsystem performs take longer, so you need to purge the instance state information from the persistence database periodically to have services perform at the level that satisfy your performance needs.

# Instance Locked Exception Action

5/4/2018 • 2 minutes to read • [Edit Online](#)

The [InstanceLockedExceptionAction](#) property of the SQL Workflow Instance Store lets you specify what action the SQL persistence provider should take when it receives an [InstanceLockedException](#). The persistence provider receives this exception when it tries to lock a workflow service instance that is currently locked by another service host. The values for this property are [NoRetry](#), [BasicRetry](#), and [AggressiveRetry](#). The default value is [NoRetry](#). The following list describes the three options:

- [NoRetry](#). The service host does not attempt to lock the workflow service instance and passes the [InstanceLockedException](#) to the caller. If your workflow stays in memory for a period exceeding 60 seconds, use [NoRetry](#) as the retry. The default value is [NoRetry](#).
- [BasicRetry](#). The service host reattempts to lock the workflow service instance with a linear interval between retry attempts and passes the [InstanceLockedException](#) to the caller at the end of the sequence. If your workflow stays in memory approximately between 5-60 seconds, and messages arrive in batches where it is more likely for messages being sent to the same instance on the same host to process all messages before unloading the workflow, use [BasicRetry](#) to achieve the best latency without wasting resources.
- [AggressiveRetry](#). The service host reattempts to lock the workflow service instance with an exponential backoff interval between retry attempts, and passes the exception to the caller at the end of the sequence. If your workflow stays in memory for a very short time (less than 5 seconds), or a Web farm is large and the chance of another message being delivered to the same host is not very high, use [AggressiveRetry](#) to achieve the best latency.

The Instance Locked Exception Action feature supports the following scenarios. In all scenarios, if the `instanceLockedExceptionAction` property of the `SqlWorkflowInstanceStore` is set to [BasicRetry](#) or [AggressiveRetry](#), the host transparently retries to acquire the lock on instances periodically.

1. **Enabling graceful shutdown and overlapped recycling of application domains.** Suppose an **AppDomain** with a service host running workflow service instances is being recycled and a new **AppDomain** is brought up to handle new requests in parallel while the old **AppDomain** is brought down gracefully. The shutdown waits until workflow service instances are idle, and then persists and unloads the instances. Any attempts by hosts in the new **AppDomain** to lock an instance will cause an [InstanceLockedException](#).
2. **Horizontally scaling durable workflows across a homogeneous farm of servers.** Suppose a node of a server farm on which a workflow instance is running crashes and the workflow host cannot remove locks on the instance it is running. When a service host running on another node of the farm receives a message for that workflow instance, it tries to acquire locks on these instances it will receive the [InstanceLockedException](#). The locks will expire after some time because the host that was supposed to renew the lock no longer exists.

**Horizontally scaling durable workflows across a homogeneous farm of servers.** Suppose you want to horizontally scale a durable workflow using multiple hosts behind a NLB (Network Load Balancer), the workflow host running on one node of the farm loads a workflow instance and is processing a message, and the next message to the instance is routed to the host that is running on another node because the NLB does not have routing algorithm to deliver messages to the host that is already running the instance. Upon receiving the message, the second host attempts to load the workflow instance and receives the [InstanceLockedException](#) because the first host has a lock on the instance. The first host unlocks the instance when it is finished with processing the first message and the second host acquires the lock when it

retries the next time, loads the instance, and processes the second message.

# Host Lock Renewal Period

5/4/2018 • 2 minutes to read • [Edit Online](#)

The **Host Lock Renewal Period** property of the SQL Workflow Instance Store lets you specify the time period within which the host renews its lock on a workflow instance. The lock remains valid for Host Lock Renewal Period + 30 seconds. If the host fails to renew the lock (in other words, extend the lease) within this time period, the lock expires and the persistence provider unlocks the instance. The value for this property is of type TimeSpan of the form "hh:mm:ss". The minimum permitted value is "00:00:01" (1 second). The default value of this property is "00:00:30" (30 seconds).

This property is significant in scenarios where a workflow service host fails before it can unlock a workflow service instance that it owns. In this scenario, the lock on the workflow service instance in the persistence database is removed by the persistence provider after the lock expires so that another workflow service host running on the same computer or another computer in a server farm can acquire the lock and load the workflow service instance into memory to resume its execution from its last persisted state.

Setting a higher value for this property causes the workflow service instances to be locked in the persistence database for a longer time and therefore delays the recovery of the instance from the last persistence point. Setting a short interval for this property causes the new instance of the workflow service host to pick up the failed workflow service instance quickly, but causes an increase in workload for the workflow service host and the SQL Server database.

The SQL Workflow Instance Store runs an internal task that periodically wakes up and detects instances with expired locks on them. When it finds instances with expired locks, it places the instances in the RunnableInstances table so that a workflow host can pick up and run these instances.

# Runnable Instances Detection Period

3/9/2019 • 2 minutes to read • [Edit Online](#)

The SQL Workflow Instance Store runs an internal task that periodically wakes up and detects runnable or activatable instances in the persistence database. The **Runnable Instances Detection Period** property of the SQL Workflow Instance Store specifies the time period after which the SQL Workflow Instance Store runs a detection task to detect any runnable or activatable workflow instances in the persistence database after the previous detection cycle.

Setting a shorter interval for this property reduces the time between the expiration of a timer associated with a workflow instance and the signaling of the event and subsequent loading of the instance. However, it also increases the processing load on a host and may not be desirable in scenarios where durable timers and/or host failures are rare. The type of the property is `TimeSpan` and the value of the property follows the format: `hh:mm:ss`. The minimum value for this property is `00:00:01`. The default value for the property is `00:00:05`.

For more information detecting and activating runnable and activatable workflow instances, see [Instance Activation](#).

# Connection String and Connection String Name

5/4/2018 • 2 minutes to read • [Edit Online](#)

**Connection String** property specifies the connection string that the SQL Workflow Instance Store should use to connect to a persistence database. This parameter is an optional parameter. **Connection String Name** property specifies the name of the named connection string that the SQL Workflow Instance Store should use to connect to the persistence database. This parameter is an optional parameter. You should specify a value for the Connection String Name property or the Connection String property if you do not want the SQL Workflow Instance Store to use the default named connection string **DefaultSqlWorkflowInstanceStoreConnectionString**.

# How to: Enable SQL Persistence for Workflows and Workflow Services

3/9/2019 • 4 minutes to read • [Edit Online](#)

This topic describes how to configure the SQL Workflow Instance Store feature to enable persistence for your workflows and workflow services both programmatically and by using a configuration file.

Windows Server App Fabric simplifies the process of configuring persistence. For more information, see [App Fabric Persistence Configuration](#)

Before using the SQL Workflow Instance Store feature, create a database that the feature uses to persist workflow instances. The .NET Framework 4.6.1 set-up program copies SQL script files associated with the SQL Workflow Instance Store feature to the %WINDIR%\Microsoft.NET\Framework\v4.x\SQL\EN folder. Run these script files against a SQL Server 2005 or SQL Server 2008 database that you want the SQL Workflow Instance Store to use to persist workflow instances. Run the SqlWorkflowInstanceStoreSchema.sql file first and then run the SqlWorkflowInstanceStoreLogic.sql file.

## NOTE

To clean up the persistence database to have a fresh database, run the scripts in %WINDIR%\Microsoft.NET\Framework\v4.x\SQL\EN in the following order.

1. SqlWorkflowInstanceStoreSchema.sql
2. SqlWorkflowInstanceStoreLogic.sql

## IMPORTANT

If you do not create a persistence database, the SQL Workflow Instance Store feature throws an exception similar to the following one when a host tries to persist workflows.

System.Data.SqlClient.SqlException: Could not find stored procedure 'System.Activities.DurableInstancing.CreateLockOwner'

The following sections describe how to enable persistence for workflows and workflow services using the SQL Workflow Instance Store. For more information about properties of the SQL Workflow Instance Store, see [Properties of SQL Workflow Instance Store](#).

## Enabling Persistence for Self-Hosted Workflows that use WorkflowApplication

You can enable persistence for self-hosted workflows that use [WorkflowApplication](#) programmatically by using the [SqlWorkflowInstanceStore](#) object model. The following procedure contains steps to do this.

### To enable persistence for self-hosted workflows

1. Add a reference to System.Activities.DurableInstancing.dll.
2. Add the following statement at the top of the source file after the existing "using" statements.

```
using System.Activities.DurableInstancing;
```

3. Construct a `SqlWorkflowInstanceStore` and assign it to the `InstanceStore` of the `WorkflowApplication` as shown in the following code example.

```
SqlWorkflowInstanceStore store =
    new SqlWorkflowInstanceStore("Server=.\\SQLEXPRESS;Initial Catalog=Persistence;Integrated
Security=SSPI");

WorkflowApplication wfApp =
    new WorkflowApplication(new Workflow1());

wfApp.InstanceStore = store;
```

**NOTE**

Depending on your edition of SQL Server, the connection string server name may be different.

4. Invoke the `Persist` method on the `WorkflowApplication` object to persist a workflow, or `Unload` method to persist and unload a workflow. You can also handle the `PersistableIdle` event raised by the `WorkflowApplication` object and return appropriate (`Persist` or `Unload`) member of `PersistableIdleAction`.

```
wfApp.PersistableIdle = delegate(WorkflowApplicationIdleEventArgs e)
{
    return PersistableIdleAction.Persist;
};
```

**NOTE**

See the [How to: Create and Run a Long Running Workflow](#) step of the [Getting Started Tutorial](#) for step by step instructions.

## Enabling Persistence for Self-Hosted Workflow Services that use the `WorkflowServiceHost`

You can enable persistence for self-hosted workflow services that use `WorkflowServiceHost` programmatically by using the `SqlWorkflowInstanceStoreBehavior` class or the `DurableInstancingOptions` class.

### Using the `SqlWorkflowInstanceStoreBehavior` Class

The following procedure contains steps to use the `SqlWorkflowInstanceStoreBehavior` class to enable persistence for self-hosted workflow services.

#### To enable persistence using `SqlWorkflowInstanceStoreBehavior`

1. Add a reference to the `System.ServiceModel.dll`.
2. Add the following statement at the top of the source file after the existing "using" statements.

```
using System.ServiceModel.Activities.Description;
```

3. Create an instance of the `WorkflowServiceHost` and add endpoints for the workflow service.

```
WorkflowServiceHost host = new WorkflowServiceHost(new CountingWorkflow(), new Uri(hostBaseAddress));
host.AddServiceEndpoint("ICountingWorkflow", new BasicHttpBinding(), "");
```

4. Construct a `SqlWorkflowInstanceStoreBehavior` object and to set properties of the behavior object.

```
SqlWorkflowInstanceStoreBehavior instanceStoreBehavior = new  
SqlWorkflowInstanceStoreBehavior(connectionString);  
instanceStoreBehavior.HostLockRenewalPeriod = new TimeSpan(0, 0, 5);  
instanceStoreBehavior.InstanceCompletionAction = InstanceCompletionAction.DeleteAll;  
instanceStoreBehavior.InstanceLockedExceptionAction = InstanceLockedExceptionAction.AggressiveRetry;  
instanceStoreBehavior.InstanceEncodingOption = InstanceEncodingOption.GZip;  
instanceStoreBehavior.RunnableInstancesDetectionPeriod = new TimeSpan("00:00:02");  
host.Description.Behaviors.Add(instanceStoreBehavior);
```

5. Open the workflow service host.

```
host.Open();
```

### Using the DurableInstancingOptions Property

When the `SqlWorkflowInstanceStoreBehavior` is applied, the `DurableInstancingOptions.InstanceStore` on the `WorkflowServiceHost` is set to the `SqlWorkflowInstanceStore` object created using the configuration values. You can do the same programmatically to set the `DurableInstancingOptions` property of the `WorkflowServiceHost` without using the `SqlWorkflowInstanceStoreBehavior` class as shown in the following code example.

```
workflowServiceHost.DurableInstancingOptions.InstanceStore = sqlInstanceStoreObject;
```

## Enabling Persistence for WAS-Hosted Workflow Services that use the WorkflowServiceHost using a Configuration File

You can enable persistence for self-hosted or Windows Process Activation Service (WAS)-hosted workflow services by using a configuration file. A WAS-hosted workflow service uses the `WorkflowServiceHost` as the self-hosted workflow services do.

The `SqlWorkflowInstanceStoreBehavior`, a service behavior that allows you to conveniently change the [SQL Workflow Instance Store](#) properties through XML configuration. For WAS-hosted workflow services, use the `Web.config` file. The following configuration example shows how to configure the SQL Workflow Instance Store by using the `sqlWorkflowInstanceStore` behavior element in a configuration file.

```
<serviceBehaviors>  
    <behavior name="">  
        <sqlWorkflowInstanceStore  
            connectionString="Data Source=(local);Initial  
Catalog=DefaultPersistenceProviderDb;Integrated Security=True;Async=true"  
            instanceEncodingOption="GZip | None"  
            instanceCompletionAction="DeleteAll | DeleteNothing"  
            instanceLockedExceptionAction="NoRetry | BasicRetry | AggressiveRetry"  
            hostLockRenewalPeriod="00:00:30"  
            runnableInstancesDetectionPeriod="00:00:05">  
  
        </sqlWorkflowInstanceStore>  
    </behavior>  
</serviceBehaviors>
```

If you do not set values for the `connectionString` or the `connectionStringName` property, the SQL Workflow Instance Store uses the default named connection string `DefaultSqlWorkflowInstanceStoreConnectionString`.

When the `SqlWorkflowInstanceStoreBehavior` is applied, the `DurableInstancingOptions.InstanceStore` on the `WorkflowServiceHost` is set to the `SqlWorkflowInstanceStore` object created using the configuration values. You can do the same programmatically to use the `SqlWorkflowInstanceStore` with `WorkflowServiceHost` without using the

service behavior element.

```
workflowServiceHost.DurableInstancingOptions.InstanceStore = sqlInstanceStoreObject;
```

#### IMPORTANT

It is recommended that you do not store sensitive information such as user names and passwords in the Web.config file. If you do store sensitive information in the Web.config file, you should secure access to the Web.config file by using file system Access Control Lists (ACLs). In addition, you can also secure the configuration values within a configuration file as mentioned in [Encrypting Configuration Information Using Protected Configuration](#).

### Machine.config Elements Related to the SQL Workflow Instance Store Feature

The .NET Framework 4.6.1 installation adds the following elements related to the SQL Workflow Instance Store feature to the Machine.config file:

- Adds the following behavior extension element to the Machine.config file so that you can use the <sqlWorkflowInstanceStore> service behavior element in the configuration file to configure persistence for your services.

```
<configuration>
  <system.serviceModel>
    <extensions>
      <behaviorExtensions>
        <add name="sqlWorkflowInstanceStore"
          type="System.Activities.DurableInstancing.SqlWorkflowInstanceStoreElement,
          System.Activities.DurableInstancing, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
        />
      </behaviorExtensions>
    </extensions>
  <system.serviceModel>
<configuration>
```

# Instance Activation

3/9/2019 • 2 minutes to read • [Edit Online](#)

The SQL Workflow Instance Store runs an internal task that periodically wakes up and detects runnable or activatable workflow instances in the persistence database. If it finds a runnable workflow instance, it notifies the workflow host that is capable of activating the instance. If the instance store finds an activatable workflow instance, it notifies a generic host that activates a workflow host, which in turn runs the workflow instance. The following sections in this topic explain the instance activation process in detail.

## Detecting and Activating Runnable Workflow Instances

The SQL Workflow Instance Store considers a workflow instance *Runnable* if the instance is not in the suspended state or the completed state and satisfies the following conditions:

- The instance is unlocked and has a pending timer that has expired.
- The instance has an expired lock on it.
- The instance is unlocked and its status is **Executing**.

The SQL Workflow Instance Store raises the [HasRunnableWorkflowEvent](#) when it finds a runnable instance. After this, the `SqlWorkflowInstanceStateStore` stops monitoring until the [TryLoadRunnableWorkflowCommand](#) is called once on the store.

A workflow host that has subscribed for the [HasRunnableWorkflowEvent](#) and capable of loading the instance executes the [TryLoadRunnableWorkflowCommand](#) against the instance store to load the instance into memory. A workflow host is considered capable of loading a workflow instance if the host and the instance have metadata property **WorkflowServiceType** set to the same value.

## Detecting and Activating Activatable Workflow Instances

A workflow instance is considered *activatable* if the instance is runnable and there is no workflow host that is capable of loading the instance is running on the computer. See Detecting and Activating Runnable Workflow Instances above for the definition of a runnable workflow instance.

The SQL Workflow Instance Store raises the [HasActivatableWorkflowEvent](#) when it finds an activatable workflow instance in the database. After this, the `SqlWorkflowInstanceStateStore` stops monitoring until the [QueryActivatableWorkflowsCommand](#) is called once on the store.

When a generic host that has subscribed for the [HasActivatableWorkflowEvent](#) receives the event, it executes the [QueryActivatableWorkflowsCommand](#) against the instance store to obtain activation parameters required to create a workflow host. The generic host uses these activation parameters to create a workflow host, which in turn loads and runs the runnable service instance.

## Generic Hosts

A generic host is a host with the value of the metadata property **WorkflowServiceType** for generic hosts is set to **WorkflowServiceType.Any** to indicate that it can handle any workflow type. A generic host has an XName parameter named **ActivationType**.

Currently, the SQL Workflow Instance Store supports generic hosts with value of the ActivationType parameter set to **WAS**. If the ActivationType is not set to WAS, the SQL Workflow Instance Store throws an

[InstancePersistenceException](#). The Workflow Management Service that ships with the hosting features of Windows Server AppFabric is a generic host that has the activation type set to **WAS**.

For WAS activation, a generic host requires a set of activation parameters to derive the endpoint address at which new hosts can be activated. The activation parameters for WAS activation are name of the site, path to the application relative to the site, and path to the service relative to the application. The SQL Workflow Instance Store stores these activation parameters during the execution of the [SaveWorkflowCommand](#).

## Runnable Instances Detection Period

The **Runnable Instances Detection Period** property of the SQL Workflow Instance Store specifies the time period after which the SQL Workflow Instance Store runs a detection task to detect any runnable or activatable workflow instances in the persistence database after the previous detection cycle. See [Runnable Instances Detection Period](#) for more details on this property.

# Support for Queries

3/9/2019 • 2 minutes to read • [Edit Online](#)

The SQL Workflow Instance Store records a set of well-known properties in the store. Users can query for instances based on these properties. The following list contains some of these well-known properties:

- **Site Name.** Name of the Web site that contains the service.
- **Relative Application Path.** Path of the application relative to the Web site.
- **Relative Service Path.** Path of the service relative to the application.
- **Service Name.** Name of the service.
- **Service Namespace.** Name of the namespace that the service uses.
- **Current Machine.**
- **Last Machine.** The computer on which the workflow service instance ran the last time.

## NOTE

For self-hosted scenarios using Workflow Service Host, only the last four properties are populated. For Workflow Application scenarios, only the last property is populated.

The workflow runtime supplies values for the first three properties. The workflow service host supplies the value for the **Suspend Reason** property. The SQL Workflow Instance Store itself supplies values for the **Last Updated Machine** property.

The SQL Workflow Instance Store feature also lets you specify the custom properties for which you want to store the values in the persistence database and that you want to use in queries. For more information about custom promotions, see [Store Extensibility](#).

## Views

The instance store contains the following views. See [Persistence Database Schema](#) for further details.

### The Instances View

The Instances view contains the following fields:

1. **Id**
2. **PendingTimer**
3. **CreationTime**
4. **LastUpdatedTime**
5. **ServiceDeploymentId**
6. **SuspensionExceptionName**
7. **SuspensionReason**
8. **ActiveBookmarks**

9. **CurrentMachine**
10. **LastMachine**
11. **ExecutionStatus**
12. **IsInitialized**
13. **IsSuspended**
14. **IsCompleted**
15. **EncodingOption**
16. **ReadWritePrimitiveDataProperties**
17. **WriteOnlyPrimitiveDataProperties**
18. **ReadWriteComplexDataProperties**
19. **WriteOnlyComplexDataProperties**

#### **The ServiceDeployments view**

The ServiceDeployments view contains the following fields:

1. **SiteName**
2. **RelativeServicePath**
3. **RelativeApplicationPath**
4. **ServiceName**
5. **ServiceNamespace**

#### **The InstancePromotedProperties view**

The InstancePromotedProperties view contains the following fields. For details on promoted properties, see the [Store Extensibility](#) topic.

1. **InstanceId**
2. **EncodingOption**
3. **PromotionName**
4. **Value#** (a range of fields from **Value1** to **Value64**).

# Store Extensibility

3/9/2019 • 2 minutes to read • [Edit Online](#)

[SqlWorkflowInstanceStore](#) allows users to promote custom, application-specific properties that can be used to query for instances in the persistence database. The act of promoting a property causes the value to be available within a special view in the database. These promoted properties (properties that can be used in user queries) can be of simple types such as Int64, Guid, String, and DateTime or of a serialized binary type (byte[]).

The [SqlWorkflowInstanceStore](#) class has the [Promote](#) method that you can use to promote a property as a property that can be used in queries. The following example is an end-to-end example of store extensibility.

1. In this example scenario, a document processing (DP) application has workflows, each of which uses custom activities for document processing. These workflows have a set of state variables that need to be made visible to the end user. To achieve this, the DP application provides an instance extension of type [PersistenceParticipant](#), which is used by activities to supply the state variables.

```
class DocumentStatusExtension : PersistenceParticipant
{
    public string DocumentId;
    public string ApprovalStatus;
    public string UserName;
    public DateTime LastUpdateTime;
}
```

2. The new extension is then added to the host.

```
static Activity workflow = CreateWorkflow();
WorkflowApplication application = new WorkflowApplication(workflow);
DocumentStatusExtension documentStatusExtension = new DocumentStatusExtension ();
application.Extensions.Add(documentStatusExtension);
```

For more details about adding a custom persistence participant, see the [Persistence Participants](#) sample.

3. The custom activities in the DP application populate various status fields in the [Execute](#) method.

```
public override void Execute(CodeActivityContext context)
{
    // ...
    context.GetExtension<DocumentStatusExtension>().DocumentId = Guid.NewGuid();
    context.GetExtension<DocumentStatusExtension>().UserName = "John Smith";
    context.GetExtension<DocumentStatusExtension>().ApprovalStatus = "Approved";
    context.GetExtension<DocumentStatusExtension>().LastUpdateTime = DateTime.Now();
    // ...
}
```

4. When a workflow instance reaches a persistence point, the [CollectValues](#) method of the **DocumentStatusExtension** persistence participant saves these properties into the persistence data collection.

```

class DocumentStatusExtension : PersistenceParticipant
{
    const XNamespace xNS = XNamespace.Get("http://contoso.com/DocumentStatus");

    protected override void CollectValues(out IDictionary<XName, object> readOnlyValues, out
IDictionary<XName, object> writeOnlyValues)
    {
        readOnlyValues = new Dictionary<XName, object>();
        readOnlyValues.Add(xNS.GetName("UserName"), this.UserName);
        readOnlyValues.Add(xNS.GetName("ApprovalStatus"), this.ApprovalStatus);
        readOnlyValues.Add(xNS.GetName("DocumentId"), this.DocumentId);
        readOnlyValues.Add(xNS.GetName("LastModifiedTime"), this.LastUpdateTime);

        writeOnlyValues = null;
    }
    // ...
}

```

#### NOTE

All these properties are passed to **SqlWorkflowInstanceStore** by the persistence framework through the **SaveWorkflowCommand.InstanceData** collection.

5. The DP application initializes the SQL Workflow Instance Store and invokes the **Promote** method to promote this data.

```

SqlWorkflowInstanceStore store = new SqlWorkflowInstanceStore(connectionString);

List<XName> variantProperties = new List<XName>()
{
    xNS.GetName("UserName"),
    xNS.GetName("ApprovalStatus"),
    xNS.GetName("DocumentId"),
    xNS.GetName("LastModifiedTime")
};

store.Promote("DocumentStatus", variantProperties, null);

```

Based on this promotion information, **SqlWorkflowInstanceStore** places the data properties in the columns of the **InstancePromotedProperties** view.

6. To query a subset of the data from the promotion table, the DP application adds a customized view on top of the promotion view.

```

create view [dbo].[DocumentStatus] with schemabinding
as
    select P.[InstanceId] as [InstanceId],
           P.Value1 as [UserName],
           P.Value2 as [ApprovalStatus],
           P.Value3 as [DocumentId],
           P.Value4 as [LastUpdatedTime]
    from [System.Activities.DurableInstancing].[InstancePromotedProperties] as P
    where P.PromotionName = N'DocumentStatus'
go

```

[System.Activities.DurableInstancing.InstancePromotedProperties]  
view

| COLUMN NAME                             | COLUMN TYPE    | DESCRIPTION  |
|---|----------------|--|
| InstanceId                              | GUID           | The workflow instance that this promotion belongs to.  |
| PromotionName                           | nvarchar(400)  | The name of the promotion itself.  |
| Value1, Value2, Value3,...,Value32      | sql_variant    | The value of the promoted property itself. Most SQL primitive data types except binary blobs and strings over 8000 bytes in length can fit in sql_variant. |
| Value33, Value34, Value35, ..., Value64 | varbinary(max) | The value of promoted properties that are explicitly declared as varbinary(max).   |

# Security

3/9/2019 • 2 minutes to read • [Edit Online](#)

The SQL Workflow Instance Store uses the following database security roles to secure access to instance state information in the persistence database.

- **System.Activities.DurableInstancing.InstanceStoreUsers.** This role has read and write access to public views and execution rights to stored procedures that are involved in creating, loading and saving instances.
- **System.Activities.DurableInstancing.InstanceStoreObservers.** This role has read-only access to public views.
- **System.Activities.DurableInstancing.WorkflowActivationUsers.** This role has execution rights to stored procedures that are involved in the instance activation process. For more information about instance activation, see [Instance Activation](#). The user account under which a generic host (such as the Workflow Management Service or hosting features of Windows Server AppFabric) runs should be added to this database role.

For more information about security for persistence stores with Windows Server App Fabric, see [Security Configuration for Persistence Stores in App Fabric](#)

**Caution**

A client that has access to its own instance data in the instance store has access to all other instances in the same instance store. The instance store does not support specifying security permissions at the instance level. You should create separate instance stores and map different groups/users to have access to different stores.

# SQL Server Persistence Database

3/9/2019 • 2 minutes to read • [Edit Online](#)

This section provides details about public database views supported by the SQL Workflow Instance Store and shows how to de-serialize primitive instance data properties and how to query for non-persisted instances.

## In This Section

- [Persistence Database Schema](#)
- [How to: Deserialize Instance Data Properties](#)
- [How to: Query for Non-persisted Instances](#)

## See also

- [App Fabric SQL Server Persistence Database](#)

# Persistence Database Schema

8/31/2018 • 5 minutes to read • [Edit Online](#)

This topic describes the public views supported by the SQL Workflow Instance Store.

## Instances view

The **Instances** view contains general information about all workflow Instances in the Database.

| COLUMN NAME             | COLUMN TYPE      | DESCRIPTION   |
|-------------------------|------------------|---|
| InstanceId              | UniqueIdentifier | The ID of a workflow instance.  |
| PendingTimer            | DateTime         | Indicates that the workflow is blocked on a Delay activity and will be resumed after the timer expires. This value can be null if the workflow is not blocked waiting on a timer to expire.   |
| CreationTime            | DateTime         | Indicates when the workflow was created.  |
| LastUpdatedTime         | DateTime         | Indicates the last time that the workflow was persisted to the database.  |
| ServiceDeploymentId     | BigInt           | Acts as a foreign key to the [ServiceDeployments] view. If the current workflow instance is an instance of a web-hosted service, then this column has a value, otherwise it is set to NULL.   |
| SuspensionExceptionName | Nvarchar(450)    | Indicates the type of exception (e.g. InvalidOperationException) that caused the workflow to suspend.   |
| SuspensionReason        | Nvarchar(max)    | Indicates why the Workflow Instance was suspended. If an exception caused the instance to suspend, then this column contains the message associated with the exception.<br><br>If the instance was manually suspended, then this column contains the user-specified reason for suspending the instance. |
| ActiveBookmarks         | Nvarchar(max)    | If the workflow Instance is Idle, this property indicates what bookmarks the instance is blocked on. If the Instance is not idle, then this column is NULL.   |

| COLUMN NAME                      | COLUMN TYPE    | DESCRIPTION   |
|----------------------------------|----------------|---|
| CurrentMachine                   | Nvarchar(128)  | Indicates the name of the computer currently has the workflow instance loaded in memory.  |
| LastMachine                      | Nvarchar(450)  | Indicates the last computer that loaded the workflow instance.  |
| ExecutionStatus                  | Nvarchar(450)  | Indicates the current execution state of the Workflow. Possible states include <b>Executing</b> , <b>Idle</b> , <b>Closed</b> .   |
| IsInitialized                    | Bit            | Indicates whether the workflow instance has been initialized. An initialized workflow instance is a workflow instance that has been persisted at least once.  |
| IsSuspended                      | Bit            | Indicates whether the workflow instance has been suspended.   |
| IsCompleted                      | Bit            | Indicates whether the Workflow Instance has finished executing. <b>Note:</b> If the <b>InstanceCompletionAction</b> property is set to <b>DeleteAll</b> , the instances are removed from the view upon completion.  |
| EncodingOption                   | TinyInt        | <p>Describes the encoding used to serialize the data properties.</p> <ul style="list-style-type: none"> <li>- 0 – No encoding</li> <li>- 1 – GzipStream</li> </ul>  |
| ReadWritePrimitiveDataProperties | Varbinary(max) | <p>Contains serialized instance data properties that will be provided back to the workflow Runtime when the instance is loaded.</p> <p>Each primitive property is a native CLR type, which means that no special assemblies are needed to deserialize the blob.</p> |
| WriteOnlyPrimitiveDataProperties | Varbinary(max) | <p>Contains serialized instance data properties that are not provided back to the workflow runtime when the instance is loaded.</p> <p>Each primitive property is a native CLR type, which means that no special assemblies are needed to deserialize the blob.</p> |

| COLUMN NAME                    | COLUMN TYPE    | DESCRIPTION  |
|--------------------------------|----------------|--|
| ReadWriteComplexDataProperties | Varbinary(max) | <p>Contains serialized instance data properties that will be provided back to the workflow runtime when the instance is loaded.</p> <p>A deserializer would require knowledge of all object types stored in this blob.</p> |
| WriteOnlyComplexDataProperties | Varbinary(max) | <p>Contains serialized instance data properties that are not provided back to the workflow runtime when the instance is loaded.</p> <p>A deserializer would require knowledge of all object types stored in this blob.</p> |
| IdentityName                   | Nvarchar(max)  | The name of the workflow definition.   |
| IdentityPackage                | Nvarchar(max)  | The package information given when the workflow was created (such as the assembly name).   |
| Build                          | BigInt         | The build number of the workflow version.  |
| Major                          | BigInt         | The major number of the workflow version.  |
| Minor                          | BigInt         | The minor number of the workflow version.  |
| Revision                       | BigInt         | The revision number of the workflow version.   |

#### Caution

The **Instances** view also contains a Delete trigger. Users with the appropriate permissions can execute delete statements against this view that will forcefully remove workflow Instances from the Database. We recommend deleting directly from the view only as a last resort because deleting an instance from underneath the workflow runtime could result in unintended consequences. Instead, use the Workflow Instance Management Endpoint to have the workflow runtime terminate the instance. If you want to delete a large number of Instances from the view, make sure there are no active runtimes that could be operating on these instances.

## ServiceDeployments view

The **ServiceDeployments** view contains deployment information for all Web (IIS/WAS) hosted workflow services. Each workflow instance that is Web-hosted will contain a **ServiceDeploymentId** that refers to a row in this view.

| COLUMN NAME         | COLUMN TYPE | DESCRIPTION                    |
|---------------------|-------------|--------------------------------|
| ServiceDeploymentId | BigInt      | The primary key for this view. |

| COLUMN NAME             | COLUMN TYPE   | DESCRIPTION  |
|-------------------------|---------------|--|
| SiteName                | Nvarchar(max) | Represents the name of the site that contains the workflow service (e.g. <b>Default Web Site</b> ).                                      |
| RelativeServicePath     | Nvarchar(max) | Represents the virtual path relative to the site that points to the workflow service. (e.g. <b>/app1/PurchaseOrderService.svc</b> ).     |
| RelativeApplicationPath | Nvarchar(max) | Represents the virtual path relative to the site that points to an application that contains the workflow service. (e.g. <b>/app1</b> ). |
| ServiceName             | Nvarchar(max) | Represents the name of the workflow Service. (e.g. <b>PurchaseOrderService</b> ).  |
| ServiceNamespace        | Nvarchar(max) | Represents the namespace of the workflow Service. (e.g. <b>MyCompany</b> ).  |

The ServiceDeployments View also contains a Delete trigger. Users with the appropriate permissions can execute delete statements against this view to remove ServiceDeployment entries from the Database. Note that:

1. Deleting entries from this view is costly since the entire Database must be locked prior to performing this operation. This is necessary to avoid the scenario where a workflow Instance could refer to a non-existent ServiceDeployment entry. Delete from this view only during down times / maintenance windows.
2. Any attempt to delete a ServiceDeployment row which is referenced to by entries in the **Instances** view will result in a no-op. You can only delete ServiceDeployment rows with zero references.

## InstancePromotedProperties view

The **InstancePromotedProperties** view contains information for all the promoted properties that are specified by the user. A promoted property functions as a first-class property, which a user can use in queries to retrieve instances. For example, a user could add a PurchaseOrder promotion which always stores the cost of an order in the **Value1** column. This would enable a user to query for all purchase orders whose cost exceeds a certain value.

| COLUMN TYPE    | COLUMN TYPE       | DESCRIPTION   |
|----------------|-------------------|---|
| Instanceld     | UniquedIdentifier | The ID of the Workflow Instance   |
| EncodingOption | TinyInt           | <p>Describes the encoding used to serialize the promoted binary properties.</p> <ul style="list-style-type: none"> <li>- 0 – No encoding</li> <li>- 1 – GZipStream</li> </ul> |

| COLUMN TYPE   | COLUMN TYPE    | DESCRIPTION   |
|---------------|----------------|---|
| PromotionName | Nvarchar(400)  | The name of the Promotion associated with this instance. The PromotionName is needed to add context to the generic columns in this row.<br><br>For example, a PromotionName of PurchaseOrder could indicate that Value1 contains the cost of the order, Value2 contains the name of the customer who placed the order, Value 3 contains the address of the customer, and so on. |
| Value[1-32]   | SqVariant      | Value[1-32] contains values that can be stored in a SqVariant column. A single promotion cannot contain more than 32 SqVariants.  |
| Value[33-64]  | Varbinary(max) | Value[33-64] contains serialized values. For instance, Value33 could contain a JPEG of an item being purchased. A single promotion cannot contain more than 32 binary properties  |

The InstancePromotedProperties view is schema bound, which means that users can add indices on one or more columns in order to optimize queries against this view.

**NOTE**

An indexed view requires more storage and adds additional processing overhead. Please refer to [Improving Performance with SQL Server 2008 Indexed Views](#) for more information.

# How to: Deserialize Instance Data Properties

5/4/2018 • 3 minutes to read • [Edit Online](#)

There may be situations when a user or workflow administrator may want to manually inspect the state of a persisted workflow instance. [SqlWorkflowInstanceStateStore](#) provides a view on the Instances table that exposes the following four columns:

- `ReadWritePrimitiveDataProperties`
- `WriteOnlyPrimitiveDataProperties`
- `ReadWriteComplexDataProperties`
- `WriteOnlyComplexDataProperties`

Primitive data properties refer to properties whose .NET Framework types are considered to be "common" (for example, `Int32` and `String`), while complex data properties refer to all other types. An exact enumeration of primitive types is found later in this code example.

Read/write properties refer to properties that are returned back to the Workflow Runtime when an instance is loaded. `WriteOnly` properties are written to the database and then never read again.

This example provides code that enables a user to deserialize primitive data properties. Given a byte array read from either the `ReadWritePrimitiveDataProperties` or `WriteOnlyPrimitiveDataProperties` column, this code will convert the binary large object (BLOB) into a [Dictionary< TKey, TValue >](#) of type `< XName, object >` where each key value pair represents a property name and its corresponding value.

This example does not demonstrate how to deserialize complex data properties because this is currently not a supported operation.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.IO.Compression;
using System.Xml.Linq;
using System.Xml;
using System.Globalization;
using System.Data.SqlClient;

namespace PropertyReader
{
    class Program
    {
        const string ConnectionString = @"Data Source=localhost;Initial Catalog=Persistence;Integrated Security=True;Asynchronous Processing=True";
        static void Main(string[] args)
        {
            string queryString = "SELECT TOP 10 * FROM [System.Activities.DurableInstancing].[Instances]";

            using (SqlConnection connection =
                  new SqlConnection(ConnectionString))
            {
                SqlCommand command =
                    new SqlCommand(queryString, connection);
                connection.Open();

                SqlDataReader reader = command.ExecuteReader();
```

```

        . . .

        byte encodingOption;

        while (reader.Read())
        {
            if (reader["ReadWritePrimitiveDataProperties"] != DBNull.Value)
            {
                encodingOption = (byte)reader["EncodingOption"];
                Console.WriteLine("Printing the ReadWritePrimitiveDataProperties of the instance with
Id:" + reader["InstanceId"]);
                foreach (KeyValuePair<XName, object> pair in (Dictionary<XName,
object>)ReadDataProperties((byte[])reader["ReadWritePrimitiveDataProperties"], encodingOption))
                {
                    Console.WriteLine("{0}, {1}" , pair.Key, pair.Value);
                }
                Console.WriteLine();
            }
            if (reader["WriteOnlyPrimitiveDataProperties"] != DBNull.Value)
            {
                encodingOption = (byte)reader["EncodingOption"];
                Console.WriteLine("Printing the WriteOnlyPrimitiveDataProperties of the instance with
Id:" + reader["InstanceId"]);
                foreach (KeyValuePair<XName, object> pair in (Dictionary<XName,
object>)ReadDataProperties((byte[])reader["WriteOnlyPrimitiveDataProperties"], encodingOption))
                {
                    Console.WriteLine("{0}, {1}" , pair.Key, pair.Value);
                }
                Console.WriteLine();
            }
        }

        // Call Close when done reading.
        reader.Close();
    }

    Console.ReadLine();

}

static Dictionary<XName, object> ReadDataProperties(byte[] serializedDataProperties, byte
encodingOption)
{
    if (serializedDataProperties != null)
    {
        Dictionary<XName, object> propertyBag = new Dictionary<XName, object>();
        bool isCompressed = (encodingOption == 1);

        using (MemoryStream memoryStream = new MemoryStream(serializedDataProperties))
        {
            // if the instance state is compressed using GZip algorithm
            if (isCompressed)
            {
                // decompress the data using the GZip
                using (GZipStream stream = new GZipStream(memoryStream, CompressionMode.Decompress))
                {
                    // create an XmlReader object and pass it on to the helper method
                    ReadPrimitiveDataProperties
                        using (XmlReader reader = XmlDictionaryReader.CreateBinaryReader(stream,
XmlDictionaryReaderQuotas.Max))
                        {
                            // invoke the helper method
                            ReadPrimitiveDataProperties(reader, propertyBag);
                        }
                }
            }
            else
            {
                // if the instance data is not compressed
                // create an XmlReader object and pass it on to the helper method

```

```

        // Create an XmlReader object and pass it on to the helper method
    ReadPrimitiveDataProperties
        using (XmlReader reader = XmlDictionaryReader.CreateBinaryReader(memoryStream,
    XmlDictionaryReaderQuotas.Max))
        {
            // invoke the helper method
            ReadPrimitiveDataProperties(reader, propertyBag);
        }
    return propertyBag;
}

return null;
}

// reads the primitive data properties from the XML stream
// invoked by the ReadDataProperties method
static void ReadPrimitiveDataProperties(XmlReader reader, Dictionary<XName, object> propertyBag)
{
    const string xmlElementName = "Property";

    if (reader.ReadToDescendant(xmlElementName))
    {
        do
        {
            // get the name of the property
            reader.MoveToFirstAttribute();
            string propertyName = reader.Value;

            // get the type of the property
            reader.MoveToNextAttribute();
            PrimitiveType type = (PrimitiveType)Int32.Parse(reader.Value,
CultureInfo.InvariantCulture);

            // get the value of the property
            reader.MoveToNextAttribute();
            object PropertyValue = ConvertStringToNativeType(reader.Value, type);

            // add the name and value of the property to the property bag
            propertyBag.Add(propertyName, PropertyValue);
        }

        while (reader.ReadToNextSibling(xmlElementName));
    }
}

// invoked by the ReadPrimitiveDataProperties method
// Given a property value as parsed from an XML attribute, and the .NET Type of the Property,
// recreates the actual property value
// (e.g. Given a property value of "1" and a PrimitiveType of Int32, this method will return an object
// of type Int32 with value 1)
static object ConvertStringToNativeType(string value, PrimitiveType type)
{
    switch (type)
    {
        case PrimitiveType.Bool:
            return XmlConvert.ToBoolean(value);
        case PrimitiveType.Byte:
            return XmlConvert.ToByte(value);
        case PrimitiveType.Char:
            return XmlConvert.ToChar(value);
        case PrimitiveType.DateTime:
            return XmlConvert.ToDateTime(value, XmlDateTimeSerializationMode.RoundtripKind);
        case PrimitiveType.DateTimeOffset:
            return XmlConvert.ToDateTimeOffset(value);
        case PrimitiveType.Decimal:
            return XmlConvert.ToDecimal(value);
        case PrimitiveType.Double:
            return XmlConvert.ToDouble(value);
    }
}

```

```

        return XmlConvert.ToDouble(value);
    case PrimitiveType.Float:
        return float.Parse(value, CultureInfo.InvariantCulture);
    case PrimitiveType.Guid:
        return XmlConvert.ToGuid(value);
    case PrimitiveType.Int:
        return XmlConvert.ToInt32(value);
    case PrimitiveType.Long:
        return XmlConvert.ToInt64(value);
    case PrimitiveType.SByte:
        return XmlConvert.ToSByte(value);
    case PrimitiveType.Short:
        return XmlConvert.ToInt16(value);
    case PrimitiveType.String:
        return value;
    case PrimitiveType.TimeSpan:
        return XmlConvert.ToDateTime(value);
    case PrimitiveType.Type:
        return Type.GetType(value);
    case PrimitiveType.UInt:
        return XmlConvert.ToInt32(value);
    case PrimitiveType.ULong:
        return XmlConvert.ToInt64(value);
    case PrimitiveType.Uri:
        return new Uri(value);
    case PrimitiveType.UShort:
        return XmlConvert.ToInt16(value);
    case PrimitiveType.XmlQualifiedName:
        return new XmlQualifiedName(value);
    case PrimitiveType.Null:
    case PrimitiveType.Unavailable:
        default:
            return null;
    }
}
// .NET Types which SQL Workflow Instance Store considers to be Primitive. Any other .NET type not
// listed in this enumeration is a "Complex" property.
enum PrimitiveType
{
    Bool = 0,
    Byte,
    Char,
    DateTime,
    DateTimeOffset,
    Decimal,
    Double,
    Float,
    Guid,
    Int,
    Null,
    Long,
    SByte,
    Short,
    String,
    TimeSpan,
    Type,
    UInt,
    ULong,
    Uri,
    UShort,
    XmlQualifiedName,
    Unavailable = 99
}
}
}

```

# How to: Query for Non-persisted Instances

5/4/2018 • 3 minutes to read • [Edit Online](#)

When a new instance of a service is created and the service has the SQL Workflow Instance Store behavior defined, the service host creates a initial entry for that service instance in the instance store. Subsequently when the service instance persists for the first time, the SQL Workflow Instance Store behavior stores the current instance state together with additional data that is required for activation, recovery, and control.

If an instance is not persisted after the initial entry for the instance is created, the service instance is said to be in the non-persisted state. All the persisted service instances can be queried and controlled. Non-persisted service instances can neither be queried nor controlled. If a non-persisted instance is suspended due to an unhandled exception it can be queried but not controlled.

Durable service instances that are not yet persisted remain in a non-persisted state in the following scenarios:

- The service host crashes before the instance persisted for the first time. The initial entry for the instance remains in the instance store. The instance is not recoverable. If a correlated message arrives, the instance becomes active again.
- The instance experiences an unhandled exception before it persisted for the first time. The following scenarios arise
  - If the value of the **UnhandledExceptionAction** property is set to **Abandon**, the service deployment information is written to the instance store and the instance is unloaded from memory. The instance remains in non-persisted state in the persistence database.
  - If the value of the **UnhandledExceptionAction** property is set to **AbandonAndSuspend**, the service deployment information is written to the persistence database and the instance state is set to **Suspended**. The instance cannot be resumed, canceled, or terminated. The service host cannot load the instance because the instance hasn't persisted yet and, hence the database entry for the instance is not complete.
  - If the value of the **UnhandledExceptionAction** property is set to **Cancel** or **Terminate**, the service deployment information is written to the instance store and the instance state is set to **Completed**.

The following sections provide sample queries to find non-persisted instances in the SQL persistence database and to delete these instances from the database.

## To find all instances not persisted yet

The following SQL query returns the ID and creation time for all instances that are not persisted in to the persistence database yet.

```
select InstanceId, CreationTime from [System.Activities.DurableInstancing].[Instances] where IsInitialized = 0;
```

## To find all instances not persisted yet and also not loaded

The following SQL query returns ID and creation time for all instances that are not persisted and also are not loaded.

```
select InstanceId, CreationTime from [System.Activities.DurableInstancing].[Instances] where IsInitialized = 0  
and CurrentMachine is NULL;
```

## To find all suspended instances not persisted yet

The following SQL query returns ID, creation time, suspension reason, and suspension exception name for all instances that are not persisted and also in a suspended state.

```
select InstanceId, CreationTime, SuspensionReason, SuspensionExceptionName from  
[System.Activities.DurableInstancing].[Instances] where IsInitialized = 0 and IsSuspended = 1;
```

## To delete non-persisted instances from the persistence database

You should periodically check the instance store for non-persisted instances and remove instances from the instance store if you are sure that the instance will not receive a correlated message. For example, if the instance has been in the database for several months and you know that the workflow typically has a lifetime of a few days, it would be safe to assume that this is an uninitialized instance that had crashed.

In general, it is safe to delete non-persisted instances that are not suspended or not loaded. You should not delete **all** the non-persisted instances because this instance set includes instances that are just created but are not persisted yet. You should only delete non-persisted instances that are left over because the workflow service host that had the instance loaded caused an exception or the instance itself caused an exception.

### WARNING

Deleting non-persisted instances from the instance store decreases the size of the store and may improve the performance of store operations.

# Instance Stores

8/31/2018 • 2 minutes to read • [Edit Online](#)

An instance store is a logical container of instances. It is the place where the instance data and metadata is stored. An instance store does not imply dedicated physical storage. An instance store could contain durable information in a SQL Server database or non-durable state information in memory. The .NET Framework 4.6.1 ships with the SQL Workflow Instance Store, which is a concrete implementation of an instance store that allows workflows to persist instance data and metadata into a SQL Server 2005 or SQL Server 2008 database. In addition Windows Server App Fabric also provides a concrete implementation of an instance store. For more information, see [Windows Server App Fabric Instance Store, Query, and Control Providers](#).

The persistence API is the interface between a host and an instance store that allows the host to send command requests (for example, [LoadWorkflowCommand](#) and [SaveWorkflowCommand](#)) to the instance store. The concrete implementation of this API is called a persistence provider. The persistence provider receives requests from a host and modifies the instance store.

Hosts and instance stores are pluggable so that a host can be used with many instance stores, and an instance store can be used with many hosts. An instance store is typically optimized for the usage patterns of a particular host, although the instance store and host may evolve on independent life cycles. For example, the **WorkflowServiceHost** and the **SqlWorkflowInstanceStore** are designed to work well together. You can create your own instance store to persist data and metadata of workflow service instances and use that instance store with the **WorkflowServiceHost**. For example, you can create an OracleWorkflowInstanceStore that lets workflows persist information into an Oracle database instead of saving them into a SQL Server database.

It is common for hosts to be extended with additional functionality that modifies the persisted objects. For example, an instance persistence system may have a workflow host, an extension that supports the "Suspend" operation, and an SQL instance store. The workflow host might send a standard command such as Save or Load to save or load a workflow from an instance store or to save a workflow into an instance store. The suspend extension might add additional semantics to the commands for saving and loading workflow instances so that a suspended workflow instance cannot be loaded. The persistence provider for the SQL instance store understands the commands for saving and loading workflow instances, and implements the commands by calling appropriate stored procedures that change the tables of persistent objects in an SQL Server database.

A host acts as an instance owner within an instance store. A host may act as more than one instance owner with more than one instance store at the same time. The host provides GUIDs for instance keys associated with the instances. An instance key is a unique alias that identifies an instance. The persistence system creates, updates, and deletes instance owner information as it executes commands requested by hosts.

The following list contains the important steps involved in the host's interaction with the instance store:

1. Obtain an **InstanceStore** from a persistence provider.
2. Obtain the handle to an instance by calling the [CreateInstanceHandle](#) method on the **InstanceStore**.
3. Invoke commands against the instance handle by calling the [Execute](#) method on the **InstanceStore**.
4. Examine the [InstanceView](#) returned by **InstanceStore.Execute** to determine the results of the commands.

# How to: Enable Persistence for Workflows and Workflow Services

5/4/2018 • 2 minutes to read • [Edit Online](#)

This topic describes how to enable persistence for workflows and workflow services.

## Enable Persistence for Workflows

You can associate an instance store with a **WorkflowApplication** by using the **InstanceStore** property of the **WorkflowApplication** class. The **Persist** method saves or persists a workflow into the instance store associated with the application. The **Unload** method persists a workflow into the instance store and then unloads the instance from the memory. The **Load** method loads a workflow into memory using the workflow data stored in the instance persistence store.

The **Persist** method performs the following steps:

1. Pauses the workflow scheduler and waits until the workflow enters the idle state.
2. Persists or saves the workflow into the persistence store.
3. Resumes the workflow scheduler.

The **Unload** method performs the following steps:

1. Pauses the workflow scheduler and waits until the workflow enters the idle state.
2. Persists or saves the workflow into the persistence store.
3. Disposes the workflow instance in the memory.

Both the **Persist** and **Unload** methods will block while a workflow is in a no-persist zone until the workflow exits the no-persist zone. The method continues with the persist or unload operation after the no-persist zone completes. If the no-persist zone does not complete before the time-out elapses, or if the persistence process takes too long, a **TimeoutException** will be thrown.

## Enable Persistence for Workflow Services in Code

The **DurableInstancingOptions** member of the **WorkflowServiceHost** class has a property named **InstanceStore** that you can use to associate an instance store with the **WorkflowServiceHost**.

```
// wsh is an instance of WorkflowServiceHost class  
wsh.DurableInstancingOptions.InstanceStore = new SqlWorkflowInstanceStore();
```

When the **WorkflowServiceHost** is opened, persistence is automatically enabled if the **DurableInstancingOptions.InstanceStore** is not null.

Typically, a service behavior provides the concrete instance store to be used with a workflow service host by using the **InstanceStore** property. For example, the **SqlWorkflowInstanceStoreBehavior** creates an instance of the **SqlWorkflowInstanceStore**, configures it, and assigns it to the **DurableInstancingOptions.InstanceStore**.

## Enable Persistence for Workflow Services Using an Application

## Configuration File

Persistence can be enabled using an application configuration file by adding the following code to your app.config or web.config file:

```
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="myBehavior">
          <SqlWorkflowInstanceStore connectionString="Data Source=myDatabaseServer;Initial Catalog=myPersistenceDatabase">
          </behavior>
        </serviceBehaviors>
      <behaviors>
    </system.serviceModel>
</configuration>
```

# How to: Create a Custom Instance Store

3/9/2019 • 7 minutes to read • [Edit Online](#)

.NET Framework 4.6.1 contains [SqlWorkflowInstanceStore](#), an instance store that uses SQL Server to persist workflow data. If your application is required to persist workflow data to another medium, such as a different database or a file system, you can implement a custom instance store. A custom instance store is created by extending the abstract [InstanceStore](#) class and implementing the methods that are required for the implementation. For a complete implementation of a custom instance store, see the [Corporate Purchase Process](#) sample.

## Implementing the BeginTryCommand method

The [BeginTryCommand](#) is sent to the instance store by the persistence engine. The type of the `command` parameter indicates which command is being executed; this parameter can be of the following types:

- [SaveWorkflowCommand](#): The persistence engine sends this command to the instance store when a workflow is to be persisted to the storage medium. The workflow persistence data is provided to the method in the [InstanceData](#) member of the `command` parameter.
- [LoadWorkflowCommand](#): The persistence engine sends this command to the instance store when a workflow is to be loaded from the storage medium. The instance ID of the workflow to be loaded is provided to the method in the `instanceId` parameter of the [InstanceView](#) property of the `context` parameter.
- [CreateWorkflowOwnerCommand](#): The persistence engine sends this command to the instance store when a [WorkflowServiceHost](#) must be registered as a lock owner. The instance ID of the current workflow should be provided to the instance store using [BindInstanceOwner](#) method of the `context` parameter.

The following code snippet demonstrates how to implement the [CreateWorkflowOwner](#) command to assign an explicit lock owner.

```
XName WFInstanceScopeName = XName.Get(scopeName, "<namespace>");  
...  
CreateWorkflowOwnerCommand createCommand = new CreateWorkflowOwnerCommand()  
{  
    InstanceOwnerMetadata =  
    {  
        { WorkflowHostName, new InstanceValue(WFInstanceScopeName) },  
    }  
};  
InstanceHandle ownerHandle = store.CreateInstanceHandle();  
store.DefaultInstanceOwner = store.Execute(  
    ownerHandle,  
    createCommand,  
    TimeSpan.FromSeconds(30)).InstanceOwner;  
childInstance.AddInitialInstanceValues(new Dictionary<XName, object>() { { WorkflowHostName,  
WFInstanceScopeName } });
```

- [DeleteWorkflowOwnerCommand](#): The persistence engine sends this command to the instance store when the instance ID of a lock owner can be removed from the instance store. As with [CreateWorkflowOwnerCommand](#), the ID of the lock owner should be provided by the application.

The following code snippet demonstrates how to release a lock using [DeleteWorkflowOwnerCommand](#).

```

static void FreeHandleAndDeleteOwner(InstanceStore store, InstanceHandle handle)
{
    handle.Free();
    handle = store.CreateInstanceHandle(store.DefaultInstanceOwner);
    try
    {
        // We need this sleep so that we dont prematurely delete the owner, we need time to update the
        database.
        Thread.Sleep(10000);
        store.Execute(handle, new DeleteWorkflowOwnerCommand(), TimeSpan.FromSeconds(10));
    }
    catch (InstancePersistenceCommandException ex) { Log.Inform(ex.Message); }
    catch (InstanceOwnerException ex) { Log.Inform(ex.Message); }
    catch (OperationCanceledException ex) { Log.Inform(ex.Message); }
    catch (Exception ex) { Log.Inform(ex.Message); }
    finally
    {
        handle.Free();
        store.DefaultInstanceOwner = null;
    }
}

```

The above method should be called in a Try/Catch block when a child instance is run.

```

try
{
    childInstance.Run();
}
catch (Exception)
{
    throw ;
}
finally
{
    FreeHandleAndDeleteOwner(store, ownerHandle);
}

```

- [LoadWorkflowByInstanceKeyCommand](#): The persistence engine sends this command to the instance store when a workflow instance is to be loaded using the workflow's instance key. The ID of the instance key can be determined by using the [LookupInstanceKey](#) parameter of the command.
- [QueryActivatableWorkflowsCommand](#): The persistence engine sends this command to the instance store to retrieve activation parameters for persisted workflows in order to create a workflow host that can then load workflows. This command is sent by the engine in response to the instance store raising the [HasActivatableWorkflowEvent](#) to the host when it locates an instance that can be activated. The instance store should be polled to determine if there are workflows that can be activated.
- [TryLoadRunnableWorkflowCommand](#): The persistence engine sends this command to the instance store to load runnable workflow instances. This command is sent by the engine in response to the instance store raising the [HasRunnableWorkflowEvent](#) to the host when it locates an instance that can be run. The instance store should poll for workflows that can be run. The following code snippet demonstrates polling an instance store for workflows that can be run or activated.

```

public void PollForEvents()
{
    InstanceOwner[] storeOwners = this.GetInstanceOwners();

    foreach (InstanceOwner owner in storeOwners)
    {
        foreach (InstancePersistenceEvent ppEvent in this.GetEvents(owner))
        {
            if (ppEvent.Equals(HasRunnableWorkflowEvent.Value))
            {
                bool hasRunnable = GetRunnableEvents(this.StoreId, owner.InstanceOwnerId,
isActivatable);
                if (hasRunnable)
                {
                    this.SignalEvent(ppEvent, owner);
                }
                else
                {
                    this.ResetEvent(ppEvent, owner);
                }
            }
            else if(ppEvent.Equals(HasActivatableWorkflowEvent.Value))
            {
                bool hasActivatable = GetActivatableEvents(this.StoreId, owner.InstanceOwnerId);
                if (hasActivatable)
                {
                    this.SignalEvent(ppEvent, owner);
                }
                else
                {
                    this.ResetEvent(ppEvent, owner);
                }
            }
        }
    }
}

```

In the above code snippet, the instance store queries the events available and examines each one to determine if it is a [HasRunnableWorkflowEvent](#) event. If one is found, [SignalEvent](#) is called to signal the host to send a command to the instance store. The following code snippet demonstrates an implementation of a handler for this command.

```

If (command is TryLoadRunnableWorkflowCommand)
{
    Owner owner;
    CheckOwner(context, command.Name, out owner);
    // Checking instance.Owner is like an InstanceLockQueryResult.
    context.QueriedInstanceStore(new InstanceLockQueryResult(context.InstanceView.InstanceId,
context.InstanceView.InstanceOwner.InstanceOwnerId));

    XName ownerService = null;
    InstanceValue value;
    Instance runnableInstance = default(Instance);
    bool foundRunnableInstance = false;

    value = QueryPropertyBag(WorkflowNamespace.WorkflowHostType, owner.Data);
    if (value != null && value.Value is XName)
    {
        ownerService = (XName)value.Value;
    }

    foreach (KeyValuePair<Guid, Instance> instance in MemoryStore.instances)
    {
        if (instance.Value.Owner != Guid.Empty || instance.Value.Completed)
        {

```

```

        continue;
    }

    if (ownerService != null)
    {
        value = QueryPropertyBag(WorkflowNamespace.WorkflowHostType, instance.Value.Metadata);
        if (value == null || ((XName)value.Value) != ownerService)
        {
            continue;
        }
    }

    value = QueryPropertyBag(WorkflowServiceNamespace.SuspendReason, instance.Value.Metadata);
    if (value != null && value.Value != null && value.Value is string)
    {
        continue;
    }

    value = QueryPropertyBag(WorkflowNamespace.Status, instance.Value.Data);
    if (value != null && value.Value is string && ((string)value.Value) == "Executing")
    {
        runnableInstance = instance.Value;
        foundRunnableInstance = true;
    }

    if (!foundRunnableInstance)
    {
        value = QueryPropertyBag(XNamespace.Get("urn:schemas-microsoft-
com:System.Activities/4.0/properties").GetName("TimerExpirationTime"), instance.Value.Data);
        if (value != null && value.Value is DateTime && ((DateTime)value.Value) <= DateTime.UtcNow)
        {
            runnableInstance = instance.Value;
            foundRunnableInstance = true;
        }
    }

    if (foundRunnableInstance)
    {
        runnableInstance.LockVersion++;
        runnableInstance.Owner = context.InstanceView.InstanceOwner.InstanceOwnerId;
        MemoryStore.instances[instance.Key] = runnableInstance;
        context.BindInstance(instance.Key);
        context.BindAcquiredLock(runnableInstance.LockVersion);

        Dictionary<Guid, IDictionary<XName, InstanceValue>> associatedKeys = new Dictionary<Guid,
IDictionary<XName, InstanceValue>>();
        Dictionary<Guid, IDictionary<XName, InstanceValue>> completedKeys = new Dictionary<Guid,
IDictionary<XName, InstanceValue>>();
        foreach (KeyValuePair<Guid, Key> keyEntry in MemoryStore.keys)
        {
            if (keyEntry.Value.Instance == context.InstanceView.InstanceId)
            {
                if (keyEntry.Value.Completed)
                {
                    completedKeys.Add(keyEntry.Key, DeserializePropertyBag(keyEntry.Value.Metadata));
                }
                else
                {
                    associatedKeys.Add(keyEntry.Key, DeserializePropertyBag(keyEntry.Value.Metadata));
                }
            }
        }
    }

    context.LoadedInstance(InstanceState.Initialized, DeserializePropertyBag(runnableInstance.Data),
DeserializePropertyBag(runnableInstance.Metadata), associatedKeys, completedKeys);
    break;
}
}

```

---

In the above code snippet, the instance store searches for runnable instances. If an instance is found, it is bound to the execution context and loaded.

## Using a custom instance store

To implement a custom instance store, assign an instance of the instance store to the `InstanceStore`, and implement the `PersistableIdle` method. See the [How to: Create and Run a Long Running Workflow](#) tutorial for specifics.

## A sample instance store

The following code sample is a complete instance store implementation, taken from the [Corporate Purchase Process](#) sample. This instance store persists workflow data to a file using XML.

```
using System;
using System.Activities.DurableInstancing;
using System.Collections.Generic;
using System.IO;
using System.Runtime.DurableInstancing;
using System.Runtime.Serialization;
using System.ServiceModel.Persistence;
using System.Xml;
using System.Xml.Linq;

namespace Microsoft.Samples.WF.PurchaseProcess
{

    public class XmlWorkflowInstanceStore : InstanceStore
    {
        Guid ownerInstanceID;

        public XmlWorkflowInstanceStore() : this(Guid.NewGuid())
        {

        }

        public XmlWorkflowInstanceStore(Guid id)
        {
            ownerInstanceID = id;
        }

        //Synchronous version of the Begin/EndTryCommand functions
        protected override bool TryCommand(InstancePersistenceContext context, InstancePersistenceCommand command, TimeSpan timeout)
        {
            return EndTryCommand(BeginTryCommand(context, command, timeout, null, null));
        }

        //The persistence engine will send a variety of commands to the configured InstanceStore,
        //such as CreateWorkflowOwnerCommand, SaveWorkflowCommand, and LoadWorkflowCommand.
        //This method is where we will handle those commands
        protected override IAsyncResult BeginTryCommand(InstancePersistenceContext context,
        InstancePersistenceCommand command, TimeSpan timeout, AsyncCallback callback, object state)
        {
            IDictionary<XName, InstanceValue> data = null;

            //The CreateWorkflowOwner command instructs the instance store to create a new instance owner bound
            //to the instance handle
            if (command is CreateWorkflowOwnerCommand)
            {
                context.BindInstanceOwner(ownerInstanceID, Guid.NewGuid());
            }
            //The SaveWorkflow command instructs the instance store to modify the instance bound to the
            //instance handle or an instance key
            else if (command is SaveWorkflowCommand)
```

```

        {
            SaveWorkflowCommand saveCommand = (SaveWorkflowCommand)command;
            data = saveCommand.InstanceData;

            Save(data);
        }
        //The LoadWorkflow command instructs the instance store to lock and load the instance bound to the
        identifier in the instance handle
        else if (command is LoadWorkflowCommand)
        {
            string fileName = IOHelper.GetFileName(this.ownerInstanceID);

            try
            {
                using (FileStream inputStream = new FileStream(fileName, FileMode.Open))
                {
                    data = LoadInstanceDataFromFile(inputStream);
                    //load the data into the persistence Context
                    context.LoadedInstance(InstanceState.Initialized, data, null, null, null);
                }
            }
            catch (Exception exception)
            {
                throw new PersistenceException(exception.Message);
            }
        }

        return new CompletedAsyncResult<bool>(true, callback, state);
    }

    protected override bool EndTryCommand(IAsyncResult result)
    {
        return CompletedAsyncResult<bool>.End(result);
    }

    //Reads data from xml file and creates a dictionary based off of that.
    IDictionary<XName, InstanceValue> LoadInstanceDataFromStream(Stream inputStream)
    {
        IDictionary<XName, InstanceValue> data = new Dictionary<XName, InstanceValue>();

        NetDataContractSerializer s = new NetDataContractSerializer();

        XmlReader rdr = XmlReader.Create(inputStream);
        XmlDocument doc = new XmlDocument();
        doc.Load(rdr);

        XmlNodeList instances = doc.GetElementsByTagName("InstanceValue");
        foreach (XmlElement instanceElement in instances)
        {
            XElement keyElement = (XmlElement)instanceElement.SelectSingleNode("descendant::key");
            XName key = (XName)DeserializeObject(s, keyElement);

            XElement valueElement = (XmlElement)instanceElement.SelectSingleNode("descendant::value");
            object value = DeserializeObject(s, valueElement);
            InstanceValue instVal = new InstanceValue(value);

            data.Add(key, instVal);
        }
    }

    return data;
}

object DeserializeObject(NetDataContractSerializer serializer, XElement element)
{
    object serializedObject = null;

    MemoryStream stm = new MemoryStream();
    XmlDictionaryWriter wtr = XmlDictionaryWriter.CreateTextWriter(stm);
    element.WriteContentTo(wtr);
}

```

```

        wtr.Flush();
        stm.Position = 0;

        deserializedObject = serializer.Deserialize(stm);

        return deserializedObject;
    }

    //Saves the persistence data to an xml file.
    void Save(IDictionary<XName, InstanceValue> instanceData)
    {
        string fileName = IOHelper.GetFileName(this.ownerInstanceID);
        XmlDocument doc = new XmlDocument();
        doc.LoadXml("<InstanceValues/>");

        foreach (KeyValuePair<XName, InstanceValue> valPair in instanceData)
        {
            XmlElement newInstance = doc.CreateElement("InstanceValue");

            XmlElement newKey = SerializeObject("key", valPair.Key, doc);
            newInstance.AppendChild(newKey);

            XmlElement newValue = SerializeObject("value", valPair.Value.Value, doc);
            newInstance.AppendChild(newValue);

            doc.DocumentElement.AppendChild(newInstance);
        }
        doc.Save(fileName);
    }

    XElement SerializeObject(string elementName, object o, XmlDocument doc)
    {
        NetDataContractSerializer s = new NetDataContractSerializer();
        XElement newElement = doc.CreateElement(elementName);
        MemoryStream stm = new MemoryStream();

        s.Serialize(stm, o);
        stm.Position = 0;
        StreamReader rdr = new StreamReader(stm);
        newElement.InnerXml = rdr.ReadToEnd();

        return newElement;
    }
}
}

```

# Persistence Participants

3/9/2019 • 3 minutes to read • [Edit Online](#)

A persistence participant can participate in a persistence operation (Save or Load) triggered by an application host. The .NET Framework 4.6.1 ships with two abstract classes, **PersistenceParticipant** and **PersistenceIOParticipant**, which you can use to create a persistence participant. A persistence participant derives from one of these classes, implements the methods of interest, and then adds an instance of the class to the [WorkflowExtensions](#) collection on the [WorkflowServiceHost](#). The application host may look for such workflow extensions when persisting a workflow instance and invoke appropriate methods on the persistence participants at appropriate times.

The following list describes the tasks performed by the persistence subsystem in different stages of the Persist (Save) operation. The persistence participants are used in the third and fourth stages. If the participant is an I/O participant (a persistence participant that also participates in I/O operations), the participant is also used in the sixth stage.

1. Gathers built-in values, including workflow state, bookmarks, mapped variables, and timestamp.
2. Gathers all persistence participants that were added to the extension collection associated with the workflow instance.
3. Invokes the [CollectValues](#) method implemented by all persistence participants.
4. Invokes the [MapValues](#) method implemented by all persistence participants.
5. Persist or save the workflow into the persistence store.
6. Invokes the [BeginOnSave](#) method on all of the persistence I/O participants. If the participant is not an I/O participant, this task is skipped. If the persistence episode is transactional, the transaction is provided in `Transaction.Current` property.
7. Waits for all persistence participants to complete. If all the participants succeed in persisting instance data, commits the transaction.

A persistence participant derives from the **PersistenceParticipant** class and may implement the [CollectValues](#) and [MapValues](#) methods. A persistence I/O participant derives from the **PersistenceIOParticipant** class and may implement the [BeginOnSave](#) method in addition to implementing the [CollectValues](#) and [MapValues](#) methods.

Each stage is completed before the next stage begins. For example, values are collected from **all** persistence participants in the first stage. Then all the values collected in the first stage are provided to all persistence participants in the second stage for mapping. Then all the values collected and mapped in the first and second stages are provided to the persistence provider in the third stage, and so on.

The following list describes the tasks performed by the persistence subsystem in different stages of the Load operation. The persistence participants are used in the fourth stage. The persistence I/O participants (persistence participants that also participate in I/O operations) are also used in the third stage.

1. Gathers all persistence participants that were added to the extension collection associated with the workflow instance.
2. Loads the workflow from the persistence store.
3. Invokes the [BeginOnLoad](#) on all persistence I/O participants and waits for all the persistence participants

to complete. If the persistence episode is transactional, the transaction is provided in `Transaction.Current`.

4. Loads the workflow instance in memory based on the data retrieved from the persistence store.

5. Invokes `PublishValues` on each persistence participant.

A persistence participant derives from the **PersistenceParticipant** class and may implement the **PublishValues** method. A persistence I/O participant derives from the **PersistenceIOParticipant** class and may implement the **BeginOnLoad** method in addition to implementing the **PublishValues** method.

When loading a workflow instance the persistence provider creates a lock on that instance. This prevents the instance from being loaded by more than one host in a multi-node scenario. If you attempt to load a workflow instance that has been locked you will see an exception like the following: The exception "

`System.ServiceModel.Persistence.InstanceLockException: The requested operation could not complete because the lock for instance '00000000-0000-0000-0000-000000000000' could not be acquired`". This error is caused when one of the following occurs:

- In a multi-node scenario the instance is loaded by another host. There are a few different ways to resolve these types of conflicts: forward the processing to the node which owns the lock and retry, or force the load which will cause the other host to be unable to save their work.
- In a single-node scenario and the host crashed. When the host starts up again (a process recycle or creating a new persistence provider factory) the new host attempts to load an instance which is still locked by the old host because the lock hasn't expired yet.
- In a single-node scenario and the instance in question was aborted at some point and a new persistence provider instance is created which has a different host ID.

The lock timeout value has a default value of 5 minutes, you can specify a different timeout value when calling `Load`.

## In This Section

- [How to: Create a Custom Persistence Participant](#)

## See also

- [Store Extensibility](#)

# How to: Create a Custom Persistence Participant

3/9/2019 • 2 minutes to read • [Edit Online](#)

The following procedure has steps to create a persistence participant. See the [Participating in Persistence](#) sample and [Store Extensibility](#) topic for sample implementations of persistence participants.

1. Create a class deriving from the [PersistenceParticipant](#) or the [PersistenceOParticipant](#) class. The [PersistenceOParticipant](#) class offers the same extensibility points as the [PersistenceParticipant](#) class in addition to being able to participate in I/O operations. Follow one or more of the following steps.
2. Implement the [CollectValues](#) method. The **CollectValues** method has two dictionary parameters, one for storing read/write values and the other one for storing write-only values (used later in queries). In this method, you should populate these dictionaries with data that is specific to a persistence participant. Each dictionary contains the name of the value as the key and the value itself as an [InstanceValue](#) object.

The values in the `readWriteValues` dictionary are packaged as **InstanceValue** objects. The values in the write-only dictionary are packaged as **InstanceValue** objects with `InstanceValueOptions.Optional` and `InstanceValueOption.WriteOnly` set. Each **InstanceValue** provided by the **CollectValues** implementations across all persistence participants must have a unique name.

```
protected virtual void CollectValues (out IDictionary<XName,Object> readWriteValues, out  
IDictionary<XName,Object> writeOnlyValues)
```

3. Implement the [MapValues](#) method. The **MapValues** method takes two parameters that are similar to the parameters that the **CollectValues** method receives. All the values collected in the **CollectValues** stage are passed through these dictionary parameters. The new values added by the **MapValues** stage are added to the write-only values. The write-only dictionary is used to provide data to an external source not directly associated with the instance values. Each value provided by implementations of the **MapValues** method across all persistence participants must have a unique name.

```
protected virtual IDictionary<XName,Object> MapValues (IDictionary<XName,Object>  
readWriteValues, IDictionary<XName,Object> writeOnlyValues)
```

The [MapValues](#) method provides functionality that [CollectValues](#) does not, in that it allows for a dependency on another value provided by another persistence participant that hasn't been processed by [CollectValues](#) yet.

4. Implement the [PublishValues](#) method. The **PublishValues** method receives a dictionary containing all the values loaded from the persistence store.

```
protected virtual void PublishValues (IDictionary<XName,Object> readWriteValues)
```

5. Implement the **BeginOnSave** method if the participant is a persistence I/O participant. This method is called during a Save operation. In this method, you should perform I/O adjunct to the persisting (saving) workflow instances. If the host is using a transaction for the corresponding persistence command, the same transaction is provided in `Transaction.Current`. Additionally, [PersistenceOParticipants](#) may advertise a transactional consistency requirement, in which case the host creates a transaction for the persistence episode if one would not otherwise be used.

```
protected virtual IAsyncResult BeginOnSave (IDictionary<XName,Object> readWriteValues,  
IDictionary<XName,Object> writeOnlyValues, TimeSpan timeout, AsyncCallback callback, Object state)
```

6. Implement the **BeginOnLoad** method if the participant is a persistence I/O participant. This method is called during a Load operation. In this method, you should perform I/O adjunct to the loading of workflow instances. If the host is using a transaction for the corresponding persistence command, the same transaction is provided in Transaction.Current. Additionally, Persistence I/O participants may advertise a transactional consistency requirement, in which case the host creates a transaction for the persistence episode if one would not otherwise be used.

```
protected virtual IAsyncResult BeginOnLoad (IDictionary<XName,Object> readWriteValues, TimeSpan timeout,  
AsyncCallback callback, Object state)
```

# Persistence Best Practices

8/31/2018 • 5 minutes to read • [Edit Online](#)

This document covers best practices for workflow design and configuration related to workflow persistence.

## Design and Implementation of Durable Workflows

In general, workflows perform work in short periods that are interleaved with times during which the workflow is idle because it is waiting for an event. This event can be such things as a message or an expiring timer. To be able to unload the workflow instance when it becomes idle, the service host must persist the workflow instance. This is possible only if the workflow instance is not in a no-persist zone (for example, waiting for a transaction to complete, or waiting for an asynchronous callback). To allow an idle workflow instance to unload, the workflow author should use transaction scopes and asynchronous activities for short-lived actions only. In particular, the author should keep delay activities within these no-persist zones as short as possible.

A workflow can only be persisted if all of the data types used by the workflow are serializable. In addition, custom types used in persisted workflows must be serializable with [NetDataContractSerializer](#) in order to be persisted by [SqlWorkflowInstanceStateStore](#).

A workflow instance cannot be recovered in case of a host or computer failure if it has not been persisted. In general, we recommend that you persist a workflow instance early in the workflow's life cycle.

If your workflow is busy for a long time, we recommend that you persist the workflow instance regularly throughout its busy period. You can do this by adding [Persist](#) activities throughout the sequence of activities that keep the workflow instance busy. In this manner, application domain recycling, host failures, or computer failures do not cause the system to be rolled back to the start of the busy period. Be aware that adding [Persist](#) activities to your workflow could lead to a degradation of performance.

Windows Server App Fabric greatly simplifies the configuration and use of persistence. For more information, see [Windows Server App Fabric Persistence](#)

## Configuration of Scalability Parameters

Scalability and performance requirements determine the settings of the following parameters:

- [TimeToPersist](#)
- [TimeToUnload](#)
- [InstanceLockedExceptionAction](#)

These parameters should be set as follows, according to the current scenario.

### Scenario: A Small Number of Workflow Instances That Require Optimal Response Time

In this scenario, all workflow instances should remain loaded when they become idle. Set [TimeToUnload](#) to a large value. The use of this setting prevents a workflow instance from moving between computers. Use this setting only if one or more of the following are true:

- A workflow instance receives a single message throughout its lifetime.
- All workflow instances run on a single computer
- All messages that are received by a workflow instance are received by the same computer.

Use [Persist](#) activities or set [TimeToPersist](#) to 0 to enable recovery of your workflow instance after service host or computer failures.

### Scenario: Workflow Instances Are Idle for Long Periods of Time

In this scenario, set [TimeToUnload](#) to 0 to release resources as soon as possible.

### Scenario: Workflow Instances Receive Multiple Messages in a Short Period of Time

In this scenario, set [TimeToUnload](#) to 60 seconds if these messages are received by the same computer. This prevents a rapid sequence of unloading and loading of a workflow instance. This also does not keep the instance in memory for too long.

Set [TimeToUnload](#) to 0, and set [InstanceLockedExceptionAction](#) to BasicRetry or AggressiveRetry if these messages may be received by different computers. This allows the workflow instance to be loaded by another computer.

### Scenario: Workflow Uses Delay Activities with Short Durations

In this scenario, the [SqlWorkflowInstanceStateStore](#) regularly polls the persistence database for instances that should be loaded because of an expired [Delay](#) activity. If the [SqlWorkflowInstanceStateStore](#) finds a timer that will expire in the next polling interval, the SQL Workflow Instance Store shortens the polling interval. The next poll will then occur right after the timer has expired. This way, the SQL Workflow Instance Store achieves a high accuracy of timers that run longer than the polling interval, which is set by [RunnableInstancesDetectionPeriod](#). To enable timely processing of shorter delays, the workflow instance must remain in memory for at least one polling interval.

Set [TimeToPersist](#) to 0 to write the expiration time to the persistence database.

Set [TimeToUnload](#) to longer than or equal to [RunnableInstancesDetectionPeriod](#) to keep the instance in memory for at least one polling interval.

We do not recommend reducing the [RunnableInstancesDetectionPeriod](#) because this leads to an increased load on the persistence database. Each service host that uses the [SqlWorkflowInstanceStateStore](#) polls the database one time per detection period. Setting [RunnableInstancesDetectionPeriod](#) to too small a time interval may cause your system's performance to decrease if the number of service hosts is large.

## Configuring the SQL Workflow Instance Store

The SQL Workflow Instance Store has the following configuration parameters:

#### [InstanceEncodingOption](#)

This parameter instructs the [SqlWorkflowInstanceStateStore](#) to compress the workflow instance state. Compression reduces the amount of data that is stored in the persistence database and reduces network traffic in case the persistence database resides on a dedicated database server. If compression is used, it requires computational resources to compress and extract the instance state. In most cases, compression yields increased performance.

#### [InstanceCompletionAction](#)

This parameter instructs the [SqlWorkflowInstanceStateStore](#) to either keep or delete completed instances. Keeping completed instances increases the persistence database storage requirements and leads to larger tables, which increases table lookup times. Unless completed instances are required for debugging or auditing, it is best to instruct the [SqlWorkflowInstanceStateStore](#) to delete completed instances. Deleted instances should be kept only if the user establishes a process for eventually removing them. Note that correlation keys cannot be reused as long as the completed workflow instance resides in the instance store.

#### [RunnableInstancesDetectionPeriod](#)

This parameter defines the maximum interval with which the [SqlWorkflowInstanceStateStore](#) polls the persistence database for instances that should be loaded when a [Delay](#) activity expires. If the [SqlWorkflowInstanceStateStore](#) finds a timer that will expire in the next polling interval, it shortens the polling interval so that the next poll will occur right after the timer has expired. This way, the SQL Workflow Instance Store achieves a high accuracy of timers that run

longer than [RunnableInstancesDetectionPeriod](#).

We do not recommend reducing the [RunnableInstancesDetectionPeriod](#), because this leads to an increased load on the persistence database. Each service host that uses the [SqlWorkflowInstanceStore](#) polls the database one time per detection period. Setting [RunnableInstancesDetectionPeriod](#) to too small an interval may cause your system's performance to decrease if the number of service hosts is large.

#### [HostLockRenewalPeriod](#)

This parameter defines the interval with which the host renews its lock in the persistence database. Shortening this interval will enable a quicker recovery of the workflow instances in case a host or computer fails. On the other hand, a short lock renewal period increases the load on the persistence database. Each service host that uses the [SqlWorkflowInstanceStore](#) will update its locks in the database one time per renewal period. If a computer runs many service hosts, make sure that the load caused by lock renewal does not decrease your system's performance. If it does, consider increasing the [HostLockRenewalPeriod](#).

#### [InstanceLockedExceptionAction](#)

If enabled, the [SqlWorkflowInstanceStore](#) retries to load a locked instance for the next 30 seconds. Set [InstanceLockedExceptionAction](#) to BasicRetry or AggressiveRetry if the workflow receives multiple messages in a short time, and these messages are received by different computers.

Because the load retry mechanism does not introduce any performance overhead as long as load retries are not tried, [InstanceLockedExceptionAction](#) should always be enabled.

# Non-Persisted Workflow Instances

5/4/2018 • 2 minutes to read • [Edit Online](#)

When a new instance of a workflow is created that persists its state in the [SqlWorkflowInstanceStateStore](#), the service host creates an entry for that service in the instance store. Subsequently, when the workflow instance is persisted for the first time, the [SqlWorkflowInstanceStateStore](#) stores the current instance state. If the workflow is hosted in the Windows Process Activation Service, the service deployment data is also written to the instance store when the instance is persisted for the first time.

As long as the workflow instance has not been persisted, it is in a **non-persisted** state. While in this state, the workflow instance cannot be recovered after an application domain recycle, host failure, or computer failure.

## The Non-Persisted State

Durable workflow instances that have not been persisted remain in a non-persisted state in the following cases:

- The service host crashes before the workflow instance is persisted for the first time. The workflow instance remains in the instance store and is not recovered. If a correlated message arrives, the workflow instance becomes active again.
- The workflow instance experiences an exception before it is persisted for the first time. Depending on the [UnhandledExceptionAction](#) returned, the following scenarios occur:
  - [UnhandledExceptionAction](#) is set to [Abort](#): When an exception occurs, service deployment information is written to the instance store, and the workflow instance is unloaded from memory. The workflow instance remains in a non-persisted state and cannot be reloaded.
  - [UnhandledExceptionAction](#) is set to [Cancel](#) or [Terminate](#): When an exception occurs, service deployment information is written to the instance store, and the activity instance state is set to [Closed](#).

To minimize the risk of encountering unloaded non-persisted workflow instances, we recommend persisting the workflow early in its life cycle.

## Detection and Removal of Non-Persisted Instances

The [SqlWorkflowInstanceStateStore](#) does not remove any non-persisted workflow instances from the instance store. It also does not remove any expired lock owners that have non-persisted workflow instances associated with them.

We recommend that the administrator periodically checks the instance store for non-persisted instances. Administrators can remove those instances from the instance store as long as they know that this workflow will not receive correlated messages. For example, if the instance has been in the database for several months and it is known that the workflow typically has a lifetime of several days, it would be safe to assume that this was an initialized instance that had crashed.

To find non-persisted instances in the SQL Workflow Instance Store you can use the following SQL queries:

- This query finds all instances that have not been persisted, and returns the ID and creation time (stored in UTC time) for them.

```
select InstanceId, CreationTime  
  from [System.Activities.DurableInstancing].[Instances]  
 where IsInitialized = 0
```

- This query finds all instances that have not been persisted, and that are not loaded, and returns the ID and creation time (stored in UTC time) for them.

```
select InstanceId, CreationTime  
  from [System.Activities.DurableInstancing].[Instances]  
 where IsInitialized = 0  
   and CurrentMachine is NULL
```

- This query finds all suspended instances that have not been persisted, and returns the ID, creation time (stored in UTC time), suspension reason, and exception name for them.

```
select InstanceId, CreationTime, SuspensionReason, SuspensionExceptionName  
  from [System.Activities.DurableInstancing].[Instances]  
 where IsInitialized = 0  
   and IsSuspended = 1
```

Use care when you are deleting non-persisted instances. In general, it is safe to remove non-persisted instances created by [WorkflowServiceHost](#) that are suspended or are not loaded. Those specific instances can be deleted from the store by deleting them from the `[System.Activities.DurableInstancing].[Instances]` view by using the following SQL command, substituting the correct instance ID.

```
delete [System.Activities.DurableInstancing].[Instances]  
 where InstanceId='078a9bc4-ada5-4f9e-8cce-b0eb0009995f'
```

#### WARNING

We do not recommend removing all non-persisted instances because this includes instances that have just been created and have not yet been persisted.

# Pausing and Resuming a Workflow

5/4/2018 • 2 minutes to read • [Edit Online](#)

Workflows will pause and resume in response to bookmarks and blocking activities such as [Delay](#), but a workflow can also be explicitly paused, unloaded, and resumed by using persistence.

## Pausing a Workflow

To pause a workflow, use [Unload](#). This method requests that the workflow persist and unload, and will throw a [TimeoutException](#) if the workflow does not unload in 30 seconds.

```
try
{
    // attempt to unload will fail if the workflow is idle within a NoPersistZone
    application.Unload(TimeSpan.FromSeconds(5));
}
catch (TimeoutException e)
{
    Console.WriteLine(e.Message);
}
```

## Resuming a Workflow

To resume a previously paused and unloaded workflow, use [Load](#). This method loads a workflow from a persistence store into memory.

```
WorkflowApplication application = new WorkflowApplication(activity);
application.InstanceStore = instanceStore;
application.Load(id);
```

## Example

The following code sample demonstrates how to pause and resume a workflow by using persistence.

```
static string bkName = "bkName";
static void Main(string[] args)
{
    StartAndUnloadInstance();
}

static void StartAndUnloadInstance()
{
    AutoResetEvent waitHandler = new AutoResetEvent(false);
    WorkflowApplication wfApp = new WorkflowApplication(GetDelayedWF());
    SqlWorkflowInstanceStore instanceStore = SetupSqlPersistenceStore();
    wfApp.InstanceStore = instanceStore;
    wfApp.Extensions.Add(SetupMyFileTrackingParticipant);
    wfApp.PersistableIdle = (e) => {           //persists application state and remove it from memory
        return PersistableIdleAction.Unload;
    };
    wfApp.Unloaded = (e) => {
        waitHandler.Set();
    };
    Guid id = wfApp.Id;
    wfApp.Run();
```

```

wfApp.Run();
waitHandler.WaitOne();
LoadAndCompleteInstance(id);
}

static void LoadAndCompleteInstance(Guid id)
{
    Console.WriteLine("Press <enter> to load the persisted workflow");
    Console.ReadLine();
    AutoResetEvent waitHandler = new AutoResetEvent(false);
    WorkflowApplication wfApp = new WorkflowApplication(new Workflow1());
    wfApp.InstanceStore =
        new
    SqlWorkflowInstanceStore(ConfigurationManager.AppSettings["SqlWF4PersistenceConnectionString"].ToString());
    wfApp.Completed = (workflowApplicationCompletedEventArgs) => {
        Console.WriteLine("\nWorkflowApplication has Completed in the {0} state.",
            workflowApplicationCompletedEventArgs.CompletionState);
    };
    wfApp.Unloaded = (workflowApplicationEventArgs) => {
        Console.WriteLine("WorkflowApplication has Unloaded\n");
        waitHandler.Set();
    };
    wfApp.Load(id);
    wfApp.Run();
    waitHandler.WaitOne();
}

public static Activity GetDelayedWF()
{
    return new Sequence {
        Activities ={ 
            new WriteLine{Text="Workflow Started"}, 
            new Delay{Duration=TimeSpan.FromSeconds(10)}, 
            new WriteLine{Text="Workflow Ended"} 
        }
    };
}

private static SqlWorkflowInstanceStore SetupSqlPersistenceStore()
{
    string connectionString =
    ConfigurationManager.AppSettings["SqlWF4PersistenceConnectionString"].ToString();
    SqlWorkflowInstanceStore sqlWFIInstanceStore = new SqlWorkflowInstanceStore(connectionString);
    sqlWFIInstanceStore.InstanceCompletionAction = InstanceCompletionAction.DeleteAll;
    InstanceHandle handle = sqlWFIInstanceStore.CreateInstanceHandle();
    InstanceView view = sqlWFIInstanceStore.Execute(handle, new CreateWorkflowOwnerCommand(),
    TimeSpan.FromSeconds(5));
    handle.Free();
    sqlWFIInstanceStore.DefaultInstanceOwner = view.InstanceOwner;
    return sqlWFIInstanceStore;
}

```

# Windows Workflow Foundation (WF) Migration Guidance

9/26/2018 • 2 minutes to read • [Edit Online](#)

This section contains topics that deal with migrating workflows from previous versions of Windows Workflow Foundation (WF).

## [Migration Guidance](#)

Contains a list of links to whitepapers on migration topics.

## [Using .NET Framework 3.0 WF Activities in .NET Framework 4 with the Interop Activity](#)

Includes information on how to use the Interop activity to embed .NET Framework 3.5 activities within a .NET Framework 4.6.1 workflow.

# Migration Guidance

3/9/2019 • 2 minutes to read • [Edit Online](#)

In the .NET Framework 4, Microsoft is releasing the second major version of Windows Workflow Foundation (WF). WF was released in WinFX (this included the types in the System.Workflow.\* namespaces; now referred to as WF3) and enhanced in .NET Framework 3.5. WF3 is also part of the .NET Framework 4, but it exists there alongside new workflow technology (the types in the System.Activities.\* namespaces; referred to as WF4). When considering when to adopt WF4, it is important to first recognize that you control the timing.

- WF3 is a fully supported part of the .NET Framework 4.
- WF3 applications run on the .NET Framework 4 without modification and continue to be fully supported.
- New WF3 applications can be created and your existing applications can be edited in Visual Studio 2012 and are fully supported.

Thus, the decision to adopt the .NET Framework 4 is decoupled from your decision to move to WF4 (System.Activities.\*) from WF3 (System.Workflow.\*). This topic provides links to WF migration guidance that provides information about working with WF3 and WF4.

## WF Migration Whitepapers and Cookbooks

The [WF Migration Overview](#) topic provides a broad overview of the relationship between WF3 and WF4 and migration strategies. Companion topics drill into specific topics.

### [WF Migration Overview](#)

Describes the relationship between WF3 and WF4, and the choices you have as a user or a potential user of workflow technology in .NET 4.

### [WF Migration: Best Practices for WF3 Development](#)

Discusses how to design WF3 artifacts so they can be more easily migrated to WF4.

### [WF Guidance: Rules](#)

Discusses how to bring rules-related investments forward into .NET Framework 4 solutions.

### [WF Guidance: State Machine](#)

Discusses WF4 control flow modeling in the absence of a State-Machine activity.

Note that this guidance only applies to workflow projects that target .NET Framework 4. State Machine workflows were added in .NET 4.0.1 with the release of Platform Update 1, and were included as part of .NET Framework 4.5. For more information about state machine workflows in .NET 4.0.1 - 4.0.3 and .NET Framework 4.5, see [Update 4.0.1 for Microsoft .NET Framework 4 Features](#) and [State Machine Workflows](#).

### [WF Migration Cookbook: Custom Activities](#)

Provides examples and instructions for redesigning WF3 custom activities on WF4.

### [WF Migration Cookbook: Advanced Custom Activities](#)

Provides guidance for redesigning advanced WF3 custom activities that use WF3 queues and schedule child activities as WF4 custom activities.

### [WF Migration Cookbook: Workflows](#)

Provides examples and instructions for redesigning WF3 workflows on WF4.

### [WF Migration Cookbook: Workflow Hosting](#)

Provides guidance for redesigning WF3 hosting code as WF4 hosting code. The goal is to cover the key differences in workflow hosting between WF3 and WF4.

#### [WF Migration Cookbook: Workflow Tracking](#)

Provides guidance for redesigning WF3 tracking code and configuration using equivalent WF4 tracking code and configuration.

#### [WF Guidance: Workflow Services](#)

Provides example-oriented step-by-step instructions for redesigning workflows that implement Windows Communication Foundation (WCF) web services (commonly referred to as workflow services) created in WF3 to use WF4, for common scenarios for out-of-box activities.

## See also

- [Interop](#)

# Using .NET Framework 3.0 WF Activities in .NET Framework 4 with the Interop Activity

10/1/2018 • 3 minutes to read • [Edit Online](#)

The [Interop](#) activity is a .NET Framework 4.6.1 (WF 4.5) activity that wraps a .NET Framework 3.5 (WF 3.5) activity within a .NET Framework 4.6.1 workflow. The WF 3 activity can be a single leaf activity or an entire tree of activities. The execution (including cancellation and exception handling) and the persistence of the .NET Framework 3.5 activity occur within the context of the .NET Framework 4.6.1 workflow instance that is executing.

## NOTE

The [Interop](#) activity does not appear in the workflow designer toolbox unless the workflow's project has its **Target Framework** setting set to **.NET Framework 4.5**.

## Criteria for Using a WF 3 Activity with an Interop Activity

For a WF 3 activity to successfully execute within an [Interop](#) activity, the following criteria must be met:

- The WF 3 activity must derive from [System.Workflow.ComponentModel.Activity](#).
- The WF 3 activity must be declared as `public` and cannot be `abstract`.
- The WF 3 activity must have a public default constructor.
- Due to limitations in the interface types that the [Interop](#) activity can support, [HandleExternalEventActivity](#) and [CallExternalMethodActivity](#) cannot be used directly, but derivative activities created using the Workflow Communication Activity tool (WCA.exe) can be used. See [Windows Workflow Foundation Tools](#) for details.

## Configuring a WF 3 Activity Within an Interop Activity

To configure and pass data into and out of a WF 3 activity, across the interoperation boundary, the WF 3 activity's properties and metadata properties are exposed by the [Interop](#) activity. The WF 3 activity's metadata properties (such as [Name](#)) are exposed through the [ActivityMetaProperties](#) collection. This is a collection of name-value pairs used to define the values for the WF 3 activity's metadata properties. A metadata property is a property backed by dependency property for which the [Metadata](#) flag is set.

The WF 3 activity's properties are exposed through the [ActivityProperties](#) collection. This is a set of name-value pairs, where each value is a [Argument](#) object, used to define the arguments for the WF 3 activity's properties. Because the direction of a WF 3 activity property cannot be inferred, every property is surfaced as an [InArgument](#)/[OutArgument](#) pair. Depending on the activity's usage of the property, you may want to provide an [InArgument](#) entry, an [OutArgument](#) entry, or both. The expected name of the [InArgument](#) entry in the collection is the name of the property as defined on the WF 3 activity. The expected name of the [OutArgument](#) entry in the collection is a concatenation of the name of the property and the string "Out".

## Limitations of Using a WF 3 Activity Within an Interop Activity

The WF 3 system-provided activities cannot be directly wrapped in an [Interop](#) activity. For some WF 3 activities, such as [DelayActivity](#), this is because there is an analogous WF 4.5 activity. For others, this is because the functionality of the activity is not supported. Many WF 3 system-provided activities can be used within workflows wrapped by the [Interop](#) activity, subject to the following restrictions:

1. [Send](#) and [Receive](#) cannot be used in an [Interop](#) activity.
2. [WebServiceInputActivity](#), [WebServiceOutputActivity](#), and [WebServiceFaultActivity](#) cannot be used within an [Interop](#) activity.
3. [InvokeWorkflowActivity](#) cannot be used within an [Interop](#) activity.
4. [SuspendActivity](#) cannot be used within an [Interop](#) activity.
5. Compensation-related activities cannot be used within an [Interop](#) activity.

There are also some behavioral specifics to understand regarding the use of WF 3 activities within the [Interop](#) activity:

1. WF 3 activities contained within an [Interop](#) activity are initialized when the [Interop](#) activity is executed. In WF 4.5 there is no initialization phase for a workflow instance prior to its execution.
2. The WF 4.5 runtime does not checkpoint workflow instance state when a transaction begins, regardless of where that transaction begins (within or outside of an [Interop](#) activity).
3. WF 3 tracking records for activities within an [Interop](#) activity are provided to WF 4.5 tracking participants as [InteropTrackingRecord](#) objects. [InteropTrackingRecord](#) is a derivative of [CustomTrackingRecord](#).
4. A WF 3 custom activity can access data using workflow queues within the interoperation environment in exactly the same way as within the WF 3 workflow runtime. No custom activity code changes are required. On the host, data is enqueued to a WF 3 workflow queue by resuming a [Bookmark](#). The name of the bookmark is the string form of the [IComparable](#) workflow queue name.

# Workflow Tracking and Tracing

3/26/2019 • 2 minutes to read • [Edit Online](#)

Windows Workflow tracking is a .NET Framework 4.6.1 feature designed to provide visibility into workflow execution. It provides a tracking infrastructure to track the execution of a workflow instance. The WF tracking infrastructure transparently instruments a workflow to emit records reflecting key events during the execution. This functionality is available by default for any .NET Framework 4.6.1 workflow. No changes are required to be made to a .NET Framework 4.6.1 workflow for tracking to occur. It is just a matter of deciding how much tracking data you want to receive. When a workflow instance starts or completes, its processing tracking records are emitted. Tracking can also extract business-relevant data associated with the workflow variables. For example, if the workflow represents an order processing system, the order ID can be extracted along with the [TrackingRecord](#) object. In general, enabling WF tracking facilitates diagnostics or business analytics data to be accessed from a workflow execution.

These tracking components are equivalent to the tracking service in WinFX. In .NET Framework 4.6.1, the performance has been improved and the programming model simplified for the WF tracking feature. The tracking runtime instruments a workflow instance to emit events related to the workflow life cycle, workflow activities and custom events.

Windows Server App Fabric also provides the ability to monitor the execution of a WCF and workflow services. For more information, see [Windows Server App Fabric Monitoring](#) and [Monitoring Applications with Windows Server AppFabric](#)

To troubleshoot the workflow runtime, you can turn on diagnostic workflow tracing. For more information, see [Workflow Tracing](#).

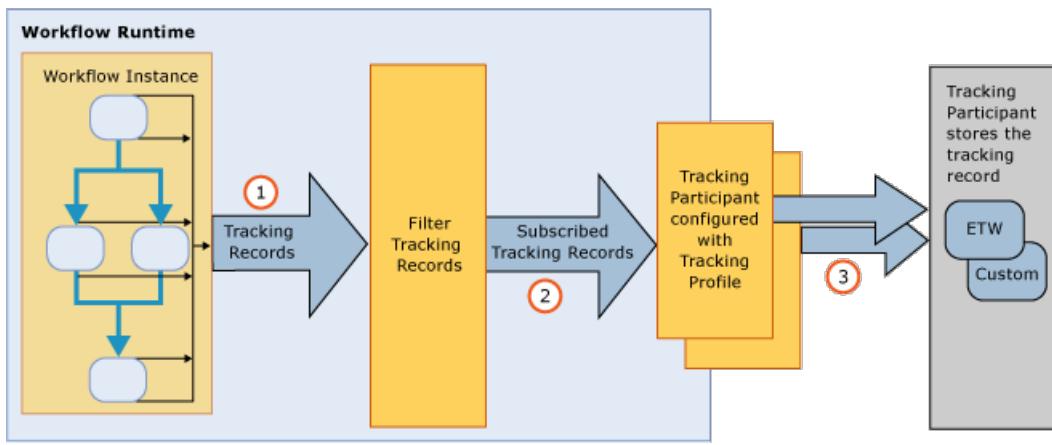
To understand the programming model, the primary components of the tracking infrastructure are discussed in this topic:

- [TrackingRecord](#) objects emitted from the workflow runtime. For more information, see [Tracking Records](#).
- [TrackingParticipant](#) objects subscribe to [TrackingRecord](#) objects. The tracking participants contain the logic to process the payload from the [TrackingRecord](#) objects (for example, they could choose to write to a file). For more information, see [Tracking Participants](#).
- [TrackingProfile](#) objects filter tracking records emitted from a workflow instance. For more information, see [Tracking Profiles](#).

## Workflow Tracking Infrastructure

The workflow tracking infrastructure follows a publish-and-subscribe paradigm. The workflow instance is the publisher of tracking records, while subscribers of the tracking records are registered as extensions to the workflow. These extensions that subscribe to [TrackingRecord](#) objects are called tracking participants. Tracking participants are extensibility points that access [TrackingRecord](#) objects and process them in whatever manner they are written to do so. The tracking infrastructure allows the application of a filter on the outgoing tracking records to allow a participant to subscribe to a subset of the records. This filtering mechanism is accomplished through a tracking profile file.

A high level view of the tracking infrastructure is shown in the following illustration:



## In This Section

### [Tracking Records](#)

Describes the tracking records that the workflow runtime emits.

### [Tracking Profiles](#)

Discusses how tracking profiles are used.

### [Tracking Participants](#)

Describes how to use system-provided tracking participant or how to create custom tracking participants.

### [Configuring Tracking for a Workflow](#)

Describes how to configure tracking for a workflow.

### [Workflow Tracing](#)

Describes the two ways to enable debug tracing for a workflow.

## See also

- [SQL Tracking](#)

# Tracking Records

1/23/2019 • 3 minutes to read • [Edit Online](#)

The workflow runtime is instrumented to emit tracking records to follow the execution of a workflow instance.

## Tracking Records

The following table details the tracking records that the workflow runtime emits.

| TRACKING RECORD             | DESCRIPTION   |
|-----------------------------|---|
| Workflow life cycle records | Emitted during various stages of the life cycle of the workflow instance. For example, a record is emitted when the workflow starts or completes.                               |
| Activity life cycle records | Details activity execution. These records indicate the state of a workflow activity such as when an activity is scheduled, when the activity completes, or when a fault occurs. |
| Bookmark resumption records | Emitted whenever a bookmark within a workflow instance is resumed.  |
| Custom tracking records     | A workflow author can create custom tracking records and emit them within a custom activity.  |

All tracking-related records emitted from the WF runtime derive from the base class [TrackingRecord](#), which contains the common set of data. Tracking records show the life cycle for a simple workflow. Each tracking record contains details about the associated tracking event, such as the [InstanceId](#), [RecordNumber](#), and additional information specific to the type of tracking record.

The following types of [TrackingRecord](#) objects are emitted by the workflow runtime:

- **WorkflowInstanceRecord** - This [TrackingRecord](#) describes the life cycle of the workflow instance. For example, a record is emitted when the workflow starts or completes, and contains the state of the workflow instance. The details of this record can be found at [WorkflowInstanceRecord](#).
- **WorkflowInstanceAbortedRecord** - This [TrackingRecord](#) is emitted when a workflow instance aborts. The record contains the reason for the workflow instance being aborted. The details of this record can be found at [WorkflowInstanceAbortedRecord](#).
- **WorkflowInstanceUnhandledExceptionRecord** - This [TrackingRecord](#) is emitted if an exception occurs in the workflow instance and is not handled by any activity. The record contains the exception details. The details of this record can be found at [WorkflowInstanceUnhandledExceptionRecord](#).
- **WorkflowInstanceSuspendedRecord** - This [TrackingRecord](#) is emitted whenever a workflow instance is suspended. The record contains the reason for the workflow instance being suspended. The details of this record can be found at [WorkflowInstanceSuspendedRecord](#).
- **WorkflowInstanceTerminatedRecord** - This [TrackingRecord](#) is emitted whenever a workflow instance is terminated. The record contains the reason for the workflow instance being terminated. The details of this record can be found at [WorkflowInstanceTerminatedRecord](#).
- **ActivityStateRecord** - This [TrackingRecord](#) is emitted when an activity within a workflow executes. These

records indicate the state of the activity within the workflow instance. The details of this record can be found at [ActivityStateRecord](#).

- **ActivityScheduledRecord** - This [TrackingRecord](#) is emitted when an activity schedules a child activity. This record contains details for both the parent activity (scheduling activity) and the scheduled child activity. The details of this record can be found at [ActivityScheduledRecord](#).
- **FaultPropagationRecord** - This [TrackingRecord](#) is emitted for each handler that looks at the record until it is handled. It is used to denote the path a fault took within the workflow instance. The details of this record can be found at [FaultPropagationRecord](#).
- **CancelRequestedRecord** - This [TrackingRecord](#) is emitted whenever an activity tries to cancel a child activity. This record contains details for both the parent activity and the child activity that is being canceled. The details of this record can be found at [CancelRequestedRecord](#).
- **BookmarkResumptionRecord** - This [TrackingRecord](#) tracks any bookmark that is successfully resumed. The details of this record can be found at [BookmarkResumptionRecord](#).
- **CustomTrackingRecord** - This [TrackingRecord](#) is created and emitted by a workflow author within a custom workflow activity. Custom tracking records can be populated with data to be emitted along with the records. The details of this record can be found at [CustomTrackingRecord](#).

For example, there could be a simple [Sequence](#) activity that contains a [WriteLine](#) operation with tracking records emitted in the following order:

1. [WorkflowInstanceRecord](#) indicates that the workflow is starting.
2. [ActivityScheduledRecord](#) indicates that an activity has been scheduled. In this case it is a [Sequence](#) activity.
3. [ActivityScheduledRecord](#) represents the [WriteLine](#) activity.
4. There are two [ActivityStateRecord](#) records that represent the two activities completing.
5. [WorkflowInstanceRecord](#) indicates that the workflow is completing.

## See also

- [Windows Server App Fabric Monitoring](#)
- [Monitoring Applications with App Fabric](#)

# Tracking Profiles

3/9/2019 • 8 minutes to read • [Edit Online](#)

Tracking profiles contain tracking queries that permit a tracking participant to subscribe to workflow events that are emitted when the state of a workflow instance changes at runtime.

## Tracking Profiles

Tracking profiles are used to specify which tracking information is emitted for a workflow instance. If no profile is specified, then all tracking events are emitted. If a profile is specified, then the tracking events specified in the profile will be emitted. Depending on your monitoring requirements, you may write a profile that is very general, which subscribes to a small set of high-level state changes on a workflow. Conversely, you may create a very detailed profile whose resulting events are rich enough to reconstruct a detailed execution flow later.

Tracking profiles manifest themselves as XML elements within a standard .NET Framework configuration file or specified in code. The following example is of a .NET Framework 4.6.1 tracking profile in a configuration file that allows a tracking participant to subscribe to the `Started` and `Completed` workflow events.

```
<system.serviceModel>
    ...
    <tracking>
        <trackingProfile name="Sample Tracking Profile">
            <workflow activityDefinitionId="*">
                <workflowInstanceQueries>
                    <workflowInstanceQuery>
                        <states>
                            <state name="Started"/>
                            <state name="Completed"/>
                        </states>
                    </workflowInstanceQuery>
                </workflowInstanceQueries>
            </workflow>
        </trackingProfile>
    </profiles>
</tracking>
    ...
</system.serviceModel>
```

The following example shows the equivalent tracking profile created using code.

```
TrackingProfile profile = new TrackingProfile()
{
    Name = "CustomTrackingProfile",
    Queries =
    {
        new WorkflowInstanceQuery()
        {
            // Limit workflow instance tracking records for started and
            // completed workflow states.
            States = { WorkflowInstanceState.Started, WorkflowInstanceState.Completed },
        }
    }
};
```

Tracking records are filtered through the visibility mode within a tracking profile by using the

[ImplementationVisibility](#) attribute. A composite activity is a top-level activity that contains other activities that form its implementation. The visibility mode specifies the tracking records emitted from composite activities within a workflow activity, to specify if activities that form the implementation are being tracked. The visibility mode applies at the tracking profile level. The filtering of tracking records for individual activities within a workflow is controlled by the queries within the tracking profile. For more information, see the **Tracking Profile Query Types** section in this document.

The two visibility modes specified by the `implementationVisibility` attribute in the tracking profile are `RootScope` and `All`. Using the `RootScope` mode suppresses the tracking records for activities that form the implementation of an activity in the case where a composite activity is not the root of a workflow. This implies that, when an activity that is implemented using other activities is added to a workflow, and the `implementationVisibility` set to `RootScope`, only the top-level activity within that composite activity is tracked. If an activity is the root of the workflow, then the implementation of the activity is the workflow itself and tracking records are emitted for activities that form the implementation. Using the All mode permits all tracking records to be emitted for the root activity and all its composite activities.

For example, suppose *MyActivity* is a composite activity whose implementation contains two activities, *Activity1* and *Activity2*. When this activity is added to a workflow and tracking is enabled with a tracking profile with `implementationVisibility` set to `RootScope`, tracking records are emitted only for *MyActivity*. However, no records are emitted for activities *Activity1* and *Activity2*.

However, if the `implementationVisibility` attribute for the tracking profile is set to `All`, then tracking records are emitted not only for *MyActivity*, but also for activities *Activity1* and *Activity2*.

The `implementationVisibility` flag applies to following tracking record types:

- `ActivityStateRecord`
- `FaultPropagationRecord`
- `CancelRequestedRecord`
- `ActivityScheduledRecord`

#### NOTE

CustomTrackingRecords emitted from activity implementation are not filtered out by the `implementationVisibility` setting.

The `implementationVisibility` functionality is specified as `RootScope` on the tracking profile in code as follows:

```
TrackingProfile sampleTrackingProfile = new TrackingProfile()
{
    Name = "Sample Tracking Profile",
    ImplementationVisibility = ImplementationVisibility.RootScope
};
```

The `implementationVisibility` functionality is specified as `All` on the tracking profile in a configuration file as follows:

```

<tracking>
    <profiles>
        <trackingProfile name="Shipping Monitoring" implementationVisibility="All">
            <workflow activityDefinitionId="*"/>
        ...
        </workflow>
    </trackingProfile>
</profiles>
</tracking>

```

The `ImplementationVisibility` setting on the tracking profile is optional. By default, its value is set to `RootScope`. The values for this attribute are also case-sensitive.

### Tracking Profile Query Types

Tracking profiles are structured as declarative subscriptions for tracking records that allow you to query the workflow runtime for specific tracking records. There are several query types that allow you subscribe to different classes of `TrackingRecord` objects. Tracking profiles can be specified in configuration or through code. Here are the most common query types:

- `WorkflowInstanceQuery` - Use this to track workflow instance life cycle changes like the previously-demonstrated `Started` and `Completed`. The `WorkflowInstanceQuery` is used to subscribe to the following `TrackingRecord` objects:
  - `WorkflowInstanceRecord`
  - `WorkflowInstanceAbortedRecord`
  - `WorkflowInstanceUnhandledExceptionRecord`
  - `WorkflowInstanceTerminatedRecord`
  - `WorkflowInstanceSuspendedRecord`

The states that can be subscribed to are specified in the `WorkflowInstanceState` class.

The configuration or code used to subscribe to workflow instance-level tracking records for the `Started` instance state using the `WorkflowInstanceQuery` is shown in the following example.

```

<workflowInstanceQueries>
    <workflowInstanceQuery>
        <states>
            <state name="Started"/>
        </states>
    </workflowInstanceQuery>
</workflowInstanceQueries>

```

```

TrackingProfile sampleTrackingProfile = new TrackingProfile()
{
    Name = "Sample Tracking Profile",
    Queries =
    {
        new WorkflowInstanceQuery()
        {
            States = { WorkflowInstanceState.Started}
        }
    }
};

```

- `ActivityStateQuery` - Use this to track life cycle changes of the activities that make up a workflow instance.

For example, you may want to keep track of every time the "Send E-Mail" activity completes within a workflow instance. This query is necessary for a [TrackingParticipant](#) to subscribe to [ActivityStateRecord](#) objects. The available states to subscribe to are specified in [ActivityStates](#).

The configuration and code used to subscribe activity state tracking records that use the [ActivityStateQuery](#) for the `SendEmailActivity` activity is shown in the following example.

```
<activityStateQueries>
    <activityStateQuery activityName="SendEmailActivity">
        <states>
            <state name="Closed"/>
        </states>
    </activityStateQuery>
</activityStateQueries>
```

```
TrackingProfile sampleTrackingProfile = new TrackingProfile()
{
    Name = "Sample Tracking Profile",
    Queries =
    {
        new ActivityStateQuery()
        {
            ActivityName = "SendEmailActivity",
            States = { ActivityStates.Closed }
        }
    }
};
```

#### NOTE

If multiple `activityStateQuery` elements have the same name, only the states in the last element are used in the tracking profile.

- [ActivityScheduledQuery](#) - This query allows you to track an activity scheduled for execution by a parent activity. The query is necessary for a [TrackingParticipant](#) to subscribe to [ActivityScheduledRecord](#) objects.

The configuration and code used to subscribe to records related to the `SendEmailActivity` child activity being scheduled using the [ActivityScheduledQuery](#) is shown in the following example.

```
<activityScheduledQueries>
    <activityScheduledQuery activityName="ProcessNotificationsActivity"
        childActivityName="SendEmailActivity" />
</activityScheduledQueries>
```

```
TrackingProfile sampleTrackingProfile = new TrackingProfile()
{
    Name = "Sample Tracking Profile",
    Queries =
    {
        new ActivityScheduledQuery()
        {
            ActivityName = "ProcessNotificationsActivity",
            ChildActivityName = "SendEmailActivity"
        }
    }
};
```

- [FaultPropagationQuery](#) - Use this to track the handling of faults that occur within an activity. The query is necessary for a [TrackingParticipant](#) to subscribe to [FaultPropagationRecord](#) objects.

The configuration and code used to subscribe to records related to fault propagation using [FaultPropagationQuery](#) is shown in the following example.

```
<faultPropagationQueries>
  <faultPropagationQuery faultSourceActivityName="SendEmailActivity"
    faultHandlerActivityName="NotificationsFaultHandler" />
</faultPropagationQueries>
```

```
TrackingProfile sampleTrackingProfile = new TrackingProfile()
{
    Name = "Sample Tracking Profile",
    Queries =
    {
        new FaultPropagationQuery()
        {
            FaultSourceActivityName = "SendEmailActivity",
            FaultHandlerActivityName = "NotificationsFaultHandler"
        }
    }
};
```

- [CancelRequestedQuery](#) - Use this to track requests to cancel a child activity by the parent activity. The query is necessary for a [TrackingParticipant](#) to subscribe to [CancelRequestedRecord](#) objects.

The configuration and code used to subscribe to records related to activity cancellation using [CancelRequestedQuery](#) is shown in the following example.

```
<cancelRequestedQueries>
  <cancelRequestedQuery activityName="ProcessNotificationsActivity"
    childActivityName="SendEmailActivity" />
</cancelRequestedQueries>
```

```
TrackingProfile sampleTrackingProfile = new TrackingProfile()
{
    Name = "Sample Tracking Profile",
    Queries =
    {
        new CancelRequestedQuery()
        {
            ActivityName = "ProcessNotificationsActivity",
            ChildActivityName = "SendEmailActivity"
        }
    }
};
```

- [CustomTrackingQuery](#) - Use this to track events that you define in your code activities. The query is necessary for a [TrackingParticipant](#) to subscribe to [CustomTrackingRecord](#) objects.

The configuration and code used to subscribe to records related to custom tracking records using [CustomTrackingQuery](#) is shown in the following example.

```
<customTrackingQueries>
  <customTrackingQuery name="EmailAddress" activityName="SendEmailActivity" />
</customTrackingQueries>
```

```

TrackingProfile sampleTrackingProfile = new TrackingProfile()
{
    Name = "Sample Tracking Profile",
    Queries =
    {
        new CustomTrackingQuery()
        {
            Name = "EmailAddress",
            ActivityName = "SendEmailActivity"
        }
    }
};

```

- [BookmarkResumptionQuery](#) - Use this to track resumption of a bookmark within a workflow instance. This query is necessary for a [TrackingParticipant](#) to subscribe to [BookmarkResumptionRecord](#) objects.

The configuration and code used to subscribe to records related to bookmark resumption using [BookmarkResumptionQuery](#) is shown in the following example.

```

<bookmarkResumptionQueries>
    <bookmarkResumptionQuery name="SentEmailBookmark" />
</bookmarkResumptionQueries>

```

```

TrackingProfile sampleTrackingProfile = new TrackingProfile()
{
    Name = "Sample Tracking Profile",
    Queries =
    {
        new BookmarkResumptionQuery()
        {
            Name = "sentEmailBookmark"
        }
    }
};

```

## Annotations

Annotations allow you to arbitrarily tag tracking records with a value that can be configured after build time. For example, you might want several tracking records across several workflows to be tagged with "Mail Server" == "Mail Server1". This makes it easy to find all records with this tag when querying tracking records later.

To accomplish this, an annotation is added to a tracking query as shown in the following example.

```

<activityStateQuery activityName="SendEmailActivity">
    <states>
        <state name="Closed"/>
    </states>
    <annotations>
        <annotation name="MailServer" value="Mail Server1"/>
    </annotations>
</activityStateQuery>

```

## How to Create a Tracking Profile

Tracking query elements are used to create a tracking profile using either an XML configuration file or .NET Framework 4.6.1 code. Here is an example of a tracking profile created using a configuration file.

```

<system.serviceModel>
  <tracking>
    <profiles>
      <trackingProfile name="Sample Tracking Profile ">
        <workflow activityDefinitionId="*"
          <!--Specify the tracking profile query elements to subscribe for tracking records-->
        </workflow>
      </trackingProfile>
    </profiles>
  </tracking>
</system.serviceModel>

```

### **WARNING**

For a WF using the Workflow service host, the tracking profile is typically created using a configuration file. It is also possible to create a tracking profile with code using the tracking profile and tracking query API.

A profile configured as an XML configuration file is applied to a tracking participant using a behavior extension. This is added to a `WorkflowServiceHost` as described in the later section [Configuring Tracking for a Workflow](#).

The verbosity of the tracking records emitted by the host is determined by configuration settings within the tracking profile. A tracking participant subscribes to tracking records by adding queries to a tracking profile. To subscribe to all tracking records, the tracking profile needs to specify all tracking queries using "\*" in the name fields in each of the queries.

Here are some of the common examples of tracking profiles.

- A tracking profile to obtain workflow instance records and faults.

```

<trackingProfile name="Instance and Fault Records">
  <workflow activityDefinitionId="*"
    <workflowInstanceQueries>
      <workflowInstanceQuery>
        <states>
          <state name="*" />
        </states>
      </workflowInstanceQuery>
    </workflowInstanceQueries>
    <activityStateQueries>
      <activityStateQuery activityName="*"
        <states>
          <state name="Faulted"/>
        </states>
      </activityStateQuery>
    </activityStateQueries>
  </workflow>
</trackingProfile>

```

1. A tracking profile to obtain all custom tracking records.

```

<trackingProfile name="Instance_And_Custom_Records">
  <workflow activityDefinitionId="*"
    <customTrackingQueries>
      <customTrackingQuery name="*" activityName="*" />
    </customTrackingQueries>
  </workflow>
</trackingProfile>

```

## See also

- [SQL Tracking](#)
- [Windows Server App Fabric Monitoring](#)
- [Monitoring Applications with App Fabric](#)

# Tracking Participants

3/26/2019 • 5 minutes to read • [Edit Online](#)

Tracking participants are extensibility points that allow a workflow developer to access [TrackingRecord](#) objects and process them. .NET Framework 4.6.1 includes a standard tracking participant that writes tracking records as Event Tracing for Windows (ETW) events. If that does not meet your requirements, you can also write a custom tracking participant.

## Tracking Participants

The tracking infrastructure allows the application of a filter on the outgoing tracking records such that a participant can subscribe to a subset of the records. The mechanism to apply a filter is through a tracking profile.

Windows Workflow Foundation (WF) in .NET Framework 4.6.1 provides a tracking participant that writes the tracking records to an ETW session. The participant is configured on a workflow service by adding a tracking-specific behavior in a configuration file. Enabling an ETW tracking participant allows tracking records to be viewed in the event viewer. The SDK sample for ETW-based tracking is a good way to get familiar with WF tracking using the ETW-based tracking participant.

## ETW Tracking Participant

.NET Framework 4.6.1 includes an ETW Tracking Participant that writes the tracking records to an ETW session. This is done in a very efficient manner with minimal impact to the application's performance or to the server's throughput. An advantage of using the standard ETW tracking participant is that the tracking records it receives can be viewed with the other application and system logs in the Windows Event Viewer.

The standard ETW tracking participant is configured in the Web.config file as shown in the following example.

```
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.0" />
  </system.web>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <etwTracking profileName="Sample Tracking Profile"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <tracking>
      <profiles>
        <trackingProfile name="Sample Tracking Profile">
          ...
        </trackingProfile>
      </profiles>
    </tracking>
  </system.serviceModel>
</configuration>
```

#### NOTE

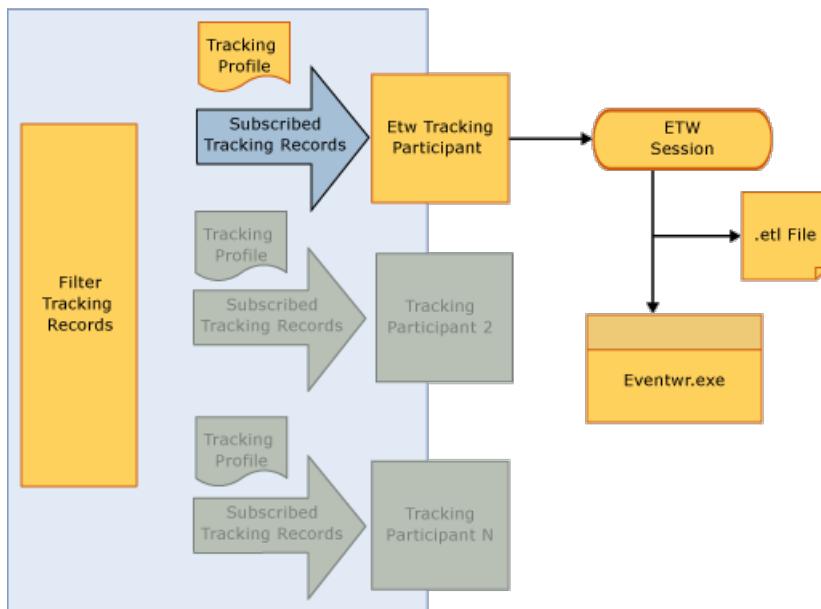
If a `trackingProfile` name is not specified, such as just `<etwTracking/>` or `<etwTracking profileName="" />`, then the default tracking profile installed with the .NET Framework 4.6.1 in the Machine.config file is used.

In the Machine.config file, the default tracking profile subscribes to workflow instance records and faults.

In ETW, events are written to the ETW session through a provider ID. The provider ID that the ETW tracking participant uses for writing the tracking records to ETW is defined in the diagnostics section of the Web.config file (under `<system.serviceModel><diagnostics>`). By default, the ETW tracking participant uses a default provider ID when one has not been specified, as shown in the following example.

```
<system.serviceModel>
  <diagnostics etwProviderId="52A3165D-4AD9-405C-B1E8-7D9A257EAC9F" />
```

The following illustration shows the flow of tracking data through the ETW tracking participant. Once the tracking data reaches the ETW session, it can be accessed in a number of ways. One of the most useful ways to access these events is through Event Viewer, a common Windows tool used for viewing logs and traces from applications and services.



## Tracking Participant Event Data

A tracking participant serializes tracked event data to an ETW session in the format of one event per tracking record. An event is identified using an ID within the range of 100 through 199. For definitions of the tracking event records emitted by a tracking participant, see the [Tracking Events Reference](#) topic.

The size of an ETW event is limited by the ETW buffer size, or the by the maximum payload for an ETW event, whichever value is smaller. If the size of the event exceeds either of these ETW limits, the event is truncated and its content removed in an arbitrary manner. Variables, arguments, annotations and custom data are not selectively removed. In the case of truncation, all of these are truncated regardless of the value that caused the event size to exceed the ETW limit. The removed data is replaced with `<item>..<item>`.

Complex types in variables, arguments, and custom data items are serialized to the ETW event record using the [NetDataContractSerializer Class](#). This class includes CLR-type information in the serialized XML stream.

Truncation of payload data due to ETW limits can result in duplicate tracking records being sent to an ETW session. This can occur if more than one session is listening for the events and the sessions have different payload

limits for the events.

For the session with the lower limit the event may be truncated. The ETW tracking participant does not have any knowledge of the number of sessions listening for the events; if an event is truncated for a session then the ETW participant retries sending the event once. In this case the session that is configured to accept a larger payload size will get the event twice (the non-truncated and truncated event). Duplication can be prevented by configuring all the ETW sessions with same buffer size limits.

## Accessing Tracking Data from an ETW Participant in the Event Viewer

Events that are written to an ETW session by the ETW tracking participant can be accessed through the Event Viewer (when using the default provider ID). This allows for rapidly viewing of tracking records that have been emitted by the workflow.

### NOTE

Tracking record events emitted to an ETW session use event IDs in the range of 100 through 199.

#### To enable viewing the Tracking Records in Event Viewer

1. Start the Event Viewer (EVENTVWR.EXE)
2. Select **Event Viewer, Applications and Services Logs, Microsoft, Windows, Application Server-Applications**.
3. Right-click and ensure that **View, Show Analytic and Debug logs** is selected. If not, select it so the check mark appears next to it. This displays the **Analytic, Perf, and Debug** logs.
4. Right-click the **Analytic** log and then select **Enable Log**. The log will exist in the %SystemRoot%\System32\Winevt\Logs\Microsoft-Windows-Application Server-Applications%4Analytic.etl file.

## Custom Tracking Participant

The tracking participant API allows extension of the tracking runtime with a user-provided tracking participant that can include custom logic to handle tracking records emitted by the workflow runtime. To write a custom tracking participant, the developer must implement the `Track` method on the `TrackingParticipant` class. This method is called when a tracking record is emitted by the workflow runtime.

Tracking participants derive from the `TrackingParticipant` class. The system-provided `EtwTrackingParticipant` emits an Event Tracking for Windows (ETW) event for each tracking record that is received. To create a custom tracking participant, a class is created that derives from `TrackingParticipant`. To provide basic tracking functionality, override `Track`. `Track` is called when a tracking record is sent by the runtime and can be processed in the desired manner. In the following example, a custom tracking participant class is defined that emits all tracking records to the console window. You can also implement a `TrackingParticipant` object that processes the tracking records asynchronously using its `BeginTrack` and `EndTrack` methods

```

class ConsoleTrackingParticipant : TrackingParticipant
{
    protected override void Track(TrackingRecord record, TimeSpan timeout)
    {
        if (record != null)
        {
            Console.WriteLine("=====");
            Console.WriteLine(record);
        }
    }
}

```

To use a particular tracking participant, register it with the workflow instance that you want to track, as shown in the following example.

```
myInstance.Extensions.Add(new ConsoleTrackingParticipant());
```

In the following example, a workflow that consists of a [Sequence](#) activity that contains a [WriteLine](#) activity is created. The `ConsoleTrackingParticipant` is added to the extensions and the workflow is invoked.

```

Activity activity= new Sequence()
{
    Activities =
    {
        new WriteLine()
        {
            Text = "Hello World."
        }
    }
};

WorkflowApplication instance = new WorkflowApplication(activity);

instance.Extensions.Add(new ConsoleTrackingParticipant());
instance.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
{
    Console.WriteLine("workflow instance completed, Id = " + instance.Id);
    resetEvent.Set();
};
instance.Run();
Console.ReadLine();

```

## See also

- [Windows Server App Fabric Monitoring](#)
- [Monitoring Applications with App Fabric](#)

# Configuring Tracking for a Workflow

3/16/2019 • 6 minutes to read • [Edit Online](#)

A workflow can execute in three ways:

- Hosted in [WorkflowServiceHost](#)
- Executed as a [WorkflowApplication](#)
- Executed directly using [WorkflowInvoker](#)

Depending on the workflow hosting option, a tracking participant can be added either through code or through a configuration file. This topic describes how tracking is configured by adding a tracking participant to a [WorkflowApplication](#) and to a [WorkflowServiceHost](#), and how to enable tracking when using [WorkflowInvoker](#).

## Configuring Workflow Application Tracking

A workflow can run using the [WorkflowApplication](#) class. This topic demonstrates how tracking is configured for a .NET Framework 4.6.1 workflow application by adding a tracking participant to the [WorkflowApplication](#) workflow host. In this case, the workflow runs as a workflow application. You configure a workflow application through code (rather than by using a configuration file), which is a self-hosted .exe file using the [WorkflowApplication](#) class. The tracking participant is added as an extension to the [WorkflowApplication](#) instance. This is done by adding the [TrackingParticipant](#) to the extensions collection for the WorkflowApplication instance.

For a workflow application, you can add the [EtwTrackingParticipant](#) behavior extension as shown in the following code.

```
LogActivity activity = new LogActivity();

WorkflowApplication instance = new WorkflowApplication(activity);
EtwTrackingParticipant trackingParticipant =
    new EtwTrackingParticipant
{
    TrackingProfile = new TrackingProfile
    {
        Name = "SampleTrackingProfile",
        ActivityDefinitionId = "ProcessOrder",
        Queries = new WorkflowInstanceQuery
        {
            States = { "*" }
        }
    }
};

instance.Extensions.Add(trackingParticipant);
```

## Configuring Workflow Service Tracking

A workflow can be exposed as a WCF service when hosted in the [WorkflowServiceHost](#) service host. [WorkflowServiceHost](#) is a specialized .NET ServiceHost implementation for a workflow-based service. This section explains how to configure tracking for a .NET Framework 4.6.1 workflow service running in [WorkflowServiceHost](#). It is configured through a Web.config file (for a Web-hosted service) or an App.config file (for a service hosted in a stand-alone application, such as a console application) by specifying a service behavior or through code by adding a tracking-specific behavior to the [Behaviors](#) collection for the service host.

For a workflow service hosted in [WorkflowServiceHost](#), you can add the [EtwTrackingParticipant](#) using the <

`behavior` > element in a configuration file, as shown in the following example.

```
<behaviors>
  <serviceBehaviors>
    <behavior>
      <etwTracking profileName="Sample Tracking Profile" />
    </behavior>
  </serviceBehaviors>
</behaviors>
```

Alternatively, for a workflow service hosted in [WorkflowServiceHost](#), you can add the [EtwTrackingParticipant](#) behavior extension through code. To add a custom tracking participant, create a new behavior extension and add it to the [ServiceHost](#) as shown in the following example code.

**NOTE**

If you want to view sample code that shows how to create a custom behavior element that adds a custom tracking participant, refer to the [Tracking](#) samples.

```
ServiceHost svcHost = new ServiceHost(typeof(WorkflowService), new
                                         Uri("http://localhost:8001/Sample"));
EtwTrackingBehavior trackingBehavior =
  new EtwTrackingBehavior
  {
    ProfileName = "Sample Tracking Profile"
  };
svcHost.Description.Behaviors.Add(trackingBehavior);
svcHost.Open();
```

The tracking participant is added to the workflow service host as an extension to the behavior.

This sample code below shows how to read a tracking profile from configuration file.

```

TrackingProfile GetProfile(string profileName, string displayName)
{
    TrackingProfile trackingProfile = null;
    TrackingSection trackingSection =
(TrackingSection)WebConfigurationManager.GetSection("system.serviceModel/tracking");
    if (trackingSection == null)
    {
        return null;
    }

    if (profileName == null)
    {
        profileName = "";
    }

    //Find the profile with the specified profile name in the list of profile found in config
    var match = from p in new List<TrackingProfile>(trackingSection.TrackingProfiles)
               where (p.Name == profileName) && ((p.ActivityDefinitionId == displayName) ||
(p.ActivityDefinitionId == "*"))
               select p;

    if (match.Count() == 0)
    {
        //return an empty profile
        trackingProfile = new TrackingProfile()
        {
            ActivityDefinitionId = displayName
        };

    }
    else
    {
        trackingProfile = match.First();
    }

    return trackingProfile;
}

```

This sample code shows how to add a tracking profile to a workflow host.

```

WorkflowServiceHost workflowServiceHost = serviceHostBase as WorkflowServiceHost;
if (null != workflowServiceHost)
{
    string workflowDisplayName = workflowServiceHost.Activity.DisplayName;
    TrackingProfile trackingProfile = GetProfile(this.profileName, workflowDisplayName);
    workflowServiceHost.WorkflowExtensions.Add(() => new EtwTrackingParticipant {
        TrackingProfile = trackingProfile
    });
}

```

#### **NOTE**

For more information on tracking profiles, refer to [Tracking Profiles](#).

### Configuring tracking using WorkflowInvoker

To configure tracking for a workflow executed using [WorkflowInvoker](#), add the tracking provider as an extension to a [WorkflowInvoker](#) instance. The following code example is from the [Custom Tracking](#) sample.

```

WorkflowInvoker invoker = new WorkflowInvoker(BuildSampleWorkflow());
invoker.Extensions.Add(customTrackingParticipant);
invoker.Invoke();

```

## Viewing tracking records in Event Viewer

There are two Event Viewer logs of particular interest to view when tracking WF execution - the Analytic log and the Debug log. Both reside under the Microsoft|Windows|Application Server-Applications node. Logs within this section contain events from a single application rather than events that have an impact on the entire system.

Debug trace events are written to the Debug Log. To collect WF debug trace events in the Event Viewer, enable the Debug Log.

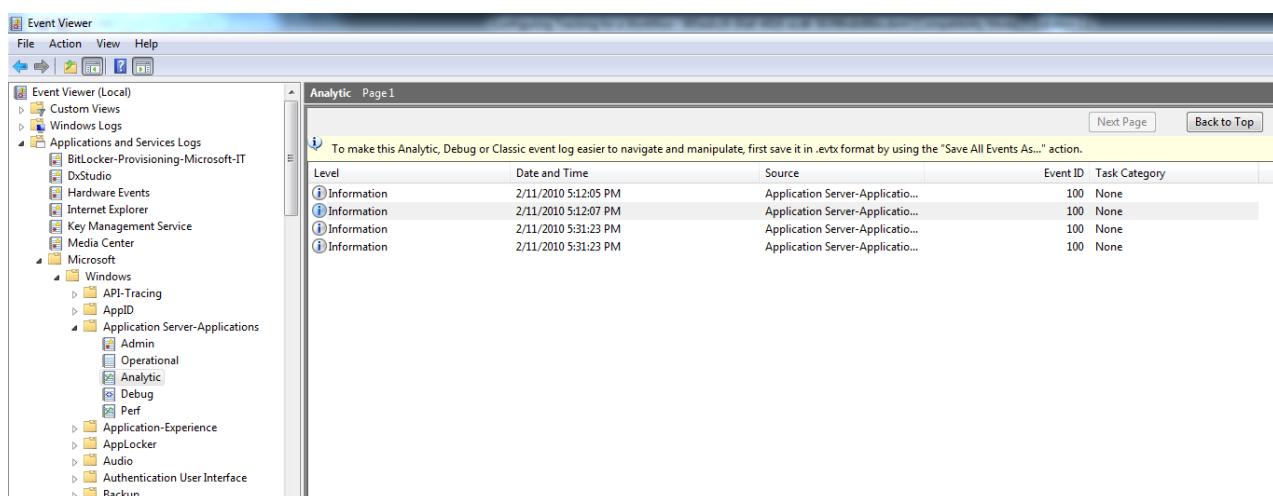
1. To open Event Viewer, click **Start**, and then click **Run**. In the Run dialog, type `eventvwr`.
2. In the Event Viewer dialog, expand the **Applications and Services Logs** node.
3. Expand the **Microsoft Windows**, and **Application Server-Applications** nodes.
4. Right-click the **Debug** node under the **Application Server-Applications** node, and select **Enable Log**.
5. Execute your tracing-enabled application to generate tracing events.
6. Right-click the **Debug** node and select **Refresh**. Tracing events should be visible in the center pane.

WF 4 provides a tracking participant that writes tracking records to an ETW (Event Tracing for Windows) session. The ETW tracking participant is configured with a tracking profile to subscribe to tracking records. When tracking is enabled, errors tracking records are emitted to ETW. ETW tracking events (between the range of 100-113) corresponding to the tracking events emitted by the ETW tracking participant are written to the Analytic Log.

To view tracking records, follow these steps.

1. To open Event Viewer, click **Start**, and then click **Run**. In the Run dialog, type `eventvwr`.
2. In the Event Viewer dialog, expand the **Applications and Services Logs** node.
3. Expand the **Microsoft Windows**, and **Application Server-Applications** nodes.
4. Right-click the **Analytic** node under the **Application Server-Applications** node, and select **Enable Log**.
5. Execute your tracking-enabled application to generate tracking records.
6. Right-click the **Analytic** node and select **Refresh**. Tracking records should be visible in the center pane.

The following image shows tracking events in the event viewer:



## Registering an application-specific provider ID

If events need to be written to a specific application log, follow these steps to register the new provider manifest.

1. Declare the provider ID in the application configuration file.

```
<system.serviceModel>
  <diagnostics etwProviderId="2720e974-9fe9-477a-bb60-81fe3bf91eec"/>
</system.serviceModel>
```

2. Copy the manifest file from %windir%\Microsoft.NET\Framework\<latest version of .NET Framework 4.6.1>\Microsoft.Windows.ApplicationServer.Applications.man to a temporary location, and rename it to Microsoft.Windows.ApplicationServer.Applications\_Provider1.man
3. Change the GUID in the manifest file to the new GUID.

```
<provider name="Microsoft-Windows-Application Server-Applications" guid="{2720e974-9fe9-477a-bb60-81fe3bf91eec}">
```

4. Change the provider name if you do not want to uninstall the default provider.

```
<provider name="Microsoft-Windows-Application Server-Applications" guid="{2720e974-9fe9-477a-bb60-81fe3bf91eec}">
```

5. If you changed the provider name in the previous step, change the channel names in the manifest file to the new provider name.

```
<channel name="Microsoft-Windows-Application Server-Applications_Provider1/Admin" chid="ADMIN_CHANNEL" symbol="ADMIN_CHANNEL" type="Admin" enabled="false" isolation="Application" message="$(string.MICROSOFT_WINDOWS_APPLICATIONSERVER_APPLICATIONS.channel.ADMIN_CHANNEL.message)" />
<channel name="Microsoft-Windows-Application Server-Applications_Provider1/Operational" chid="OPERATIONAL_CHANNEL" symbol="OPERATIONAL_CHANNEL" type="Operational" enabled="false" isolation="Application" message="$(string.MICROSOFT_WINDOWS_APPLICATIONSERVER_APPLICATIONS.channel.OPERATIONAL_CHANNEL.message)" />
<channel name="Microsoft-Windows-Application Server-Applications_Provider1/Analytic" chid="ANALYTIC_CHANNEL" symbol="ANALYTIC_CHANNEL" type="Analytic" enabled="false" isolation="Application" message="$(string.MICROSOFT_WINDOWS_APPLICATIONSERVER_APPLICATIONS.channel.ANALYTIC_CHANNEL.message)" />
<channel name="Microsoft-Windows-Application Server-Applications_Provider1/Debug" chid="DEBUG_CHANNEL" symbol="DEBUG_CHANNEL" type="Debug" enabled="false" isolation="Application" message="$(string.MICROSOFT_WINDOWS_APPLICATIONSERVER_APPLICATIONS.channel.DEBUG_CHANNEL.message)" />
<channel name="Microsoft-Windows-Application Server-Applications_Provider1/Perf" chid="PERF_CHANNEL" symbol="PERF_CHANNEL" type="Analytic" enabled="false" isolation="Application" message="$(string.MICROSOFT_WINDOWS_APPLICATIONSERVER_APPLICATIONS.channel.PERF_CHANNEL.message)" />
```

6. Generate the resource DLL by following these steps.

- a. Install the Windows SDK. The Windows SDK includes the message compiler ([mc.exe](#)) and resource compiler ([rc.exe](#)).
- b. In a Windows SDK command prompt, run mc.exe on the new manifest file.

```
mc.exe Microsoft.Windows.ApplicationServer.Applications_Provider1.man
```

- c. Run rc.exe on the resource file generated in the previous step.

```
rc.exe Microsoft.Windows.ApplicationServer.Applications_Provider1.rc
```

- d. Create an empty cs file called NewProviderReg.cs.

- e. Create a resource DLL using the C# compiler.

```
csc /target:library /win32res:Microsoft.Windows.ApplicationServer.Applications_Provider1.res  
NewProviderReg.cs /out:Microsoft.Windows.ApplicationServer.Applications_Provider1.dll
```

- f. Change the resource and message dll name in the manifest file from

`Microsoft.Windows.ApplicationServer.Applications.Provider1.man` to the new dll name.

```
<provider name="Microsoft-Windows-Application Server-Applications_Provider1" guid="{2720e974-  
9fe9-477a-bb60-81fe3bf91eec}" symbol="Microsoft_Windows_ApplicationServer_ApplicationEvents"  
resourceFileName="<dll directory>\Microsoft.Windows.ApplicationServer.Applications_Provider1.dll"  
messageFileName="<dll directory>\Microsoft.Windows.ApplicationServer.Applications_Provider1.dll">
```

- g. Use [wevtutil](#) to register the manifest.

```
wevtutil im Microsoft.Windows.ApplicationServer.Applications_Provider1.man
```

## See also

- [Windows Server App Fabric Monitoring](#)
- [Monitoring Applications with App Fabric](#)

# Using Tracking to Troubleshoot Applications

1/23/2019 • 2 minutes to read • [Edit Online](#)

Windows Workflow Foundation (WF) enables you to track workflow-related information to give details into the execution of a Windows Workflow Foundation application or service. Windows Workflow Foundation hosts are able to capture workflow events during the execution of a workflow instance. If your workflow generates faults or exceptions, you can use the Windows Workflow Foundation tracking details to troubleshooting its processing.

## Troubleshooting a WF using WF Tracking

To detect faults within the processing of a Windows Workflow Foundation activity, you can enable tracking with a tracking profile that queries for an [ActivityStateRecord](#) with the state of Faulted. The corresponding query is specified in the following code.

```
<activityStateQueries>
    <activityStateQuery activityName="*">
        <states>
            <state name="Faulted" />
        </states>
    </activityStateQuery>
</activityStateQueries>
```

If a fault is propagated and handled within a fault handler (such as a [TryCatch](#) activity) this can be detected using a [FaultPropagationRecord](#). The [FaultPropagationRecord](#) indicates the source activity of the fault and the name of the fault handler. The [FaultPropagationRecord](#) contains fault details in form of the exception stack for the fault. The query to subscribe for a [FaultPropagationRecord](#) is shown in the following example.

```
<faultPropagationQueries>
    <faultPropagationQuery faultSourceActivityName ="" faultHandlerActivityName="*"/>
</faultPropagationQueries>
```

If a fault is not handled within the workflow it results in an unhandled exception at the workflow instance and the workflow instance is aborted. To understand the details of the unhandled exception, the tracking profile must query the workflow instance record with `state name="UnhandledException"` as specified in the following example.

```
<workflowInstanceQueries>
    <workflowInstanceQuery>
        <states>
            <state name="UnhandledException" />
        </states>
    </workflowInstanceQuery>
</workflowInstanceQueries>
```

When a workflow instance encounters an unhandled exception, a [WorkflowInstanceUnhandledExceptionRecord](#) object is emitted if Windows Workflow Foundation tracking has been enabled.

This tracking record contains the fault details in the form of the exception stack. It has details of the source of the fault (for example, the activity) that faulted and resulted in the unhandled exception. To subscribe to fault events from a Windows Workflow Foundation, enable tracking by adding a tracking participant. Configure this participant with a tracking profile that queries for `ActivityStateQuery (state="Faulted")`, [FaultPropagationRecord](#), and `WorkflowInstanceQuery (state="UnhandledException")`.

If tracking is enabled using the ETW tracking participant, the fault events are emitted to an ETW session. The events can be viewed using the Event Viewer event viewer. This can be found under the node **Event Viewer->Applications and Services Logs->Microsoft->Windows->Application Server-Applications** in the analytic channel.

## See also

- [Windows Server App Fabric Monitoring](#)
- [Monitoring Applications with App Fabric](#)

# Workflow Tracing

1/23/2019 • 2 minutes to read • [Edit Online](#)

Workflow tracing offers a way to capture diagnostic information using .NET Framework trace listeners. Tracing can be enabled if a problem is detected with the application and then disabled again once the problem is resolved. There are two ways you could enable debug tracing for workflows. You can configure it using the Event Trace viewer or you can use [System.Diagnostics](#) to send trace events to a file.

## Enabling Debug Tracing in ETW

To enable tracing using ETW, enable the Debug channel in Event Viewer:

1. Navigate to analytic and debug logs node in Event Viewer.
2. In the tree view in Event Viewer, navigate to **Event Viewer->Applications and Services Logs->Microsoft->Windows->Application Server-Applications**. Right-click **Application Server-Applications** and select **View->Show Analytic and Debug Logs**. Right-click **Debug** and select **Enable Log**.
3. When a workflow runs the debug and traces are emitted to ETW debug channel, they can be viewed in the Event Viewer. Navigate to **Event Viewer->Applications and Services Logs->Microsoft->Windows->Application Server-Applications**. Right-click **Debug** and select **Refresh**.
4. The default analytic trace buffer size is only 4 kilobytes (KB); it is recommended to increase the size to 32 KB. To do this, perform the following steps.
  - a. Execute the following command in the current framework directory (for example, C:\Windows\Microsoft.NET\Framework\v4.0.21203):

```
wEvtutil um Microsoft.Windows.ApplicationServer.Applications.man
```
  - b. Change the <bufferSize> value in the Windows.ApplicationServer.Applications.man file to 32.

```
<channel name="Microsoft-Windows-Application Server-Applications/Analytic" chid="ANALYTIC_CHANNEL" symbol="ANALYTIC_CHANNEL" type="Analytic" enabled="false" isolation="Application" message="$(string.MICROSOFT_WINDOWS_APPLICATIONSERVER_APPLICATIONS.channel.ANALYTIC_CHANNEL.message)"> <publishing> <bufferSize>32</bufferSize> </publishing> </channel>
```

- c. Execute the following command in the current framework directory (for example, C:\Windows\Microsoft.NET\Framework\v4.0.21203):

```
wEvtutil im Microsoft.Windows.ApplicationServer.Applications.man
```

### NOTE

If you are using the .NET Framework 4 Client Profile, you must first register the ETW manifest by running the following command from the .NET Framework 4 directory: 

```
ServiceModelReg.exe -i -c:etw
```

## Enabling Debug Tracing using System.Diagnostics

These listeners can be configured in the App.config file of the workflow application, or the Web.config for a workflow service. In this example, a [TextWriterTraceListener](#) is configured to save tracing information to the MyTraceLog.txt file in the current directory.

```
<configuration>
  <system.diagnostics>
    <sources>
      <source name="System.Activities" switchValue="Information">
        <listeners>
          <add name="textListener" />
          <remove name="Default" />
        </listeners>
      </source>
    </sources>
    <sharedListeners>
      <add name="textListener"
           type="System.Diagnostics.TextWriterTraceListener"
           initializeData="MyTraceLog.txt"
           traceOutputOptions="ProcessId, DateTime" />
    </sharedListeners>
    <trace autoflush="true" indentsize="4">
      <listeners>
        <add name="textListener" />
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

## See also

- [Windows Server App Fabric Monitoring](#)
- [Monitoring Applications with App Fabric](#)

# Tracking Events Reference

3/9/2019 • 12 minutes to read • [Edit Online](#)

During execution a workflow in .NET Framework 4.6.1 raises tracking events as it moves through various stages in its lifetime. The host can subscribe to these events and keep updated on the status of the workflow's progress during its lifetime. The tracking events raised are discussed in this section.

## Event Reference

| Event ID   | Event Level | Event Message  | Keywords  |
|--|-------------|--|---|
| <a href="#">100 - WorkflowInstanceRecord</a>                   | Information | TrackRecord=WorkflowInstanceRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, State = %5, Annotations = %6, ProfileName = %7  | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTracking |
| <a href="#">101 - WorkflowInstanceUnhandledExceptionRecord</a> | Error       | TrackRecord = WorkflowInstanceUnhandledExceptionRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, SourceName = %5, SourceId = %6, SourceInstanceId = %7, SourceTypeName=%8, Exception=%9, Annotations= %10, ProfileName = %11 | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTracking |
| <a href="#">102 - WorkflowInstanceAbortedRecord</a>            | Warning     | TrackRecord = WorkflowInstanceAbortedRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, Reason = %5, Annotations = %6, ProfileName = %7  | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTracking |
| <a href="#">103 - ActivityStateRecord</a>                      | Information | TrackRecord = ActivityStateRecord, InstanceID = %1, RecordNumber=%2, EventTime=%3, State = %4, Name=%5, ActivityId=%6, ActivityInstanceId=%7, ActivityTypeName=%8, Arguments=%9, Variables=%10, Annotations=%11, ProfileName = %12                                 | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTracking |

| Event ID                      | Event Level | Event Message  | Keywords   |
|-------------------------------|-------------|--|--|
| 104 - ActivityScheduledRecord | Information | TrackRecord = ActivityScheduledRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, Name = %4, ActivityId = %5, ActivityInstanceId = %6, ActivityTypeName = %7, ChildActivityName = %8, ChildActivityId = %9, ChildActivityInstanceId = %10, ChildActivityTypeName = %11, Annotations = %12, ProfileName = %13  | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTacking |
| 105 - FaultPropagationRecord  | Warning     | TrackRecord = FaultPropagationRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, FaultSourceActivityName = %4, FaultSourceActivityId = %5, FaultSourceActivityInstanceId = %6, FaultSourceActivityTypeName = %7, FaultHandlerActivityName = %8, FaultHandlerActivityId = %9, FaultHandlerActivityInstanceId = %10, FaultHandlerActivityTypeName = %11, Fault = %12, IsFaultSource = %13, Annotations = %14, ProfileName = %15 | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTacking |
| 106 - CancelRequestRecord     | Information | TrackRecord = CancelRequestedRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, Name = %4, ActivityId = %5, ActivityInstanceId = %6, ActivityTypeName = %7, ChildActivityName = %8, ChildActivityId = %9, ChildActivityInstanceId = %10, ChildActivityTypeName = %11, Annotations = %12, ProfileName = %13  | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTacking |

| Event ID                             | Event Level | Event Message  | Keywords  |
|--------------------------------------|-------------|--|---|
| 107 --<br>BookmarkResumptionRecord   | Information | TrackRecord = BookmarkResumptionRecord , InstanceID=%1, RecordNumber=%2, EventTime=%3, Name=%4, SubInstanceId=%5, OwnerActivityName=%6, OwnerActivityId=%7, OwnerActivityInstanceId=%8, OwnerActivityTypeName=%9, Annotations=%10, ProfileName = %11 | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTracking             |
| 108 -<br>CustomTrackingRecordInfo    | Information | TrackRecord = CustomTrackingRecord, InstanceID = %1, RecordNumber=%2, EventTime=%3, Name=%4, ActivityName=%5, ActivityId=%6, ActivityInstanceId=%7, ActivityTypeName=%8, Data=%9, Annotations=%10, ProfileName = %11                                 | UserEvents, EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTracking |
| 110 -<br>CustomTrackingRecordWarning | Warning     | TrackRecord = CustomTrackingRecord, InstanceID = %1, RecordNumber=%2, EventTime=%3, Name=%4, ActivityName=%5, ActivityId=%6, ActivityInstanceId=%7, ActivityTypeName=%8, Data=%9, Annotations=%10, ProfileName = %11                                 | UserEvents, EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTracking |
| 111 -<br>CustomTrackingRecordError   | Error       | TrackRecord = CustomTrackingRecord, InstanceID = %1, RecordNumber=%2, EventTime=%3, Name=%4, ActivityName=%5, ActivityId=%6, ActivityInstanceId=%7, ActivityTypeName=%8, Data=%9, Annotations=%10, ProfileName = %11                                 | UserEvents, EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTracking |

| Event ID                                     | Event Level | Event Message   | Keywords  |
|--|-------------|---|---|
| 112 - WorkflowInstanceSuspended Record       | Information | TrackRecord = WorkflowInstanceSuspended Record, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, Reason = %5, Annotations = %6, ProfileName = %7                                  | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTracking |
| 113 - WorkflowInstanceTerminated Record      | Error       | TrackRecord = WorkflowInstanceTerminated Record, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, Reason = %5, Annotations = %6, ProfileName = %7                                 | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTracking |
| 114 - WorkflowInstanceRecordWithId           | Information | TrackRecord= WorkflowInstanceRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, State = %5, Annotations = %6, ProfileName = %7, WorkflowDefinitionIdentity = %8             | HealthMonitoring, WFTracking                                      |
| 115 - WorkflowInstanceAbortedRecordWithId    | Warning     | TrackRecord = WorkflowInstanceAbortedRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, Reason = %5, Annotations = %6, ProfileName = %7, WorkflowDefinitionIdentity = %8    | HealthMonitoring, WFTracking                                      |
| 116 - WorkflowInstanceSuspended RecordWithId | Information | TrackRecord = WorkflowInstanceSuspended Record, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, Reason = %5, Annotations = %6, ProfileName = %7, WorkflowDefinitionIdentity = %8 | HealthMonitoring, WFTracking                                      |

| Event ID  | Event Level | Event Message   | Keywords   |
|---|-------------|---|--|
| <a href="#">117 - WorkflowInstanceTerminated RecordWithId</a>         | Error       | TrackRecord = WorkflowInstanceTerminated Record, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, Reason = %5, Annotations = %6, ProfileName = %7, WorkflowDefinitionIdentity = %8  | HealthMonitoring, WFTracking                               |
| <a href="#">118 - WorkflowInstanceUnhandled ExceptionRecordWithId</a> | Error       | TrackRecord = WorkflowInstanceUnhandled ExceptionRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, SourceName = %5, SourceId = %6, SourceInstanceId = %7, SourceTypeName=%8, Exception=%9, Annotations= %10, ProfileName = %11, WorkflowDefinitionIdentity = %12 | HealthMonitoring, WFTracking, HealthMonitoring, WFTracking |
| <a href="#">119 - WorkflowInstanceUpdatedRecord</a>                   | Information | TrackRecord= WorkflowInstanceUpdatedRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, State = %5, OriginalDefinitionIdentity = %6, UpdatedDefinitionIdentity = %7, Annotations = %8, ProfileName = %9  | HealthMonitoring, WFTracking                               |
| <a href="#">225 - TraceCorrelationKeys</a>                            | Information | Calculated correlation key '%1' using values '%2' in parent scope '%3'.   | Troubleshooting WFServices                                 |
| <a href="#">440 - StartSignpostEvent1</a>                             | Information | Activity boundary.  | Troubleshooting WFServices                                 |
| <a href="#">441 - StopSignpostEvent1</a>                              | Information | Activity boundary.  | Troubleshooting WFServices                                 |
| <a href="#">1001 - WorkflowApplicationCompleted</a>                   | Information | WorkflowInstance Id: "%1" has completed in the Closed state.  | WFRuntime  |
| <a href="#">1002 - WorkflowApplicationTerminated</a>                  | Information | WorkflowApplication Id: "%1" was terminated. It has completed in the Faulted state with an exception.   | WFRuntime  |

| Event ID                                     | Event Level | Event Message   | Keywords  |
|--|-------------|---|-----------|
| 1003 - WorkflowInstanceCanceled              | Information | WorkflowInstance Id: '%1' has completed in the Canceled state.  | WFRuntime |
| 1004 - WorkflowInstanceAborted               | Information | WorkflowInstance Id: '%1' was aborted with an exception.  | WFRuntime |
| 1005 - WorkflowApplicationIdled              | Information | WorkflowApplication Id: '%1' went idle.   | WFRuntime |
| 1006 - WorkflowApplicationUnhandledException | Error       | WorkflowInstance Id: '%1' has encountered an unhandled exception. The exception originated from Activity '%2', DisplayName: '%3'. The following action will be taken: %4. | WFRuntime |
| 1007 - WorkflowApplicationPersisted          | Information | WorkflowApplication Id: '%1' was Persisted.   | WFRuntime |
| 1008 - WorkflowApplicationUnloaded           | Information | WorkflowInstance Id: '%1' was Unloaded.   | WFRuntime |
| 1009 - ActivityScheduled                     | Information | Parent Activity '%1', DisplayName: '%2', InstanceId: '%3' scheduled child Activity '%4', DisplayName: '%5', InstanceId: '%6'.   | WFRuntime |
| 1010 - ActivityCompleted                     | Information | Parent Activity '%1', DisplayName: '%2', InstanceId: '%3' scheduled child Activity '%4', DisplayName: '%5', InstanceId: '%6'.   | WFRuntime |
| 1011 - ScheduleExecuteActivityWorkItem       | Verbose     | An ExecuteActivityWorkItem has been scheduled for Activity '%1', DisplayName: '%2', InstanceId: '%3'.   | WFRuntime |
| 1012 - StartExecuteActivityWorkItem          | Verbose     | Starting execution of an ExecuteActivityWorkItem for Activity '%1', DisplayName: '%2', InstanceId: '%3'.  | WFRuntime |
| 1013 - CompleteExecuteActivityWorkItem       | Verbose     | An ExecuteActivityWorkItem has completed for Activity '%1', DisplayName: '%2', InstanceId: '%3'.  | WFRuntime |

| Event ID                              | Event Level | Event Message   | Keywords  |
|---------------------------------------|-------------|---|-----------|
| 1014 - ScheduleCompletionWorkItem     | Verbose     | A CompletionWorkItem has been scheduled for parent Activity '%1', DisplayName: '%2', InstanceId: '%3'. Completed Activity '%4', DisplayName: '%5', InstanceId: '%6'.    | WFRuntime |
| 1015 - StartCompletionWorkItem        | Verbose     | Starting execution of a CompletionWorkItem for parent Activity '%1', DisplayName: '%2', InstanceId: '%3'. Completed Activity '%4', DisplayName: '%5', InstanceId: '%6'. | WFRuntime |
| 1016 - CompleteCompletionWorkItem     | Verbose     | A CompletionWorkItem has completed for parent Activity '%1', DisplayName: '%2', InstanceId: '%3'. Completed Activity '%4', DisplayName: '%5', InstanceId: '%6'.         | WFRuntime |
| 1017 - ScheduleCancelActivityWorkItem | Verbose     | A CancelActivityWorkItem has been scheduled for Activity '%1', DisplayName: '%2', InstanceId: '%3'.   | WFRuntime |
| 1018 - StartCancelActivityWorkItem    | Verbose     | Starting execution of a CancelActivityWorkItem for Activity '%1', DisplayName: '%2', InstanceId: '%3'.  | WFRuntime |
| 1019 - CompleteCancelActivityWorkItem | Verbose     | A CancelActivityWorkItem has completed for Activity '%1', DisplayName: '%2', InstanceId: '%3'.  | WFRuntime |
| 1020 - CreateBookmark                 | Verbose     | A Bookmark has been created for Activity '%1', DisplayName: '%2', InstanceId: '%3'. BookmarkName: %4, BookmarkScope: %5.  | WFRuntime |
| 1021 - ScheduleBookmarkWorkItem       | Verbose     | A BookmarkWorkItem has been scheduled for Activity '%1', DisplayName: '%2', InstanceId: '%3'. BookmarkName: %4, BookmarkScope: %5.                                      | WFRuntime |

| Event ID                                  | Event Level | Event Message   | Keywords  |
|---|-------------|---|-----------|
| 1022 - StartBookmarkWorkItem              | Verbose     | Starting execution of a BookmarkWorkItem for Activity '%1', DisplayName: '%2', InstanceId: '%3'. BookmarkName: %4, BookmarkScope: %5.   | WFRuntime |
| 1023 - CompleteBookmarkWorkItem           | Verbose     | A BookmarkWorkItem has completed for Activity '%1', DisplayName: '%2', InstanceId: '%3'. BookmarkName: %4, BookmarkScope: %5.   | WFRuntime |
| 1024 - CreateBookmarkScope                | Verbose     | A BookmarkScope has been created: %1.   | WFRuntime |
| 1025 - BookmarkScopeInitialized           | Verbose     | The BookmarkScope that had TemporaryId: '%1' has been initialized with Id: '%2'.  | WFRuntime |
| 1026 - ScheduleTransactionContextWorkItem | Verbose     | A TransactionContextWorkItem has been scheduled for Activity '%1', DisplayName: '%2', InstanceId: '%3'.   | WFRuntime |
| 1027 - StartTransactionContextWorkItem    | Verbose     | Starting execution of a TransactionContextWorkItem for Activity '%1', DisplayName: '%2', InstanceId: '%3'.  | WFRuntime |
| 1028 - CompleteTransactionContextWorkItem | Verbose     | A TransactionContextWorkItem has completed for Activity '%1', DisplayName: '%2', InstanceId: '%3'.  | WFRuntime |
| 1029 - ScheduleFaultWorkItem              | Verbose     | A FaultWorkItem has been scheduled for Activity '%1', DisplayName: '%2', InstanceId: '%3'. The exception was propagated from Activity '%4', DisplayName: '%5', InstanceId: '%6'.    | WFRuntime |
| 1030 - StartFaultWorkItem                 | Verbose     | Starting execution of a FaultWorkItem for Activity '%1', DisplayName: '%2', InstanceId: '%3'. The exception was propagated from Activity '%4', DisplayName: '%5', InstanceId: '%6'. | WFRuntime |

| Event ID                                     | Event Level | Event Message   | Keywords     |
|--|-------------|---|--------------|
| 1031 - CompleteFaultWorkItem                 | Verbose     | A FaultWorkItem has completed for Activity '%1', DisplayName: '%2', InstanceId: '%3'. The exception was propagated from Activity '%4', DisplayName: '%5', InstanceId: '%6'. | WFRuntime    |
| 1032 - ScheduleRuntimeWorkItem               | Verbose     | A runtime work item has been scheduled for Activity '%1', DisplayName: '%2', InstanceId: '%3'.  | WFRuntime    |
| 1033 - StartRuntimeWorkItem                  | Verbose     | Starting execution of a runtime work item for Activity '%1', DisplayName: '%2', InstanceId: '%3'.   | WFRuntime    |
| 1034 - CompleteRuntimeWorkItem               | Verbose     | A runtime work item has completed for Activity '%1', DisplayName: '%2', InstanceId: '%3'.   | WFRuntime    |
| 1035 - RuntimeTransactionSet                 | Verbose     | The runtime transaction has been set by Activity '%1', DisplayName: '%2', InstanceId: '%3'. Execution isolated to Activity '%4', DisplayName: '%5', InstanceId: '%6'.       | WFRuntime    |
| 1036 - RuntimeTransactionCompletionRequested | Verbose     | Activity '%1', DisplayName: '%2', InstanceId: '%3' has scheduled completion of the runtime transaction.   | WFRuntime    |
| 1037 - RuntimeTransactionComplete            | Verbose     | The runtime transaction has completed with the state '%1'.  | WFRuntime    |
| 1038 - EnterNoPersistBlock                   | Verbose     | Entering a no persist block.  | WFRuntime    |
| 1039 - ExitNoPersistBlock                    | Verbose     | Exiting a no persist block.   | WFRuntime    |
| 1040 - InArgumentBound                       | Verbose     | In argument '%1' on Activity '%2', DisplayName: '%3', InstanceId: '%4' has been bound with value: %5.   | WFActivities |
| 1041 - WorkflowApplicationPersistenceIdle    | Information | WorkflowApplication Id: '%1' is idle and persistable. The following action will be taken: %2.   | WFRuntime    |

| Event ID                                   | Event Level | Event Message   | Keywords     |
|--|-------------|---|--------------|
| 1101 - WorkflowActivityStart               | Information | WorkflowInstance Id: '%1' E2E Activity  | WFRuntime    |
| 1102 - WorkflowActivityStop                | Information | WorkflowInstance Id: '%1' E2E Activity  | WFRuntime    |
| 1103 - WorkflowActivitySuspend             | Information | WorkflowInstance Id: '%1' E2E Activity  | WFRuntime    |
| 1104 - WorkflowActivityResume              | Information | WorkflowInstance Id: '%1' E2E Activity  | WFRuntime    |
| 1124 - InvokeMethodIsStatic                | Information | InvokeMethod '%1' - method is Static.   | WFRuntime    |
| 1125 - InvokeMethodIsNotStatic             | Information | InvokeMethod '%1' - method is not Static.   | WFRuntime    |
| 1126 - InvokedMethodThrewException         | Information | An exception was thrown in the method called by the activity '%1'. %2   | WFRuntime    |
| 1131 - InvokeMethodUseAsyncPattern         | Information | InvokeMethod '%1' - method uses asynchronous pattern of '%2' and '%3'.  | WFRuntime    |
| 1132 - InvokeMethodDoesNotUseA syncPattern | Information | InvokeMethod '%1' - method does not use asynchronous pattern.   | WFRuntime    |
| 1140 - FlowchartStart                      | Information | Flowchart '%1' - Start has been scheduled.  | WFActivities |
| 1141 - FlowchartEmpty                      | Warning     | Flowchart '%1' - was executed with no Nodes. An exception was thrown in the method called by the activity '%1'. %2                              | WFActivities |
| 1143 - FlowchartNextNull                   | Information | Flowchart '%1'/FlowStep - Next node is null. Flowchart execution will end.  | WFActivities |
| 1146 - FlowchartSwitchCase                 | Information | Flowchart '%1'/FlowSwitch - Case '%2' was selected.   | WFActivities |
| 1147 - FlowchartSwitchDefault              | Information | Flowchart '%1'/FlowSwitch - Default Case was selected.  | WFActivities |
| 1148 - FlowchartSwitchCaseNotFound         | Information | Flowchart '%1'/FlowSwitch - could find neither a Case activity nor a Default Case matching the Expression result. Flowchart execution will end. | WFActivities |

| Event ID                                   | Event Level | Event Message   | Keywords     |
|--|-------------|---|--------------|
| 1150 - CompensationState                   | Information | CompensableActivity '%1' is in the '%2' state.  | WFActivities |
| 1223 - SwitchCaseNotFound                  | Information | The Switch activity '%1' could not find a Case activity matching the Expression result.                 | WFActivities |
| 1449 - WfMessageReceived                   | Information | Message received by workflow  | WFServices   |
| 1450 - WfMessageSent                       | Information | Message sent from workflow  | WFServices   |
| 2021 - ExecuteWorkItemStart                | Verbose     | Execute work item start   | WFRuntime    |
| 2022 - ExecuteWorkItemStop                 | Verbose     | Execute work item stop  | WFRuntime    |
| 2023 - SendMessageChannelCache Miss        | Verbose     | SendMessageChannelCache miss  | WFRuntime    |
| 2024 - InternalCacheMetadataStart          | Verbose     | InternalCacheMetadata started on activity '%1'.   | WFRuntime    |
| 2025 - InternalCacheMetadataStop           | Verbose     | InternalCacheMetadata stopped on activity '%1'.   | WFRuntime    |
| 2026 - CompileVbExpressionStart            | Verbose     | Compiling VB expression '%1'  | WFRuntime    |
| 2027 - CacheRootMetadataStart              | Verbose     | CacheRootMetadata started on activity '%1'  | WFRuntime    |
| 2028 - CacheRootMetadataStop               | Verbose     | CacheRootMetadata stopped on activity %1.   | WFRuntime    |
| 2029 - CompileVbExpressionStop             | Verbose     | Finished compiling VB expression.   | WFRuntime    |
| 2576 - TryCatchExceptionFromTry            | Information | The TryCatch activity '%1' has caught an exception of type '%2'.  | WFActivities |
| 2577 - TryCatchExceptionDuringCancellation | Warning     | A child activity of the TryCatch activity '%1' has thrown an exception during cancellation.             | WFActivities |
| 2578 - TryCatchExceptionFromCatchOrFinally | Warning     | A Catch or Finally activity that is associated with the TryCatch activity '%1' has thrown an exception. | WFActivities |

| Event ID                                     | Event Level | Event Message   | Keywords         |
|--|-------------|---|------------------|
| 3501 - InferredContractDescription           | Information | ContractDescription with Name='%1' and Namespace='%2' has been inferred from WorkflowService.   | WFServices       |
| 3502 - InferredOperationDescription          | Information | OperationDescription with Name='%1' in contract '%2' has been inferred from WorkflowService. IsOneWay=%3.   | WFServices       |
| 3503 - DuplicateCorrelationQuery             | Warning     | A duplicate CorrelationQuery was found with Where='%1'. This duplicate query will not be used when calculating correlation.   | WFServices       |
| 3507 - ServiceEndpointAdded                  | Information | A service endpoint has been added for address '%1', binding '%2', and contract '%3'.  | WFServices       |
| 3508 - TrackingProfileNotFound               | Verbose     | TrackingProfile '%1' for the ActivityDefinitionId '%2' not found. Either the TrackingProfile is not found in the config file or the ActivityDefinitionId does not match.          | WFServices       |
| 3550 - BufferOutOfOrderMessageN oInstance    | Information | Operation '%1' cannot be performed at this time. Another attempt will be made when the service instance is ready to process this particular operation.                            | WFServices       |
| 3551 - BufferOutOfOrderMessageN oBookmark    | Information | Operation '%2' on service instance '%1' cannot be performed at this time. Another attempt will be made when the service instance is ready to process this particular operation.   | WFServices       |
| 3552 - MaxPendingMessagesPerCh annelExceeded | Warning     | The throttle 'MaxPendingMessagesPerCh annel' limit of '%1' was hit. To increase this limit, adjust the MaxPendingMessagesPerCh annel property on BufferedReceiveServiceBehav ior. | Quota WFServices |

| Event ID  | Event Level | Event Message   | Keywords              |
|---|-------------|---|-----------------------|
| <a href="#">3557 - TransactedReceiveScopeEnd CommitFailed</a> | Information | The call to EndCommit on the CommittableTransaction with id = '%1' threw a TransactionException with the following message: '%2'.   | WFServices            |
| <a href="#">4201 - EndSqlCommandExecute</a>                   | Verbose     | End SQL command execution: %1   | WFInstanceStore       |
| <a href="#">4202 - StartSqlCommandExecute</a>                 | Verbose     | Starting SQL command execution: %1  | WFInstanceStore       |
| <a href="#">4203 - RenewLockSystemError</a>                   | Error       | Failed to extend lock expiration, lock expiration already passed or the lock owner was deleted. Aborting SqlWorkflowInstanceStore.  | WFInstanceStore       |
| <a href="#">4205 - FoundProcessingError</a>                   | Error       | Command failed: %1  | WFInstanceStore       |
| <a href="#">4206 - UnlockInstanceException</a>                | Error       | Encountered exception %1 while attempting to unlock instance.   | WFInstanceStore       |
| <a href="#">4207 - MaximumRetriesExceededForSqlCommand</a>    | Information | Giving up retrying a SQL command as the maximum number of retries have been performed.  | Quota WFInstanceStore |
| <a href="#">4208 - RetryingSqlCommandDueToSqlError</a>        | Information | Retrying a SQL command due to SQL error number %1.  | WFInstanceStore       |
| <a href="#">4209 - TimeoutOpeningSqlConnection</a>            | Error       | Timeout trying to open a SQL connection. The operation did not complete within the allotted timeout of %1. The time allotted to this operation may have been a portion of a longer timeout. | WFInstanceStore       |
| <a href="#">4210 - SqlExceptionCaught</a>                     | Warning     | Caught SQL Exception number %1 message %2.  | WFInstanceStore       |
| <a href="#">4211 - QueuingSqlRetry</a>                        | Warning     | Queuing SQL retry with delay %1 milliseconds.   | WFInstanceStore       |

| Event ID                                | Event Level | Event Message  | Keywords        |
|---|-------------|--|-----------------|
| 4212 - LockRetryTimeout                 | Warning     | Timeout trying to acquire the instance lock. The operation did not complete within the allotted timeout of %1. The time allotted to this operation may have been a portion of a longer timeout.  | WFInstanceStore |
| 4213 - RunnableInstancesDetection Error | Error       | Detection of runnable instances failed due to the following exception  | WFInstanceStore |
| 4214 - InstanceLocksRecoveryError       | Error       | Recovering instance locks failed due to the following exception  | WFInstanceStore |
| 39456 - TrackingRecordDropped           | Warning     | Size of tracking record %1 exceeds maximum allowed by the ETW session for provider %2  | WFTtracking     |
| 39457 - TrackingRecordRaised            | Information | Tracking Record %1 raised to %2.   | WFRuntime       |
| 39458 - TrackingRecordTruncated         | Warning     | Truncated tracking record %1 written to ETW session with provider %2.<br>Variables/annotations/user data have been removed   | WFTtracking     |
| 39459 - TrackingDataExtracted           | Verbose     | Tracking data %1 extracted in activity %2.   | WFRuntime       |
| 39460 - TrackingValueNotSerializable    | Warning     | The extracted argument/variable '%1' is not serializable.  | WFTtracking     |
| 57398 - MaxInstancesExceeded            | Warning     | The system hit the limit set for throttle 'MaxConcurrentInstances'. Limit for this throttle was set to %1. Throttle value can be changed by modifying attribute 'maxConcurrentInstances' in serviceThrottle element or by modifying 'MaxConcurrentInstances' property on behavior ServiceThrottlingBehavior. | WFServices      |

# 100 - WorkflowInstanceRecord

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| Id       | 100  |
| Keywords | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTacking |
| Level    | Information  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic       |

## Description

This event is emitted by the ETW tracking participant when a workflow instance emits WorkflowInstanceRecord for workflow states : Started, Resumed, Persisted, Idle, Deleted, Completed, Canceled, Unloaded, Unsuspended.

## Message

TrackRecord= WorkflowInstanceRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3,  
ActivityDefinitionId = %4, State = %5, Annotations = %6, ProfileName = %7

## Details

| DATA ITEM NAME       | DATA ITEM TYPE | DESCRIPTION                                   |
|----------------------|----------------|---|
| InstanceId           | xs:GUID        | The instance id for the workflow              |
| RecordNumber         | xs:long        | The sequence number of the emitted record     |
| EventTime            | xs:dateTime    | The time in UTC when the event was emitted    |
| ActivityDefinitionId | xs:string      | The name of the root activity in the workflow |
| State                | xs:string      | The current state of the Workflow.            |

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Annotations    | xs:string      | The annotations that were added to this event. The values are stored in an xml element in the format <items> < item name = "annotationName" type="System.String">annotationValue </item> </items> . If no annotations are specified then the string contains <items/>. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with <items>... </items>. |
| ProfileName    | xs:string      | The name or the tracking profile that resulted in this event being emitted   |
| HostReference  | xs:string      | For web hosted services, this field uniquely identifies the service in the web hierarchy. Its format is defined as 'Web Site Name Application Virtual Path Service Virtual Path ServiceName' Example: 'Default Web Site/CalculatorApplication CalculatorService.svc CalculatorService'   |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.   |

# 101 - WorkflowInstanceUnhandledExceptionRecord

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| Id       | 101   |
| Keywords | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTracking |
| Level    | Error   |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic        |

## Description

This event is emitted by the ETW tracking participant when a workflow instance emits WorkflowInstanceUnhandledExceptionRecord.

## Message

TrackRecord = WorkflowInstanceUnhandledExceptionRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, SourceName = %5, Sourceld = %6, SourceInstanceId = %7, SourceTypeName=%8, Exception=%9, Annotations= %10, ProfileName = %11

## Details

| DATA ITEM NAME       | DATA ITEM TYPE | DESCRIPTION   |
|----------------------|----------------|---|
| Instanceld           | xs:GUID        | The instance id for the workflow  |
| RecordNumber         | xs:long        | The sequence number of the emitted record                                 |
| EventTime            | xs:dateTime    | The time in UTC when the event was emitted                                |
| ActivityDefinitionId | xs:string      | The name of the root activity in the workflow                             |
| SourceName           | xs:string      | The source activity name that faulted resulting in the unhandledException |
| Sourceld             | xs:string      | The activity id of the fault source activity                              |
| SourceInstanceId     | xs:string      | The activity instance id of the fault source activity                     |

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION   |
|----------------|----------------|---|
| SourceTypeName | xs:string      | The source activity type name that faulted resulting in the unhandledException  |
| Exception      | xs:string      | The exception details for the unhandled exception   |
| Annotations    | xs:string      | The annotations that were added to this event. The values are stored in an xml element in the format <items> < item name = "annotationName" type="System.String">annotationValue </item> </items>. If no annotations are specified then the string contains <items/>. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with <items>... </items>. |
| ProfileName    | xs:string      | The name or the tracking profile that resulted in this event being emitted  |
| HostReference  | xs:string      | For web hosted services, this field uniquely identifies the service in the web hierarchy. Its format is defined as 'Web Site Name Application Virtual Path Service Virtual Path ServiceName' Example: 'Default Web Site/CalculatorApplication CalculatorService.svc CalculatorService'  |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.  |

# 102 - WorkflowInstanceAbortedRecord

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| Id       | 102  |
| Keywords | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTacking |
| Level    | Warning  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic       |

## Description

This event is emitted by the ETW tracking participant when a workflow instance emits WorkflowInstanceAbortedRecord.

## Message

TrackRecord = WorkflowInstanceAbortedRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, Reason = %5, Annotations = %6, ProfileName = %7

## Details

| DATA ITEM NAME       | DATA ITEM TYPE | DESCRIPTION                                   |
|----------------------|----------------|---|
| InstanceId           | xs:GUID        | The instance id for the workflow              |
| RecordNumber         | xs:long        | The sequence number of the emitted record     |
| EventTime            | xs:dateTime    | The time in UTC when the event was emitted    |
| ActivityDefinitionId | xs:string      | The name of the root activity in the workflow |
| Reason               | xs:string      | The reason the workflow was aborted           |

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Annotations    | xs:string      | The annotations that were added to this event. The values are stored in an xml element in the format <items> < item name = "annotationName" type="System.String">annotationValue </item> </items> . If no annotations are specified then the string contains <items/>. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with <items>... </items>. |
| ProfileName    | xs:string      | The name or the tracking profile that resulted in this event being emitted   |
| HostReference  | xs:string      | For web hosted services, this field uniquely identifies the service in the web hierarchy. Its format is defined as 'Web Site Name Application Virtual Path Service Virtual Path ServiceName' Example: 'Default Web Site/CalculatorApplication CalculatorService.svc CalculatorService'   |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.   |

# 103 - ActivityStateRecord

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| Id       | 103   |
| Keywords | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTracking |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic        |

## Description

This event is emitted by the ETW tracking participant when a activity within a workflow instance emits ActivityStateRecord

## Message

TrackRecord = ActivityStateRecord, InstanceID = %1, RecordNumber=%2, EventTime=%3, State = %4, Name=%5, ActivityId=%6, ActivityInstanceId=%7, ActivityTypeName=%8, Arguments=%9, Variables=%10, Annotations=%11, ProfileName = %12

## Details

| DATA ITEM NAME     | DATA ITEM TYPE | DESCRIPTION   |
|--------------------|----------------|---|
| InstanceId         | xs:GUID        | The instance id for the workflow                        |
| RecordNumber       | xs:long        | The sequence number of the emitted record               |
| EventTime          | xs:dateTime    | The time in UTC when the event was emitted              |
| State              | xs:string      | The state of the activity                               |
| Name               | xs:string      | The display name of the activity that emitted the event |
| ActivityId         | xs:string      | The activity id of the emitting activity                |
| ActivityInstanceId | xs:string      | The activity instance id of the emitting activity       |
| ActivityTypeName   | xs:string      | The type name of the emitting activity                  |

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION   |
|----------------|----------------|---|
| Arguments      | xs:string      | <p>The arguments that were tracked with this event. The values are stored in an xml element in the format &lt;items&gt;&lt; item name = "argumentName" type="System.String"&gt;argumentValue&lt;/item&gt;&lt;/items&gt;. If no arguments were tracked then the string contains &lt;items/&gt;. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with &lt;items&gt;...&lt;/items&gt;. The following types are stored as their value as returned by <code>ToString()</code>; string,char,bool,int,short,long,uint,ushort,ulong,System.Single,float,double,System.Guid,System.DateTimeOffset,System.DateTime. All other types are serialized using <code>System.Runtime.Serialization.NetDataContractSerializer</code>.</p> |
| Variables      | xs:string      | <p>The variables that were tracked with this event. The values are stored in an xml element in the format &lt;items&gt;&lt; item name = "variableName" type="System.String"&gt;variableValue&lt;/item&gt;&lt;/items&gt;. If no variables were tracked then the string contains &lt;items/&gt;. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the variables value with &lt;items&gt;...&lt;/items&gt;. The following types are stored as their value as returned by <code>ToString()</code>; string,char,bool,int,short,long,uint,ushort,ulong,System.Single,float,double,System.Guid,System.DateTimeOffset,System.DateTime. All other types are serialized using <code>System.Runtime.Serialization.NetDataContractSerializer</code>.</p>  |

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Annotations    | xs:string      | The annotations that were added to this event. The values are stored in an xml element in the format <items> < item name = "annotationName" type="System.String">annotationValue </item> </items> . If no annotations are specified then the string contains <items/>. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with <items>... </items>. |
| ProfileName    | xs:string      | The name or the tracking profile that resulted in this event being emitted   |
| HostReference  | xs:string      | For web hosted services, this field uniquely identifies the service in the web hierarchy. Its format is defined as 'Web Site Name Application Virtual Path Service Virtual Path ServiceName' Example: 'Default Web Site/CalculatorApplication CalculatorService.svc CalculatorService'   |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.   |

# 104 - ActivityScheduledRecord

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| Id       | 104   |
| Keywords | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTracking |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic        |

## Description

This event is emitted by the ETW tracking participant when an activity within a workflow instance emits ActivityScheduledRecord

## Message

TrackRecord = ActivityScheduledRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, Name = %4, ActivityId = %5, ActivityInstanceId = %6, ActivityTypeName = %7, ChildActivityName = %8, ChildActivityId = %9, ChildActivityInstanceId = %10, ChildActivityTypeName = %11, Annotations=%12, ProfileName = %13

## Details

| DATA ITEM NAME     | DATA ITEM TYPE | DESCRIPTION   |
|--------------------|----------------|---|
| Instanceid         | xs:GUID        | The instance id for the workflow                                  |
| RecordNumber       | xs:long        | The sequence number of the emitted record                         |
| EventTime          | xs:dateTime    | The time in UTC when the event was emitted                        |
| Name               | xs:string      | The name of the activity that scheduled the child activity        |
| ActivityId         | xs:string      | The id of the activity that scheduled the child activity          |
| ActivityInstanceId | xs:string      | The instance id of the activity that scheduled the child activity |
| ActivityTypeName   | xs:string      | The type of the activity that requested the cancel operation      |

| DATA ITEM NAME          | DATA ITEM TYPE | DESCRIPTION  |
|-------------------------|----------------|--|
| ChildActivityName       | xs:string      | The name of the scheduled activity   |
| ChildActivityId         | xs:string      | The id of the scheduled activity   |
| ChildActivityInstanceId | xs:string      | The instance id of the scheduled activity  |
| ChildActivityTypeName   | xs:string      | The type of the scheduled activity   |
| Annotations             | xs:string      | The annotations that were added to this event. The values are stored in an xml element in the format <items> < item name = "annotationName" type="System.String">annotationValue </item> </items>. If no annotations are specified then the string contains <items/>. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with <items> ... </items>. |
| ProfileName             | xs:string      | The name or the tracking profile that resulted in this event being emitted   |
| HostReference           | xs:string      | For web hosted services, this field uniquely identifies the service in the web hierarchy. Its format is defined as 'Web Site Name Application Virtual Path Service Virtual Path ServiceName' Example: 'Default Web Site/CalculatorApplication CalculatorService.svc CalculatorService'   |
| AppDomain               | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.   |

# 105 - FaultPropagationRecord

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| Id       | 105  |
| Keywords | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTacking |
| Level    | Warning  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic       |

## Description

This event is emitted by the ETW tracking participant when a activity with the workflow instance emits FaultPropagationRecord.

## Message

TrackRecord = FaultPropagationRecord, InstanceID=%1, RecordNumber=%2, EventTime=%3, FaultSourceActivityName=%4, FaultSourceActivityId=%5, FaultSourceActivityInstanceId=%6, FaultSourceActivityTypeName=%7, FaultHandlerActivityName=%8, FaultHandlerActivityId = %9, FaultHandlerActivityInstanceId =%10, FaultHandlerActivityTypeName=%11, Fault=%12, IsFaultSource=%13, Annotations=%14, ProfileName = %15

## Details

| DATA ITEM NAME                | DATA ITEM TYPE | DESCRIPTION  |
|-------------------------------|----------------|--|
| InstanceId                    | xs:GUID        | The instance id for the workflow                       |
| RecordNumber                  | xs:long        | The sequence number of the emitted record              |
| EventTime                     | xs:dateTime    | The time in UTC when the event was emitted             |
| FaultSourceActivityName       | xs:string      | The name of activity that emitted the fault            |
| FaultSourceActivityId         | xs:string      | The id of the activity that emitted the fault          |
| FaultSourceActivityInstanceId | xs:string      | The instance id of the activity that emitted the fault |

| DATA ITEM NAME                 | DATA ITEM TYPE  | DESCRIPTION   |
|--------------------------------|-----------------|---|
| FaultSourceActivityTypeName    | xs:string       | The type of the activity that emitted the fault   |
| FaultHandlerActivityName       | xs:string       | The display name of the fault handler activity  |
| FaultHandlerActivityId         | xs:string       | The id of the fault handler activity  |
| FaultHandlerActivityInstanceId | xs:string       | The instance id of the fault handler activity   |
| FaultHandlerActivityTypeName   | xs:string       | The type of the fault handler activity  |
| Fault                          | xs:string       | The fault details   |
| IsFaultSource                  | xs:unsignedByte | Indicates if the event was emitted from the fault source  |
| Annotations                    | xs:string       | The annotations that were added to this event. The values are stored in an xml element in the format <items> < item name = "annotationName" type="System.String">annotationValue </item> </items>. If no annotations are specified then the string contains <items/>. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with <items>... </items>. |
| ProfileName                    | xs:string       | The name or the tracking profile that resulted in this event being emitted  |
| HostReference                  | xs:string       | For web hosted services, this field uniquely identifies the service in the web hierarchy. Its format is defined as 'Web Site Name Application Virtual Path Service Virtual Path ServiceName' Example: 'Default Web Site/CalculatorApplication CalculatorService.svc CalculatorService'  |
| AppDomain                      | xs:string       | The string returned by AppDomain.CurrentDomain.FriendlyName.  |

# 106 - CancelRequestRecord

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| Id       | 106  |
| Keywords | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTacking |
| Level    | Information  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic       |

## Description

This event is emitted by the ETW tracking participant when a activity within a workflow instance emits cancelrequestedrecord.

## Message

TrackRecord = CancelRequestedRecord, InstanceID=%1, RecordNumber=%2, EventTime=%3, Name=%4, ActivityId=%5, ActivityInstanceId=%6, ActivityTypeName = %7, ChildActivityName = %8, ChildActivityId = %9, ChildActivityInstanceId = %10, ChildActivityTypeName = %11, Annotations=%12, ProfileName = %13

## Details

| DATA ITEM NAME     | DATA ITEM TYPE | DESCRIPTION   |
|--------------------|----------------|---|
| InstanceId         | xs:GUID        | The instance id for the workflow                                    |
| RecordNumber       | xs:long        | The sequence number of the emitted record                           |
| EventTime          | xs:dateTime    | The time in UTC when the event was emitted                          |
| Name               | xs:string      | The name of the activity that requested the cancel operation        |
| ActivityId         | xs:string      | The id of the activity that requested the cancel operation          |
| ActivityInstanceId | xs:string      | The instance id of the activity that requested the cancel operation |
| ActivityTypeName   | xs:string      | The type of the activity that requested the cancel operation        |

| DATA ITEM NAME          | DATA ITEM TYPE | DESCRIPTION   |
|-------------------------|----------------|---|
| ChildActivityName       | xs:string      | The name of the activity being canceled   |
| ChildActivityId         | xs:string      | The id of the activity being canceled   |
| ChildActivityInstanceId | xs:string      | The instance id of the activity being canceled  |
| ChildActivityTypeName   | xs:string      | The type of the activity being canceled   |
| Annotations             | xs:string      | The annotations that were added to this event. The values are stored in an xml element in the format <items> < item name = "annotationName" type="System.String">annotationValue </item> </items>. If no annotations are specified then the string contains <items/>. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with <items>... </items>. |
| ProfileName             | xs:string      | The name or the tracking profile that resulted in this event being emitted  |
| HostReference           | xs:string      | For web hosted services, this field uniquely identifies the service in the web hierarchy. Its format is defined as 'Web Site Name Application Virtual Path Service Virtual Path ServiceName' Example: 'Default Web Site/CalculatorApplication /CalculatorService.svc CalculatorService'   |
| AppDomain               | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.  |

# 107 -- BookmarkResumptionRecord

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| Id       | 107   |
| Keywords | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTracking |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic        |

## Description

This event is emitted by the ETW tracking participant when a workflow instance emits a BookmarkResumptionRecord.

## Message

TrackRecord = BookmarkResumptionRecord, InstanceID=%1, RecordNumber=%2, EventTime=%3, Name=%4, SubInstanceId=%5, OwnerActivityName=%6, OwnerActivityId=%7, OwnerActivityInstanceId=%8, OwnerActivityTypeName=%9, Annotations=%10, ProfileName=%11

## Details

| DATA ITEM NAME          | DATA ITEM TYPE | DESCRIPTION                                |
|-------------------------|----------------|--|
| InstanceId              | xs:GUID        | The instance id for the workflow           |
| RecordNumber            | xs:long        | The sequence number of the emitted record  |
| EventTime               | xs:dateTime    | The time in UTC when the event was emitted |
| Name                    | xs:string      | The name of the bookmark that was resumed  |
| SubInstanceId           | xs:GUID        | The id of the bookmark scope               |
| OwnerActivityName       | xs:string      | The name of the bookmark activity          |
| OwnerActivityId         | xs:string      | The id of the bookmark activity            |
| OwnerActivityInstanceId | xs:string      | The instance id of the bookmark activity   |

| DATA ITEM NAME        | DATA ITEM TYPE | DESCRIPTION  |
|-----------------------|----------------|--|
| OwnerActivityTypeName | xs:string      | The type of the bookmark activity  |
| Annotations           | xs:string      | The annotations that were added to this event. The values are stored in an xml element in the format <items> < item name = "annotationName" type="System.String">annotationValue </item> </items>. If no annotations are specified then the string contains <items/>. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with <items> ... </items>. |
| ProfileName           | xs:string      | The name or the tracking profile that resulted in this event being emitted   |
| HostReference         | xs:string      | For web hosted services, this field uniquely identifies the service in the web hierarchy. Its format is defined as 'Web Site Name Application Virtual Path Service Virtual Path ServiceName' Example: 'Default Web Site/CalculatorApplication CalculatorService.svc CalculatorService'   |
| AppDomain             | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.   |

# 108 - CustomTrackingRecordInfo

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| Id       | 108  |
| Keywords | UserEvents, EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTacking |
| Level    | Information  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic                   |

## Description

This event is emitted by the ETW tracking participant when a activity within a workflow instance emits CustomTrackingRecord with level Info.

## Message

TrackRecord = CustomTrackingRecord, InstanceID = %1, RecordNumber=%2, EventTime=%3, Name=%4, ActivityName=%5, ActivityId=%6, ActivityInstanceId=%7, ActivityTypeName=%8, Data=%9, Annotations=%10, ProfileName = %11

## Details

| DATA ITEM NAME     | DATA ITEM TYPE | DESCRIPTION   |
|--------------------|----------------|---|
| InstanceId         | xs:GUID        | The instance id for the workflow                                      |
| RecordNumber       | xs:long        | The sequence number of the emitted record                             |
| EventTime          | xs:dateTime    | The time in UTC when the event was emitted                            |
| Name               | xs:string      | The name of the CustomTrackingRecord                                  |
| ActivityName       | xs:string      | The name of the activity that emitted the CustomTrackingRecord        |
| ActivityId         | xs:string      | The id of the activity that emitted the CustomTrackingRecord          |
| ActivityInstanceId | xs:string      | The instance id of the activity that emitted the CustomTrackingRecord |

| DATA ITEM NAME   | DATA ITEM TYPE | DESCRIPTION  |
|------------------|----------------|--|
| ActivityTypeName | xs:string      | The name of the activity that emitted the CustomTrackingRecord   |
| Data             | xs:string      | <p>The data that was tracked with this event. The values are stored in an xml element in the format &lt;items&gt; &lt; item name = "dataName" type="System.String"&gt;dataValue&lt;/item&gt; &lt;/items&gt;. If no data was tracked then the string contains &lt;items/&gt;. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the data value with &lt;items&gt;...&lt;/items&gt;. The following types are stored as their value as returned by ToString();</p> <p>string,char,bool,int,short,long,uint,ushort,ulong,System.Single,float,double,System.Guid,System.DateTimeOffset,System.DateTime. All other types are serialized using System.Runtime.Serialization.NetDataContractSerializer.</p> |
| Annotations      | xs:string      | <p>The annotations that were added to this event. The values are stored in an xml element in the format &lt;items&gt; &lt; item name = "annotationName" type="System.String"&gt;annotationValue&lt;/item&gt; &lt;/items&gt;. If no annotations are specified then the string contains &lt;items/&gt;. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with &lt;items&gt;...&lt;/items&gt;.</p>   |
| ProfileName      | xs:string      | The name or the tracking profile that resulted in this event being emitted   |
| HostReference    | xs:string      | For web hosted services, this field uniquely identifies the service in the web hierarchy. Its format is defined as 'Web Site Name Application Virtual Path Service Virtual Path ServiceName' Example: 'Default Web Site/CalculatorApplication CalculatorService.svc CalculatorService'   |
| AppDomain        | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.   |

# 110 - CustomTrackingRecordWarning

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| Id       | 110   |
| Keywords | UserEvents, EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTracking |
| Level    | Warning   |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic                    |

## Description

This event is emitted by the ETW tracking participant when an activity within a workflow instance emits CustomTrackingRecord with level warning

## Message

TrackRecord = CustomTrackingRecord, InstanceID = %1, RecordNumber=%2, EventTime=%3, Name=%4, ActivityName=%5, ActivityId=%6, ActivityInstanceId=%7, ActivityTypeName=%8, Data=%9, Annotations=%10, ProfileName = %11

## Details

| DATA ITEM NAME     | DATA ITEM TYPE | DESCRIPTION   |
|--------------------|----------------|---|
| Instanceid         | xs:GUID        | The instance id for the workflow                                      |
| RecordNumber       | xs:long        | The sequence number of the emitted record                             |
| EventTime          | xs:dateTime    | The time in UTC when the event was emitted                            |
| Name               | xs:string      | The name of the CustomTrackingRecord                                  |
| ActivityName       | xs:string      | The name of the activity that emitted the CustomTrackingRecord        |
| ActivityId         | xs:string      | The id of the activity that emitted the CustomTrackingRecord          |
| ActivityInstanceId | xs:string      | The instance id of the activity that emitted the CustomTrackingRecord |

| DATA ITEM NAME   | DATA ITEM TYPE | DESCRIPTION  |
|------------------|----------------|--|
| ActivityTypeName | xs:string      | The name of the activity that emitted the CustomTrackingRecord   |
| Data             | xs:string      | <p>The data that was tracked with this event. The values are stored in an xml element in the format &lt;items&gt; &lt; item name = "dataName" type="System.String"&gt;dataValue&lt;/item&gt; &lt;/items&gt;. If no data was tracked then the string contains &lt;items/&gt;. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the data value with &lt;items&gt;...&lt;/items&gt;. The following types are stored as their value as returned by ToString();</p> <p>string,char,bool,int,short,long,uint,ushort,ulong,System.Single,float,double,System.Guid,System.DateTimeOffset,System.DateTime. All other types are serialized using System.Runtime.Serialization.NetDataContractSerializer.</p> |
| Annotations      | xs:string      | <p>The annotations that were added to this event. The values are stored in an xml element in the format &lt;items&gt; &lt; item name = "annotationName" type="System.String"&gt;annotationValue&lt;/item&gt; &lt;/items&gt;. If no annotations are specified then the string contains &lt;items/&gt;. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with &lt;items&gt;...&lt;/items&gt;.</p>   |
| ProfileName      | xs:string      | The name or the tracking profile that resulted in this event being emitted   |
| HostReference    | xs:string      | For web hosted services, this field uniquely identifies the service in the web hierarchy. Its format is defined as 'Web Site Name Application Virtual Path Service Virtual Path ServiceName' Example: 'Default Web Site/CalculatorApplication CalculatorService.svc CalculatorService'   |
| AppDomain        | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.   |

# 111 - CustomTrackingRecordError

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| Id       | 111   |
| Keywords | UserEvents, EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTracking |
| Level    | Error   |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic                    |

## Description

This event is emitted by the ETW tracking participant when an activity within a workflow instance emits CustomTrackingRecord with level error.

## Message

TrackRecord = CustomTrackingRecord, InstanceID = %1, RecordNumber=%2, EventTime=%3, Name=%4, ActivityName=%5, ActivityId=%6, ActivityInstanceId=%7, ActivityTypeName=%8, Data=%9, Annotations=%10, ProfileName = %11

## Details

| DATA ITEM NAME     | DATA ITEM TYPE | DESCRIPTION   |
|--------------------|----------------|---|
| Instanceid         | xs:GUID        | The instance id for the workflow                                      |
| RecordNumber       | xs:long        | The sequence number of the emitted record                             |
| EventTime          | xs:dateTime    | The time in UTC when the event was emitted                            |
| Name               | xs:string      | The name of the CustomTrackingRecord                                  |
| ActivityName       | xs:string      | The name of the activity that emitted the CustomTrackingRecord        |
| ActivityId         | xs:string      | The id of the activity that emitted the CustomTrackingRecord          |
| ActivityInstanceId | xs:string      | The instance id of the activity that emitted the CustomTrackingRecord |

| DATA ITEM NAME   | DATA ITEM TYPE | DESCRIPTION  |
|------------------|----------------|--|
| ActivityTypeName | xs:string      | The name of the activity that emitted the CustomTrackingRecord   |
| Data             | xs:string      | <p>The data that was tracked with this event. The values are stored in an xml element in the format &lt;items&gt; &lt; item name = "dataName" type="System.String"&gt;dataValue&lt;/item&gt; &lt;/items&gt;. If no data was tracked then the string contains &lt;items/&gt;. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the data value with &lt;items&gt;...&lt;/items&gt;. The following types are stored as their value as returned by ToString();</p> <p>string,char,bool,int,short,long,uint,ushort,ulong,System.Single,float,double,System.Guid,System.DateTimeOffset,System.DateTime. All other types are serialized using System.Runtime.Serialization.NetDataContractSerializer.</p> |
| Annotations      | xs:string      | <p>The annotations that were added to this event. The values are stored in an xml element in the format &lt;items&gt; &lt; item name = "annotationName" type="System.String"&gt;annotationValue&lt;/item&gt; &lt;/items&gt;. If no annotations are specified then the string contains &lt;items/&gt;. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with &lt;items&gt;...&lt;/items&gt;.</p>   |
| ProfileName      | xs:string      | The name or the tracking profile that resulted in this event being emitted   |
| HostReference    | xs:string      | For web hosted services, this field uniquely identifies the service in the web hierarchy. Its format is defined as 'Web Site Name Application Virtual Path Service Virtual Path ServiceName' Example: 'Default Web Site/CalculatorApplication CalculatorService.svc CalculatorService'   |
| AppDomain        | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.   |

# 112 - WorkflowInstanceSuspendedRecord

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| Id       | 112  |
| Keywords | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTacking |
| Level    | Information  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic       |

## Description

This event is emitted by the ETW tracking participant when a workflow instance emits WorkflowInstanceSuspended Record.

## Message

TrackRecord = WorkflowInstanceSuspendedRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, Reason = %5, Annotations = %6, ProfileName = %7

## Details

| DATA ITEM NAME       | DATA ITEM TYPE | DESCRIPTION                                   |
|----------------------|----------------|---|
| InstanceId           | xs:GUID        | The instance id for the workflow              |
| RecordNumber         | xs:long        | The sequence number of the emitted record     |
| EventTime            | xs:dateTime    | The time in UTC when the event was emitted    |
| ActivityDefinitionId | xs:string      | The name of the root activity in the workflow |
| Reason               | xs:string      | The reason the workflow was suspended         |

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Annotations    | xs:string      | The annotations that were added to this event. The values are stored in an xml element in the format <items> < item name = "annotationName" type="System.String">annotationValue </item> </items> . If no annotations are specified then the string contains <items/>. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with <items>... </items>. |
| ProfileName    | xs:string      | The name or the tracking profile that resulted in this event being emitted   |
| HostReference  | xs:string      | For web hosted services, this field uniquely identifies the service in the web hierarchy. Its format is defined as 'Web Site Name Application Virtual Path Service Virtual Path ServiceName' Example: 'Default Web Site/CalculatorApplication CalculatorService.svc CalculatorService'   |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.   |

# 113 - WorkflowInstanceTerminatedRecord

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| Id       | 113  |
| Keywords | EndToEndMonitoring, Troubleshooting, HealthMonitoring, WFTacking |
| Level    | Error  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic       |

## Description

This event is emitted by the ETW tracking participant when a workflow instance emits WorkflowInstanceTerminatedRecord.

## Message

TrackRecord = WorkflowInstanceTerminatedRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, Reason = %5, Annotations = %6, ProfileName = %7

## Details

| DATA ITEM NAME       | DATA ITEM TYPE | DESCRIPTION                                   |
|----------------------|----------------|---|
| InstanceId           | xs:GUID        | The instance id for the workflow              |
| RecordNumber         | xs:long        | The sequence number of the emitted record     |
| EventTime            | xs:dateTime    | The time in UTC when the event was emitted    |
| ActivityDefinitionId | xs:string      | The name of the root activity in the workflow |
| Reason               | xs:string      | The reason the workflow was terminated        |

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Annotations    | xs:string      | The annotations that were added to this event. The values are stored in an xml element in the format <items> < item name = "annotationName" type="System.String">annotationValue </item> </items> . If no annotations are specified then the string contains <items/>. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with <items>... </items>. |
| ProfileName    | xs:string      | The name or the tracking profile that resulted in this event being emitted   |
| HostReference  | xs:string      | For web hosted services, this field uniquely identifies the service in the web hierarchy. Its format is defined as 'Web Site Name Application Virtual Path Service Virtual Path ServiceName' Example: 'Default Web Site/CalculatorApplication CalculatorService.svc CalculatorService'   |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.   |

# 114 - WorkflowInstanceRecordWithId

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 114  |
| Keywords | HealthMonitoring, WFTtracking                              |
| Level    | Information  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

This event is emitted by the ETW tracking participant when a workflow instance emits WorkflowInstanceRecord for workflow states : Started, Resumed, Persisted, Idle, Deleted, Completed, Canceled, Unloaded, Unsuspended.

## Message

TrackRecord= WorkflowInstanceRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, State = %5, Annotations = %6, ProfileName = %7, WorkflowDefinitionIdentity = %8

## Details

| DATA ITEM NAME       | DATA ITEM TYPE | DESCRIPTION                                   |
|----------------------|----------------|---|
| Instanceld           | xs:GUID        | The instance id for the workflow              |
| RecordNumber         | xs:long        | The sequence number of the emitted record     |
| EventTime            | xs:dateTime    | The time in UTC when the event was emitted    |
| ActivityDefinitionId | xs:string      | The name of the root activity in the workflow |
| State                | xs:string      | The current state of the Workflow.            |

| DATA ITEM NAME             | DATA ITEM TYPE | DESCRIPTION  |
|----------------------------|----------------|--|
| Annotations                | xs:string      | The annotations that were added to this event. The values are stored in an xml element in the format <items> < item name = "annotationName" type="System.String">annotationValue </item> </items> . If no annotations are specified then the string contains <items/>. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with <items>... </items>. |
| ProfileName                | xs:string      | The name or the tracking profile that resulted in this event being emitted   |
| WorkflowDefinitionIdentity | xs:string      | The workflow definition id   |
| AppDomain                  | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.   |

# 115 - WorkflowInstanceAbortedRecordWithId

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 115  |
| Keywords | HealthMonitoring, WFTracking                               |
| Level    | Warning  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

This event is emitted by the ETW tracking participant when a workflow instance emits WorkflowInstanceAbortedRecord.

## Message

TrackRecord = WorkflowInstanceAbortedRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, Reason = %5, Annotations = %6, ProfileName = %7, WorkflowDefinitionIdentity = %8

## Details

| DATA ITEM NAME       | DATA ITEM TYPE | DESCRIPTION                                   |
|----------------------|----------------|---|
| InstanceId           | xs:GUID        | The instance id for the workflow              |
| RecordNumber         | xs:long        | The sequence number of the emitted record     |
| EventTime            | xs:dateTime    | The time in UTC when the event was emitted    |
| ActivityDefinitionId | xs:string      | The name of the root activity in the workflow |
| State                | xs:string      | The current state of the Workflow.            |

| DATA ITEM NAME             | DATA ITEM TYPE | DESCRIPTION  |
|----------------------------|----------------|--|
| Annotations                | xs:string      | The annotations that were added to this event. The values are stored in an xml element in the format <items> < item name = "annotationName" type="System.String">annotationValue </item> </items> . If no annotations are specified then the string contains <items/>. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with <items>... </items>. |
| ProfileName                | xs:string      | The name or the tracking profile that resulted in this event being emitted   |
| WorkflowDefinitionIdentity | xs:string      | The workflow definition id   |
| AppDomain                  | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.   |

# 116 - WorkflowInstanceSuspendedRecordWithId

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 116  |
| Keywords | HealthMonitoring, WFTracking                               |
| Level    | Information  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

This event is emitted by the ETW tracking participant when a workflow instance emits WorkflowInstanceSuspended Record.

## Message

TrackRecord = WorkflowInstanceSuspendedRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, Reason = %5, Annotations = %6, ProfileName = %7, WorkflowDefinitionIdentity = %8

## Details

| DATA ITEM NAME       | DATA ITEM TYPE | DESCRIPTION                                   |
|----------------------|----------------|---|
| InstanceId           | xs:GUID        | The instance id for the workflow              |
| RecordNumber         | xs:long        | The sequence number of the emitted record     |
| EventTime            | xs:dateTime    | The time in UTC when the event was emitted    |
| ActivityDefinitionId | xs:string      | The name of the root activity in the workflow |
| State                | xs:string      | The current state of the Workflow.            |

| DATA ITEM NAME             | DATA ITEM TYPE | DESCRIPTION  |
|----------------------------|----------------|--|
| Annotations                | xs:string      | The annotations that were added to this event. The values are stored in an xml element in the format <items> < item name = "annotationName" type="System.String">annotationValue </item> </items> . If no annotations are specified then the string contains <items/>. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with <items>... </items>. |
| ProfileName                | xs:string      | The name or the tracking profile that resulted in this event being emitted   |
| WorkflowDefinitionIdentity | xs:string      | The workflow definition id   |
| AppDomain                  | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.   |

# 117 - WorkflowInstanceTerminatedRecordWithId

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 117  |
| Keywords | HealthMonitoring, WFTracking                               |
| Level    | Error  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

This event is emitted by the ETW tracking participant when a workflow instance emits WorkflowInstanceTerminatedRecord.

## Message

TrackRecord = WorkflowInstanceTerminatedRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, Reason = %5, Annotations = %6, ProfileName = %7, WorkflowDefinitionIdentity = %8

## Details

| DATA ITEM NAME       | DATA ITEM TYPE | DESCRIPTION                                   |
|----------------------|----------------|---|
| InstanceId           | xs:GUID        | The instance id for the workflow              |
| RecordNumber         | xs:long        | The sequence number of the emitted record     |
| EventTime            | xs:dateTime    | The time in UTC when the event was emitted    |
| ActivityDefinitionId | xs:string      | The name of the root activity in the workflow |
| State                | xs:string      | The current state of the Workflow.            |

| DATA ITEM NAME             | DATA ITEM TYPE | DESCRIPTION  |
|----------------------------|----------------|--|
| Annotations                | xs:string      | The annotations that were added to this event. The values are stored in an xml element in the format <items> < item name = "annotationName" type="System.String">annotationValue </item> </items> . If no annotations are specified then the string contains <items/>. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with <items>... </items>. |
| ProfileName                | xs:string      | The name or the tracking profile that resulted in this event being emitted   |
| WorkflowDefinitionIdentity | xs:string      | The workflow definition id   |
| AppDomain                  | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.   |

118 -

## WorkflowInstanceUnhandledExceptionRecordWithId

5/4/2018 • 2 minutes to read • [Edit Online](#)

### Properties

|          |  |
|----------|--|
| ID       | 118  |
| Keywords | HealthMonitoring, WFTtracking                              |
| Level    | Error  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

### Description

This event is emitted by the ETW tracking participant when a workflow instance emits WorkflowInstanceUnhandledExceptionRecord.

### Message

TrackRecord = WorkflowInstanceUnhandledExceptionRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, SourceName = %5, Sourceld = %6, SourceInstanceId = %7, SourceTypeName=%8, Exception=%9, Annotations= %10, ProfileName = %11, WorkflowDefinitionIdentity = %12

### Details

| DATA ITEM NAME       | DATA ITEM TYPE | DESCRIPTION   |
|----------------------|----------------|---|
| Instanceld           | xs:GUID        | The instance id for the workflow  |
| RecordNumber         | xs:long        | The sequence number of the emitted record                                 |
| EventTime            | xs:dateTime    | The time in UTC when the event was emitted                                |
| ActivityDefinitionId | xs:string      | The name of the root activity in the workflow                             |
| SourceName           | xs:string      | The source activity name that faulted resulting in the unhandledException |
| Sourceld             | xs:string      | The activity id of the fault source activity                              |

| DATA ITEM NAME             | DATA ITEM TYPE | DESCRIPTION   |
|----------------------------|----------------|---|
| SourceInstanceId           | xs:string      | The activity instance id of the fault source activity   |
| SourceTypeName             | xs:string      | The source activity type name that faulted resulting in the unhandledException  |
| Exception                  | xs:string      | The exception details for the unhandled exception   |
| State                      | xs:string      | The current state of the Workflow.  |
| Annotations                | xs:string      | The annotations that were added to this event. The values are stored in an xml element in the format <items> < item name = "annotationName" type="System.String">annotationValue </item> </items>. If no annotations are specified then the string contains <items/>. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with <items>... </items>. |
| ProfileName                | xs:string      | The name or the tracking profile that resulted in this event being emitted  |
| WorkflowDefinitionIdentity | xs:string      | The workflow definition id  |
| AppDomain                  | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.  |

# 119 - WorkflowInstanceUpdatedRecord

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 119  |
| Keywords | HealthMonitoring, WFTtracking                              |
| Level    | Information  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

This event is emitted by the ETW tracking participant when a workflow instance is updated.

## Message

TrackRecord= WorkflowInstanceUpdatedRecord, InstanceID = %1, RecordNumber = %2, EventTime = %3, ActivityDefinitionId = %4, State = %5, OriginalDefinitionIdentity = %6, UpdatedDefinitionIdentity = %7, Annotations = %8, ProfileName = %9

## Details

| DATA ITEM NAME             | DATA ITEM TYPE | DESCRIPTION                                   |
|----------------------------|----------------|---|
| InstanceId                 | xs:GUID        | The instance id for the workflow              |
| RecordNumber               | xs:long        | The sequence number of the emitted record     |
| EventTime                  | xs:dateTime    | The time in UTC when the event was emitted    |
| ActivityDefinitionId       | xs:string      | The name of the root activity in the workflow |
| State                      | xs:string      | The current state of the Workflow.            |
| OriginalDefinitionIdentity | xs:string      | The original workflow definition id           |
| UpdatedDefinitionIdentity  | xs:string      | The updated workflow definition id            |

| DATA ITEM NAME             | DATA ITEM TYPE | DESCRIPTION  |
|----------------------------|----------------|--|
| Annotations                | xs:string      | The annotations that were added to this event. The values are stored in an xml element in the format <items> < item name = "annotationName" type="System.String">annotationValue </item> </items> . If no annotations are specified then the string contains <items/>. The ETW event size is limited by the ETW buffer size or the max payload for an ETW event. If the size of the event exceeds the ETW limits, then the event is truncated by dropping the annotations and replacing the annotation value with <items>... </items>. |
| ProfileName                | xs:string      | The name or the tracking profile that resulted in this event being emitted   |
| WorkflowDefinitionIdentity | xs:string      | The workflow definition id   |
| AppDomain                  | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.   |

# 225 - TraceCorrelationKeys

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 225  |
| Keywords | Troubleshooting, ServiceModel                              |
| Level    | Information  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

This event is emitted when content-based correlation is used for a workflow service. It contains the correlation keys that are applied to correlate a message to an instance.

## Message

Calculated correlation key '%1' using values '%2' in parent scope '%3'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Instance Key   | xs:GUID        | The key that was generated from the correlation values.  |
| Values         | xs:string      | The values that were used to compute the correlation instance key.   |
| Parent Scope   | xs:string      |  |
| HostReference  | xs:string      | For Web hosted services, this field uniquely identifies the service in the Web hierarchy. Its format is defined as 'Web Site Name Application Virtual Path Service Virtual Path ServiceName'. Example: 'Default Web Site/CalculatorApplication CalculatorService.svc CalculatorService'. |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.   |

# 440 - StartSignpostEvent1

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 440  |
| Keywords | Troubleshooting  |
| Level    | Information  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

In activity tracing, indicates a message has started crossing an activity boundary in send or receive.

## Message

Activity boundary.

## Details

| DATA ITEM NAME | DATA ITEM TYPE         | DESCRIPTION  |
|----------------|------------------------|--|
| ExtendedData   | <code>xs:string</code> | The name of the activity.                                    |
| AppDomain      | <code>xs:string</code> | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 441- StopSignpostEvent1

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 441  |
| Keywords | Troubleshooting  |
| Level    | Information  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

In activity tracing, indicates a message has completed crossing an activity boundary in send or receive.

## Message

Activity boundary.

## Details

| DATA ITEM NAME | DATA ITEM TYPE         | DESCRIPTION  |
|----------------|------------------------|--|
| Extended Data  | <code>xs:string</code> | The name of the activity.                                    |
| AppDomain      | <code>xs:string</code> | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1001 - WorkflowApplicationCompleted

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1001  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a workflow application has completed in the Closed state.

## Message

WorkflowInstanceId: '%1' has completed in the Closed state.

## Details

| DATA ITEM NAME     | DATA ITEM TYPE | DESCRIPTION  |
|--------------------|----------------|--|
| WorkflowInstanceId | xs:string      | The instance id for the workflow                             |
| AppDomain          | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1002 - WorkflowApplicationTerminated

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1002  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a workflow application has terminated in the Faulted state with an exception.

## Message

WorkflowApplication Id: '%1' was terminated. It has completed in the Faulted state with an exception.

## Details

| DATA ITEM NAME        | DATA ITEM TYPE | DESCRIPTION  |
|-----------------------|----------------|--|
| WorkflowApplicationId | xs:string      | The workflow application id                                  |
| Exception             | xs:string      | The exception details for the exception                      |
| AppDomain             | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1003 - WorkflowInstanceCanceled

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1003  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a workflow instance has completed in the Canceled state.

## Message

WorkflowInstanceId: '%1' has completed in the Canceled state.

## Details

| DATA ITEM NAME     | DATA ITEM TYPE | DESCRIPTION  |
|--------------------|----------------|--|
| WorkflowInstanceId | xs:string      | The instance id for the workflow                             |
| AppDomain          | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1004 - WorkflowInstanceAborted

3/6/2019 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1004  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a workflow instance has aborted with an exception.

## Message

WorkflowInstanceId: '%1' was aborted with an exception.

## Details

| DATA ITEM NAME     | DATA ITEM TYPE | DESCRIPTION  |
|--------------------|----------------|--|
| WorkflowInstanceId | xs:string      | The instance id for the workflow                             |
| Exception          | xs:string      | The exception details for the exception                      |
| AppDomain          | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1005 - WorkflowApplicationIdled

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1005  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a workflow application has idled.

## Message

WorkflowApplication Id: '%1' went idle.

## Details

| DATA ITEM NAME     | DATA ITEM TYPE | DESCRIPTION  |
|--------------------|----------------|--|
| WorkflowInstanceId | xs:string      | The workflow application id                                  |
| AppDomain          | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1006 - WorkflowApplicationUnhandledException

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1006  |
| Keywords | WFRuntime   |
| Level    | Error   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a workflow application has encountered an unhandled exception.

## Message

WorkflowInstanceId: '%1' has encountered an unhandled exception. The exception originated from Activity '%2', DisplayName: '%3'. The following action will be taken: %4.

## Details

| DATA ITEM NAME     | DATA ITEM TYPE | DESCRIPTION  |
|--------------------|----------------|--|
| WorkflowInstanceId | xs:string      | The instance id for the workflow                             |
| Exception          | xs:string      | The exception details for the exception                      |
| AppDomain          | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1007 - WorkflowApplicationPersisted

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1007  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a workflow application has persisted.

## Message

WorkflowApplication Id: '%1' was Persisted.

## Details

| DATA ITEM NAME        | DATA ITEM TYPE | DESCRIPTION  |
|-----------------------|----------------|--|
| WorkflowApplicationId | xs:string      | The workflow application id                                  |
| AppDomain             | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1008 - WorkflowApplicationUnloaded

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1008  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a workflow application has unloaded.

## Message

WorkflowInstanceId: '%1' was Unloaded.

## Details

| DATA ITEM NAME     | DATA ITEM TYPE | DESCRIPTION  |
|--------------------|----------------|--|
| WorkflowInstanceId | xs:string      | The instance id for the workflow                             |
| AppDomain          | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1009 - ActivityScheduled

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1009  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates an activity is being scheduled for execution.

## Message

Parent Activity '%1', DisplayName: '%2', InstanceId: '%3' scheduled child Activity '%4', DisplayName: '%5', InstanceId: '%6'.

## Details

| DATA ITEM NAME    | DATA ITEM TYPE | DESCRIPTION  |
|-------------------|----------------|--|
| ParentActivity    | xs:string      | The type name of the parent activity.                        |
| ParentDisplayName | xs:string      | The display name of the parent activity.                     |
| ParentInstanceId  | xs:string      | The instance id of the parent activity.                      |
| ChildActivity     | xs:string      | The type name of the scheduled child activity.               |
| ChildDisplayName  | xs:string      | The display name of the scheduled child activity.            |
| ChildInstanceId   | xs:string      | The instance id of the scheduled child activity.             |
| AppDomain         | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1010 - ActivityCompleted

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1010  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates an activity has completed execution.

## Message

Activity '%1', DisplayName: '%2', InstanceId: '%3' has completed in the '%4' state.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| State          | xs:string      | The state of the activity.                                   |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1011 - ScheduleExecuteActivityWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1011  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates an ExecuteActivityWorkItem has been scheduled.

## Message

An ExecuteActivityWorkItem has been scheduled for Activity '%1', DisplayName: '%2', InstanceId: '%3'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1012 - StartExecuteActivityWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1012  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates an ExecuteActivityWorkItem is starting execution.

## Message

Starting execution of an ExecuteActivityWorkItem for Activity '%1', DisplayName: '%2', InstanceId: '%3'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1013 - CompleteExecuteActivityWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1013  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates an ExecuteActivityWorkItem has completed

## Message

An ExecuteActivityWorkItem has completed for Activity '%1', DisplayName: '%2', InstanceId: '%3'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1014 - ScheduleCompletionWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1014  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a CompletionWorkItem has been scheduled.

## Message

A CompletionWorkItem has been scheduled for parent Activity '%1', DisplayName: '%2', InstanceId: '%3'. Completed Activity '%4', DisplayName: '%5', InstanceId: '%6'.

## Details

| DATA ITEM NAME               | DATA ITEM TYPE | DESCRIPTION  |
|------------------------------|----------------|--|
| ParentActivity               | xs:string      | The type name of the parent activity.                        |
| ParentDisplayName            | xs:string      | The display name of the parent activity.                     |
| ParentInstanceId             | xs:string      | The instance id of the parent activity.                      |
| CompletedActivity            | xs:string      | The type name of the completed activity.                     |
| CompletedActivityDisplayName | xs:string      | The display name of the completed activity.                  |
| CompletedActivityInstanceId  | xs:string      | The instance id of the completed activity.                   |
| AppDomain                    | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1015 - StartCompletionWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1015  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a CompletionWorkItem is starting execution.

## Message

Starting execution of a CompletionWorkItem for parent Activity '%1', DisplayName: '%2', InstanceId: '%3'. Completed Activity '%4', DisplayName: '%5', InstanceId: '%6'.

## Details

| DATA ITEM NAME               | DATA ITEM TYPE | DESCRIPTION  |
|------------------------------|----------------|--|
| ParentActivity               | xs:string      | The type name of the parent activity.                        |
| ParentDisplayName            | xs:string      | The display name of the parent activity.                     |
| ParentInstanceId             | xs:string      | The instance id of the parent activity.                      |
| CompletedActivity            | xs:string      | The type name of the completed activity.                     |
| CompletedActivityDisplayName | xs:string      | The display name of the completed activity.                  |
| CompletedActivityInstanceId  | xs:string      | The instance id of the completed activity.                   |
| AppDomain                    | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1016 - CompleteCompletionWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1016  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a CompletionWorkItem has completed.

## Message

A CompletionWorkItem has completed for parent Activity '%1', DisplayName: '%2', InstanceId: '%3'. Completed Activity '%4', DisplayName: '%5', InstanceId: '%6'.

## Details

| DATA ITEM NAME               | DATA ITEM TYPE | DESCRIPTION  |
|------------------------------|----------------|--|
| ParentActivity               | xs:string      | The type name of the parent activity.                        |
| ParentDisplayName            | xs:string      | The display name of the parent activity.                     |
| ParentInstanceId             | xs:string      | The instance id of the parent activity.                      |
| CompletedActivity            | xs:string      | The type name of the completed activity.                     |
| CompletedActivityDisplayName | xs:string      | The display name of the completed activity.                  |
| CompletedActivityInstanceId  | xs:string      | The instance id of the completed activity.                   |
| AppDomain                    | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1017 - ScheduleCancelActivityWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1017  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a CancelActivityWorkItem has been scheduled.

## Message

A CancelActivityWorkItem has been scheduled for Activity '%1', DisplayName: '%2', InstanceId: '%3'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1018 - StartCancelActivityWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1018  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a CancelActivityWorkItem is starting execution.

## Message

Starting execution of a CancelActivityWorkItem for Activity '%1', DisplayName: '%2', InstanceId: '%3'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1019 - CompleteCancelActivityWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1019  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a CancelActivityWorkItem has completed.

## Message

A CancelActivityWorkItem has completed for Activity '%1', DisplayName: '%2', InstanceId: '%3'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1020 - CreateBookmark

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1020  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a bookmark has been created for an activity.

## Message

A Bookmark has been created for Activity '%1', DisplayName: '%2', InstanceId: '%3'. BookmarkName: %4, BookmarkScope: %5.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| BookmarkName   | xs:string      | The name of the bookmark.                                    |
| BookmarkScope  | xs:string      | The scope of the bookmark.                                   |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1021 - ScheduleBookmarkWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1021  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a BookmarkWorkItem has been scheduled.

## Message

A BookmarkWorkItem has been scheduled for Activity '%1', DisplayName: '%2', InstanceId: '%3'. BookmarkName: %4, BookmarkScope: %5.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| BookmarkName   | xs:string      | The name of the bookmark.                                    |
| BookmarkScope  | xs:string      | The scope of the bookmark.                                   |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1022 - StartBookmarkWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1022  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a BookmarkWorkItem is starting execution.

## Message

Starting execution of a BookmarkWorkItem for Activity '%1', DisplayName: '%2', InstanceId: '%3'. BookmarkName: %4, BookmarkScope: %5.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| BookmarkName   | xs:string      | The name of the bookmark.                                    |
| BookmarkScope  | xs:string      | The scope of the bookmark.                                   |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1023 - CompleteBookmarkWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1023  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a BookmarkWorkItem has completed.

## Message

A BookmarkWorkItem has completed for Activity '%1', DisplayName: '%2', InstanceId: '%3'. BookmarkName: %4, BookmarkScope: %5.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| BookmarkName   | xs:string      | The name of the bookmark.                                    |
| BookmarkScope  | xs:string      | The scope of the bookmark.                                   |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1024 - CreateBookmarkScope

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1024  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a BookmarkScope has been created.

## Message

A BookmarkScope has been created: %1.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| BookmarkScope  | xs:string      | The scope of the bookmark.                                   |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1025 - BookmarkScopeInitialized

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1025  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a BookmarkScope has been initialized.

## Message

The BookmarkScope that had TemporaryId: '%1' has been initialized with Id: '%2'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| TemporaryId    | xs:string      | The temporary id of the bookmark.                            |
| InitializedId  | xs:string      | The initialized id of the bookmark.                          |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1026 - ScheduleTransactionContextWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1026  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a TransactionContextWorkItem has been scheduled.

## Message

A TransactionContextWorkItem has been scheduled for Activity '%1', DisplayName: '%2', InstanceId: '%3'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1027 - StartTransactionContextWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1027  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a TransactionContextWorkItem is starting execution.

## Message

Starting execution of a TransactionContextWorkItem for Activity '%1', DisplayName: '%2', InstanceId: '%3'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1028 - CompleteTransactionContextWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1028  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a TransactionContextWorkItem has completed.

## Message

A TransactionContextWorkItem has completed for Activity '%1', DisplayName: '%2', InstanceId: '%3'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1029 - ScheduleFaultWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1029  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a FaultWorkItem has been scheduled.

## Message

A FaultWorkItem has been scheduled for Activity '%1', DisplayName: '%2', InstanceId: '%3'. The exception was propagated from Activity '%4', DisplayName: '%5', InstanceId: '%6'.

## Details

| DATA ITEM NAME               | DATA ITEM TYPE | DESCRIPTION  |
|------------------------------|----------------|--|
| FaultActivity                | xs:string      | The type name of the fault activity.                         |
| FaultActivityDisplayName     | xs:string      | The display name of the fault activity.                      |
| FaultActivityInstanceId      | xs:string      | The instance id of the fault activity.                       |
| ExceptionActivity            | xs:string      | The type name of the activity that threw the exception.      |
| ExceptionActivityDisplayName | xs:string      | The display name of the activity that threw the exception.   |
| ExceptionActivityInstanceId  | xs:string      | The instance id of the activity that threw the exception.    |
| Exception                    | xs:string      | The exception details for the exception                      |
| AppDomain                    | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1030 - StartFaultWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1030  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a FaultWorkItem is starting execution.

## Message

Starting execution of a FaultWorkItem for Activity '%1', DisplayName: '%2', InstanceId: '%3'. The exception was propagated from Activity '%4', DisplayName: '%5', InstanceId: '%6'.

## Details

| DATA ITEM NAME               | DATA ITEM TYPE | DESCRIPTION  |
|------------------------------|----------------|--|
| FaultActivity                | xs:string      | The type name of the fault activity.                         |
| FaultActivityDisplayName     | xs:string      | The display name of the fault activity.                      |
| FaultActivityInstanceId      | xs:string      | The instance id of the fault activity.                       |
| ExceptionActivity            | xs:string      | The type name of the activity that threw the exception.      |
| ExceptionActivityDisplayName | xs:string      | The display name of the activity that threw the exception.   |
| ExceptionActivityInstanceId  | xs:string      | The instance id of the activity that threw the exception.    |
| Exception                    | xs:string      | The exception details for the exception                      |
| AppDomain                    | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1031 - CompleteFaultWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1031  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a FaultWorkItem has completed.

## Message

A FaultWorkItem has completed for Activity '%1', DisplayName: '%2', InstanceId: '%3'. The exception was propagated from Activity '%4', DisplayName: '%5', InstanceId: '%6'.

## Details

| DATA ITEM NAME               | DATA ITEM TYPE | DESCRIPTION  |
|------------------------------|----------------|--|
| FaultActivity                | xs:string      | The type name of the fault activity.                         |
| FaultActivityDisplayName     | xs:string      | The display name of the fault activity.                      |
| FaultActivityInstanceId      | xs:string      | The instance id of the fault activity.                       |
| ExceptionActivity            | xs:string      | The type name of the activity that threw the exception.      |
| ExceptionActivityDisplayName | xs:string      | The display name of the activity that threw the exception.   |
| ExceptionActivityInstanceId  | xs:string      | The instance id of the activity that threw the exception.    |
| Exception                    | xs:string      | The exception details for the exception                      |
| AppDomain                    | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1032 - ScheduleRuntimeWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1032  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a RuntimeWorkItem has been scheduled.

## Message

A runtime work item has been scheduled for Activity '%1', DisplayName: '%2', InstanceId: '%3'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1033 - StartRuntimeWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1033  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a RuntimeWorkItem is starting execution.

## Message

Starting execution of a runtime work item for Activity '%1', DisplayName: '%2', InstanceId: '%3'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1034 - CompleteRuntimeWorkItem

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1034  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a RuntimeWorkItem has completed.

## Message

A runtime work item has completed for Activity '%1', DisplayName: '%2', InstanceId: '%3'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1035 - RuntimeTransactionSet

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1035  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates an activity has set the runtime transaction.

## Message

The runtime transaction has been set by Activity '%1', DisplayName: '%2', InstanceId: '%3'. Execution isolated to Activity '%4', DisplayName: '%5', InstanceId: '%6'.

## Details

| DATA ITEM NAME              | DATA ITEM TYPE | DESCRIPTION   |
|-----------------------------|----------------|---|
| Activity                    | xs:string      | The type name of the activity.  |
| DisplayName                 | xs:string      | The display name of the activity.                                     |
| InstanceId                  | xs:string      | The instance id of the activity.                                      |
| IsolatedActivity            | xs:string      | The type name of the activity that the transaction is isolated to.    |
| IsolatedActivityDisplayName | xs:string      | The display name of the activity that the transaction is isolated to. |
| IsolatedActivityInstanceId  | xs:string      | The instance id of the activity that the transaction is isolated to.  |
| AppDomain                   | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName.          |

# 1036 - RuntimeTransactionCompletionRequested

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1036  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates an activity has scheduled the completion of the runtime transaction.

## Message

Activity '%1', DisplayName: '%2', InstanceId: '%3' has scheduled completion of the runtime transaction.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1037 - RuntimeTransactionComplete

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1037  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates the runtime transaction has completed.

## Message

The runtime transaction has completed with the state '%1'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| State          | xs:string      | The state of the transaction.                                |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1038 - EnterNoPersistBlock

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1038  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a no persist block has been entered.

## Message

Entering a no persist block.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1039 - ExitNoPersistBlock

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1039  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a no persist block has been exited.

## Message

Exiting a no persist block.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1040 - InArgumentBound

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1040  |
| Keywords | WFActivities  |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates an In argument has been bound.

## Message

In argument '%1' on Activity '%2', DisplayName: '%3', InstanceId: '%4' has been bound with value: %5.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| InArgument     | xs:string      | The name of the InArgument.                                  |
| Activity       | xs:string      | The type name of the activity.                               |
| DisplayName    | xs:string      | The display name of the activity.                            |
| InstanceId     | xs:string      | The instance id of the activity.                             |
| Value          | xs:string      | The value bound to the InArgument.                           |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1041 - WorkflowApplicationPersistableIdle

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1041  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates that a workflow application is idle and persistable. The workflow application will be idled or persisted.

## Message

WorkflowApplication Id: '%1' is idle and persistable. The following action will be taken: %2.

## Details

| DATA ITEM NAME        | DATA ITEM TYPE | DESCRIPTION  |
|-----------------------|----------------|--|
| WorkflowApplicationId | xs:string      | The workflow application id                                  |
| ActionTaken           | xs:string      | The action that will be taken on the workflow application.   |
| AppDomain             | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1101 - WorkflowActivityStart

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1101  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a workflow activity has started.

## Message

WorkflowInstanceId: '%1' E2E Activity

## Details

| DATA ITEM NAME     | DATA ITEM TYPE | DESCRIPTION  |
|--------------------|----------------|--|
| WorkflowInstanceId | xs:string      | The workflow instance id.                                    |
| AppDomain          | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1102 - WorkflowActivityStop

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1102  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a workflow activity has stopped.

## Message

WorkflowInstanceId: '%1' E2E Activity

## Details

| DATA ITEM NAME     | DATA ITEM TYPE | DESCRIPTION  |
|--------------------|----------------|--|
| WorkflowInstanceId | xs:string      | The workflow instance id.                                    |
| AppDomain          | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1103 - WorkflowActivitySuspend

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1103  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a workflow activity has been suspended.

## Message

WorkflowInstanceId: '%1' E2E Activity

## Details

| DATA ITEM NAME     | DATA ITEM TYPE | DESCRIPTION  |
|--------------------|----------------|--|
| WorkflowInstanceId | xs:string      | The workflow instance id.                                    |
| AppDomain          | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1104 - WorkflowActivityResume

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1104  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a workflow activity has been resumed.

## Message

WorkflowInstanceId: '%1' E2E Activity

## Details

| DATA ITEM NAME     | DATA ITEM TYPE | DESCRIPTION  |
|--------------------|----------------|--|
| WorkflowInstanceId | xs:string      | The workflow instance id.                                    |
| AppDomain          | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1124 - InvokeMethodIsStatic

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1124  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

During CacheMetadata step, InvokeMethod activity indicates the method to be invoked is static.

## Message

InvokeMethod '%1' - method is Static.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| InvokeMethod   | xs:string      | The display name of the InvokeMethod activity.               |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1125 - InvokeMethodIsNotStatic

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1125  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

During CacheMetadata step, InvokeMethod activity indicates the method to be invoked is not static.

## Message

InvokeMethod '%1' - method is not Static.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| InvokeMethod   | xs:string      | The display name of the InvokeMethod activity.               |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1126 - InvokedMethodThrewException

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1126  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates an exception was thrown by the method called by the InvokeMethod activity.

## Message

An exception was thrown in the method called by the activity '%1'. %2

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| InvokeMethod   | xs:string      | The display name of the InvokeMethod activity.               |
| Exception      | xs:string      | The exception details for the exception                      |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1131 - InvokeMethodUseAsyncPattern

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1131  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

During CacheMetadata step, InvokeMethod activity indicates that it is using the async pattern when invoking the method.

## Message

InvokeMethod '%1' - method uses asynchronous pattern of '%2' and '%3'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| InvokeMethod   | xs:string      | The display name of the InvokeMethod activity.               |
| BeginMethod    | xs:string      | The name of the begin method.                                |
| EndMethod      | xs:string      | The name of the end method.                                  |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1132 - InvokeMethodDoesNotUseAsyncPattern

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1132  |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

During CacheMetadata step, InvokeMethod activity indicates that it is not using the async pattern when invoking the method.

## Message

InvokeMethod '%1' - method does not use asynchronous pattern.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| InvokeMethod   | xs:string      | The display name of the InvokeMethod activity.               |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1140 - FlowchartStart

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1140  |
| Keywords | WFActivities  |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a Flowchart start has been scheduled.

## Message

Flowchart '%1' - Start has been scheduled.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| FlowChart      | xs:string      | The display name of the FlowChart.                           |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1141 - FlowchartEmpty

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1141  |
| Keywords | WFActivities  |
| Level    | Warning   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a Flowchart was executed with no nodes.

## Message

Flowchart '%1' - was executed with no Nodes.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| FlowChart      | xs:string      | The display name of the FlowChart.                           |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1143 - FlowchartNextNull

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1143  |
| Keywords | WFActivities  |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates the next node in a Flowchart is null. Flowchart execution will end.

## Message

Flowchart '%1'/FlowStep - Next node is null. Flowchart execution will end.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| FlowChart      | xs:string      | The display name of the FlowChart.                           |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1146 - FlowchartSwitchCase

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1146  |
| Keywords | WFActivities  |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates which case has been selected in a Flowchart switch.

## Message

Flowchart '%1'/FlowSwitch - Case '%2' was selected.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| FlowChart      | xs:string      | The display name of the FlowChart.                           |
| Case           | xs:string      | The switch case that selected.                               |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1147 - FlowchartSwitchDefault

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1147  |
| Keywords | WFActivities  |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates the default case has been selected in a Flowchart switch.

## Message

Flowchart '%1'/FlowSwitch - Default Case was selected.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| FlowChart      | xs:string      | The display name of the FlowChart.                           |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1148 - FlowchartSwitchCaseNotFound

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1148  |
| Keywords | WFActivities  |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates that neither a matching case or a default case in a Flowchart switch could be found. Flowchart execution will end.

## Message

Flowchart '%1'/FlowSwitch - could find neither a Case activity nor a Default Case matching the Expression result. Flowchart execution will end.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| FlowChart      | xs:string      | The display name of the FlowChart.                           |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1150 - CompensationState

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1150  |
| Keywords | WFActivities  |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a change of state in a CompensableActivity.

## Message

CompensableActivity '%1' is in the '%2' state.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| DisplayName    | xs:string      | The display name of the activity.                            |
| State          | xs:string      | The compensation state.                                      |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1223 - SwitchCaseNotFound

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1223  |
| Keywords | WFActivities  |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a case activity with a matching Expression result could not be found in a Switch activity.

## Message

The Switch activity '%1' could not find a Case activity matching the Expression result.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| DisplayName    | xs:string      | The display name of the activity.                            |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1449 - WfMessageReceived

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1449  |
| Keywords | WFServices  |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a message has been received by a workflow.

## Message

Message received by workflow

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 1450 - WfMessageSent

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 1450  |
| Keywords | WFServices  |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a message has been sent by a workflow.

## Message

Message sent from workflow

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 2021 - ExecuteWorkItemStart

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 2021  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates an ExecuteWorkItem is starting execution.

## Message

Execute work item start

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 2022 - ExecuteWorkItemStop

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 2022  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates an ExecuteWorkItem has completed.

## Message

Execute work item stop

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 2023 - SendMessageChannelCacheMiss

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 2023  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a cache miss in the SendMessageChannelCache.

## Message

SendMessageChannelCache miss

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 2024 - InternalCacheMetadataStart

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 2024  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates the start of InternalCacheMetadata on an activity.

## Message

InternalCacheMetadata started on activity '%1'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| DisplayName    | xs:string      | The display name of the activity.                            |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 2025 - InternalCacheMetadataStop

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 2025  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates the completion of InternalCacheMetadata on an activity.

## Message

InternalCacheMetadata stopped on activity '%1'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| DisplayName    | xs:string      | The display name of the activity.                            |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 2026 - CompileVbExpressionStart

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 2026  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates the start of a VB expression compilation.

## Message

Compiling VB expression '%1'

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Expression     | xs:string      | The VisualBasic expression to compile.                       |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 2027 - CacheRootMetadataStart

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 2027  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates the start of CacheRootMetadata on an activity.

## Message

CacheRootMetadata started on activity '%1'

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| DisplayName    | xs:string      | The display name of the activity.                            |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 2028 - CacheRootMetadataStop

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 2028  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates the completion of CacheRootMetadata on an activity.

## Message

CacheRootMetadata stopped on activity %1.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| DisplayName    | xs:string      | The display name of the activity.                            |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 2029 - CompileVbExpressionStop

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 2029  |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates the completion of a VB expression compilation.

## Message

Finished compiling VB expression.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 2576 - TryCatchExceptionFromTry

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 2576  |
| Keywords | WFActivities  |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates the TryCatch activity has caught an exception.

## Message

The TryCatch activity '%1' has caught an exception of type '%2'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| DisplayName    | xs:string      | The display name of the activity.                            |
| Exception      | xs:string      | The type name of the exception.                              |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 2577 - TryCatchExceptionDuringCancellation

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 2577  |
| Keywords | WFActivities  |
| Level    | Warning   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a child activity of the TryCatch activity has thrown an exception during cancelation.

## Message

A child activity of the TryCatch activity '%1' has thrown an exception during cancelation.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| DisplayName    | xs:string      | The display name of the activity.                            |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 2578 - TryCatchExceptionFromCatchOrFinally

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 2578  |
| Keywords | WFActivities  |
| Level    | Warning   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a Catch or Finally activity has thrown an exception.

## Message

A Catch or Finally activity that is associated with the TryCatch activity '%1' has thrown an exception.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| DisplayName    | xs:string      | The display name of the activity.                            |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 3501 - InferredContractDescription

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 3501   |
| Keywords | WFServices   |
| Level    | Information  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

Indicates a ContractDescription has been inferred from WorkflowService.

## Message

ContractDescription with Name='%1' and Namespace='%2' has been inferred from WorkflowService.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Name           | xs:string      | The name of the ContractDescription.                         |
| Namespace      | xs:string      | The namespace of the ContractDescription.                    |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 3502 - InferredOperationDescription

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 3502   |
| Keywords | WFServices   |
| Level    | Information  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

Indicates an OperationDescription has been inferred from WorkflowService.

## Message

OperationDescription with Name='%1' in contract '%2' has been inferred from WorkflowService. IsOneWay=%3.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| OperationName  | xs:string      | The name of the operation.                                   |
| ContractName   | xs:string      | The name of the contract.                                    |
| IsOneWay       | xs:string      | True or False indicating if the contract is one-way.         |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 3503 - DuplicateCorrelationQuery

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 3503   |
| Keywords | WFServices   |
| Level    | Warning  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

Indicates a duplicate CorrelationQuery was found. The duplicate query will not be used when calculating correlation.

## Message

A duplicate CorrelationQuery was found with Where='%'1'. This duplicate query will not be used when calculating correlation.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Where          | xs:string      | The Where portion of the correlation query.                  |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 3507 - ServiceEndpointAdded

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 3507   |
| Keywords | WFServices   |
| Level    | Information  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

Indicates a service endpoint has been added.

## Message

A service endpoint has been added for address '%1', binding '%2', and contract '%3'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Address        | xs:string      | The address of the endpoint.                                 |
| Binding        | xs:string      | The binding of the endpoint.                                 |
| Contract       | xs:string      | The contract of the endpoint.                                |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 3508 - TrackingProfileNotFound

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 3508   |
| Keywords | WFServices   |
| Level    | Verbose  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

Indicates either a TrackingProfile is not found in the config file or the ActivityDefinitionId does not match the profile.

## Message

TrackingProfile '%1' for the ActivityDefinitionId '%2' not found. Either the TrackingProfile is not found in the config file or the ActivityDefinitionId does not match.

## Details

| DATA ITEM NAME       | DATA ITEM TYPE | DESCRIPTION  |
|----------------------|----------------|--|
| TrackingProfile      | xs:string      | The name of the tracking profile.                            |
| ActivityDefinitionId | xs:string      | The activity definition id.                                  |
| AppDomain            | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 3550 - BufferOutOfOrderMessageNoInstance

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 3550   |
| Keywords | WFServices   |
| Level    | Information  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

Indicates a buffered receive has failed. The operation will be attempted again when the service instance is ready to process this particular operation.

## Message

Operation '%1' cannot be performed at this time. Another attempt will be made when the service instance is ready to process this particular operation.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| OperationName  | xs:string      | The name of the operation.                                   |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 3551 - BufferOutOfOrderMessageNoBookmark

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 3551   |
| Keywords | WFServices   |
| Level    | Information  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

Indicates a bookmark resumption has failed. The buffered receive operation will be attempted again when the service instance is ready to process this particular operation.

## Message

Operation '%2' on service instance '%1' cannot be performed at this time. Another attempt will be made when the service instance is ready to process this particular operation.

## Details

| DATA ITEM NAME    | DATA ITEM TYPE | DESCRIPTION  |
|-------------------|----------------|--|
| OperationName     | xs:string      | The name of the operation.                                   |
| ServiceInstanceId | xs:string      | The id of the service instance.                              |
| AppDomain         | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 3552 - MaxPendingMessagesPerChannelExceeded

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 3552   |
| Keywords | Quota, WFServices  |
| Level    | Warning  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

Indicates the throttle 'MaxPendingMessagesPerChannel' limit was hit.

## Message

The throttle 'MaxPendingMessagesPerChannel' limit of '%1' was hit. To increase this limit, adjust the MaxPendingMessagesPerChannel property on BufferedReceiveServiceBehavior.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Limit          | xs:string      | The limit of the MaxPendingMessagesPerChannel throttle.      |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 3557 - TransactedReceiveScopeEndCommitFailed

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 3557   |
| Keywords | WFServices   |
| Level    | Information  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

Indicates the call to EndCommit on a CommitableTransaction threw a TransactionException.

## Message

The call to EndCommit on the CommitableTransaction with id = '%1' threw a TransactionException with the following message: '%2'.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| TransactionId  | xs:string      | The id of the CommitableTransaction.                         |
| Exception      | xs:string      | The exception details for the exception                      |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 4201 - EndSqlCommandExecute

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 4201  |
| Keywords | WFInstanceStore   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a SQL command has finished executing.

## Message

End SQL command execution: %1

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| SqlCommand     | xs:string      | The SQL command that was executed.                           |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 4202 - StartSqlCommandExecute

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 4202  |
| Keywords | WFInstanceStore   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a SQL command is being executed.

## Message

Starting SQL command execution: %1

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| SqlCommand     | xs:string      | The SQL command that was executed.                           |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 4203 - RenewLockSystemError

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 4203  |
| Keywords | WFInstanceStore   |
| Level    | Error   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates SQL provider has failed to extend lock expiration due to either lock expiration already passed or the lock owner was deleted. The SqlWorkflowInstanceStore will be aborted.

## Message

Failed to extend lock expiration, lock expiration already passed or the lock owner was deleted. Aborting SqlWorkflowInstanceStore.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 4205 - FoundProcessingError

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 4205  |
| Keywords | WFInstanceStore   |
| Level    | Error   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates SQL provider command has failed.

## Message

Command failed: %1

## Details

| DATA ITEM NAME   | DATA ITEM TYPE | DESCRIPTION  |
|------------------|----------------|--|
| ExceptionMessage | xs:string      | The message from the SQL exception.                          |
| Exception        | xs:string      | The exception details for the exception                      |
| AppDomain        | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 4206 - UnlockInstanceException

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 4206  |
| Keywords | WFInstanceStore   |
| Level    | Error   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates an exception was encountered while trying to unlock an instance.

## Message

Encountered exception %1 while attempting to unlock instance.

## Details

| DATA ITEM NAME   | DATA ITEM TYPE | DESCRIPTION  |
|------------------|----------------|--|
| ExceptionMessage | xs:string      | The message from the SQL exception.                          |
| AppDomain        | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 4207 - MaximumRetriesExceededForSqlCommand

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 4207  |
| Keywords | Quota, WFInstanceStore                                  |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates SQL provider is giving up retrying a SQL command as the maximum number of retries have been performed.

## Message

Giving up retrying a SQL command as the maximum number of retries have been performed.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 4208 - RetryingSqlCommandDueToSqlError

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 4208  |
| Keywords | WFInstanceStore   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates the SQL provider is retrying a SQL command due to a SQL error.

## Message

Retrying a SQL command due to SQL error number %1.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| ErrorNumber    | xs:string      | The SQL error number.  |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 4209 - TimeoutOpeningSqlConnection

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 4209  |
| Keywords | WFInstanceStore   |
| Level    | Error   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a timeout was encountered when trying to open a SQL connection.

## Message

Timeout trying to open a SQL connection. The operation did not complete within the allotted timeout of %1. The time allotted to this operation may have been a portion of a longer timeout.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Timeout        | xs:string      | The timeout value for opening the SQL connection.            |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 4210 - SqlExceptionCaught

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 4210  |
| Keywords | WFInstanceStore   |
| Level    | Warning   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a SQL exception was caught.

## Message

Caught SQL Exception number %1 message %2.

## Details

| DATA ITEM NAME   | DATA ITEM TYPE | DESCRIPTION  |
|------------------|----------------|--|
| ErrorNumber      | xs:string      | The SQL error number.  |
| ExceptionMessage | xs:string      | The message from the SQL exception.                          |
| AppDomain        | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 4211 - QueuingSqlRetry

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 4211  |
| Keywords | WFInstanceStore   |
| Level    | Warning   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates queuing SQL retry.

## Message

Queuing SQL retry with delay %1 milliseconds.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Delay          | xs:string      | The delay between retries.                                   |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 4212 - LockRetryTimeout

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 4212  |
| Keywords | WFInstanceStore   |
| Level    | Warning   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

SQL provider encountered a timeout trying to acquire the instance lock.

## Message

Timeout trying to acquire the instance lock. The operation did not complete within the allotted timeout of %1. The time allotted to this operation may have been a portion of a longer timeout.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Delay          | xs:string      | The delay between retries.                                   |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 4213 - RunnableInstancesDetectionError

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 4213  |
| Keywords | WFInstanceStore   |
| Level    | Error   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Detection of runnable instances failed due to an exception

## Message

Detection of runnable instances failed due to the following exception

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Exception      | xs:string      | The exception details for the exception                      |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 4214 - InstanceLocksRecoveryError

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 4214  |
| Keywords | WFInstanceStore   |
| Level    | Error   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Recovering instance locks failed due to an exception.

## Message

Recovering instance locks failed due to the following exception

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Exception      | xs:string      | The exception details for the exception                      |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 39456 - TrackingRecordDropped

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 39456   |
| Keywords | WFTracking  |
| Level    | Warning   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a tracking record has been dropped because its size exceeds maximum allowed by the ETW session provider.

## Message

Size of tracking record %1 exceeds maximum allowed by the ETW session for provider %2

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Exception      | xs:string      | The exception details for the exception                      |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 39457 - TrackingRecordRaised

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 39457   |
| Keywords | WFRuntime   |
| Level    | Information   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a TrackingRecord has been raised to a TrackingParticipant.

## Message

Tracking Record %1 raised to %2.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| RecordNumber   | xs:string      | The tracking record number.                                  |
| ParticipantId  | xs:string      | The tracking participant.                                    |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 39458 - TrackingRecordTruncated

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 39458   |
| Keywords | WFTracking  |
| Level    | Warning   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates a tracking record has been truncated. Variables/annotations/user data have been removed.

## Message

Truncated tracking record %1 written to ETW session with provider %2. Variables/annotations/user data have been removed

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| RecordNumber   | xs:string      | The tracking record number.                                  |
| ProviderId     | xs:string      | The ETW provider id.   |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 39459 - TrackingDataExtracted

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 39459   |
| Keywords | WFRuntime   |
| Level    | Verbose   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates tracking data has been extracted in an activity.

## Message

Tracking data %1 extracted in activity %2.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Data           | xs:string      | The name of the data extracted.                              |
| Activity       | xs:string      | The name of the activity.                                    |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 39460 - TrackingValueNotSerializable

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |   |
|----------|---|
| ID       | 39460   |
| Keywords | WFTracking  |
| Level    | Warning   |
| Channel  | Microsoft-Windows-Application Server-Applications/Debug |

## Description

Indicates the extracted argument/variable in a tracking record is not serializable.

## Message

The extracted argument/variable '%1' is not serializable.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Name           | xs:string      | The name of the item.  |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# 57398 - MaxInstancesExceeded

5/4/2018 • 2 minutes to read • [Edit Online](#)

## Properties

|          |  |
|----------|--|
| ID       | 57398  |
| Keywords | WFServices   |
| Level    | Warning  |
| Channel  | Microsoft-Windows-Application Server-Applications/Analytic |

## Description

Indicates the system hit the limit set for throttle 'MaxConcurrentInstances'.

## Message

The system hit the limit set for throttle 'MaxConcurrentInstances'. Limit for this throttle was set to %1. Throttle value can be changed by modifying attribute 'maxConcurrentInstances' in serviceThrottle element or by modifying 'MaxConcurrentInstances' property on behavior ServiceThrottlingBehavior.

## Details

| DATA ITEM NAME | DATA ITEM TYPE | DESCRIPTION  |
|----------------|----------------|--|
| Name           | xs:string      | The name of the item.  |
| AppDomain      | xs:string      | The string returned by AppDomain.CurrentDomain.FriendlyName. |

# Custom Tracking Records

3/6/2019 • 2 minutes to read • [Edit Online](#)

This topic demonstrates how to create custom tracking records and populate them with data to be emitted along with the records.

## Emitting Custom Tracking Records

Custom tracking records can be emitted from a code activity as shown in the following example.

```
protected override void Execute(CodeActivityContext context)
{
    ...
    CustomTrackingRecord customRecord = new CustomTrackingRecord("CustomEmailSentEvent");
    customRecord.Data.Add("SendTime", sendTime);
    context.Track(customRecord);
}
```

A [CustomTrackingRecord](#) is emitted in a code activity by invoking the [Track](#) method on the [ActivityContext](#).

## See also

- [Windows Server App Fabric Monitoring](#)
- [Monitoring Applications with App Fabric](#)

# Variable and Argument Tracking

1/23/2019 • 2 minutes to read • [Edit Online](#)

When tracking the execution of a workflow, it is often useful to extract data. This provides additional context when accessing a tracking record post execution. In .NET Framework 4.6.1, you can extract any visible variable or argument within the scope of any activity in a workflow using tracking. Tracking profiles make it easy to extract data.

## Variables and Arguments

Variables and arguments are extracted when an activity emits an `ActivityStateRecord`. A variable is available for extraction only if it is within the scope of the activity. A variable to be extracted within an activity is specified in the following manner:

- If a variable is specified by the variable name, then tracking looks for the variable within the current activity being tracked and in parent activities. The variable is searched in current activity scope and in parent scope.
- If variables to be extracted are specified by using `name="*"`, then all variables within the current activity being tracked are extracted. In this case variables that are in scope but defined in parent activities are not extracted.

When extracting arguments, the arguments extracted depend on the state of the activity. When the state of an activity is `Executing`, then only the `InArguments` are available for extraction. For any other activity state (`Closed`, `Faulted`, `Canceled`), all arguments, both `InArguments` and `OutArguments`, are available for extraction.

The following example shows an activity state query that extracts variables and arguments when the activity's `Closed` tracking record is emitted. Variables and arguments can be extracted only with an `ActivityStateRecord` and thus are subscribed to within a tracking profile using `ActivityStateQuery`.

```
<activityStateQuery activityName="SendEmailActivity">
  <states>
    <state name="Closed"/>
  </states>
  <variables>
    <variable name="FromAddress"/>
  </variables>
  <arguments>
    <argument name="Result"/>
  </arguments>
</activityStateQuery>
```

## Protecting Information Stored Within Variables and Arguments

A tracked variable or argument is by default made visible by the WF runtime. A workflow developer can protect it from being accessed by taking the following steps:

1. Encrypt the value of a variable.
2. Control the authoring of a tracking profile to prevent the extraction of a variable or argument.
3. For custom tracking participants ensure that the WF code does not disclose sensitive information that is stored in variables or arguments.

## See also

- [Windows Server App Fabric Monitoring](#)
- [Monitoring Applications with App Fabric](#)

# Workflow Security

3/8/2019 • 2 minutes to read • [Edit Online](#)

Windows Workflow Foundation (WF) is integrated with several different technologies, such as Microsoft SQL Server and Windows Communication Foundation (WCF). Interacting with these technologies may introduce security issues into your workflow if done improperly.

## Persistence Security Concerns

1. Workflows that use a [Delay](#) activity and persistence need to be reactivated by a service. Windows AppFabric uses the Workflow Management Service (WMS) to reactivate workflows with expired timers. WMS creates a [WorkflowServiceHost](#) to host the reactivated workflow. If the WMS service is stopped, persisted workflows will not be reactivated when their timers expire.
2. Access to durable instancing should be protected against malicious entities external to the application domain. In addition, developers should ensure that malicious code can't be executed in the same application domain as the durable instancing code.
3. Durable instancing should not be run with elevated (Administrator) permissions.
4. Data being processed outside the application domain should be protected.
5. Applications that require security isolation should not share the same instance of the schema abstraction. Such applications should use different store providers, or store providers configured to use different store instantiations.

## SQL Server Security Concerns

- When large numbers of child activities, locations, bookmarks, host extensions, or scopes are used, or when bookmarks with very large payloads are used, memory can be exhausted, or undue amounts of database space can be allocated during persistence. This can be mitigated by using object-level and database-level security.
- When using [SqlWorkflowInstanceStateStore](#), the instance store must be secured. For more information, see [SQL Server Best Practices](#).
- Sensitive data in the instance store should be encrypted. For more information, see [SQL Security Encryption](#).
- Since the database connection string is often included in a configuration file, windows-level security (ACL) should be used to ensure that the configuration file (Web.Config usually) is secure, and that login and password information are not included in the connection string. Windows authentication should be used between the database and the web server instead.

## Considerations for WorkflowServiceHost

- Windows Communication Foundation (WCF) endpoints used in workflows should be secured. For more information, see [WCF Security Overview](#).
- Host-level authorization can be implemented by using [ServiceAuthorizationManager](#). See [How To: Create a Custom Authorization Manager for a Service](#) for details.
- The ServiceSecurityContext for the incoming message is also available from within workflow by accessing

OperationContext.

## WF Security Pack CTP

The Microsoft WF Security Pack CTP 1 is the first community technology preview (CTP) release of a set of activities and their implementation based on [Windows Workflow Foundation](#) in [.NET Framework 4](#) (WF 4) and the [Windows Identity Foundation \(WIF\)](#). The Microsoft WF Security Pack CTP 1 contains both activities and their designers which illustrate how to easily enable various security-related scenarios using workflow, including:

1. Impersonating a client identity in the workflow
2. In-workflow authorization, such as PrincipalPermission and validation of Claims
3. Authenticated messaging using ClientCredentials specified in the workflow, such as username/password or a token retrieved from a Security Token Service (STS)
4. Flowing a client security token to a back-end service (claims-based delegation) using WS-Trust ActAs

For more information and to download the WF Security Pack CTP, see: [WF Security Pack CTP](#)

# Windows Workflow Foundation 4 Performance

3/14/2019 • 31 minutes to read • [Edit Online](#)

Microsoft .NET Framework version 4 includes a major revision of the Windows Workflow Foundation (WF) with heavy investments in performance. This new revision introduces significant design changes from the previous versions of WF that shipped as part of .NET Framework 3.0 and .NET Framework 3.5. It has been re-architected from the core of the programming model, runtime, and tooling to greatly improve performance and usability. This topic shows the important performance characteristics of these revisions and compares them against those of the previous version.

Individual workflow component performance has increased by orders of magnitude between WF3 and WF4. This leaves the gap between hand-coded Windows Communication Foundation (WCF) services and WCF workflow services to be quite small. Workflow latency has been significantly reduced in WF4. Persistence performance has increased by a factor of 2.5 - 3.0. Health monitoring by means of workflow tracking has significantly less overhead. These are compelling reasons to migrate to or adopt WF4 in your applications.

## Terminology

The version of WF introduced in .NET Framework 4 will be referred to as WF4 for the rest of this topic. WF was introduced in .NET 3.0 and had a few minor revisions through .NET Framework 3.5 SP1. The .NET Framework 3.5 version of Workflow Foundation will be referred to as WF3 for the rest of this topic. WF3 is shipped in .NET Framework 4 side-by-side with WF4. For more information about migrating WF3 artifacts to WF4 see: [Windows Workflow Foundation 4 Migration Guide](#)

Windows Communication Foundation (WCF) is Microsoft's unified programming model for building service-oriented applications. It was first introduced as part of .NET 3.0 together with WF3 and now is one of the key components of the .NET Framework.

Windows Server AppFabric is a set of integrated technologies that make it easier to build, scale and manage Web and composite applications that run on IIS. It provides tools for monitoring and managing services and workflows. For more information, see [Windows Server AppFabric 1.0](#).

## Goals

The goal of this topic is to show the performance characteristics of WF4 with data measured for different scenarios. It also provides detailed comparisons between WF4 and WF3, and thus shows the great improvements that have been made in this new revision. The scenarios and data presented in this article quantify the underlying cost of different aspects of WF4 and WF3. This data is useful in understanding the performance characteristics of WF4 and can be helpful in planning migrations from WF3 to WF4 or using WF4 in application development. However, care should be taken in the conclusions drawn from the data presented in this article. The performance of a composite workflow application is highly dependent on how the workflow is implemented and how different components are integrated. One must measure each application to determine the performance characteristics of that application.

## Overview of WF4 Performance Enhancements

WF4 was carefully designed and implemented with high performance and scalability which are described in the following sections.

### WF Runtime

At the core of the WF runtime is an asynchronous scheduler that drives the execution of the activities in a

workflow. It provides a performant, predictable execution environment for activities. The environment has a well-defined contract for execution, continuation, completion, cancellation, exceptions, and a predictable threading model.

In comparison to WF3, the WF4 runtime has a more efficient scheduler. It leverages the same I/O thread pool that is used for WCF, which is very efficient at executing batched work items. The internal work item scheduler queue is optimized for most common usage patterns. The WF4 runtime also manages the execution states in a very light-weight way with minimal synchronization and event handling logic, while WF3 depends on heavy-weight event registration and invocation to perform complex synchronization for state transitions.

### **Data Storage and Flow**

In WF3, data associated with an activity is modeled through dependency properties implemented by the type [DependencyProperty](#). The dependency property pattern was introduced in Windows Presentation Foundation (WPF). In general, this pattern is very flexible to support easy data binding and other UI features. However, the pattern requires the properties to be defined as static fields in the workflow definition. Whenever the WF runtime sets or gets the property values, it involves heavily-weighted look-up logic.

WF4 uses clear data scoping logic to greatly improve how data is handled in a workflow. It separates the data stored in an activity from the data that is flowing across the activity boundaries by using two different concepts: variables and arguments. By using a clear hierarchical scope for variables and "In/Out/InOut" arguments, the data usage complexity for activities is dramatically reduced and the lifetime of the data is also automatically scoped. Activities have a well-defined signature described by its arguments. By simply inspecting an activity you can determine what data it expects to receive and what data will be produced by it as the result of its execution.

In WF3 activities were initialized when a workflow was created. In WF 4 activities are initialized only when the corresponding activities are executing. This allows a simpler activity lifecycle without performing Initialize/Uninitialize operations when a new workflow instance is created, and thus has achieved more efficiency

### **Control Flow**

Just as in any programming language, WF provides support for control flows for workflow definitions by introducing a set of control flow activities for sequencing, looping, branching and other patterns. In WF3, when the same activity needs to be re-executed, a new [ActivityExecutionContext](#) is created and the activity is cloned through a heavy-weight serialization and deserialization logic based on [BinaryFormatter](#). Usually the performance for iterative control flows is much slower than executing a sequence of activities.

WF4 handles this quite differently. It takes the activity template, creates a new [ActivityInstance](#) object, and adds it to the scheduler queue. This whole process only involves explicit object creation and is very light-weight.

### **Asynchronous Programming**

Applications usually have better performance and scalability with asynchronous programming for long running blocking operations such as I/O or distributed computing operations. WF4 provides asynchronous support through base activity types [AsyncCodeActivity](#), [AsyncCodeActivity<TResult>](#). The runtime natively understands asynchronous activities and therefore can automatically put the instance in a no-persist zone while the asynchronous work is outstanding. Custom activities can derive from these types to perform asynchronous work without holding the workflow scheduler thread and blocking any activities that may be able to run in parallel.

### **Messaging**

Initially WF3 had very limited messaging support through external events or web services invocations. In .NET 3.5, workflows could be implemented as WCF clients or exposed as WCF services through [SendActivity](#) and [ReceiveActivity](#). In WF4, the concept of workflow-based messaging programming has been further strengthened through the tight integration of WCF messaging logic into WF.

The unified message processing pipeline provided in WCF in .NET 4 helps WF4 services to have significantly better performance and scalability than WF3. WF4 also provides richer messaging programming support that can model complex Message Exchange Patterns (MEPs). Developers can use either typed service contracts to achieve

easy programming or un-typed service contracts to achieve better performance without paying serialization costs. The client-side channel caching support through the [SendMessageChannelCache](#) class in WF4 helps developers build fast applications with minimal effort. For more information, see [Changing the Cache Sharing Levels for Send Activities](#).

## Declarative Programming

WF4 provides a clean and simple declarative programming framework to model business processes and services. The programming model supports fully declarative composition of activities, with no code-beside, greatly simplifying workflow authoring. In .NET Framework 4, the XAML-based declarative programming framework has been unified into the single assembly System.Xaml.dll to support both WPF and WF.

In WF4, XAML provides a truly declarative experience and allows for the entire definition of the workflow to be defined in XML markup, referencing activities and types built using .NET. This was difficult to do in WF3 with XOML format without involving custom code-behind logic. The new XAML-stack in .NET 4 has much better performance in serializing/deserializing workflow artifacts and makes declarative programming more attractive and solid.

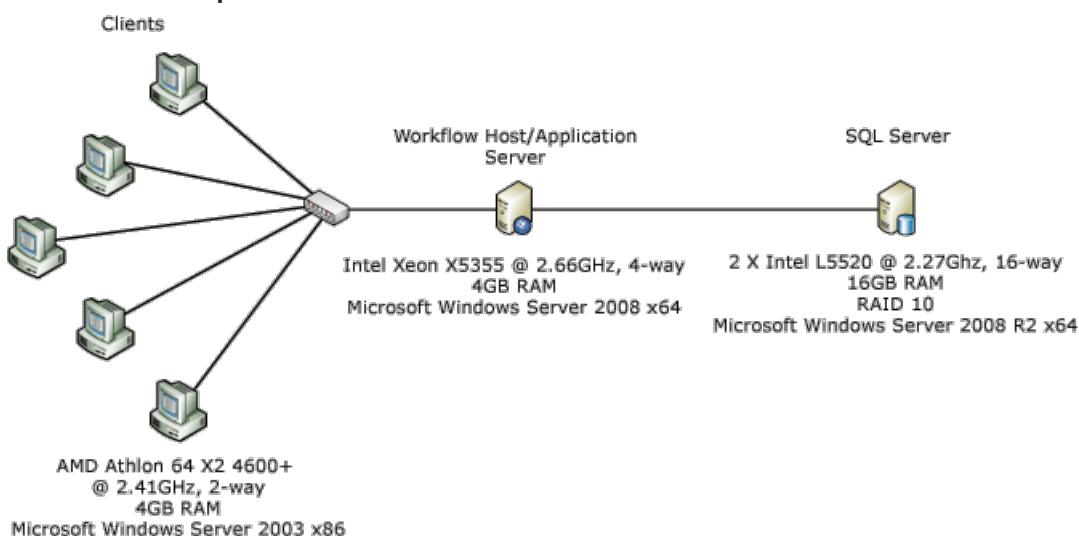
## Workflow Designer

Fully declarative programming support for WF4 explicitly imposes higher requirements for design time performance for large workflows. The Workflow designer in WF4 has much better scalability for large workflows than that for WF3. With UI virtualization support, the designer can easily load a large workflow of 1000 activities in a few seconds, while it is almost impossible to load a workflow of a few hundred activities with the WF3 designer.

# Component-level Performance Comparisons

This section contains data on direct comparisons between individual activities in WF3 and WF4 workflows. Key areas like persistence have a more profound impact on performance than the individual activity components. The performance improvements in individual components in WF4 are important though because the components are now fast enough to be compared against hand-coded orchestration logic. An example of which is covered in the next section: "Service Composition Scenario."

## Environment Setup



The above figure shows the machine configuration used for component-level performance measurement. A single server and five clients connected over one 1-Gbps Ethernet network interface. For easy measurements, the server is configured to use a single core of a dual-proc/quad-core server running Windows Server 2008 x86. The system CPU utilization is maintained at nearly 100%.

## Test Details

The WF3 [CodeActivity](#) is likely the simplest activity that can be used in a WF3 workflow. The activity calls a method

in the code-behind that the workflow programmer can put custom code into. In WF4, there is no direct analog to the WF3 [CodeActivity](#) that provides the same functionality. Note that there is a [CodeActivity](#) base class in WF4 that is not related to the WF3 [CodeActivity](#). Workflow authors are encouraged to create custom activities and build XAML-only workflows. In the tests below, an activity called `Comment` is used in place of an empty [CodeActivity](#) in WF4 workflows. The code in the `Comment` activity is as follows:

```
[ContentProperty("Body")]
public sealed class Comment : CodeActivity
{
    public Comment()
        : base()
    {

    }

    [DefaultValue(null)]
    public Activity Body
    {
        get;
        set;
    }

    protected override void Execute(CodeActivityContext context)
    {
    }
}
```

## Empty Workflow

This test uses a sequence workflow with no child activities.

## Single Activity

The workflow is a sequence workflow containing one child activity. The activity is a [CodeActivity](#) with no code in the WF3 case and a `Comment` activity in the WF4 case.

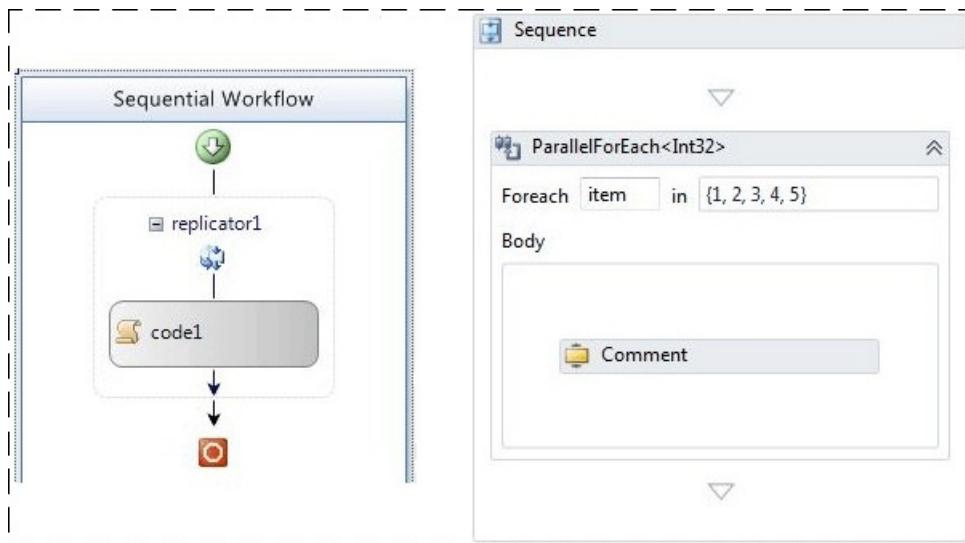
## While with 1000 Iterations

The sequence workflow contains one [While](#) activity with one child activity in the loop that does not perform any work.

## Replicator compared to ParallelForEach

[ReplicatorActivity](#) in WF3 has sequential and parallel execution modes. In sequential mode, the activity's performance is similar to the [WhileActivity](#). The [ReplicatorActivity](#) is most useful for parallel execution. The WF4 analog for this is the [ParallelForEach<T>](#) activity.

The following diagram shows the workflows used for this test. The WF3 workflow is on the left and the WF4 workflow is on the right.



### Sequential Workflow with Five Activities

This test is meant to show the effect of having several activities execute in sequence. There are five activities in the sequence.

### Transaction Scope

The transaction scope test differs from the other tests slightly in that a new workflow instance is not created for every iteration. Instead, the workflow is structured with a while loop containing a [TransactionScope](#) activity containing a single activity that does no work. Each run of a batch of 50 iterations through the while loop is counted as a single operation.

### Compensation

The WF3 workflow has a single compensatable activity named [WorkScope](#). The activity simply implements the [ICompensatableActivity](#) interface:

```
class WorkScope :
    CompositeActivity, ICompensatableActivity
{
    public WorkScope() : base() { }

    public WorkScope(string name)
    {
        this.Name = name;
    }

    public ActivityExecutionStatus Compensate(
        ActivityExecutionContext executionContext)
    {
        return ActivityExecutionStatus.Closed;
    }
}
```

The fault handler targets the [WorkScope](#) activity. The WF4 workflow is equally simplistic. A [CompensableActivity](#) has a body and a compensation handler. An explicit compensate is next in the sequence. The body activity and compensation handler activity are both empty implementations:

```

public sealed class CompensableActivityEmptyCompensation : CodeActivity
{
    public CompensableActivityEmptyCompensation()
        : base() { }

    public Activity Body { get; set; }

    protected override void Execute(CodeActivityContext context) { }
}

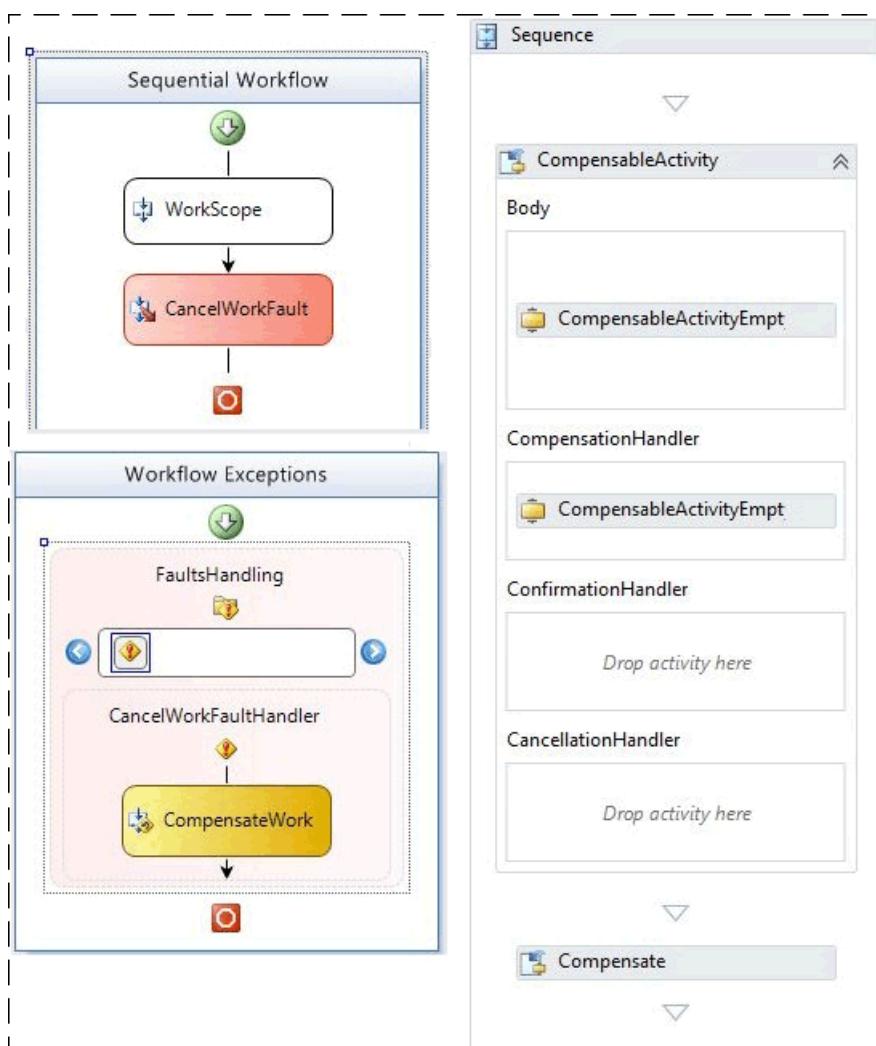
public sealed class CompensableActivityEmptyBody : CodeActivity
{
    public CompensableActivityEmptyBody()
        : base() { }

    public Activity Body { get; set; }

    protected override void Execute(CodeActivityContext context) { }
}

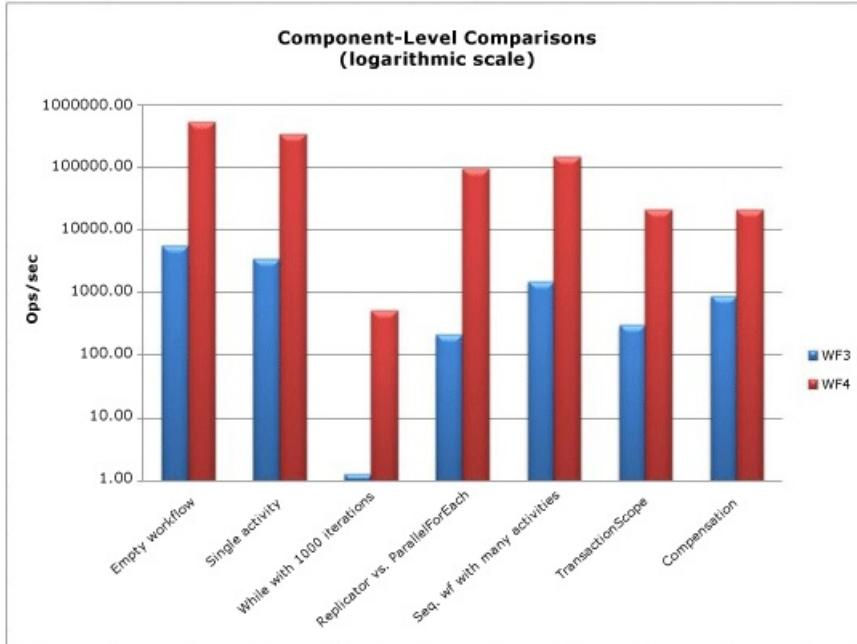
```

The following diagram shows the basic compensation workflow. The WF3 workflow is on the left and the WF4 workflow is on the right.



## Performance Test Results

| Test                                     | WF 3 (ops/sec) | WF 4 (ops/sec) |
|--|----------------|----------------|
| Empty workflow                           | 5,699          | 559,711        |
| Single activity                          | 3,622          | 352,561        |
| While with 1000 iterations               | 1.3            | 531            |
| Replicator vs. ParallelForEach           | 220            | 97,759         |
| Sequential workflow with five activities | 1,576          | 153,582        |
| TransactionScope                         | 311            | 22,099         |
| Compensation                             | 900            | 21,822         |



All tests are measured in workflows per second with the exception of the transaction scope test. As can be seen above, the WF runtime performance has improved across the board, especially in areas that require multiple executions of the same activity like the while loop.

## Service Composition Scenario

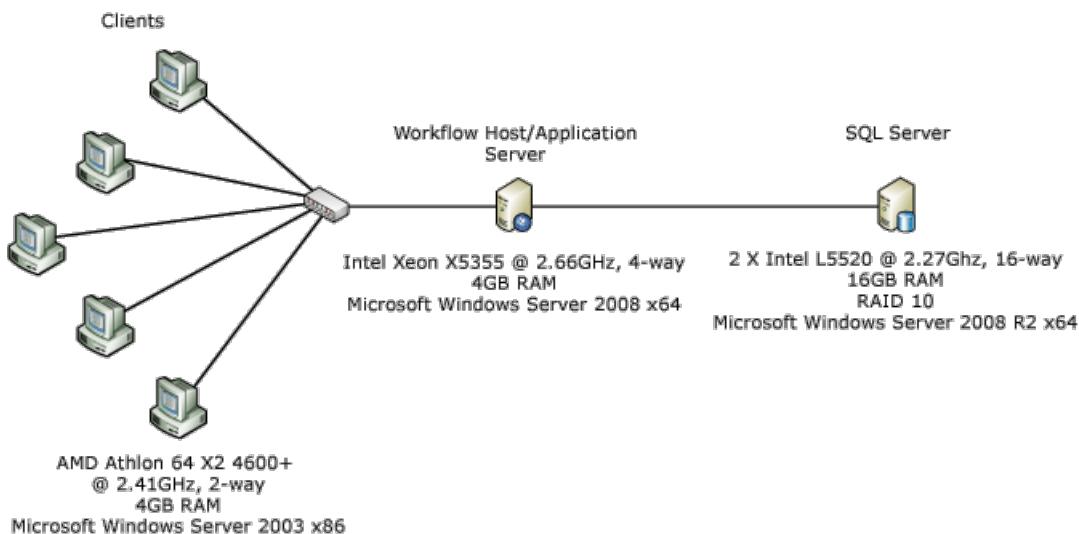
As is shown in the previous section, "Component-level Performance Comparisons," there has been a significant reduction in overhead between WF3 and WF4. WCF workflow services can now almost match the performance of hand-coded WCF services but still have all the benefits of the WF runtime. This test scenario compares a WCF service against a WCF workflow service in WF4.

### Online Store Service

One of the strengths of Windows Workflow Foundation is the ability to compose processes using several services. For this example, there is an online store service that orchestrates two service calls to purchase an order. The first step is to validate the order using an Order Validating Service. The second step is to fill the order using a Warehouse Service.

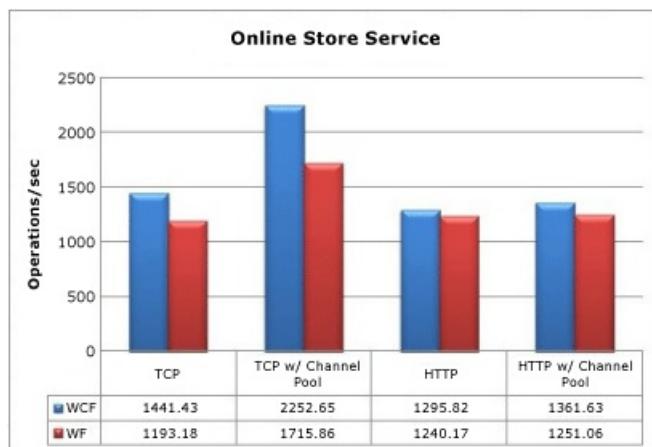
The two backend services, Order Validating Service and Warehouse Service, remain the same for both tests. The part that changes is the Online Store Service that performs the orchestration. In one case, the service is hand-coded as a WCF service. For the other case, the service is written as a WCF workflow service in WF4. WF-specific features like tracking and persistence are turned off for this test.

### Environment



Client requests are made to the Online Store Service via HTTP from multiple computers. A single computer hosts all three services. The transport layer between the Online Store Service and the backend services is TCP or HTTP. The measurement of operations/second is based on the number of completed `PurchaseOrder` calls made to the Online Store Service. Channel pooling is a new feature available in WF4. In the WCF portion of this test channel pooling is not provided out of the box so a hand-coded implementation of a simple pooling technique was used in the Online Store Service.

## Performance



Connecting to backend TCP services without channel pooling, the WF service has a 17.2% impact on throughput. With channel pooling, the penalty is about 23.8%. For HTTP, the impact is much less: 4.3% without pooling and 8.1% with pooling. It is also important to note that the channel pooling provides very little benefit when using HTTP.

While there is overhead from the WF4 runtime compared with a hand-coded WCF service in this test, it could be considered a worst-case scenario. The two backend services in this test do very little work. In a real end-to-end scenario, these services would perform more expensive operations like database calls, making the performance impact of the transport layer less important. This plus the benefits of the features available in WF4 makes Workflow Foundation a viable choice for creating orchestration services.

## Key Performance Considerations

The feature areas in this section, with the exception of interop, have dramatically changed between WF3 and WF4. This affects the design of workflow applications as well as the performance.

### Workflow Activation Latency

In a WCF workflow service application, the latency for starting a new workflow or loading an existing workflow is important as it can be blocking. This test case measures a WF3 XOML host against a WF4 XAMLX host in a typical

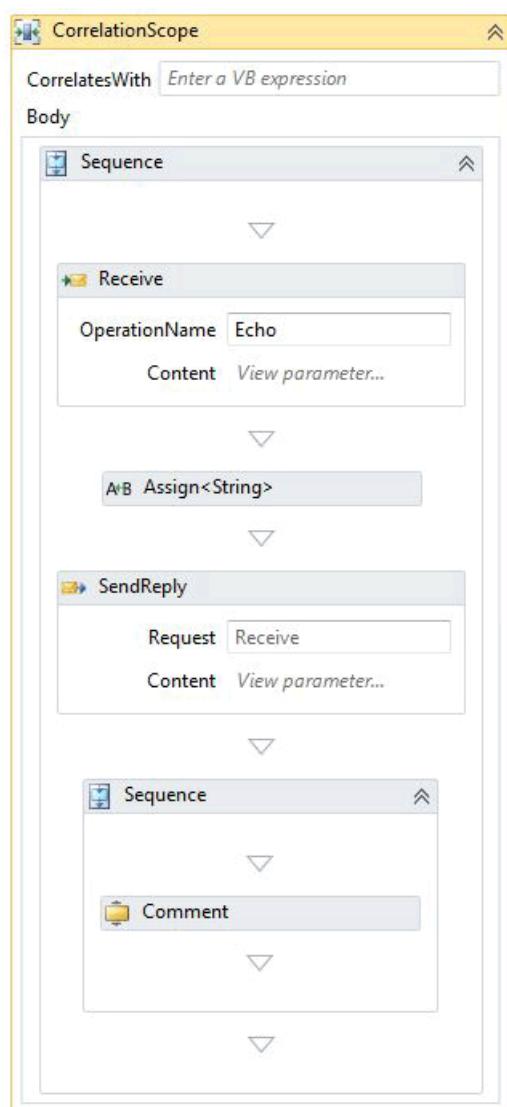
scenario.



#### Test Setup

In the scenario, a client computer contacts a WCF workflow service using context-based correlation. Context correlation requires a special context binding and uses a context header or cookie to relate messages to the correct workflow instance. It has a performance benefit in that the correlation Id is located in the message header so the message body does not need to be parsed.

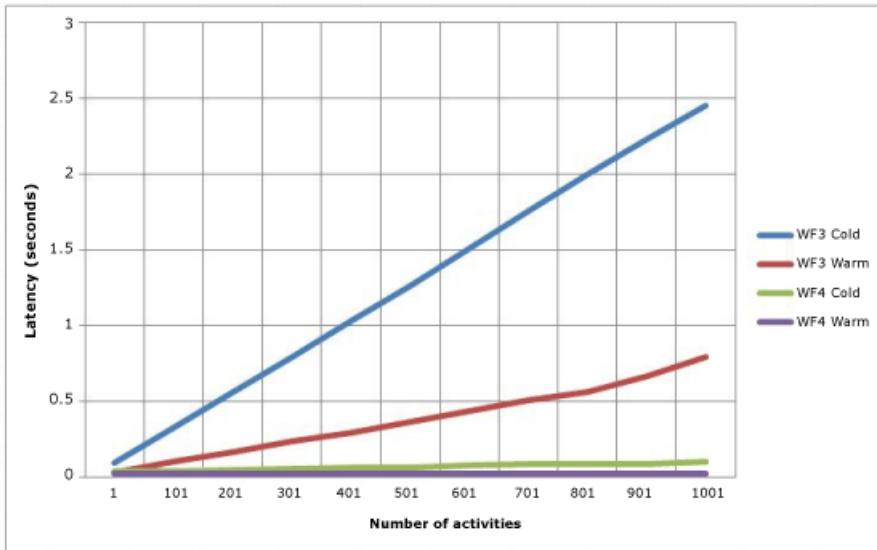
The service will create a new workflow with the request and send an immediate response so that the measurement of latency does not include the time spent running the workflow. The WF3 workflow is XOML with a code-behind and the WF4 workflow is entirely XAML. The WF4 workflow looks like this:



The **Receive** activity creates the workflow instance. A value passed in the received message is echoed in the reply message. A sequence following the reply contains the rest of the workflow. In the above case, only one comment activity is shown. The number of comment activities is changed to simulate workflow complexity. A comment activity is equivalent to a WF3 [CodeActivity](#) that performs no work. For more information about the comment activity, see the "Component-level Performance Comparison" section earlier in this article.

#### Test Results

Cold and warm latency for WCF workflow services:



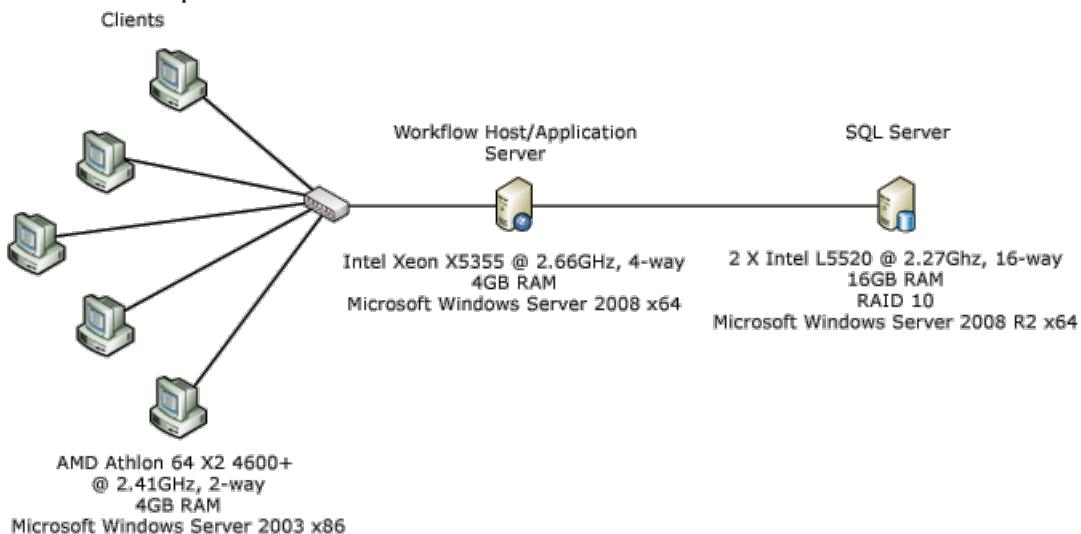
In the previous chart, cold refers to the case where there is not an existing [WorkflowServiceHost](#) for the given workflow. In other words, cold latency is when the workflow is being used for the first time and the XOML or XAML needs to be compiled. Warm latency is the time to create a new workflow instance when the workflow type has already been compiled. The complexity of the workflow makes very little difference in the WF4 case but has a linear progression in the WF3 case.

#### Correlation Throughput

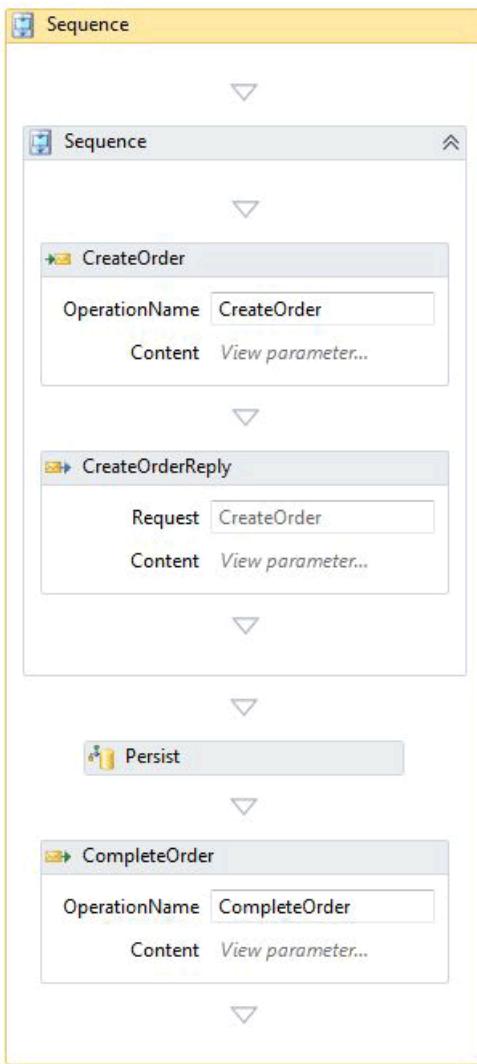
WF4 introduces a new content-based correlation feature. WF3 provided only context-based correlation. Context-based correlation could only be done over specific WCF channel bindings. The workflow Id is inserted into the message header when using these bindings. The WF3 runtime could only identify a workflow by its Id. With content-based correlation, the workflow author can create a correlation key out of a relevant piece of data like an account number or customer Id.

Context-based correlation has a performance advantage in that the correlation key is located in the message header. The key can be read from the message without de-serialization/message-copying. In content-based correlation, the correlation key is stored in the message body. An XPath expression is used to locate the key. The cost of this extra processing depends on the size of the message, depth of the key in the body, and the number of keys. This test compares context- and content-based correlation and also shows the performance degradation when using multiple keys.

#### Environment Setup

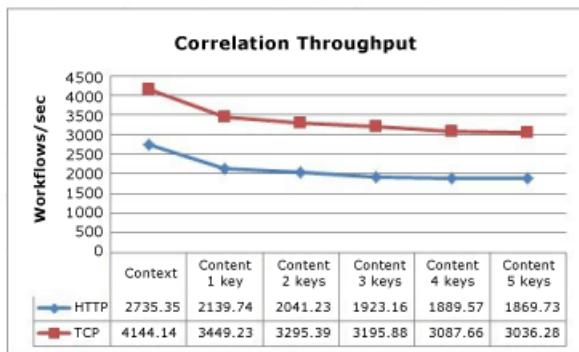


#### Test Setup



The previous workflow is the same one used in the [Persistence](#) section. For the correlation tests without persistence, there is no persistence provider installed in the runtime. Correlation occurs in two places: CreateOrder and CompleteOrder.

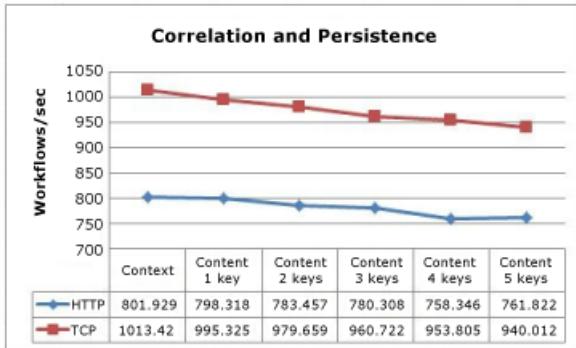
#### Test Results



This graph shows a decrease in performance as the number of keys used in content-based correlation increases. The similarity in the curves between TCP and HTTP indicates the overhead associated with these protocols.

#### Correlation with Persistence

With a persisted workflow, the CPU pressure from content-based correlation shifts from the workflow runtime to the SQL database. The stored procedures in the SQL persistence provider do the work of matching the keys to locate the appropriate workflow.



Context-based correlation is still faster than content-based correlation. However, the difference is less pronounced as persistence has more impact on performance than correlation.

### Complex Workflow Throughput

The complexity of a workflow is not measured only by the number of activities. Composite activities can contain many children and those children can also be composite activities. As the number of levels of nesting increases, so does the number of activities that can be currently in the executing state and the number of variables that can be in state. This test compares throughput between WF3 and WF4 when executing complex workflows.

### Test Setup

These tests were executed on an Intel Xeon X5355 @ 2.66GHz 4-way computer with 4GB RAM running Windows Server 2008 x64. The test code runs in a single process with one thread per core to reach 100% CPU utilization.

The workflows generated for this test have two main variables: depth and number of activities in each sequence. Each depth level includes a parallel activity, while loop, decisions, assignments, and sequences. In the WF4 designer pictured below, the top-level flow chart is pictured. Each flowchart activity resembles the main flowchart. It may be helpful to think of a fractal when picturing this workflow, where the depth is limited to the parameters of the test.

The number of activities in a given test is determined by the depth and number of activities per sequence. The following equation computes the number of activities in the WF4 test:

$$f(d, a) = \sum_{k=0}^{d-1} (4a + 22)3^k$$

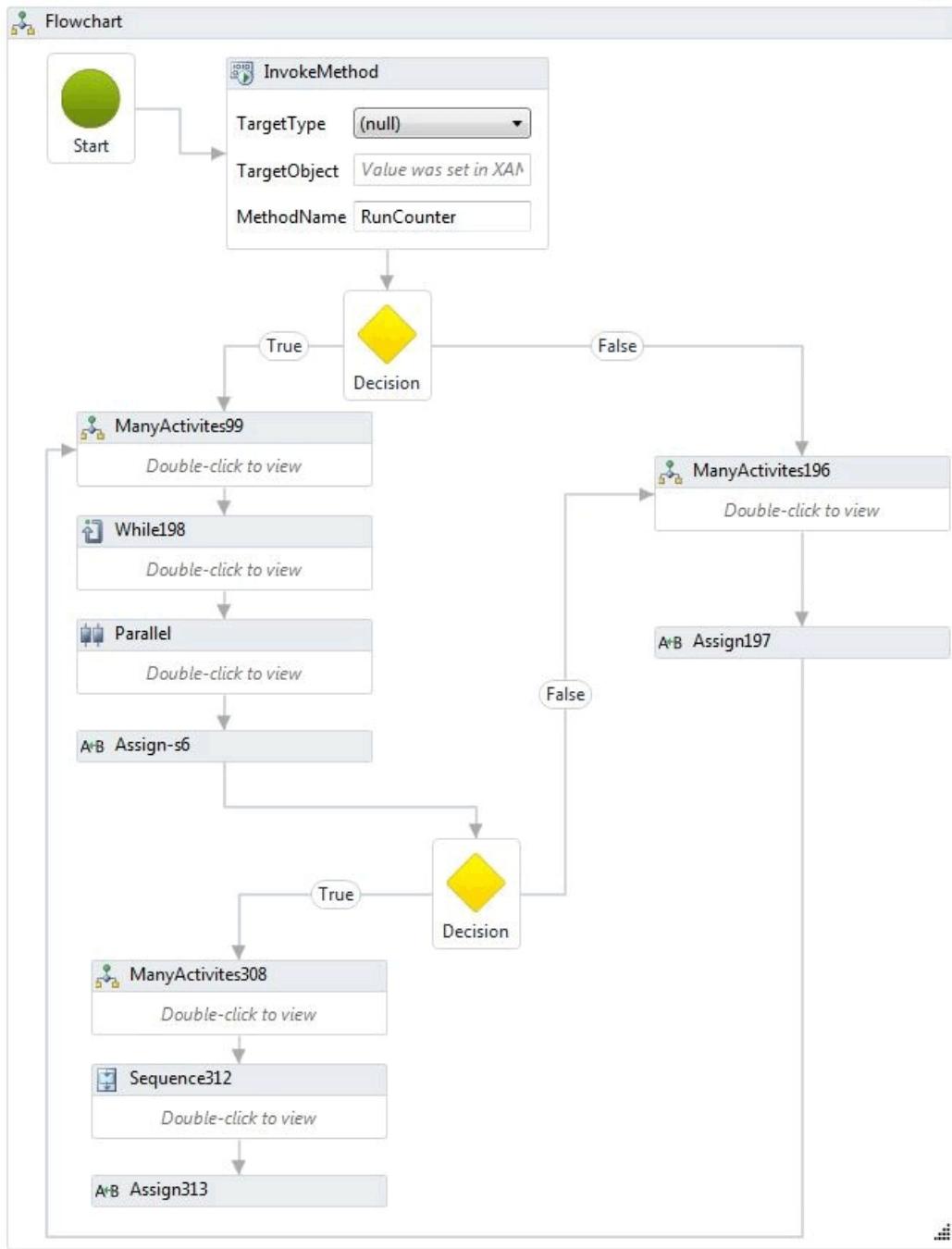
The WF3 test's activity count can be computed with a slightly different equation due to an extra sequence:

$$g(d, a) = \sum_{k=0}^{d-1} (5a + 23)3^k$$

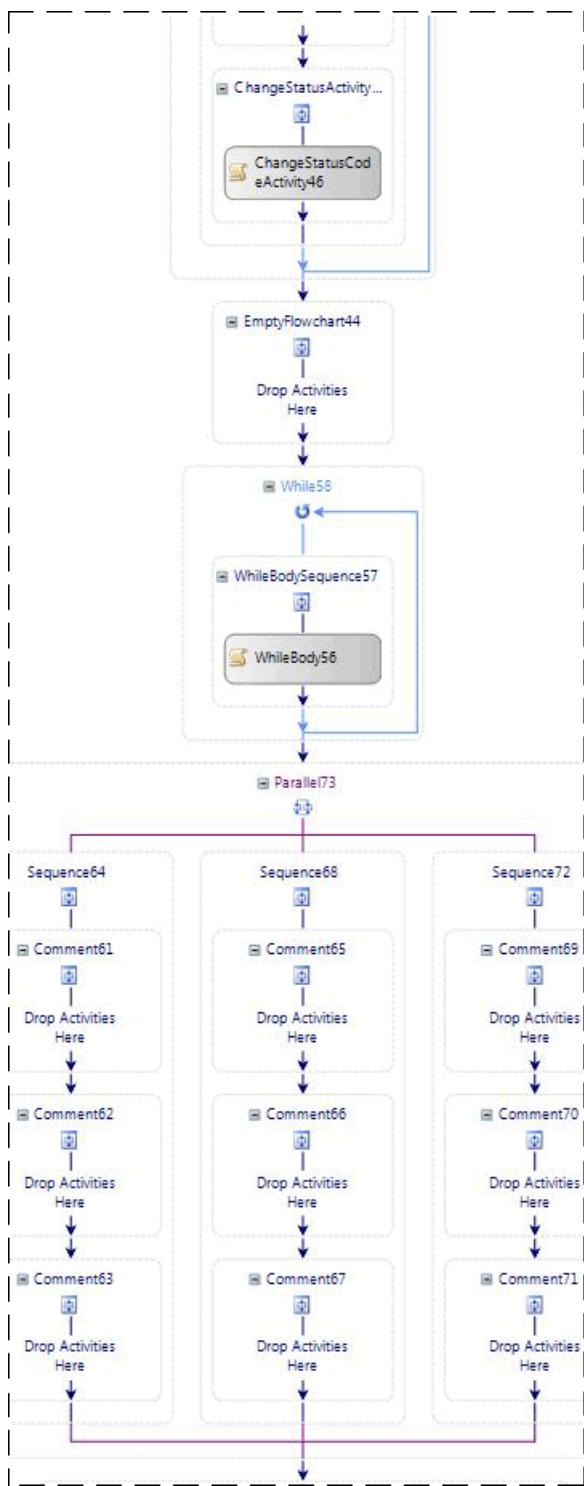
Where d is the depth and a is the number of activities per sequence. The logic behind these equations is that the first constant, multiplied by a, is the number of sequences and the second constant is the static number of activities in the current level. There are three flowchart child activities in each flowchart. At the bottom depth level, these flowcharts are empty but at the other levels they are copies of the main flowchart. The number of activities in each test variation's workflow definition is indicated in the following table:

|                      | WF3   | WF4   |
|----------------------|-------|-------|
| depth 1 sequence 1   | 28    | 26    |
| depth 1 sequence 100 | 523   | 422   |
| depth 3 sequence 3   | 494   | 442   |
| depth 5 sequence 5   | 5808  | 5082  |
| depth 7 sequence 1   | 30604 | 28418 |

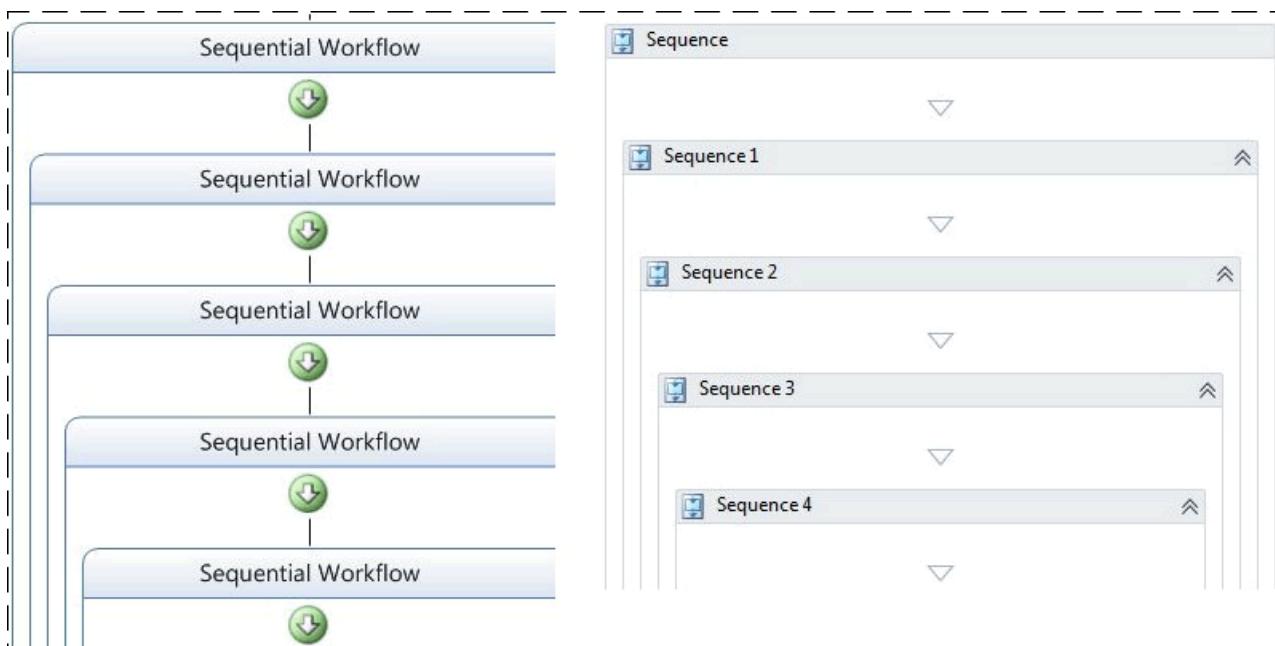
The number of activities in the workflow definition increases sharply with each depth level. But only one path per decision point is executed in a given workflow instance, so only a small subset of the actual activities are executed.



An equivalent workflow was created for WF3. The WF3 designer shows the entire workflow in the design area instead of nesting, therefore it is too big to display in this topic. A snippet of the workflow is shown below.

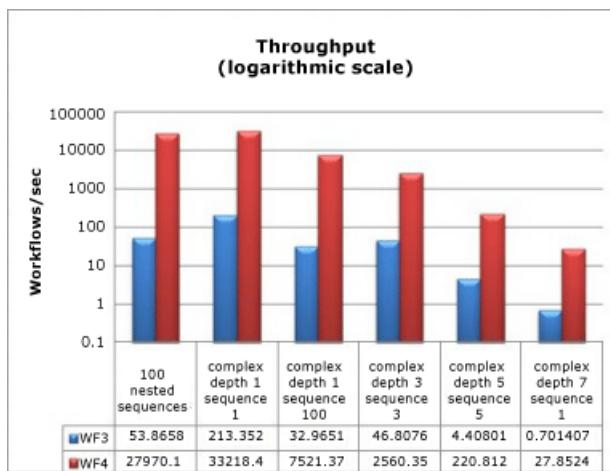


To exercise nesting in an extreme case, another workflow that is part of this test uses 100 nested sequences. In the innermost sequence is a single `Comment` or `CodeActivity`.



Tracking and persistence are not used as part of this test.

## Test Results



Even with complex workflows with lots of depth and a high number of activities, the performance results are consistent with other throughput numbers shown earlier in this article. WF4's throughput is orders of magnitude faster and has to be compared on a logarithmic scale.

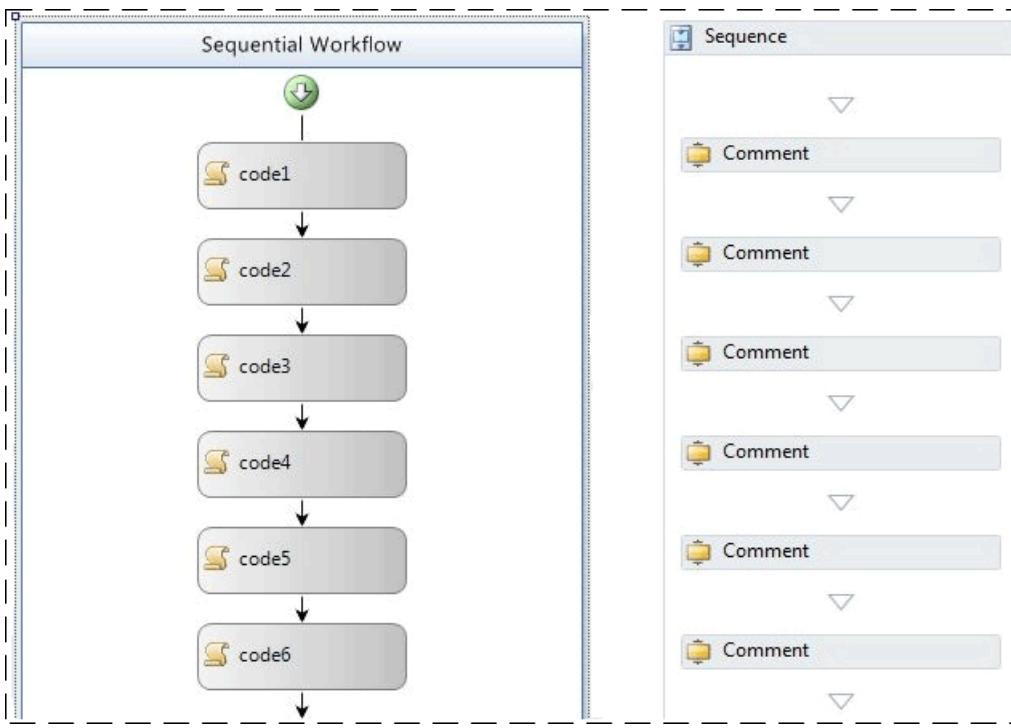
## Memory

The memory overhead of Windows Workflow Foundation is measured in two key areas: workflow complexity and number of workflow definitions. Memory measurements were taken on a Windows 7 64-bit workstation. There are many ways to obtain the measurement of working set size such as monitoring performance counters, polling Environment.WorkingSet, or using a tool like VMMMap available from [VMMMap](#). A combination of methods was used to obtain and verify the results of each test.

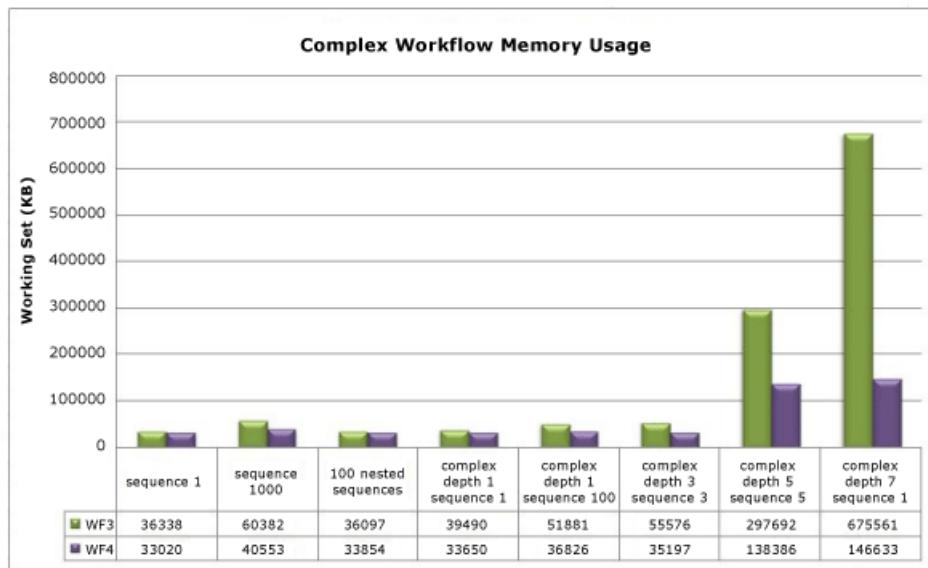
## Workflow Complexity Test

The workflow complexity test measures the working set difference based on the complexity of the workflow. In addition to the complex workflows used in the previous section, new variations are added to cover two basic cases: a single activity workflow and a sequence with 1000 activities. For these tests the workflows are initialized and executed to completion in a single serial loop for a period of one minute. Each test variation is run three times and the data recorded is the average of those three runs.

The two new basic tests have workflows that look like those shown below:



In the WF3 workflow shown above, empty [CodeActivity](#) activities are used. The WF4 workflow above uses [Comment](#) activities. The [Comment](#) activity was described in the Component-level Performance Comparisons section earlier in this article.



One of the clear trends to notice in this graph is that nesting has relatively minimal impact on memory usage in both WF3 and WF4. The most significant memory impact comes from the number of activities in a given workflow. Given the data from the sequence 1000, complex depth 5 sequence 5, and complex depth 7 sequence 1 variations, it is clear that as the number of activities enters the thousands, the memory usage increase becomes more noticeable. In the extreme case (depth 7 sequence 1) where there are ~29K activities, WF4 is using almost 79% less memory than WF3.

### Multiple Workflow Definitions Test

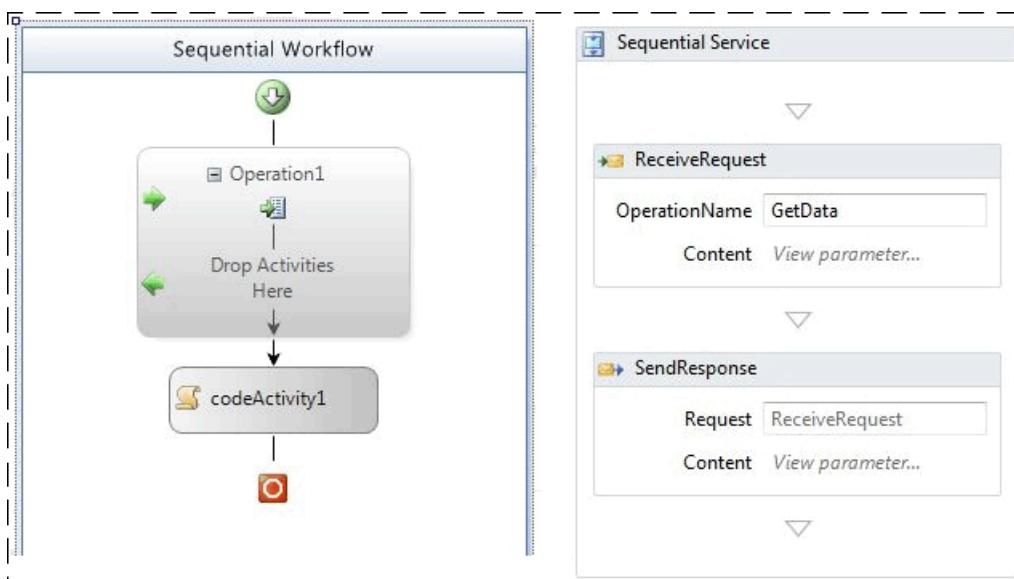
Measuring memory per workflow definition is divided into two different tests because of the available options for hosting workflows in WF3 and WF4. The tests are run in a different manner than the workflow complexity test in that a given workflow is instanced and executed only once per definition. This is because the workflow definition and its host remain in memory for the lifetime of the AppDomain. The memory used by running a given workflow instance should be cleaned up during garbage collection. The migration guidance for WF4 contains more detailed information on the hosting options. For more information, see [WF Migration Cookbook: Workflow Hosting](#).

Creating many workflow definitions for a workflow definition test can be done in several ways. For instance, one could use code generation to create a set of 1000 workflows that are identical except in name and save each of those workflows into separate files. This approach was taken for the console-hosted test. In WF3, the [WorkflowRuntime](#) class was used to run the workflow definitions. WF4 can either use [WorkflowApplication](#) to create a single workflow instance or directly use [WorkflowInvoker](#) to run the activity as if it were a method call. [WorkflowApplication](#) is a host of a single workflow instance and has closer feature parity to [WorkflowRuntime](#) so that was used in this test.

When hosting workflows in IIS it is possible to use a [VirtualPathProvider](#) to create a new [WorkflowServiceHost](#) instead of generating all of the XAMLX or XOML files. The [VirtualPathProvider](#) handles the incoming request and responds with a "virtual file" that can be loaded from a database or, in this case, generated on the fly. It is therefore unnecessary to create 1000 physical files.

The workflow definitions used in the console test were simple sequential workflows with a single activity. The single activity was an empty [CodeActivity](#) for the WF3 case and a [comment](#) activity for the WF4 case. The IIS-hosted case used workflows that start on receiving a message and end on sending a reply:

The following image shows a WF3 workflow with ReceiveActivity and a WF4 workflow with request/response pattern:



The following table shows the delta in working set between a single workflow definition and 1001 definitions:

| HOSTING OPTIONS                      | WF3 WORKING SET DELTA | WF4 WORKING SET DELTA |
|--------------------------------------|-----------------------|-----------------------|
| Console Application Hosted Workflows | 18 MB                 | 9 MB                  |
| IIS Hosted Workflow Services         | 446 MB                | 364 MB                |

Hosting workflow definitions in IIS consumes much more memory due to the [WorkflowServiceHost](#), detailed WCF service artifacts, and the message processing logic associated with the host.

For console hosting in WF3 the workflows were implemented in code instead of XOML. In WF4 the default is to use XAML. The XAML is stored as an embedded resource in the assembly and compiled during runtime to provide the implementation of the workflow. There is some overhead associated with this process. In order to make a fair comparison between WF3 and WF4, coded workflows were used instead of XAML. An example of one of the WF4 workflows is shown below:

```

public class Workflow1 : Activity
{
    protected override Func<Activity> Implementation
    {
        get
        {
            return new Func<Activity>(() =>
            {
                return new Sequence
                {
                    Activities = {
                        new Comment()
                    }
                };
            });
        }
        set
        {
            base.Implementation = value;
        }
    }
}

```

There are many other factors that can affect memory consumption. The same advice for all managed programs still applies. In IIS-hosted environments, the [WorkflowServiceHost](#) object created for a workflow definition stays in memory until the application pool is recycled. This should be kept in mind when writing extensions. Also, it is best to avoid "global" variables (variables scoped to the whole workflow) and limit the scope of variables wherever possible.

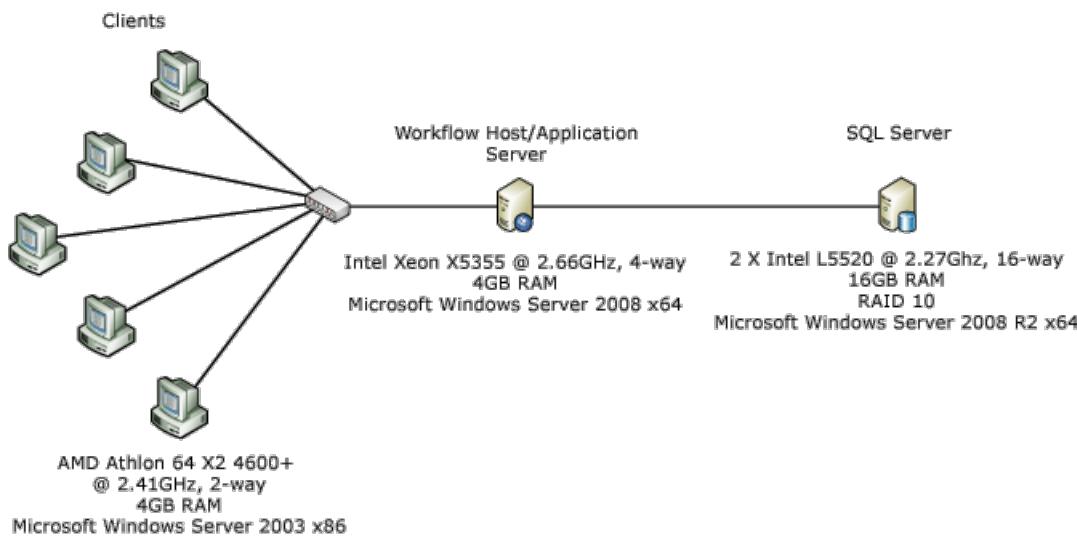
## Workflow Runtime Services

### Persistence

WF3 and WF4 both ship with a SQL persistence provider. The WF3 SQL persistence provider is a simple implementation that serializes the workflow instance and stores it in a blob. For this reason, the performance of this provider depends heavily on the size of the workflow instance. In WF3, the instance size could increase for many reasons, as is discussed previously in this paper. Many customers choose not to use the default SQL persistence provider because storing a serialized instance in a database gives no visibility into the state of the workflow. In order to find a particular workflow without knowing the workflow id, one would have to deserialize each persisted instance and examine the contents. Many developers prefer to write their own persistence providers to overcome this obstacle.

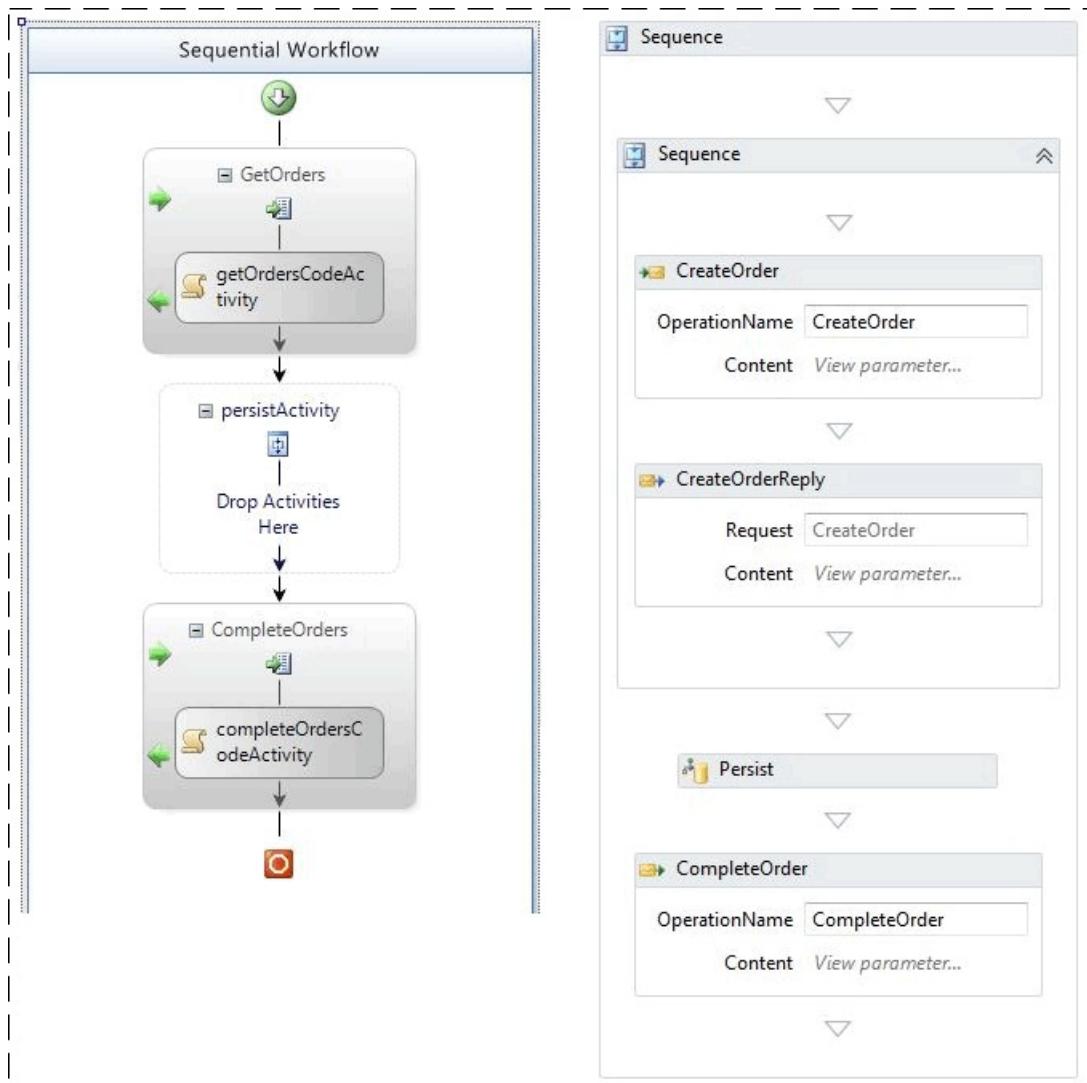
The WF4 SQL persistence provider has tried to address some of these concerns. The persistence tables expose certain information such as the active bookmarks and promotable properties. The new content-based correlation feature in WF4 would not perform well using the WF3 SQL persistence approach, which has driven some change in the organization of the persisted workflow instance. This makes the job of the persistence provider more complex and puts extra stress on the database.

### Environment Setup



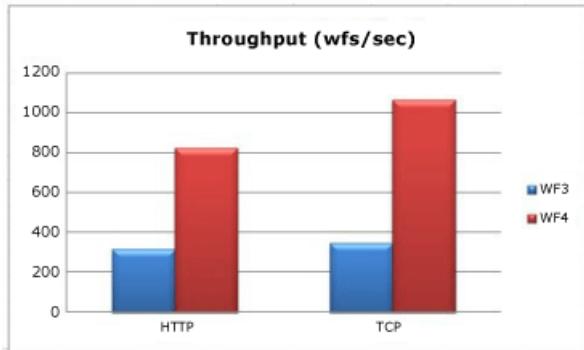
## Test Setup

Even with an improved feature set and better concurrency handling, the SQL persistence provider in WF4 is faster than the provider in WF3. To showcase this, two workflows that perform essentially the same operations in WF3 and WF4 are compared below.



The two workflows are both created by a received message. After sending an initial reply, the workflow is persisted. In the WF3 case, an empty [TransactionScopeActivity](#) is used to initiate the persistence. The same could be achieved in WF3 by marking an activity as "persist on close." A second, correlated message completes the workflow. The workflows are persisted but not unloaded.

## Test Results



When the transport between client and middle tier is HTTP, persistence in WF4 shows an improvement of 2.6 times. The TCP transport increases that factor to 3.0 times. In all cases, CPU utilization on the middle tier is 98% or higher. The reason that WF4 throughput is greater is due to the faster workflow runtime. The size of the serialized instance is low for both cases and is not a major contributing element in this situation.

Both the WF3 and WF4 workflows in this test use an activity to explicitly indicate when persistence should occur. This has the benefit of persisting the workflow without unloading it. In WF3, it is also possible to persist using the [TimeToUnload](#) feature, but this unloads the workflow instance from memory. If a developer using WF3 wants to make sure a workflow persists at certain points, they either have to alter the workflow definition or pay the cost for unloading and re-loading the workflow instance. A new feature in WF4 makes it possible to persist without unloading: [TimeToPersist](#). This feature allows the workflow instance to be persisted on idle but stay in memory until the [TimeToUnload](#) threshold is met or execution is resumed.

Note that the WF4 SQL persistence provider performs more work in the database tier. The SQL database can become a bottleneck so it is important to monitor the CPU and disk usage there. Be sure to include the following performance counters from the SQL database when performance testing workflow applications:

- PhysicalDisk\%Disk Read Time
- PhysicalDisk\% Disk Time
- PhysicalDisk\% Disk Write Time
- PhysicalDisk\% Avg. Disk Queue Length
- PhysicalDisk\Avg. Disk Read Queue Length
- PhysicalDisk\Avg. Disk Write Queue Length
- PhysicalDisk\Current Disk Queue Length
- Processor Information\% Processor Time
- SQLServer:Latches\Average Latch Wait Time (ms)
- SQLServer:Latches\Latch Waits/sec

## Tracking

Workflow tracking can be used to track the progress of a workflow. The information that is included in the tracking events is determined by a tracking profile. The more complex the tracking profile, the more expensive tracking becomes.

WF3 shipped with a SQL-based tracking service. This service could work in batched and non-batched modes. In non-batched mode, tracking events are written directly to the database. In batched mode, tracking events are collected into the same batch as the workflow instance state. The batched mode has the best performance for the widest range of workflow designs. However, batching can have a negative performance impact if the workflow runs many activities without persisting and those activities are tracked. This would commonly happen in loops and the best way to avoid this scenario is to design large loops to contain a persistence point. Introducing a persistence

point into a loop can negatively affect performance as well so it is important to measure the costs of each and come up with a balance.

WF4 is not shipped with a SQL tracking service. Recording tracking information to a SQL database can be handled better from an application server rather than built into the .NET Framework. Therefore SQL tracking is now handled by AppFabric. The out-of-the-box tracking provider in WF4 is based on Event Tracing for Windows (ETW).

ETW is a kernel-level, low-latency event system built into Windows. It uses a provider/consumer model that makes it possible to only incur the penalty for event tracing when there is actually a consumer. In addition to kernel events such as processor, disk, memory, and network usage, many applications leverage ETW as well. ETW events are more powerful than performance counters in that events can be customized to the application. An event can contain text such as a workflow ID or an informational message. Also, events are categorized with bitmasks so that consuming a certain subset of events will have less performance impact than capturing all events.

Benefits to the approach of using ETW for tracking instead of SQL include:

- Collection of tracking events can be separated to another process. This gives greater flexibility in how the events are recorded.
- ETW tracking events are easily combined with the WCF ETW events or other ETW providers such as a SQL Server or kernel provider.
- Workflow authors do not need to alter a workflow to work better with a particular tracking implementation, such as the WF3 SQL tracking service's batch mode.
- An administrator can turn tracking on or off without recycling the host process.

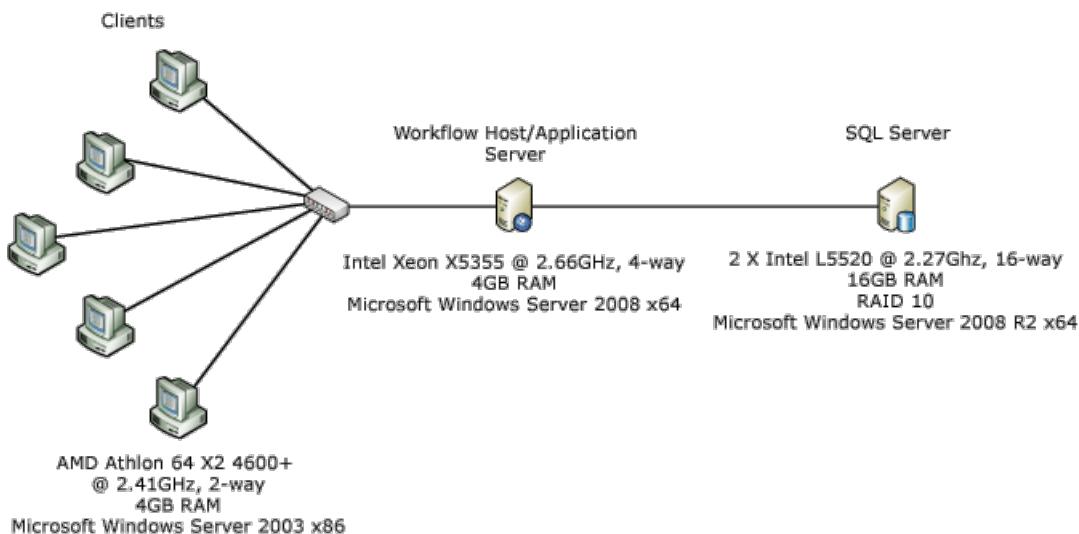
The performance benefits to ETW tracking come with a drawback. ETW events can be lost if the system is under intense resource pressure. The processing of events is not meant to block normal program execution and therefore it is not guaranteed that all ETW events will be broadcast to their subscribers. This makes ETW tracking great for health monitoring but not suitable for auditing.

While WF4 does not have a SQL tracking provider, AppFabric does. AppFabric's SQL tracking approach is to subscribe to ETW events with a Windows Service that batches the events and writes them to a SQL table designed for quick inserts. A separate job drains the data from this table and reforms it into reporting tables that can be viewed on the AppFabric dashboard. This means that a batch of tracking events is handled independent of the workflow it came from and therefore does not have to wait for a persistence point before being recorded.

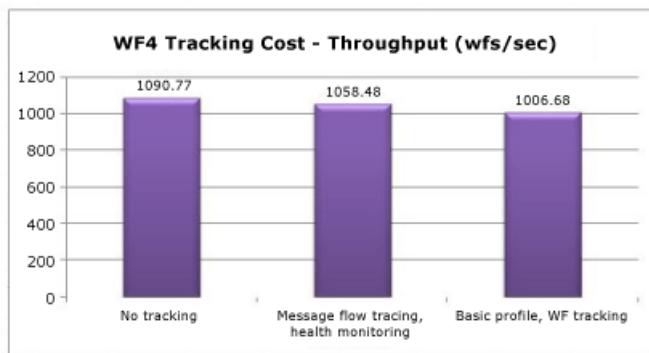
ETW events can be recorded with tools such as logman or xperf. The compact ETL file can be viewed with a tool like xperfview or converted to a more readable format, such as XML, with tracerpt. In WF3, the only option for getting tracking events without a SQL database is to create a custom tracking service. For more information about ETW, see [WCF Services and Event Tracing for Windows](#) and [Event Tracing - Windows applications](#).

Enabling workflow tracking will impact performance in varying degrees. The benchmark below uses the logman tool to consume the ETW tracking events and record them to an ETL file. The cost of the SQL tracking in AppFabric is not in the scope of this article. The basic tracking profile, also used in AppFabric, is shown in this benchmark. Also included is the cost of tracking only health monitoring events. These events are useful for troubleshooting problems and determining the average throughput of the system.

## Environment Setup



## Test Results

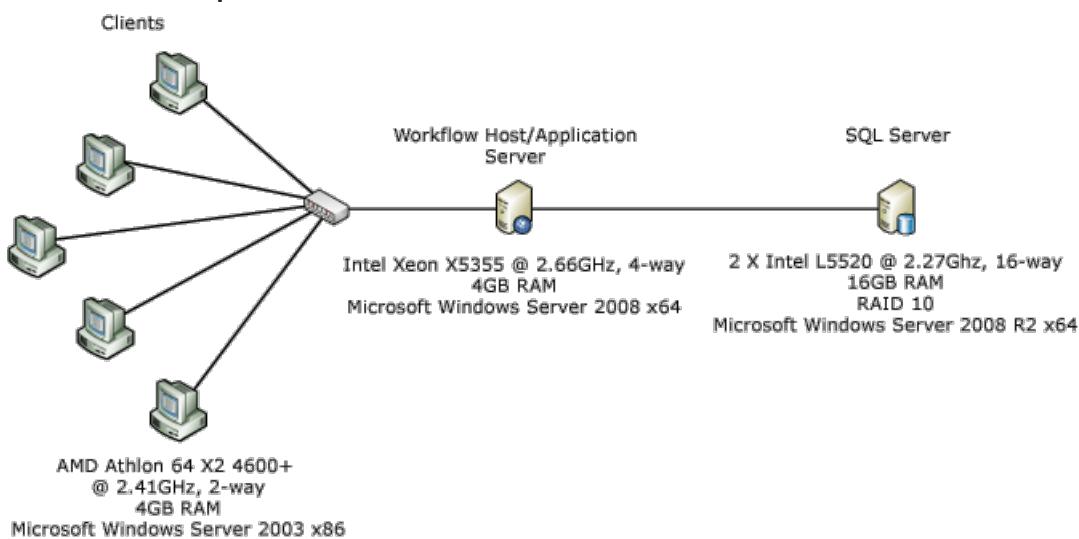


Health monitoring has roughly a 3% impact on throughput. The basic profile's cost is around 8%.

## Interop

WF4 is almost a complete rewrite of WF and therefore WF3 workflows and activities are not directly compatible with WF4. Many customers that adopted Windows Workflow Foundation early will have in-house or third-party workflow definitions and custom activities for WF3. One way to ease the transition to WF4 is to use the Interop activity, which can execute WF3 activities from within a WF4 workflow. It is recommended that the [Interop activity](#) only be used when necessary. For more information about migrating to WF4 check out the [WF4 Migration Guidance](#).

## Environment Setup



## Test Results

The following table shows the results of running a workflow containing five activities in a sequence in various configurations.

| TEST                                      | THROUGHPUT (WORKFLOWS/SEC) |
|---|----------------------------|
| WF3 Sequence in WF3 runtime               | 1,576                      |
| WF3 Sequence in WF4 runtime using Interop | 2,745                      |
| WF4 Sequence                              | 153,582                    |

There is a notable performance increase to using Interop over straight WF3. However, when compared against WF4 activities, the increase is negligible.

## Summary

Heavy investments in performance for WF4 have paid off in many crucial areas. Individual workflow component performance is in some cases hundreds of times faster in WF4 compared to WF3 due to a leaner WF runtime. Latency numbers are significantly better as well. This means the performance penalty for using WF as opposed to hand-coding WCF orchestration services is very small considering the added benefits of using WF. Persistence performance has increased by a factor of 2.5 - 3.0. Health monitoring by means of workflow tracking now has very little overhead. A comprehensive set of migration guides are available for those that are considering moving from WF3 to WF4. All of this should make WF4 an attractive option for writing complex applications.

# Extending Windows Workflow Foundation

3/9/2019 • 2 minutes to read • [Edit Online](#)

The following section describes how to extend Windows Workflow Foundation (WF) with custom activities and designers in rehosted environments outside Visual Studio 2010.

## In This Section

[Customizing the Workflow Design Experience](#) Indicates how the scenarios for designing custom activities and for rehosting the Windows Workflow Designer have been greatly simplified in .NET Framework 4. Development and deployment are now both easier and more flexible because the new activity designer programming model is built upon Windows Presentation Foundation (WPF).

## See also

- [Windows Workflow Foundation](#)

# Customizing the Workflow Design Experience

3/9/2019 • 2 minutes to read • [Edit Online](#)

The scenarios for designing custom activities and for rehosting the Windows Workflow Designer have been greatly simplified in .NET Framework 4. Development and deployment are now both easier and more flexible. The key infrastructural change is that the new activity designer programming model is built upon Windows Presentation Foundation (WPF). This gives you the ability to define activity designers declaratively and to rehost the Workflow Designer in other applications with comparative ease. When rehosting, a custom expression editor can be developed to support IntelliSense or a simplified expression domain. The integration with Windows Communication Foundation (WCF) has become more seamless with use of workflow services. Custom activity designers and the Model Item Tree can be used to enhance design time experiences in rehosted workflow designers.

## In This Section

### [Using Custom Activity Designers and Templates](#)

Describes how to create new custom activity designers and templates.

### [Rehosting the Workflow Designer](#)

Describes how to re-host the Windows Workflow Designer outside of Visual Studio and how to display validation errors.

### [Using a Custom Expression Editor](#)

Describes how to implement a custom expression editor to use with workflow designers rehosted outside of Visual Studio 2010.

## Reference

### [ActivityDesigner](#)

## See also

- [Extending Windows Workflow Foundation](#)
- [Designer](#)
- [Custom Activity Designers](#)
- [Designer ReHosting](#)

# Using Custom Activity Designers and Templates

3/9/2019 • 2 minutes to read • [Edit Online](#)

This section contains topics describing how to create custom activity designers and custom activity templates.

## In This Section

### [How to: Create a Custom Activity Designer](#)

Describes how to create a customized activity designer when the designers provided by the workflow are not appropriate to the design tasks.

### [How to: Create a Custom Activity Template](#)

Describes how to use custom activity templates to preconfigure activities so that users do not have to create each activity individually and configure their properties and other settings manually.

### [Using the ModelItem Editing Context](#)

Describes how to use the features of the ModelItem editing context to allow the designer to interact with the host.

### [Binding a custom activity property to a designer control](#)

Describes how to bind a listview control to an activity property in the designer.

## Reference

### [ActivityDesigner](#)

### [ExpressionTextBox](#)

### [ActivityAction](#)

### [IActivityTemplateFactory](#)

### [ModelTreeManager](#)

## Related Sections

### [Rehosting the Workflow Designer](#)

## External Resources

### [Custom Activities](#)

# How to: Create a Custom Activity Designer

3/9/2019 • 7 minutes to read • [Edit Online](#)

Custom activity designers are typically implemented so that their associated activities are composable with other activities whose designers can be dropped on to the design surface with them. This functionality requires that a custom activity designer provide a "drop zone" where an arbitrary activity can be placed and also the means to manage the resulting collection of elements on the design surface. This topic describes how to create a custom activity designer that contains such a drop zone and how to create a custom activity designer that provides that editing functionality needed to manage the collection of designer elements.

Custom activity designers typically inherit from [ActivityDesigner](#) which is the default base activity designer type for any activities without a specific designer. This type provides the design-time experience of interacting with the property grid and configuring basic aspects such as managing colors and icons.

[ActivityDesigner](#) uses two helper controls, [WorkflowItemPresenter](#) and [WorkflowItemsPresenter](#) to make it easier to develop custom activity designers. They handle common functionality like dragging and dropping of child elements, deletion, selection, and addition of those child elements. The [WorkflowItemPresenter](#) allows a single child UI element inside, providing the "drop zone", it while the [WorkflowItemsPresenter](#) can provide support multiple UI elements, including additional functionality like the ordering, moving, deleting, and adding of child elements.

The other key part of the story that needs highlighting in the implementation of a custom activity designer concerns the way in which the visual edits are bound using WPF data binding to the instance stored in memory of what we are editing in the designer. This is accomplished by the Model Item tree, which is also responsible for enabling change notification and the tracking of events like changes in states.

This topic outlines two procedures.

1. The first procedure describes how to create a custom activity designer with a [WorkflowItemPresenter](#) that provides the drop zone that receives other activities. This procedure is based on the [Custom Composite Designers - Workflow Item Presenter](#) sample.
2. The second procedure describes how to create a custom activity designer with a [WorkflowItemsPresenter](#) that provides the functionality needed to edit of a collection of contained elements. This procedure is based on the [Custom Composite Designers - Workflow Items Presenter](#) sample.

## To create a custom activity designer with a drop zone using WorkflowItemPresenter

1. Start Visual Studio 2010.
2. On the **File** menu, point to **New**, and then select **Project....**

The **New Project** dialog box opens.

3. In the **Installed Templates** pane, select **Windows** from your preferred language category.
4. In the **Templates** pane, select **WPF Application**.
5. In the **Name** box, enter `UsingWorkflowItemPresenter`.
6. In the **Location** box, enter the directory in which you want to save your project, or click **Browse** to navigate to it.

7. In the **Solution** box, accept the default value.
8. Click **OK**.
9. Right-click the MainWindows.xaml file in the **Solution Explorer**, select **Delete** and confirm **OK** in the **Microsoft Visual Studio** dialogue box.
10. Right-click the UsingWorkflowItemPresenter project in **Solution Explorer**, select **Add**, then **New Item...** to bring up the **Add New Item** dialogue and select the **WPF** category from the **Installed Templates** section on the left.
11. Select the **Window (WPF)** template, name it `RehostingWFDesigner`, and click **Add**.
12. Open the RehostingWFDesigner.xaml file and paste the following code into it to define the UI for the application.

```

<Window x:Class=" UsingWorkflowItemPresenter.RehostingWFDesigner"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sapt="clr-
namespace:System.Activities.Presentation.Toolbox;assembly=System.Activities.Presentation"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
    Title="Window1" Height="600" Width="900">
    <Window.Resources>
        <sys:String x:Key="AssemblyName">System.Activities, Version=4.0.0.0, Culture=neutral,
    PublicKeyToken=31bf3856ad364e35</sys:String>
    </Window.Resources>
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="2*"/>
            <ColumnDefinition Width="7*"/>
            <ColumnDefinition Width="3*"/>
        </Grid.ColumnDefinitions>
        <Border Grid.Column="0">
            <sapt:ToolboxControl Name="Toolbox">
                <sapt:ToolboxCategory CategoryName="Basic">
                    <sapt:ToolboxItemWrapper AssemblyName="{StaticResource AssemblyName}" >
                        <sapt:ToolboxItemWrapper.ToolName>
                            System.Activities.Statements.Sequence
                        </sapt:ToolboxItemWrapper.ToolName>
                    </sapt:ToolboxItemWrapper>
                    <sapt:ToolboxItemWrapper AssemblyName="{StaticResource AssemblyName}">
                        <sapt:ToolboxItemWrapper.ToolName>
                            System.Activities.Statements.WriteLine
                        </sapt:ToolboxItemWrapper.ToolName>

                    </sapt:ToolboxItemWrapper>
                    <sapt:ToolboxItemWrapper AssemblyName="{StaticResource AssemblyName}">
                        <sapt:ToolboxItemWrapper.ToolName>
                            System.Activities.Statements.If
                        </sapt:ToolboxItemWrapper.ToolName>

                    </sapt:ToolboxItemWrapper>
                    <sapt:ToolboxItemWrapper AssemblyName="{StaticResource AssemblyName}">
                        <sapt:ToolboxItemWrapper.ToolName>
                            System.Activities.Statements.While
                        </sapt:ToolboxItemWrapper.ToolName>

                    </sapt:ToolboxItemWrapper>
                </sapt:ToolboxCategory>
            </sapt:ToolboxControl>
        </Border>
        <Border Grid.Column="1" Name="DesignerBorder"/>
        <Border Grid.Column="2" Name="PropertyBorder"/>
    </Grid>
</Window>

```

13. To associate an activity designer with an activity type, you must register that activity designer with the metadata store. To do this, add the `RegisterMetadata` method to the `RehostingWFDesigner` class. Within the scope of the `RegisterMetadata` method, create an `AttributeTableBuilder` object and call the `AddCustomAttributes` method to add the attributes to it. Call the `AddAttributeTable` method to add the `AttributeTable` to the metadata store. The following code contains the rehosting logic for the designer. It registers the metadata, puts the `SimpleNativeActivity` into the toolbox, and creates the workflow. Put this code into the `RehostingWFDesigner.xaml.cs` file.

```

using System;
using System.Activities.Core.Presentation;
using System.Activities.Presentation;
using System.Activities.Presentation.Metadata;
using System.Activities.Presentation.Toolbox;
using System.Activities.Statements;
using System.ComponentModel;
using System.Windows;

namespace UsingWorkflowItemPresenter
{
    // Interaction logic for RehostingWFDesigner.xaml
    public partial class RehostingWFDesigner
    {
        public RehostingWFDesigner()
        {
            InitializeComponent();
        }

        protected override void OnInitialized(EventArgs e)
        {
            base.OnInitialized(e);
            // register metadata
            (new DesignerMetadata()).Register();
            RegisterCustomMetadata();
            // add custom activity to toolbox
            Toolbox.Categories.Add(new ToolboxCategory("Custom activities"));
            Toolbox.Categories[1].Add(new ToolboxItemWrapper(typeof(SimpleNativeActivity)));

            // create the workflow designer
            WorkflowDesigner wd = new WorkflowDesigner();
            wd.Load(new Sequence());
            DesignerBorder.Child = wd.View;
            PropertyBorder.Child = wd.PropertyInspectorView;
        }

        void RegisterCustomMetadata()
        {
            AttributeTableBuilder builder = new AttributeTableBuilder();
            builder.AddCustomAttributes(typeof(SimpleNativeActivity), new
DesignAttribute(typeof(SimpleNativeDesigner)));
            MetadataStore.AddAttributeTable(builder.CreateTable());
        }
    }
}

```

14. Right-click the References directory in Solution Explorer and select **Add Reference ...** to bring up the **Add Reference** dialogue.
15. Click the **.NET** tab, locate the assembly named **System.Activities.Core.Presentation**, select it and click **OK**.
16. Using the same procedure, add references to the following assemblies:
  - a. System.Data.DataSetExtensions.dll
  - b. System.Activities.Presentation.dll
  - c. System.ServiceModel.Activities.dll
17. Open the App.xaml file and change the value of the StartUpUri to "RehostingWFDesigner.xaml".
18. Right-click the UsingWorkflowItemPresenter project in **Solution Explorer**, select **Add**, then **New Item...** to bring up the **Add New Item** dialogue and select the **Workflow** category from the **Installed Templates**

section on the left.

19. Select the **Activity Designer** template, name it `SimpleNativeDesigner`, and click **Add**.
20. Open the SimpleNativeDesigner.xaml file and paste the following code into it. Note this code uses **ActivityDesigner** as your root element and shows how binding is used to integrate **WorkflowItemPresenter** into your designer so a child type can be displayed in your composite activity designer.

**NOTE**

The schema for **ActivityDesigner** allows the addition of only one child element to your custom activity designer definition; however, this element could be a `StackPanel`, `Grid`, or some other composite UI element.

```
<sap:ActivityDesigner x:Class=" UsingWorkflowItemPresenter.SimpleNativeDesigner"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sap="clr-namespace:System.Activities.Presentation;assembly=System.Activities.Presentation"
    xmlns:sapv="clr-
    namespace:System.Activities.Presentation.View;assembly=System.Activities.Presentation">
    <sap:ActivityDesigner.Resources>
        <DataTemplate x:Key="Collapsed">
            <StackPanel>
                <TextBlock>This is the collapsed view</TextBlock>
            </StackPanel>
        </DataTemplate>
        <DataTemplate x:Key="Expanded">
            <StackPanel>
                <TextBlock>Custom Text</TextBlock>
                <sap:WorkflowItemPresenter Item="{Binding Path=ModelItem.Body, Mode=TwoWay}"
                    HintText="Please drop an activity here" />
            </StackPanel>
        </DataTemplate>
        <Style x:Key="ExpandOrCollapsedStyle" TargetType="{x:Type ContentPresenter}">
            <Setter Property="ContentTemplate" Value="{DynamicResource Collapsed}"/>
            <Style.Triggers>
                <DataTrigger Binding="{Binding Path>ShowExpanded}" Value="true">
                    <Setter Property="ContentTemplate" Value="{DynamicResource Expanded}"/>
                </DataTrigger>
            </Style.Triggers>
        </Style>
    </sap:ActivityDesigner.Resources>
    <Grid>
        <ContentPresenter Style="{DynamicResource ExpandOrCollapsedStyle}" Content="{Binding}" />
    </Grid>
</sap:ActivityDesigner>
```

21. Right-click the UsingWorkflowItemPresenter project in **Solution Explorer**, select **Add**, then **New Item...** to bring up the **Add New Item** dialogue and select the **Workflow** category from the **Installed Templates** section on the left.
22. Select the **Code Activity** template, name it `SimpleNativeActivity`, and click **Add**.
23. Implement the `SimpleNativeActivity` class by entering the following code into the SimpleNativeActivity.cs file.

```

using System.Activities;

namespace UsingWorkflowItemPresenter
{
    public sealed class SimpleNativeActivity : NativeActivity
    {
        // this property contains an activity that will be scheduled in the execute method
        // the WorkflowItemPresenter in the designer is bound to this to enable editing
        // of the value
        public Activity Body { get; set; }

        protected override void CacheMetadata(NativeActivityMetadata metadata)
        {
            metadata.AddChild(Body);
            base.CacheMetadata(metadata);

        }

        protected override void Execute(NativeActivityContext context)
        {
            context.ScheduleActivity(Body);
        }
    }
}

```

24. Select **Build Solution** from the **Build** menu.

25. Select **Start Without Debugging** from the **Debug** menu to open the rehosted custom design window.

#### To create a custom activity designer using **WorkflowItemsPresenter**

1. The procedure for the second custom activity designer is the parallels the first with a few modifications, the first of which is to name the second application `UsingWorkflowItemsPresenter`. Also this application does not define a new custom activity.
2. Key differences are contained in the `CustomParallelDesigner.xaml` and `RehostingWFDesigner.xaml.cs` files. Here is the code from the `CustomParallelDesigner.xaml` file that defines the UI.

```

<sap:ActivityDesigner x:Class=" UsingWorkflowItemsPresenter.CustomParallelDesigner"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sap="clr-namespace:System.Activities.Presentation;assembly=System.Activities.Presentation"
    xmlns:sapv="clr-
    namespace:System.Activities.Presentation.View;assembly=System.Activities.Presentation">
    <sap:ActivityDesigner.Resources>
        <DataTemplate x:Key="Collapsed">
            <TextBlock>This is the Collapsed View</TextBlock>
        </DataTemplate>
        <DataTemplate x:Key="Expanded">
            <StackPanel>
                <TextBlock HorizontalAlignment="Center">This is the</TextBlock>
                <TextBlock HorizontalAlignment="Center">extended view</TextBlock>
                <sap:WorkflowItemsPresenter HintText="Drop Activities Here"
                    Items="{Binding Path=ModelItem.Branches}">
                    <sap:WorkflowItemsPresenter.SpacerTemplate>
                        <DataTemplate>
                            <Ellipse Width="10" Height="10" Fill="Black"/>
                        </DataTemplate>
                    </sap:WorkflowItemsPresenter.SpacerTemplate>
                    <sap:WorkflowItemsPresenter.ItemsPanel>
                        <ItemsPanelTemplate>
                            <StackPanel Orientation="Horizontal"/>
                        </ItemsPanelTemplate>
                    </sap:WorkflowItemsPresenter.ItemsPanel>
                </sap:WorkflowItemsPresenter>
            </StackPanel>
        </DataTemplate>
        <Style x:Key="ExpandOrCollapsedStyle" TargetType="{x:Type ContentPresenter}">
            <Setter Property="ContentTemplate" Value="{DynamicResource Collapsed}"/>
            <Style.Triggers>
                <DataTrigger Binding="{Binding Path>ShowExpanded}" Value="true">
                    <Setter Property="ContentTemplate" Value="{DynamicResource Expanded}"/>
                </DataTrigger>
            </Style.Triggers>
        </Style>
    </sap:ActivityDesigner.Resources>
    <Grid>
        <ContentPresenter Style="{DynamicResource ExpandOrCollapsedStyle}" Content="{Binding}"/>
    </Grid>
</sap:ActivityDesigner>

```

3. Here is the code from the RehostingWFDesigner.xaml.cs file that provides the rehosting logic.

```

using System;
using System.Activities.Core.Presentation;
using System.Activities.Presentation;
using System.Activities.Presentation.Metadata;
using System.Activities.Statements;
using System.ComponentModel;
using System.Windows;

namespace UsingWorkflowItemsPresenter
{
    public partial class RehostingWfDesigner : Window
    {
        public RehostingWfDesigner()
        {
            InitializeComponent();
        }

        protected override void OnInitialized(EventArgs e)
        {
            base.OnInitialized(e);
            // register metadata
            (new DesignerMetadata()).Register();
            RegisterCustomMetadata();

            // create the workflow designer
            WorkflowDesigner wd = new WorkflowDesigner();
            wd.Load(new Sequence());
            DesignerBorder.Child = wd.View;
            PropertyBorder.Child = wd.PropertyInspectorView;
        }

        void RegisterCustomMetadata()
        {
            AttributeTableBuilder builder = new AttributeTableBuilder();
            builder.AddCustomAttributes(typeof(Parallel), new
DesignAttribute(typeof(CustomParallelDesigner)));
            MetadataStore.AddAttributeTable(builder.CreateTable());
        }
    }
}

```

## See also

- [ActivityDesigner](#)
- [WorkflowItemPresenter](#)
- [WorkflowItemsPresenter](#)
- [WorkflowViewElement](#)
- [ModelItem](#)
- [Customizing the Workflow Design Experience](#)

# How to: Create a Custom Activity Template

3/9/2019 • 3 minutes to read • [Edit Online](#)

Custom activity templates are used to customize the configuration of activities, including custom composite activities, so that users do not have to create each activity individually and configure their properties and other settings manually. These custom templates can be made available in the **Toolbox** on the Windows Workflow Designer or from a rehosted designer, from which users can drag them onto the preconfigured design surface. Workflow Designer ships with good examples of such templates: the [SendAndReceiveReply Template Designer](#) and the [ReceiveAndSendReply Template Designer](#) in the [Messaging Activity Designers](#) category.

The first procedure in this topic describes how to create a custom activity template for a **Delay** activity and the second procedure describes briefly how to make it available in a Workflow Designer to verify that the custom template works.

Custom activity templates must implement the [IActivityTemplateFactory](#). The interface has a single [Create](#) method with which you can create and configure the activity instances used in the template.

## To create a template for the Delay activity

1. Start Visual Studio 2010.
2. On the **File** menu, point to **New**, and then select **Project**.  
The **New Project** dialog box opens.
3. In the **Project Types** pane, select **Workflow** from either the **Visual C# projects** or **Visual Basic** groupings depending on your language preference.
4. In the **Templates** pane, select **Activity Library**.
5. In the **Name** box, enter `DelayActivityTemplate`.
6. Accept the defaults in the **Location** and **Solution name** text boxes, and then click **OK**.
7. Right-click the References directory of the DelayActivityTemplate project in **Solution Explorer** and choose **Add Reference** to open the **Add Reference** dialog box.
8. Go to the **.NET** tab and select **PresentationFramework** from the **Component Name** column on the left and click **OK** to add a reference to the PresentationFramework.dll file.
9. Repeat this procedure to add references to the System.Activities.Presentation.dll and the WindowsBase.dll files.
10. Right-click the DelayActivityTemplate project in **Solution Explorer** and choose **Add** and then **New Item** to open the **Add New Item** dialog box.
11. Select the **Class** template, name it MyDelayTemplate, and then click **OK**.
12. Open the MyDelayTemplate.cs file and add the following statements.

```
//Namespaces added
using System.Activities;
using System.Activities.Statements;
using System.Activities.Presentation;
using System.Windows;
```

13. Implement the `IActivityTemplateFactory` with the `MyDelayActivity` class with the following code. This configures the delay to have a duration of 10 seconds.

```
public sealed class MyDelayActivity : IActivityTemplateFactory
{
    public Activity Create(System.Windows.DependencyObject target)
    {
        return new System.Activities.Statements.Delay
        {
            DisplayName = "DelayActivityTemplate",
            Duration = new TimeSpan(0, 0, 10)
        };
    }
}
```

14. Select **Build Solution** from the **Build** menu to generate the DelayActivityTemplate.dll file.

#### To make the template available in a Workflow Designer

1. Right-click the DelayActivityTemplate solution in **Solution Explorer** and choose **Add** and then **New Project** to open the **Add New Project** dialog box.
2. Select the **Workflow Console Application** template, name it `CustomActivityTemplateApp`, and then click **OK**.
3. Right-click the References directory of the CustomActivityTemplateApp project in **Solution Explorer** and choose **Add Reference** to open the **Add Reference** dialog box.
4. Go to the **Projects** tab and select **DelayActivityTemplate** from the **Project Name** column on the left and click **OK** to add a reference to the DelayActivityTemplate.dll file that you created in the first procedure.
5. Right-click the CustomActivityTemplateApp project in **Solution Explorer** and choose **Build** to compile the application.
6. Right-click the CustomActivityTemplateApp project in **Solution Explorer** and choose **Set as Startup Project**.
7. Select **Start Without Debugging** from the **Debug** menu and press any key to continue when prompted from the cmd.exe window.
8. Open the Workflow1.xaml file and open the **Toolbox**.
9. Locate the **MyDelayActivity** template in the **DelayActivityTemplate** category. Drag it onto the design surface. Confirm in the **Properties** window that the `Duration` property has been set to 10 seconds.

## Example

The MyDelayActivity.cs file should contain the following code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

//Namespaces added
using System.Activities;
using System.Activities.Statements;
using System.Activities.Presentation;
using System.Windows;

namespace DelayActivityTemplate
{
    public sealed class MyDelayActivity : IActivityTemplateFactory
    {
        public Activity Create(System.Windows.DependencyObject target)
        {
            return new System.Activities.Statements.Delay
            {
                DisplayName = "DelayActivityTemplate",
                Duration = new TimeSpan(0, 0, 10)

            };
        }
    }
}
```

## See also

- [IActivityTemplateFactory](#)
- [Customizing the Workflow Design Experience](#)

# Using the ModelItem Editing Context

3/14/2019 • 2 minutes to read • [Edit Online](#)

The [ModelItem](#) editing context is the object that the host application uses to communicate with the designer. [EditingContext](#) exposes two methods, [Items](#) and [Services](#), which can be used

## The Items collection

The [Items](#) collection is used to access data that is shared between the host and the designer, or data that is available to all designers. This collection has the following capabilities, accessed via the [ContextItemManager](#) class:

1. [GetValue](#)
2. [Subscribe](#)
3. [Unsubscribe](#)
4. [SetValue](#)

## The Services collection

The [Services](#) collection is used to access services that the designer uses to interact with the host, or services that all designers use. This collection has the following methods of note:

1. [Publish](#)
2. [Subscribe](#)
3. [Unsubscribe](#)
4. [GetService](#)

## Assigning a designer an activity

To specify which designer an activity uses, the `Designer` attribute is used.

```
[Designer(typeof(MyClassDesigner))]  
public sealed class MyClass : CodeActivity  
{
```

## Creating a service

To create a service that serves as a conduit of information between the designer and the host, an interface and an implementation must be created. The interface is used by the [Publish](#) method to define the members of the service, and the implementation contains the logic for the service. In the following code example, a service interface and implementation are created.

```

public interface IMyService
{
    IEnumerable<string> GetValues(string DisplayName);
}

public class MyServiceImpl : IMyService
{
    public IEnumerable<string> GetValues(string DisplayName)
    {
        return new string[] {
            DisplayName + " One",
            DisplayName + " Two",
            "Three " + DisplayName
        } ;
    }
}

```

## Publishing a service

For a designer to consume a service, it must first be published by the host using the [Publish](#) method.

```
this.Context.Services.Publish<IMyService>(new MyServiceImpl);
```

## Subscribing to a service

The designer obtains access to the service using the [Subscribe](#) method in the [OnModelItemChanged](#) method. The following code snippet demonstrates how to subscribe to a service.

```

protected override void OnModelItemChanged(object newItem)
{
    if (!subscribed)
    {
        this.Context.Services.Subscribe<IMyService>(
            servInstance =>
            {
                listBox1.ItemsSource =
                    servInstance.GetValues(this.ModelItem.Properties["DisplayName"].ComputedValue.ToString());
            }
        );
        subscribed = true;
    }
}

```

## Sharing data using the Items collection

Using the Items collection is similar to using the Services collection, except that [SetValue](#) is used instead of Publish. This collection is more appropriate for sharing simple data between the designers and the host, rather than complex functionality.

## EditingContext host items and services

The .NET Framework provides a number of built-in items and services accessed through the editing context.

Items:

- [AssemblyContextControlItem](#): Manages the list of referenced local assemblies that will be used inside the workflow for controls (such as the expression editor).

- [ReadOnlyState](#): Indicates whether the designer is in a read-only state.
- [Selection](#): Defines the collection of objects that are currently selected.
- [WorkflowCommandExtensionItem](#):
- [WorkflowFileItem](#): Provides information on the file that the current editing session is based on.

Services:

- [AttachedPropertiesService](#): Allows properties to be added to the current instance, using [AddProperty](#).
- [DesignerView](#): Allows access to the properties of the designer canvas.
- [IActivityToolboxService](#): Allows the contents of the toolbox to be updated.
- [ICommandService](#): Used to integrate designer commands (such as Context Menu) with custom-provided service implementations.
- [IDesignerDebugView](#): Provides functionality for the designer debugger.
- [IExpressionEditorService](#): Provides access to the Expression Editor dialog.
- [IIintegratedHelpService](#): Provides the designer with integrated help functionality.
- [IValidationErrorsService](#): Provides access to validation errors using [ShowValidationErrors](#).
- [IWorkflowDesignerStorageService](#): Provides an internal service to store and retrieve data. This service is used internally by the .NET Framework, and is not intended for external use.
- [IXamlLoadErrorService](#): Provides access to the XAML load error collection using [ShowXamlLoadErrors](#).
- [ModelService](#): Used by the designer to interact with the model of the workflow being edited.
- [ModelTreeManager](#): Provides access to the root of the model item tree using [Root](#).
- [UndoEngine](#): Provides undo and redo functionality.
- [ViewService](#): Maps visual elements to underlying model items.
- [ViewStateService](#): Stores view states for model items.
- [VirtualizedContainerService](#): Used to customize the virtual container UI behavior.
- [WindowHelperService](#): Used to register and unregister delegates for event notifications. Also allows a window owner to be set.

# Binding a custom activity property to a designer control

3/6/2019 • 3 minutes to read • [Edit Online](#)

Binding a text box designer control to an activity argument is fairly straightforward; binding a complex designer control (such as a combo box) to an activity argument may present challenges, however. This topic discusses how to bind an activity argument to a combo box control on a custom activity designer.

## Creating the combo box item converter

1. Create a new empty solution in Visual Studio called CustomProperty.
2. Create a new class called ComboBoxItemConverter. Add a reference to System.Windows.Data, and have the class derive from [IValueConverter](#). Have Visual Studio implement the interface to generate stubs for `Convert` and `ConvertBack`.
3. Add the following code to the `Convert` method. This code converts the activity's `InArgument<T>` of type `String` to the value to be placed in the designer.

```
ModelItem modelItem = value as ModelItem;
if (value != null)
{
    InArgument<string> inArgument = modelItem.GetCurrentValue() as InArgument<string>;
    if (inArgument != null)
    {
        Activity<string> expression = inArgument.Expression;
        VisualBasicValue<string> vbexpression = expression as VisualBasicValue<string>;
        Literal<string> literal = expression as Literal<string>;
        if (literal != null)
        {
            return "" + literal.Value + "";
        }
        else if (vbexpression != null)
        {
            return vbexpression.ExpressionText;
        }
    }
}
return null;
```

The expression in the above code snippet can also be created using `CSharpValue<TResult>` instead of `VisualBasicValue<TResult>`.

```

ModelItem modelItem = value as ModelItem;
if (value != null)
{
    InArgument<string> inArgument = modelItem.GetCurrentValue() as InArgument<string>;
    if (inArgument != null)
    {
        Activity<string> expression = inArgument.Expression;
        CSharpValue<string> csexpression = expression as CSharpValue<string>;
        Literal<string> literal = expression as Literal<string>;
        if (literal != null)
        {
            return "" + literal.Value + "";
        }
        else if (csexpression != null)
        {
            return csexpression.ExpressionText;
        }
    }
}
return null;

```

- Add the following code to the `ConvertBack` method. This code converts the incoming combo box item back to an `InArgument<T>`.

```

// Convert combo box value to InArgument<string>
string itemContent = (string)((ComboBoxItem)value).Content;
VisualBasicValue<string> vbArgument = new VisualBasicValue<string>(itemContent);
InArgument<string> inArgument = new InArgument<string>(vbArgument);
return inArgument;

```

The expression in the above code snippet can also be created using `CSharpValue<TResult>` instead of `VisualBasicValue<TResult>`.

```

// Convert combo box value to InArgument<string>
string itemContent = (string)((ComboBoxItem)value).Content;
CSharpValue<string> csArgument = new CSharpValue<string>(itemContent);
InArgument<string> inArgument = new InArgument<string>(csArgument);
return inArgument;

```

## Adding the ComboBoxItemConverter to the custom designer of an activity

- Add a new item to the project. In the New Item dialog, select the Workflow node and select Activity Designer as the type of the new item. Name the item CustomPropertyDesigner.
- Add a Combo Box to the new designer. In the Items property, add a couple of items to the combo box, with Content values of "Item1" and 'Item2'.
- Modify the XAML of the combo box to add the new item converter as the item converter to be used for the combo box. The converter is added as a resource in the ActivityDesigner.Resources segment, and specifies the converter in the Converter attribute for the `ComboBox`. Note that the namespace of the project is specified in the namespaces attributes for the activity designer; if the designer is to be used in a different project, this namespace will need to be changed.

```

<sap:ActivityDesigner x:Class="CustomProperty.CustomPropertyDesigner"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:c="clr-namespace:CustomProperty"
    xmlns:sap="clr-namespace:System.Activities.Presentation;assembly=System.Activities.Presentation"
    xmlns:sapv="clr-
    namespace:System.Activities.Presentation.View;assembly=System.Activities.Presentation">

    <sap:ActivityDesigner.Resources>
        <ResourceDictionary>
            <c:ComboBoxItemConverter x:Key="comboBoxItemConverter"/>
        </ResourceDictionary>
    </sap:ActivityDesigner.Resources>
    <Grid>
        <ComboBox SelectedValue="{Binding Path=ModelItem.Text, Mode=TwoWay, Converter={StaticResource
    comboBoxItemConverter}}" Height="23" HorizontalAlignment="Left" Margin="132,5,0,0" Name="comboBox1"
    VerticalAlignment="Top" Width="120" ItemsSource="{Binding}">
            <ComboBoxItem>item1</ComboBoxItem>
            <ComboBoxItem>item2</ComboBoxItem>
        </ComboBox>
    </Grid>
</sap:ActivityDesigner>

```

4. Create a new item of type [CodeActivity](#). The default code created by the IDE for the activity will be sufficient for this example.
5. Add the following attribute to the class definition:

```
[Designer(typeof(CustomPropertyDesigner))]
```

This line associates the new designer with the new class.

The new activity should now be associated with the designer. To test the new activity, add it to a workflow, and set the combo box to the two values. The properties window should update to reflect the combo box value.

# Rehosting the Workflow Designer

3/9/2019 • 2 minutes to read • [Edit Online](#)

The Windows Workflow Designer can be rehosted in environments outside of Visual Studio 2012 for the purposes of creating, modifying, and monitoring workflows.

The [WorkflowDesigner](#) type is a wrapper of the canvas, property grid, and other elements, and exposes a basic programming model to handle the majority of designer rehosting scenarios. Hosting the [WorkflowDesigner](#) inside a Windows Presentation Foundation (WPF) application is a common rehosting scenario for Workflow Designer.

## In This Section

[Task 1: Create a New Windows Presentation Foundation Application](#)

[Task 2: Host the Workflow Designer](#)

[Task 3: Create the Toolbox and PropertyGrid Panes](#)

[Support for New Workflow Foundation 4.5 Features in the Rehosted Workflow Designer](#)

## See also

- [Customizing the Workflow Design Experience](#)

# Task 1: Create a New Windows Presentation Foundation Application

3/9/2019 • 2 minutes to read • [Edit Online](#)

In this task, you will create an empty Windows Presentation Foundation (WPF) application by using the WPF Application Visual Studio template and add references to the appropriate .NET Framework 4.6.1 workflow assemblies.

## To create the WPF Application project

1. Open Visual Studio and on the **File** menu, point to **New**, and then click **Project**.
2. In the **New Project** dialog box, select either **Visual C#** or **Visual Basic** from the **Installed Templates** pane on the left side of the box. If the language of your choice does not appear, look under **Other Languages**.
3. Select **Windows** in the **Installed Templates** pane.
4. In the top pane, confirm that (the default value) **.NET Framework 4** has been selected in the drop-down list box, and then select **WPF Application**.
5. Set the name of the project to **HostingApplication** at the bottom of the window.
6. Set the solution name to **RehostingTheDesigner**.
7. Click **OK** to create the application project. Visual Studio creates a basic WPF UI for your application and includes the appropriate XAML and code-behind files.
8. Add references to **WorkflowModel** assemblies. To do this, in **Solution Explorer**, right-click the **HostingApplication** project and select **Add Reference**.
9. In the **Add Reference** dialog box, click the **.NET** tab, hold down the CTRL key, select the following assemblies, and then click **OK**:
  - System.Activities
  - System.Activities.Presentation
  - System.Activities.Core.Presentation
10. Click **OK**.
11. See [Task 2: Host the Workflow Designer](#) to learn how to host the workflow designer design canvas.

## See also

- [Rehosting the Workflow Designer](#)
- [Task 2: Host the Workflow Designer](#)

# Task 2: Host the Workflow Designer

3/9/2019 • 3 minutes to read • [Edit Online](#)

This topic describes the procedure for hosting an instance of the Windows Workflow Designer in a Windows Presentation Foundation (WPF) application.

The procedure configures the **Grid** control that contains the designer, programmatically creates an instance of the **WorkflowDesigner** that contains a default **Sequence** activity, registers the designer metadata to provide designer support for all built-in activities, and hosts the Workflow Designer in the WPF application.

## To host the workflow designer

1. Open the HostingApplication project you created in [Task 1: Create a New Windows Presentation Foundation Application](#).
2. Adjust the size of the window to make it easier to use the Workflow Designer. To do this, select **MainWindow** in the designer, press F4 to display the **Properties** window, and, in the **Layout** section there, set the **Width** to a value of 600 and the **Height** to a value of 350.
3. Set the grid name by selecting the **Grid** panel in the designer (click the box inside the **MainWindow**) and setting the **Name** property at the top of the **Properties** window to "grid1".
4. In the **Properties** window, click the ellipsis (...) next to the **ColumnDefinitions** property to open the **Collection Editor** dialog box.
5. In the **Collection Editor** dialog box, click the **Add** button three times to insert three columns into the layout. The first column will contain the **Toolbox**, the second column will host the Workflow Designer, and the third column will be used for the property inspector.
6. Set the **Width** property of the middle column to the value "4\*".
7. Click **OK** to save the changes. The following XAML is added to your MainWindow.xaml file:

```
<Grid Name="grid1">
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition Width="4*" />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
</Grid>
```

8. In **Solution Explorer**, right-click **MainWindow.xaml** and select **View Code**. Modify the code by following these steps:

- a. Add the following namespaces:

```
using System.Activities;
using System.Activities.Core.Presentation;
using System.Activities.Presentation;
using System.Activities.Presentation.Metadata;
using System.Activities.Presentation.Toolbox;
using System.Activities.Statements;
using System.ComponentModel;
```

- b. To declare a private member field to hold an instance of the **WorkflowDesigner**, add the following

code to the `MainWindow` class.

```
public partial class MainWindow : Window
{
    private WorkflowDesigner wd;

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

- c. Add the following `AddDesigner` method to the `MainWindow` class. The implementation creates an instance of the `WorkflowDesigner`, adds a `Sequence` activity to it, and places it in middle column of the `grid1` **Grid**.

```
private void AddDesigner()
{
    //Create an instance of WorkflowDesigner class.
    this.wd = new WorkflowDesigner();

    //Place the designer canvas in the middle column of the grid.
    Grid.SetColumn(this.wd.View, 1);

    //Load a new Sequence as default.
    this.wd.Load(new Sequence());

    //Add the designer canvas to the grid.
    grid1.Children.Add(this.wd.View);
}
```

- d. Register the designer metadata to add designer support for all the built-in activities. This enables you to drop activities from the toolbox onto the original `Sequence` activity in the Workflow Designer. To do this, add the `RegisterMetadata` method to the `MainWindow` class.

```
private void RegisterMetadata()
{
    DesignerMetadata dm = new DesignerMetadata();
    dm.Register();
}
```

For more information about registering activity designers, see [How to: Create a Custom Activity Designer](#).

- e. In the `MainWindow` class constructor, add calls to the methods declared previously to register the metadata for designer support and to create the `WorkflowDesigner`.

```
public MainWindow()
{
    InitializeComponent();

    // Register the metadata
    RegisterMetadata();

    // Add the WFF Designer
    AddDesigner();
}
```

**NOTE**

The `RegisterMetadata` method registers the designer metadata of built-in activities including the `Sequence` activity. Because the `AddDesigner` method uses the `Sequence` activity, the `RegisterMetadata` method must be called first.

9. Press F5 to build and run the solution.
10. See [Task 3: Create the Toolbox and PropertyGrid Panes](#) to learn how to add **Toolbox** and **PropertyGrid** support to your rehosted workflow designer.

## See also

- [Rehosting the Workflow Designer](#)
- [Task 1: Create a New Windows Presentation Foundation Application](#)
- [Task 3: Create the Toolbox and PropertyGrid Panes](#)

# Task 3: Create the Toolbox and PropertyGrid Panes

3/9/2019 • 3 minutes to read • [Edit Online](#)

In this task, you will create the **Toolbox** and **PropertyGrid** panes and add them to the rehosted Windows Workflow Designer.

For reference, the code that should be in the MainWindow.xaml.cs file after completing the three tasks in the [Rehosting the Workflow Designer](#) series of topics is provided at the end of this topic.

## To create the Toolbox and add it to the grid

1. Open the HostingApplication project you obtained by following the procedure described in [Task 2: Host the Workflow Designer](#).
2. In the **Solution Explorer** pane, right-click the MainWindow.xaml file and select **View Code**.
3. Add a `GetToolboxControl` method to the `MainWindow` class that creates a `ToolboxControl`, adds a new **Toolbox** category to the **Toolbox**, and assigns the `Assign` and `Sequence` activity types to that category.

```
private ToolboxControl GetToolboxControl()
{
    // Create the ToolBoxControl.
    ToolboxControl ctrl = new ToolboxControl();

    // Create a category.
    ToolboxCategory category = new ToolboxCategory("category1");

    // Create Toolbox items.
    ToolboxItemWrapper tool1 =
        new ToolboxItemWrapper("System.Activities.Statements.Assign",
            typeof(Assign).Assembly.FullName, null, "Assign");

    ToolboxItemWrapper tool2 = new ToolboxItemWrapper("System.Activities.Statements.Sequence",
        typeof(Sequence).Assembly.FullName, null, "Sequence");

    // Add the Toolbox items to the category.
    category.Add(tool1);
    category.Add(tool2);

    // Add the category to the ToolBox control.
    ctrl.Categories.Add(category);
    return ctrl;
}
```

4. Add a private `AddToolbox` method to the `MainWindow` class that places the **Toolbox** in the left column on the grid.

```
private void AddToolbox()
{
    ToolboxControl tc = GetToolboxControl();
    Grid.SetColumn(tc, 0);
    grid1.Children.Add(tc);
}
```

5. Add a call to the `AddToolbox` method in the `MainWindow()` class constructor as shown in the following code.

```

public MainWindow()
{
    InitializeComponent();
    this.RegisterMetadata();
    this.AddDesigner();

    this.AddToolBox();
}

```

6. Press F5 to build and run your solution. The **Toolbox** containing the [Assign](#) and [Sequence](#) activities should be displayed.

### To create the **PropertyGrid**

1. In the **Solution Explorer** pane, right-click the MainWindow.xaml file and select **View Code**.
2. Add the `AddPropertyInspector` method to the `MainWindow` class to place the **PropertyGrid** pane in the rightmost column on the grid.

```

private void AddPropertyInspector()
{
    Grid.SetColumn(wd.PropertyInspectorView, 2);
    grid1.Children.Add(wd.PropertyInspectorView);
}

```

3. Add a call to the `AddPropertyInspector` method in the `MainWindow()` class constructor as shown in the following code.

```

public MainWindow()
{
    InitializeComponent();
    this.RegisterMetadata();
    this.AddDesigner();
    this.AddToolBox();

    this.AddPropertyInspector();
}

```

4. Press F5 to build and run the solution. The **Toolbox**, workflow design canvas, and **PropertyGrid** panes should all be displayed, and when you drag an [Assign](#) activity or a [Sequence](#) activity onto the design canvas, the property grid should update depending on the highlighted activity.

## Example

The MainWindow.xaml.cs file should now contain the following code.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
//dlls added

```

```

using System.Activities;
using System.Activities.Core.Presentation;
using System.Activities.Presentation;
using System.Activities.Presentation.Metadata;
using System.Activities.Presentation.Toolbox;
using System.Activities.Statements;
using System.ComponentModel;

namespace HostingApplication
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private WorkflowDesigner wd;

        public MainWindow()
        {
            InitializeComponent();
            RegisterMetadata();
            AddDesigner();
            this.AddToolBox();
            this.AddPropertyInspector();
        }

        private void AddDesigner()
        {
            //Create an instance of WorkflowDesigner class.
            this.wd = new WorkflowDesigner();

            //Place the designer canvas in the middle column of the grid.
            Grid.SetColumn(this.wd.View, 1);

            //Load a new Sequence as default.
            this.wd.Load(new Sequence());

            //Add the designer canvas to the grid.
            grid1.Children.Add(this.wd.View);
        }

        private void RegisterMetadata()
        {
            DesignerMetadata dm = new DesignerMetadata();
            dm.Register();
        }

        private ToolboxControl GetToolboxControl()
        {
            // Create the ToolBoxControl.
            ToolboxControl ctrl = new ToolboxControl();

            // Create a category.
            ToolboxCategory category = new ToolboxCategory("category1");

            // Create Toolbox items.
            ToolboxItemWrapper tool1 =
                new ToolboxItemWrapper("System.Activities.Statements.Assign",
                typeof(Assign).Assembly.FullName, null, "Assign");

            ToolboxItemWrapper tool2 = new ToolboxItemWrapper("System.Activities.Statements.Sequence",
                typeof(Sequence).Assembly.FullName, null, "Sequence");

            // Add the Toolbox items to the category.
            category.Add(tool1);
            category.Add(tool2);

            // Add the category to the ToolBox control.
            ctrl.Categories.Add(category);
        }
    }
}

```

```
        return ctrl;
    }

    private void AddToolBox()
    {
        ToolboxControl tc = GetToolboxControl();
        Grid.SetColumn(tc, 0);
        grid1.Children.Add(tc);
    }

    private void AddPropertyInspector()
    {
        Grid.SetColumn(wd.PropertyInspectorView, 2);
        grid1.Children.Add(wd.PropertyInspectorView);
    }

}
```

## See also

- [Rehosting the Workflow Designer](#)
- [Task 1: Create a New Windows Presentation Foundation Application](#)
- [Task 2: Host the Workflow Designer](#)

# How to: Display Validation Errors in a Rehosted Designer

5/4/2018 • 2 minutes to read • [Edit Online](#)

This topic describes how to retrieve and publish validation errors in a rehosted Windows Workflow Designer. This provides us with a procedure to confirm that a workflow in a rehosted designer is valid.

This task has two parts. The first is to provide an implementation [IValidationErrorService](#). There is one critical method to implement on this interface, [ShowValidationErrors](#) which will pass you a list of [ValidationErrorInfo](#) objects containing information about the errors to the debug log. After implementing the interface, you retrieve the error information by publishing an instance of that implementation to the editing context.

## Implement the IValidationErrorService Interface

1. Here is a code sample for a simple implementation that will write out the validation errors to the debug log.

```
using System.Activities.Presentation.Validation;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;

namespace VariableFinderShell
{
    class DebugValidationErrorService : IValidationErrorService
    {
        public void ShowValidationErrors(IList<ValidationErrorInfo> errors)
        {
            errors.ToList().ForEach(vei => Debug.WriteLine(string.Format("Error: {0} ", vei.Message)));
        }
    }
}
```

## Publishing to the Editing Context

1. Here is the code that will publish this to the editing context.

```
wd.Context.Services.Publish<IValidationErrorService>(new DebugValidationErrorService());
```

# Support for New Workflow Foundation 4.5 Features in the Rehosted Workflow Designer

3/9/2019 • 8 minutes to read • [Edit Online](#)

Windows Workflow Foundation (WF) in .NET Framework 4.5 introduced many new features, including several enhancements to the workflow designer experience. This topic details which of these features are supported in the rehosted designer, and which ones are currently not supported.

## NOTE

For a list of all of the new Windows Workflow Foundation (WF) features introduced in .NET Framework 4.5, including those that are unrelated to designer rehosting, see [What's New in Windows Workflow Foundation in .NET 4.5](#).

## Activities

The built-in activity library contains new activities and new features for existing activities. All of these new activities are supported in the rehosted designer. For more information on these new activities, see the [Activities](#) section of [What's New in Windows Workflow Foundation in .NET 4.5](#).

## C# Expressions

Prior to .NET Framework 4.5, all expressions in workflows could only be written in Visual Basic. In .NET Framework 4.5, Visual Basic expressions are only used for projects created using Visual Basic. Visual C# projects now use C# for expressions. When authoring workflows in Visual Studio 2012, a fully functional C# expression editor is provided which capabilities such as grammar highlighting and intellisense. C# workflow projects created in previous versions that use Visual Basic expressions will continue to work.

## WARNING

C# expressions are not supported in the rehosted designer.

## New Designer Capabilities

### Designer Search

The [Quick Find](#) and [Find in Files](#) features introduced with .NET Framework 4.5 are not supported in the rehosted designer. The [Toolbox](#) search is supported in the rehosted designer. For more information on these features, see [Designer Search](#).

## WARNING

Quick Find and Find in Files are not supported in the rehosted designer.

### Delete context menu item in variable and argument designer

In .NET Framework 4, variables and arguments could only be deleted in the designer using the keyboard. Starting with .NET Framework 4.5, variables and arguments can be deleted using the context menu. This feature is supported in the rehosted designer.

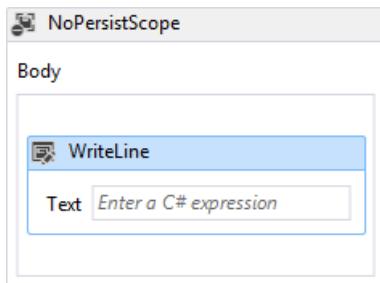
The following screenshot shows the variable and argument designer context menu.

| Name                   | Variable type | Scope    | Default               |
|------------------------|---------------|----------|-----------------------|
| Guess                  | Int32         | Sequence | Enter a C# expression |
| Target                 |               | Sequence | Enter a C# expression |
| <b>Create Variable</b> |               |          |                       |

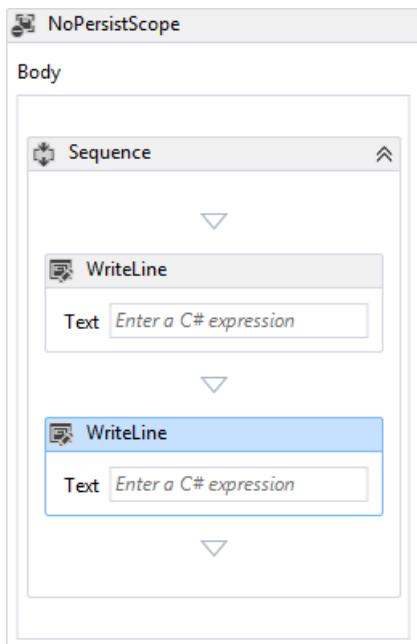
### Auto-surround with Sequence

Since a workflow or certain container activities (such as `NoPersistScope`) can only contain a single body activity, adding a second activity required the developer to delete the first activity, add a `Sequence` activity, and then add both activities to the sequence activity. Starting with .NET Framework 4.5, when adding a second activity to the designer surface, a `Sequence` activity will be automatically created to wrap both activities. This feature is supported in the rehosted designer.

The following screenshot shows a `WriteLine` activity in the `Body` of a `NoPersistScope`.



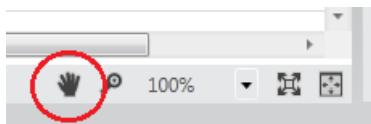
The following screenshot shows the automatically created `Sequence` activity in the `Body` when a second `WriteLine` is dropped below the first.



### Pan Mode

To more easily navigate a large workflow in the designer, pan mode can be enabled, allowing the developer to click and drag to move the visible portion of the workflow, rather than needing to use the scroll bars. The button to activate pan mode is in the lower right corner of the designer. This feature is supported in the rehosted designer.

The following screenshot shows the pan button which is located at the bottom right corner of the workflow designer.



The middle mouse button or space bar can also be used to pan the workflow designer.

### Multi-select

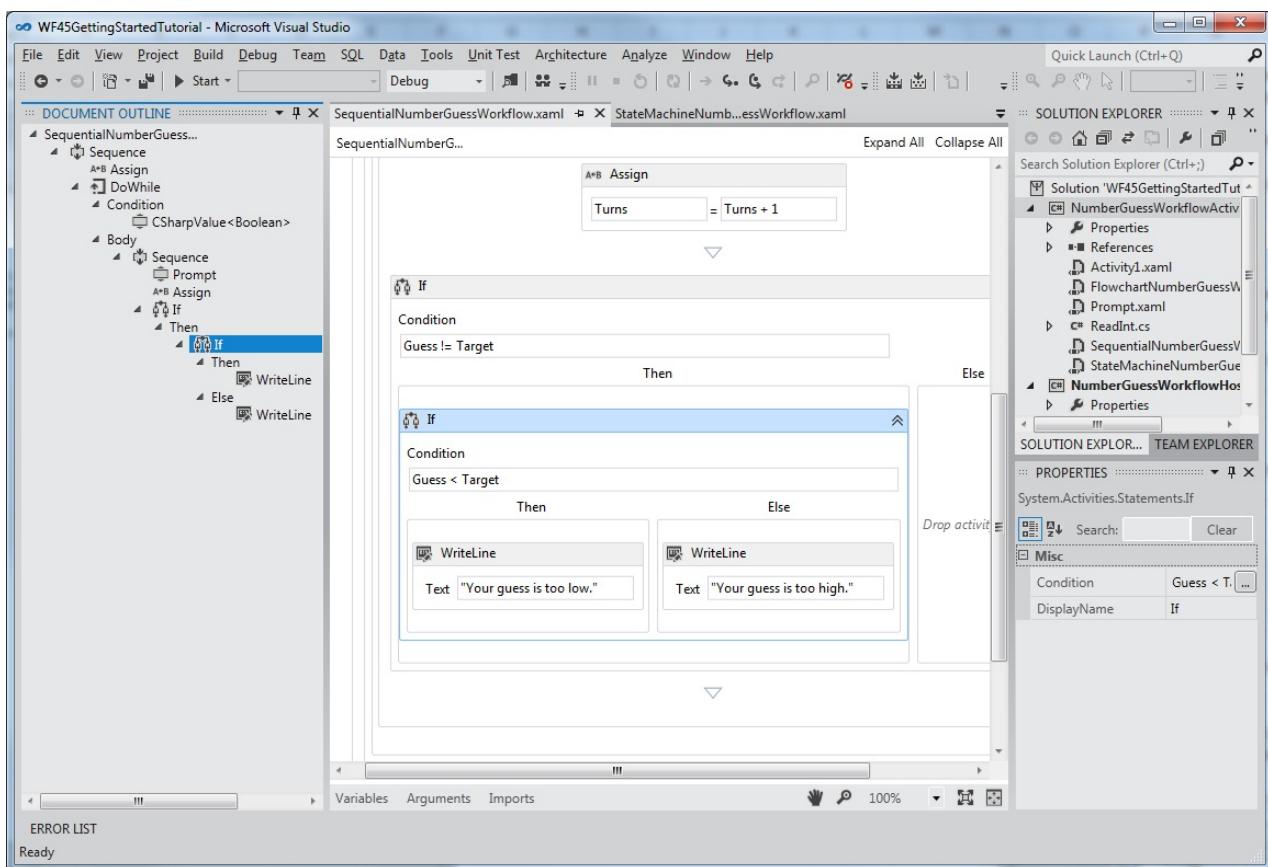
Multiple activities can be selected at one time, either by dragging a rectangle around them (when pan mode is not enabled), or by holding down Ctrl and click on the desired activities one by one. This feature is supported in the rehosted designer.

Multiple activity selections can also be dragged and dropped within the designer, and can also be interacted with using the context menu.

### Outline view of workflow items

In order to make hierarchical workflows easier to navigate, components of a workflow are shown in a tree-style outline view. The outline view is displayed in the **Document Outline** view. To open this view in Visual Studio, from the top menu, select **View, Other Windows, Document Outline**, or press Ctrl W,U. Clicking on a node in outline view will navigate to the corresponding activity in the workflow designer, and the outline view will be updated to show activities that are selected in the designer. This feature is supported in the rehosted designer.

The following screenshot of the completed workflow from the [Getting Started Tutorial](#) shows the outline view with a sequential workflow.



### More control of visibility of shell bar and header items

In a rehosted designer, some of the standard UI controls may not have meaning for a given workflow, and may be turned off. In .NET Framework 4, this customization is only supported by the shell bar at the bottom of the designer. In .NET Framework 4.5, the visibility of shell header items at the top of the designer can be adjusted by setting `WorkflowShellHeaderItemsVisibility` with the appropriate `ShellHeaderItemsVisibility` value.

### Auto-connect and auto-insert in Flowchart and State Machine workflows

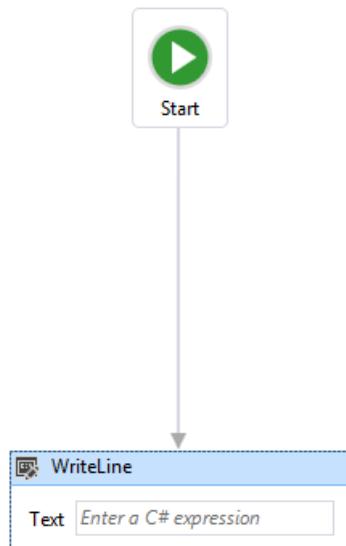
In .NET Framework 4, connections between nodes in a Flowchart workflow had to be added manually. In .NET

Framework 4.5, Flowchart and State Machine nodes have auto-connect points that become visible when an activity is dragged from the toolbox onto the designer surface. Dropping an activity on one of these points automatically adds the activity along with the necessary connection.

The following screenshot shows the attachment points that become visible when an activity is dragged from the toolbox.



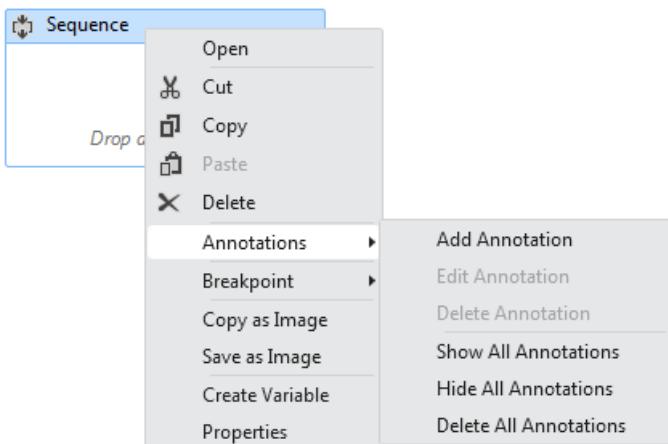
Activities can also be dragged onto connections between flowchart nodes and states to auto-insert the node between two other nodes. The following screenshot shows the highlighted connecting line where activities can be dragged from the toolbox and dropped.



Auto-connect and auto-insert are supported in the rehosted designer.

### Designer Annotations

To facilitate developing larger workflows, the designer now supports adding annotations to help keep track of the design process. Annotation can be added to activities, states, flowchart nodes, variables and arguments. The following screenshot shows the context menu used to add annotations to the designer.



Designer annotations are supported in the rehosted designer.

### Define and consume ActivityDelegate objects in the designer

Activities in .NET Framework 4 used [ActivityDelegate](#) objects to expose execution points where other parts of the workflow could interact with a workflow's execution, but using these execution points usually required a fair amount of code. In this release, developers can define and consume activity delegates using the workflow designer. For more information, see [How to: Define and consume activity delegates in the Workflow Designer](#).

Activity delegates are supported in the rehosted designer.

### Build-time validation

In .NET Framework 4, workflow validation errors weren't counted as build errors during the build of a workflow project. This meant that building a workflow project could succeed even when there were workflow validation errors. In .NET Framework 4.5, workflow validation errors cause the build to fail.

#### WARNING

Build-time validation is not supported in the rehosted designer.

### Design-time background validation

In .NET Framework 4, workflows were validated as a foreground process, which could potentially hang the UI during complex or time-consuming validation processes. Workflow validation now takes place on a background thread, so that the UI is not blocked.

Design-time background validation is supported in the rehosted designer.

### View state located in a separate location in XAML files

In .NET Framework 4, the view state information for a workflow is stored across the XAML file in many different locations. This is inconvenient for developers who want to read XAML directly, or write code to remove the view state information. In .NET Framework 4.5, the view state information in the XAML file is serialized as a separate element in the XAML file. Developers can easily locate and edit the view state information of an activity, or remove the view state altogether.

This feature is supported in the rehosted workflow designer.

### Opt-in for Workflow 4.5 features in rehosted designer

To preserve backward compatibility, some new features included in .NET Framework 4.5 are not enabled by default in the rehosted designer. This is to ensure that existing applications that use the rehosted designer are not broken by updating to the latest version. To enable new features in the rehosted designer, either set [TargetFrameworkName](#) to ".Net Framework 4.5", or set individual members of [DesignerConfigurationService](#) to enable individual features.

## New Workflow Development Models

In addition to flowchart and sequential workflow development models, this release includes State Machine workflows, and contract-first workflow services.

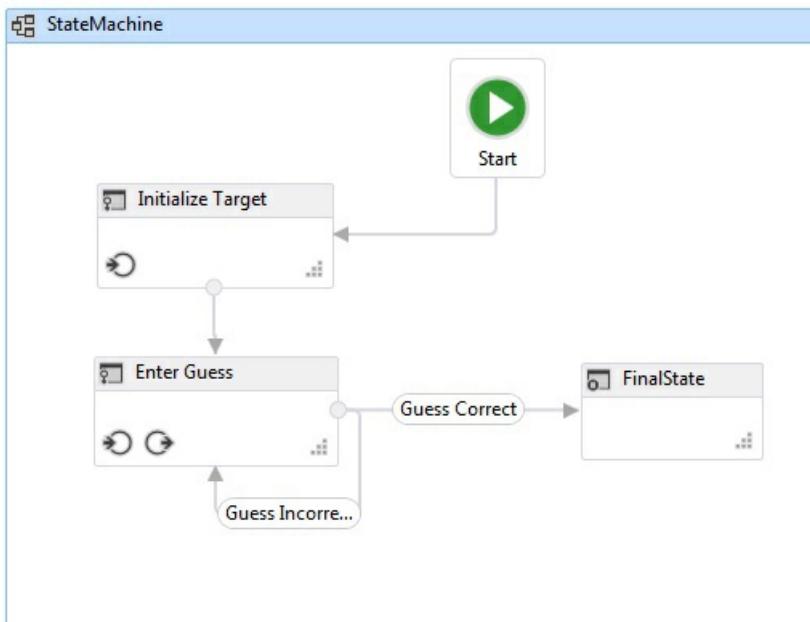
### State machine workflows

State machine workflows were introduced as part of the .NET Framework 4.0.1 in the [Microsoft .NET Framework 4 Platform Update 1](#). This update included several new classes and activities which allowed developers to create state machine workflows. These classes and activities have been updated for .NET Framework 4.5. Updates include:

1. The ability to set breakpoints on states
2. The ability to copy and paste transitions in the workflow designer
3. Designer support for shared trigger transition creation

#### 4. Activities used to create State Machine workflows, including: [StateMachine](#), [State](#), and [Transition](#)

The following screenshot shows the completed state machine workflow from the [Getting Started Tutorial](#) step [How to: Create a State Machine Workflow](#).



For more information on creating state machine workflows, see [State Machine Workflows](#). State machine workflows are supported in the rehosted designer.

#### Contract-first workflow development

The contract-first workflow development tool allows the developer to design a contract in code first, then, with a few clicks in Visual Studio, automatically generate an activity template in the toolbox representing each operation. These activities are then used to create a workflow that implements the operations defined by the contract. The workflow designer will validate the workflow service to ensure that these operations are implemented and the signature of the workflow matches the contract signature. The developer can also associate a workflow service with a collection of implemented contracts. For more information on contract-first workflow service development, see [How to: Create a workflow service that consumes an existing service contract](#).

#### WARNING

Contract-first workflow development is not supported in the workflow designer.

# Using a Custom Expression Editor

3/9/2019 • 4 minutes to read • [Edit Online](#)

A custom expression editor can be implemented to provide a richer or simpler expression editing experience. There are several scenarios in which you might want to use a custom expression editor:

- To provide support for IntelliSense and other rich editing features in a rehosted workflow designer. This functionality must be provided because the default Visual Studio expression editor cannot be used in rehosted applications.
- To simplify the expression editing experience for the business analyst users, so that they are not, for example, required to learn Visual Basic or deal with Visual Basic expressions.

Three basic steps are needed to implement a custom expression editor:

1. Implement the [IExpressionEditorService](#) interface. This interface manages the creation and destruction of expression editors.
2. Implement the [IExpressionEditorInstance](#) interface. This interface implements the UI for expression editing UI.
3. Publish the [IExpressionEditorService](#) in your rehosted workflow application.

## Implementing a Custom Expression Editor in a Class Library

Here is a sample of code for a (proof of concept) `MyEditorService` class that implements the [IExpressionEditorService](#) interface is contained in a MyExpressionEditorService library project.

```

using System;
using System.Collections.Generic;
using System.Activities.Presentation.View;
using System.Activities.Presentation.Hosting;
using System.Activities.Presentation.Model;

namespace MyExpressionEditorService
{
    public class MyEditorService : IExpressionEditorService
    {
        public void CloseExpressionEditors()
        {

        }

        public IExpressionEditorInstance CreateExpressionEditor(AssemblyContextControlItem assemblies,
ImportedNamespaceControlItem importedNamespaces, List<ModelItem> variables, string text)
        {
            MyExpressionEditorInstance instance = new MyExpressionEditorInstance();
            return instance;
        }

        public IExpressionEditorInstance CreateExpressionEditor(AssemblyContextControlItem assemblies,
ImportedNamespaceControlItem importedNamespaces, List<ModelItem> variables, string text, System.Windows.Size
initialSize)
        {
            MyExpressionEditorInstance instance = new MyExpressionEditorInstance();
            return instance;
        }

        public IExpressionEditorInstance CreateExpressionEditor(AssemblyContextControlItem assemblies,
ImportedNamespaceControlItem importedNamespaces, List<ModelItem> variables, string text, Type expressionType)
        {
            MyExpressionEditorInstance instance = new MyExpressionEditorInstance();
            return instance;
        }

        public IExpressionEditorInstance CreateExpressionEditor(AssemblyContextControlItem assemblies,
ImportedNamespaceControlItem importedNamespaces, List<ModelItem> variables, string text, Type expressionType,
System.Windows.Size initialSize)
        {
            MyExpressionEditorInstance instance = new MyExpressionEditorInstance();
            return instance;
        }

        public void UpdateContext(AssemblyContextControlItem assemblies, ImportedNamespaceControlItem
importedNamespaces)
        {

        }
    }
}

```

Here is the code for a `MyExpressionEditorInstance` class that implements the `IExpressionEditorInstance` interface in a MyExpressionEditorService library project.

```

using System;
using System.Activities.Presentation.View;
using System.Windows;
using System.Reflection;
using System.Windows.Controls;

namespace MyExpressionEditorService
{
    public class MyExpressionEditorInstance : IExpressionEditorInstance
    {
        private TextBox textBox = new TextBox();

        public bool AcceptsReturn { get; set; }
        public bool AcceptsTab { get; set; }
    }
}

```

```
public void AcceptsTab { get; set; }
public bool HasAggregateFocus {
    get
    {
        return true;
    }
}

public System.Windows.Controls.ScrollBarVisibility HorizontalScrollBarVisibility { get; set; }
public System.Windows.Controls.Control HostControl {
    get
    {
        return textBox;
    }
}
public int MaxLines { get; set; }
public int MinLines { get; set; }
public string Text { get; set; }
public System.Windows.Controls.ScrollBarVisibility VerticalScrollBarVisibility { get; set; }

public event EventHandler Closing;
public event EventHandler GotAggregateFocus;
public event EventHandler LostAggregateFocus;
public event EventHandler TextChanged;

public bool CanCompleteWord()
{
    return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
}
public bool CanCopy()
{
    return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
}
public bool CanCut()
{
    return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
}
public bool CanDecreaseFilterLevel()
{
    return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
}
public bool CanGlobalIntellisense()
{
    return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
}
public bool CanIncreaseFilterLevel()
{
    return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
}
public bool CanParameterInfo()
{
    return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
}
public bool CanPaste()
{
    return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
}
public bool CanQuickInfo()
{
    return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
}
```

```
        }
        public bool CanRedo()
        {
            return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
        }
        public bool CanUndo()
        {
            return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
        }

        public void ClearSelection()
        {
            MessageBox.Show(MethodBase.GetCurrentMethod().Name);
        }
        public void Close()
        {
            MessageBox.Show(MethodBase.GetCurrentMethod().Name);
        }
        public bool CompleteWord()
        {
            return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
        }
        public bool Copy()
        {
            return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
        }
        public bool Cut()
        {
            return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
        }
        public bool DecreaseFilterLevel()
        {
            return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
        }
        public void Focus()
        {
            MessageBox.Show(MethodBase.GetCurrentMethod().Name);
        }
        public string GetCommittedText()
        {
            return "CommittedText";
        }
        public bool GlobalIntellisense()
        {
            return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
        }
        public bool IncreaseFilterLevel()
        {
            return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
        }
        public bool ParameterInfo()
        {
            return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
        }
        public bool Paste()
        {
            return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
        }
        public bool QuickInfo()
        {
```

```
        {
            return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
        }
        public bool Redo()
        {
            return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
        }
        public bool Undo()
        {
            return (MessageBox.Show(MethodBase.GetCurrentMethod().Name, "TestEditorInstance",
MessageBoxButton.YesNo) == MessageBoxResult.Yes);
        }
    }
}
```

## Publishing a Custom Expression Editor in a WPF Project

Here is the code that shows how to rehost the designer in a WPF application and how to create and publish the `MyEditorService` service. Before using this code, add a reference to the `MyExpressionEditorService` library project from the project that contains the `avalon2` application.

```

using System.Windows;
using System.Windows.Controls;
using System.Activities.Presentation;
using System.Activities.Statements;
using System.Activities.Core.Presentation;
using System.Activities.Presentation.View;
using MyExpressionEditorService;

namespace WpfApplication1
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {

        private MyEditorService expressionEditorService;
        public MainWindow()
        {
            InitializeComponent();
            new DesignerMetadata().Register();
            createDesigner();
        }

        public void createDesigner()
        {
            WorkflowDesigner designer = new WorkflowDesigner();
            Sequence root = new Sequence()
            {
                Activities =
                {
                    new Assign(),
                    new WriteLine()
                };
            }

            designer.Load(root);

            Grid.SetColumn(designer.View, 0);

            // Create ExpressionEditorService
            this.expressionEditorService = new MyEditorService();

            // Publish the instance of MyEditorService.
            designer.Context.Services.Publish<IExpressionEditorService>(this.expressionEditorService);

            MyGrid.Children.Add(designer.View);
        }
    }
}

```

## Notes

If you are using an **ExpressionTextBox** control in a custom activity designer, it is not necessary to create and destroy expression editors using the [CreateExpressionEditor](#) and [CloseExpressionEditors](#) methods of the [IExpressionEditorService](#) interface. The [ExpressionTextBox](#) class manages this for you.

## See also

- [IExpressionEditorService](#)
- [IExpressionEditorInstance](#)
- [Using the ExpressionTextBox in a Custom Activity Designer](#)

# Windows Workflow Foundation Glossary for .NET Framework 4.5

3/6/2019 • 2 minutes to read • [Edit Online](#)

The following terms are used in the Windows Workflow Foundation documentation.

## Terms

| TERM                 | DEFINITION  |
|----------------------|---|
| activity             | A unit of program behavior in Windows Workflow Foundation. Single activities can be composed together into more complex activities.   |
| activity action      | A data structure used to expose callbacks for workflow and activity execution.  |
| argument             | Defines the data flow into and out of an activity. Each argument has a specified direction: in, out, or in/out. These represent the input, output, and input/output parameters of the activity.                         |
| bookmark             | The point at which an activity can pause and wait to be resumed.  |
| compensation         | A group of actions designed to undo or mitigate the effect of previously completed work.  |
| correlation          | The mechanism for routing messages to a workflow or service instance.   |
| expression           | A construct that takes in one or more arguments, performs an operation on the arguments and returns a single value. Expressions can be used anywhere an activity can be used.   |
| flowchart            | A well-known modeling paradigm that represents program components as symbols linked together with directional arrows. In the .NET Framework 4, workflows can be modeled as flowcharts using the Flowchart activity.     |
| long-running process | A unit of program execution that does not return immediately and may span system restarts.  |
| persistence          | Saving the state of a workflow or service to a durable medium, so that it can be unloaded from memory or recovered after a system failure.  |
| state machine        | A well-known modeling paradigm that represents program components as individual states linked together with event-driven state transitions. Workflows can be modeled as state machines using the StateMachine activity. |

| TERM           | DEFINITION   |
|----------------|--|
| substance      | Represents a group of related bookmarks under a common identifier and allows the runtime to make decisions about whether a particular bookmark resumption is valid or may become valid.  |
| type converter | A CLR type can be associated with one or more System.ComponentModel.TypeConverter derived types that enable converting instances of the CLR type to and from instances of other types. A type converter is associated with a CLR type using the System.ComponentModel.TypeConverterAttribute attribute. A TypeConverterAttribute can be specified directly on the CLR type or on a property. A type converter specified on a property always takes precedence over a type converter specified on the CLR type of the property. |
| variable       | Represents the storage of some data that must be saved and accessed later.   |
| workflow       | A single activity or tree of activities invoked by a host process.   |
| XAML           | eXtensible Application Markup Language   |

# Windows Workflow (WF) Samples

3/9/2019 • 2 minutes to read • [Edit Online](#)

You can [download Windows Workflow samples](#) that provide instruction on various aspects of Windows Workflow Foundation (WF).

The articles in this section describe some of the samples in the download package. For a complete documentation set that covers all of the samples, check the [.NET Framework 4 WF samples](#) section.

## NOTE

The downloadable samples were created with Visual Studio 2010 and .NET Framework 4, but are compatible with later versions of Visual Studio and the .NET Framework. Additional samples for Windows Workflow Foundation in .NET Framework 4.6.1 can be found on [MSDN code samples](#).

## In this section

[Application](#) - Provides samples that are related to workflow applications.

[Basic](#) - Provides samples that demonstrate basic Windows Workflow Foundation (WF) functionality.

[Scenario](#) - Provides examples of Windows Workflow Foundation (WF) scenarios.