|  |  |
| --- | --- |
| **mslogo-1.jpg** | **PluralsightColorLogoLgTrans.png** |

A Developer’s Guide to the Microsoft® .NET Workflow Service

Coordinating services in the cloud

Matt Milner, Pluralsight

May 2009

Contents

[Abstract 3](#_Toc228962799)

[Introduction 3](#_Toc228962800)

[Windows Workflow Foundation Basics 4](#_Toc228962801)

[Workflow Modeling 7](#_Toc228962802)

[Microsoft .NET Workflow Service 11](#_Toc228962803)

[Microsoft .NET Services SDK 11](#_Toc228962804)

[Microsoft .NET Workflow Service Scenarios 13](#_Toc228962805)

[Building Workflows for the Microsoft .NET Workflow Service 13](#_Toc228962806)

[Building Your First Cloud Workflow 14](#_Toc228962807)

[Cloud Activities 17](#_Toc228962808)

[CloudHttpSend 18](#_Toc228962809)

[CloudHttpReceive 20](#_Toc228962810)

[CloudXPathRead 25](#_Toc228962811)

[CloudXPathUpdate 27](#_Toc228962812)

[CloudServiceBusSend 28](#_Toc228962813)

[CloudServiceBusReceive 30](#_Toc228962814)

[CloudDelay 32](#_Toc228962815)

[Managing Cloud Workflows 32](#_Toc228962816)

[Azure Management Portal 33](#_Toc228962817)

[Custom Workflow Management using WorkflowClient 36](#_Toc228962818)

[Summary 38](#_Toc228962819)

[Additional Resources 39](#_Toc228962820)

[Microsoft® .NET Services Whitepaper Series 39](#_Toc228962821)

[.NET Workflow Service Resources 39](#_Toc228962822)

[WF Resources 39](#_Toc228962823)

[About the Author 40](#_Toc228962824)

[Acknowledgements 40](#_Toc228962825)

# Abstract

This paper provides details about the Microsoft® .NET Workflow Service, its relation to Windows Workflow Foundation, and what developers need to know to begin building workflows to orchestrate message exchanges with the .NET Service Bus. It not only explains the current tools and capabilities but also outlines the vision for future releases.

# Introduction

Microsoft® .NET Services[[1]](#footnote-1) are a set of highly scalable developer-oriented services running in Microsoft data centers as part of the Azure™ Services Platform. Microsoft .NET Services provide developers with common building blocks and infrastructure services for cloud-based and cloud-aware applications. Much like you rely on the .NET Framework for common building blocks when developing on-premise software, you will rely on Microsoft® .NET Services for the common building blocks in your cloud applications.

The Microsoft® .NET Workflow Service is one of the core service offerings found within Microsoft® .NET Services. Today it’s complemented by two other services: the Microsoft® .NET Service Bus and the Microsoft® .Access Control Service. The .NET Service Bus relies on the .NET Access Control Service for controlling access to your solutions through a claims-based security model. The .NET Workflow Service allows you to define cloud-based workflows that model service interactions through the .NET Service Bus and HTTP messaging. Together these services provide a valuable development fabric required by most cloud applications, thereby simplifying development by allowing you to focus more directly on business needs.[[2]](#footnote-2)

This whitepaper focuses on the Microsoft® .NET Workflow Service: a hosted, scalable environment for using workflow to orchestrate messages and services in the cloud. The .NET Workflow Service integrates with the .NET Service Bus and the .NET Access Control service to provide secure coordination of message interactions. In addition to an execution environment, the .NET Workflow Service provides management tools and a Web service API for creating and managing workflow types and instances.

Because the hosting environment is built on the Windows Azure platform, it is capable of scaling on demand and to great degrees without an organization or a developer having to worry about planning for more hardware and software. Because the Windows Workflow Foundation (WF) runtime is being used, workflow instances can run on a pool of servers with the ability to move from one server to another for each episode of execution. The hosting environment includes a persistence service which leverages the secure, replicated, Microsoft SQL Services to save the state of running workflows and to ensure recovery capabilities. In addition, workflows can also take advantage of other services in the Azure Services Platform such as Microsoft SQL Services for accessing or manipulating cloud-hosted application data.

As you move towards cloud computing, workflow provides a simplified approach for coordinating complex service interactions in the composite “cloud” solutions you build. The .NET Workflow Service provides the tool for defining workflows that bring together various message interactions into a logical process. In addition your workflows run in the scalable environment provided by Microsoft data centers around the world. In order to understand and appreciate how to build workflows that can be hosted in the .NET Workflow Service, it’s important to first understand some key details about how WF works.

# Windows Workflow Foundation Basics

Windows Workflow Foundation (WF) provides a runtime, programming model, and tools for adding workflow capabilities to your applications. WF provides key functionality to enable declarative flow control of your business logic, state management of your application processing, all with the thread and process agility required for building scalable systems.

The programming model for WF allows you to author workflows that are essentially a tree of activities. In fact, the workflow itself is just another activity, which happens to be the root activity in the tree that acts as the starting point for all processing. Activities are the building blocks of workflows and they provide both the processing logic and the control flow within the workflow.

You can think of activities as program statements much like you use in code. Some activities provide control flow, like the IfElse, While, and Parallel activities, much like their counterparts in .NET languages today. Other activities do the “work” of the application and are known as leaf activities. The framework provides a few activities of this type, primarily focused on communication such as the Send activity which allows you to call a service using Windows Communication Foundation (WCF).

When you get started with WF, you will likely build your own libraries of activities to help you interact with your business objects, systems and APIs to make modeling business processes in workflows simpler. Figure 1 shows the relationship of the activities to the workflow, or root activity, and illustrates the difference between control flow activities and leaf activities in the tree.

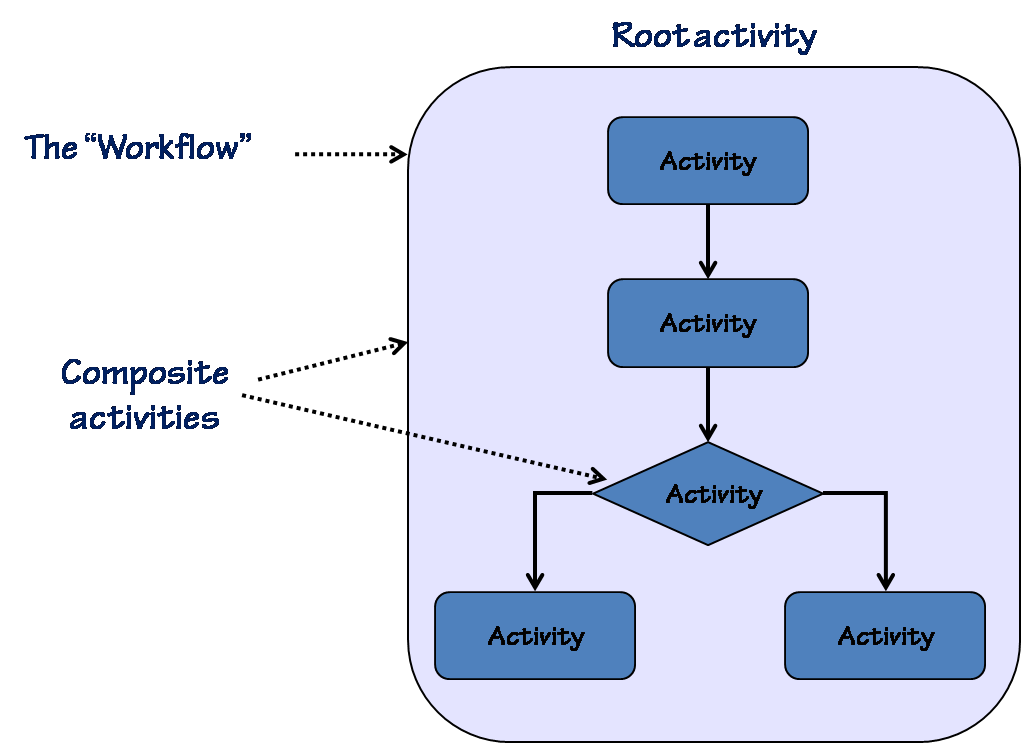


Figure 1: Workflow = Tree of Activities

If activities are program statements, you can think of the workflow itself much like a Main method in a program; it is still a standard program statement, but it’s treated differently by the managing runtime. In WF, the runtime is modeled by the WorkflowRuntime class. The runtime is responsible for creating and managing all workflow instances. Specifically, the runtime handles issues around thread management, scheduling work, persisting application state, and managing “bookmarks” or resumption points.

While the runtime manages all aspects of workflow execution, it depends on a set of runtime services to provide specific implementations. For example, while the runtime is ultimately responsible for scheduling activity execution and making sure workflow logic gets processed, it accomplishes this task by relying on a scheduler service that must be loaded into the runtime. If no scheduler is configured, the runtime uses the DefaultWorkflowScheduler service to schedule the work.

The scheduler service is responsible for taking the work, essentially .NET delegates or callbacks to be executed, and scheduling it on a CLR thread. The DefaultWorkflowScheduler uses a threadpool to manage the work and executes the callbacks asynchronously on a thread from the pool. The ManualWorkflowScheduler, on the other hand, executes the scheduled work when the hosting application requests it and it does so on the requesting thread, providing a synchronous model of execution. Figure 2 shows the runtime hosted in a .NET application and the runtime services added into the runtime to provide specific implementations that make sense for the hosting environment.

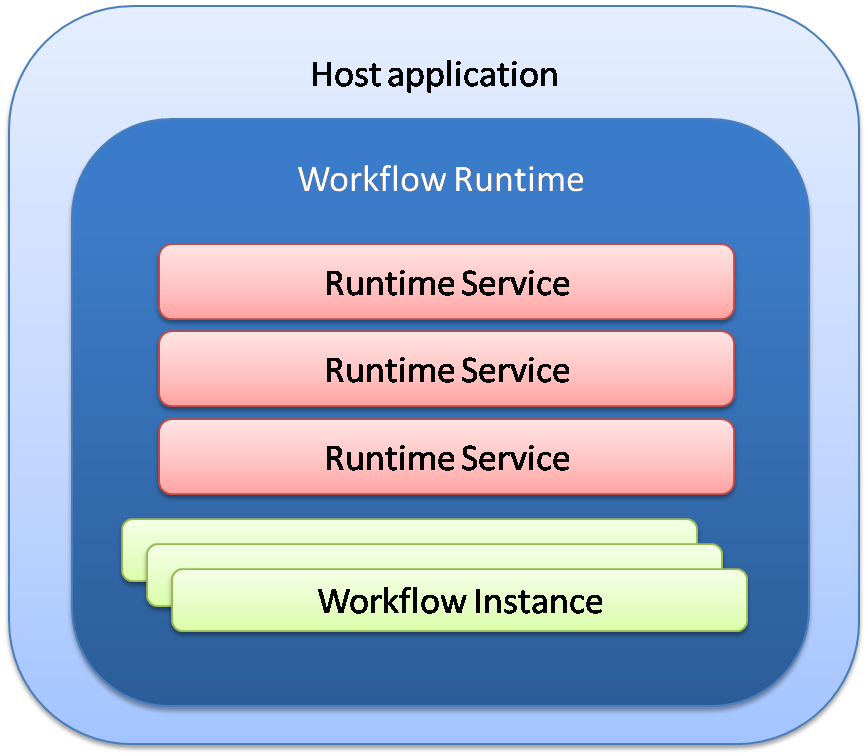


Figure 2: WF Runtime and Runtime services

One of the most important services the runtime depends on is the persistence service. The persistence service allows the runtime to serialize the current state of a workflow to a durable store, such as Microsoft SQL Server. In addition to saving the state of the workflow itself, including property values on activities, persistence also stores the “bookmarks” indicating where the workflow should resume. Workflows typically are persisted at key points such as the end of a transaction, and when the workflow goes idle, when it is waiting for external stimuli or for asynchronous work to complete.

Using the threading and persistence services, the WF runtime is able to provide both thread and process agility, meaning that a given workflow instance can execute on multiple threads over the lifetime of the workflow and the instance can begin in one process and resume in another. Figure 3 depicts process agility in action as a single workflow begins in one process, goes idle and is persisted to a Microsoft SQL Server database. At some later time, due to activity outside the workflow instance, the workflow is resumed in another process, potentially on another computer, with work picking up where it left off.

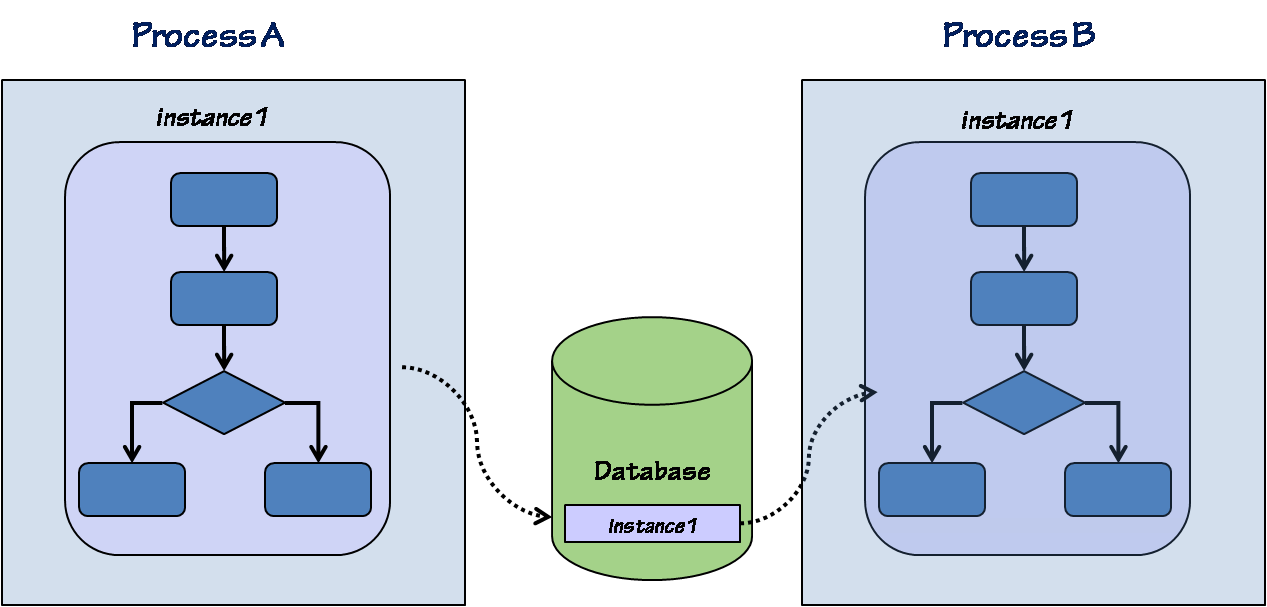


Figure 3: Process Agility via Persistence

Having thread and process agility allows WF applications to scale and more efficiently use processing resources while providing the rich model for declarative programming and state management. The benefit to developers is that much of this complexity is buried in the framework and not surfaced when writing workflows and activities. One example of this is threading. While many threads may process a single workflow instance during its lifetime, WF guarantees that only a single thread is processing a workflow at a given time. This frees activity authors from having to write threading and synchronization code in their activity logic and in many cases frees the workflow author from having to worry about concurrency issues. It should be noted however that it is possible to model a workflow in such a way as to have concurrency issues in the processing logic, but for this WF provides a synchronization scope activity that can be used to ensure locks are appropriately managed at the workflow level.

In addition to the runtime and related services, WF comes with a base activity library which includes many different activities for managing control flow and a few leaf activities focused on communications. Several of the activities provide control flow similar to their equivalents in .NET. Table 1 shows most of the activities included in the base activity library grouped according to their primary function. It is beyond the scope of this paper to provide details on each of these activities. For more detail on the function of specific activities, see the Windows Workflow Foundation documentation on MSDN.

|  |  |
| --- | --- |
| Logical group | Activities |
| Control flow | IfElse, While, Parallel, Sequence, Replicator, Delay, Listen, ConditionedActivityGroup |
| Scoping | TransactionScope, SynchronizationScope, EventHandlingScope, EventDriven, CompensatableSequence, FaultHandler, |
| Communications | InvokeWorkflow, Send, Receive, InvokeWebService, CallExternalMethod, HandleExternalEvent |
| Workflow control | Suspend, Terminate, Throw, Compensate |
| State machine | State, SetState, StateInitialization, StateFinalization |

Table 1: Base Activity Library

## Workflow Modeling

Another area of flexibility in WF is the ability to model workflows in different formats. Because the definition of a workflow is mostly the definition of an object hierarchy, there are many different options for representing the tree of activities and their property settings or configuration.

The two models supported by WF directly include a code model and an XML model. In the code model, you define your workflow with a .NET class and the tree of activities is created in the constructor of the workflow class. This is the model used in Visual Studio when you create a new workflow class using the supplied project or item templates. The XML option allows you to define the hierarchy using Xml Application Markup Language (XAML). When you use XAML to define your workflow, you are not defining a new type, rather you are defining the relationship between existing activity types.

Figure 4 shows an example of a simple workflow definition in code. Most of the time this definition is hidden from developers behind the graphical designer found in Visual Studio. Figure 5 shows the same workflow defined using XAML, which can also be created using the graphical designer in Visual Studio.

public class HelloWorld : SequentialWorkflowActivity

{

public HelloWorld()

{

WriteLineActivity hello = new WriteLineActivity();

hello.OutputText = "Hello ";

WriteLineActivity world = new WriteLineActivity();

world.OutputText = "world";

this.Activities.Add(hello);

this.Activities.Add(hello);

}

}

Figure 4: Workflow Definition in Code

<SequentialWorkflowActivity

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow"

xmlns:sample="clr-namespace:WFSamples;Assembly=WFSamples">

<sample:WriteLineActivity OutputText="Hello" Name="Hello"/>

<sample:WriteLineActivity OutputText="World" Name="world" />

</SequentialWorkflowActivity>

Figure 5: Workflow Definition in XAML

Both ways of modeling your workflow result in the tree of activities being created and initialized at runtime. The WorklowRuntime class relies on a runtime loading service to take the .NET type or XML and return an initialized tree of activities. The end result of each modeling style is essentially the same at runtime once the definition has become a workflow instance. One of the main differences between these two models is that a workflow designed in XAML cannot directly contain any code. That is, there is no code-behind for the workflow, it is entirely declarative. The code in Figure 6 shows how the workflow defined in Figure 5 can be used to create a workflow instance and start it.

using (WorkflowRuntime runtime = new WorkflowRuntime())

{

XmlReader reader =

XmlReader.Create("..\\..\\XAMLWorkflow.xml");

WorkflowInstance xmlWF = runtime.CreateWorkflow(reader);

xmlWF.Start();

}

Figure 6: Executing Declarative XAML Workflow

There are differences and tradeoffs between these models. For more details on how the loader works, and the benefits of each model, see the MSDN Magazine article “Loading Workflow Models in WF”.

In addition to providing a declarative model for defining the business process, WF also provides a rich business rule engine and allows for defining declarative business rules. In a workflow, declarative conditions can be used on activities such as the While activity to define conditional logic. These conditions are expressions that result in a Boolean value. The While activity would evaluate the condition each time it completes an execution of its child activity and use the resulting Boolean value to determine whether to execute the loop again. The condition editor can be seen in showing a simple expression testing the value of a property on the workflow in which the activity is contained.

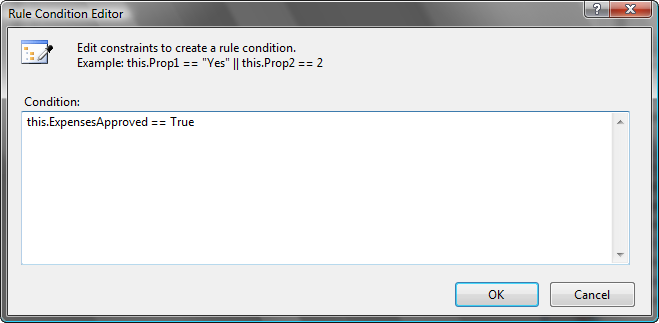


Figure 7: Activity Condition Editor

The Policy activity in the base activity library provides the ability to execute a RuleSet during the processing of a workflow. A RuleSet is a collection of rules, each with a condition, then actions and else actions; much like an If/Then/Else statement in code. Unlike If/Then/Else statements, rules can have priorities and the engine can detect dependencies between the rules and act on those dependencies to reevaluate rules. For a detailed explanation of the rules engine, rule definition and rule execution semantics, see “Introduction to Windows Workflow Foundation Rules Engine” in Additional Resources.

Much like a workflow definition can be represented in code or XML, a RuleSet can likewise be represented in memory by classes in the System.CodeDOM namespace or it can be serialized to XML. And, like the XAML workflow model, rules in an XML file can be used at the time of workflow creation to provide the conditions and policies used during execution of the workflow. Figure 8 shows the serialized form of the expression defined in the condition editor as shown in Figure 7. With serialized workflow definition and rules, all workflow logic not defined in an activity class can be contained in two XML files.

<RuleDefinitions xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow">

<RuleDefinitions.Conditions>

<RuleExpressionCondition Name="CheckFlag">

<RuleExpressionCondition.Expression>

<ns0:CodeBinaryOperatorExpression Operator="ValueEquality"

xmlns:ns0="clr-namespace:System.CodeDom;Assembly=System,

Version=2.0.0.0,

Culture=neutral,PublicKeyToken=b77a5c561934e089">

<ns0:CodeBinaryOperatorExpression.Left>

<ns0:CodePropertyReferenceExpression

PropertyName="ExpensesApproved">

<ns0:CodePropertyReferenceExpression.TargetObject>

<ns0:CodeThisReferenceExpression />

</ns0:CodePropertyReferenceExpression.TargetObject>

</ns0:CodePropertyReferenceExpression>

</ns0:CodeBinaryOperatorExpression.Left>

<ns0:CodeBinaryOperatorExpression.Right>

<ns0:CodePrimitiveExpression>

<ns0:CodePrimitiveExpression.Value>

<ns1:Boolean xmlns:ns1="clr-namespace:System;

Assembly=mscorlib, Version=2.0.0.0, Culture=neutral,

PublicKeyToken=b77a5c561934e089">

True

</ns1:Boolean>

</ns0:CodePrimitiveExpression.Value>

</ns0:CodePrimitiveExpression>

</ns0:CodeBinaryOperatorExpression.Right>

</ns0:CodeBinaryOperatorExpression>

</RuleExpressionCondition.Expression>

</RuleExpressionCondition>

</RuleDefinitions.Conditions>

</RuleDefinitions>

Figure 8: Serialized Rule Condition

One of the goals of WF is to provide a framework level toolset that allows the runtime to be hosted in any .NET application domain. This flexibility means that workflows can be hosted in rich clients built with Windows Presentation Foundation (WPF) or Windows Forms, or in web applications built with ASP.NET. Developers more and more are using WF to implement the logic in their WCF services and using Internet Information Services (IIS) or Windows Process Activation Service (WAS) to host these services. Each of these hosts provides benefits in their own right and covers many different scenarios. However, when working with the Service Bus and coordinating message exchanges in a cloud environment, the more appropriate host is a farm of servers built and managed by Microsoft in their data centers around the world.

# Microsoft .NET Workflow Service

Building any host for workflows means making decisions about what features the environment will support, and how best to make the environment secure, scalable and stable. As mentioned, today the Microsoft .NET Workflow Service is built on .NET Framework 3.5 and the WF components in that framework release. However, in order to provide the best hosting experience, Microsoft has added a few custom services and they’ve placed some restrictions on the workflows executed in their cloud.

The persistence service used in the cloud is not the SqlWorkflowPersistenceProvider used most commonly by developers using WF. In the cloud environment, in order to leverage the Azure operating environment and provide the best experience for scale and stability, a custom persistence provider uses the storage capabilities of Microsoft SQL Services to maintain the state of running workflows. After all, an internet scale service requires an internet scale data storage and retrieval technology. However, because WF is the same engine in the cloud and in your on-premise solutions, the use of a custom persistence provider is transparent to workflow developers. It feels just like any other WF environment.

To build workflows for the cloud, developers use the same familiar Visual Studio tools, including the same workflow designer to create XAML workflows and rules files. These XML files are then loaded into a server in the cloud where they can be used to create workflow instances. The Microsoft .NET Services SDK includes a project template for creating a SequentialCloudWorkflow which is a specialized version of the standard SequentialWorkflow template. When defining workflows to be run in the cloud, one of the restrictions of the current environment is that you may only use a subset of the activities in the base activity library as well as a suite of custom activities provided as part of the .NET Services SDK.

Requiring workflows to be entirely declarative and restricting the set of activities helps ensure the stability of the environment as custom code cannot be added to the environment. When building a hosting environment for workflows written by any number of developers around the world, this level of control is a must. Because WF today requires full trust to run, there is no ability to easily sandbox custom code on the servers to ensure a limited set of functionality. It should be noted that while there are a restricted set of activities available today, more will be added over time to enhance the capabilities in a cloud-hosted workflow, and as future versions of the .NET Framework are released, the Microsoft .NET Workflow Service will support them as well. In addition, if custom steps are required, on-premise workflow capabilities can be combined with cloud-hosted workflows using the .NET Service Bus.

## Microsoft .NET Services SDK

The Microsoft .NET Services SDK is recommended to build workflows to be hosted in the cloud. The SDK can be downloaded from the Azure Services Platform developer portal on MSDN. Included in the SDK are a project template to build a cloud sequential workflow, a set of cloud activities, and a client API for remotely deploying and managing workflows hosted in the cloud. In addition, several samples are included in the SDK showing basic usage of the tools and the activities. The SDK also includes the tools, documentation and samples related to using the .NET Service Bus and the .NET Access Control Service. Hence, the single SDK provides all the tools surrounding development of hosted workflows.

The WF base activity library provides many control flow activities that are commonly used when building workflows. Rather than recreate custom versions of these activities for the cloud, these framework activities are allowed in workflows that will be run in the cloud. Table 2 shows the activities that are supported in the toolbox allowing developers to add them to a workflow.

|  |  |
| --- | --- |
| Activity | Function |
| IfElse | Provides conditional processing based on a Boolean expression |
| While | Provides conditional looping / iteration of a set of activities |
| Sequence | Executes child activities in order, waiting for each to close before proceeding |
| Suspend | Suspends the execution of a workflow until the host application resumes |
| Terminate | Terminates the workflow execution immediately and all state is deleted |

Table 2: Supported Framework Activities

The SDK also comes with activities for sending and receiving messages over HTTP or via the .NET Service Bus. These activities are the leaf activities used in building workflows hosted in the cloud. They can be used in the primary flow of the workflow or in the fault and cancelation handlers. Table 3 describes these activities and their function. In a later section each of these activities will be examined in detail.

|  |  |
| --- | --- |
| Activity | Function |
| CloudHttpReceive | Receive HTTP requests posted to a specific URL for the workflow instance |
| CloudHttpSend | Invoke HTTP GET or POST operations to a specified URL and get the response |
| CloudServiceBusReceive | Receives messages from an endpoint on the ServiceBus |
| CloudServiceBusSend | Send a message to a specific endpoint on the ServiceBus |
| CloudXPathRead | Reads specified data from an input XML |
| CloudXPathUpdate | Set specified data in an input XML document |
| CloudDelay | Waits for a specified time span |

Table 3: Cloud Activities

With the SDK installed, you can begin creating workflows locally to be hosted in the cloud. Developers follow the same familiar WF development tactics for building cloud workflows, including the use of activity binding, to connect the activities together in a declarative business process. Once the workflow and rules have been defined, a Visual Studio add-in enables developers to simply right-click on the workflow design surface and automatically deploy the workflow to the cloud hosting environment.

## Microsoft .NET Workflow Service Scenarios

With the tools and technologies available today it’s possible to imagine different scenarios that can be implemented with the Microsoft .NET Workflow Service. For example, an organization might choose to use a workflow hosted in the cloud to manage vendor requests to keep their stock of goods up to date. A developer can build a workflow that loops and monitors an HTTP endpoint to receive information from a RESTful service, such as Microsoft SQL Data Services, about product inventory. Data could be extracted about items to order and then the .NET Service Bus could be used to send a request to multiple vendors using a multicast endpoint.

An alternative approach would be to have an on-premise application which determines that inventory of a particular item is low and so the application initiates a workflow process. The on-premise application can send an HTTP message to create and start an instance of a workflow in the cloud and the workflow would then take over processing by notifying vendors via the .NET Service Bus. The workflow can then listen on the .NET Service Bus for vendor replies and coordinate the collection of those replies, finally sending results back to an on premise service.

In addition, cloud based workflows can use the .NET Service Bus to send messages to workflows hosted as WCF services on-premise. This allows a cloud based workflow to instantiate an on-premise workflow with full access to the base activity library, custom activities, and enterprise data so local processing can occur. The on-premise service could then notify the cloud workflow with any results.

# Building Workflows for the Microsoft .NET Workflow Service

To get started building workflows that will be hosted in the .NET Workflow Service, you must setup an account for Microsoft .NET Services. You can register for an account at <http://www.microsoft.com/azure/register.mspx> and configure the identity information and credentials for the main solution account. In addition, once you’ve registered online and been approved, you will need to create a solution that will act as the container for your workflow instances. The next step is to install the Microsoft .NET Services SDK which will provide the activities, workflow templates, and tools to create and deploy a cloud workflow.

## Building Your First Cloud Workflow

I’m going to walk through a very simple workflow to show the steps involved in creating the workflow and how to communicate from a workflow running in the cloud to a WCF service running on-premise using the .NET Service Bus. The first step is to create a new console application project in Visual Studio 2008; this will be the host for the on-premise service. To this project add an interface and a class to define a simple service contract and implementation as shown in Figure 9. Add references to the System.ServiceModel.dll and the Microsoft.ServiceBus.dll assemblies in order to get support for both WCF and the .NET Service Bus.

[ServiceContract(Namespace="")]

public interface IHelloService

{

[OperationContract(IsOneWay=true)]

void SayHello(string input);

}

public class HelloService : IHelloService

{

public void SayHello(string input)

{

Console.WriteLine(input);

}

}

Figure 9: Simple On-Premise Service

With the service defined, write some code in the program.cs file to host the service using an endpoint on the .NET Service Bus. For this example, since the workflow will be calling the service, use the NetEventRelayBinding and configure the TransportClientEndpointBehavior to use UserNamePassword credentials to authenticate to the .NET Service Bus as shown in Figure 10. Be sure to use your own solution name if following along.

ServiceHost host = new ServiceHost(typeof(HelloService));

ServiceEndpoint ep = host.AddServiceEndpoint("HelloCloudService.IHelloService",

new NetEventRelayBinding(),

"sb://{solution-name}.servicebus.windows.net/services/Hello");

TransportClientEndpointBehavior tb = new TransportClientEndpointBehavior();

tb.CredentialType = TransportClientCredentialType.UserNamePassword;

tb.Credentials.UserName.UserName = "{solution-name}";

tb.Credentials.UserName.Password = "{solution-password}";

ep.Behaviors.Add(tb);

host.Open();

Console.WriteLine("Host is listening");

Console.ReadLine();

host.Close();

Figure 10: Hosting the On-Premise Service

To create the cloud workflow, add a new project to the solution and choose the new CloudSequentialWorkflow[[3]](#footnote-3) project. Using the CloudSequentialWorkflow project, the root workflow type will be an activity type that provides validation of the workflow structure to improve the likelihood that the workflow can be executed in the cloud. The designer for the root workflow type also participates in the filtering of the toolbox items available.

Once the project is created and the workflow is in place, the activities can be composed in many different ways to create a business process. The steps in defining the workflow are really no different from designing any workflow using WF, with the exception of the restricted activities and no code-behind model. For this example, add a CloudXPathUpdate activity to the workflow by dragging one from the Cloud Workflow Activities tab of the Visual Studio toolbox. This activity allows you to read XML from another activity, or a literal value, and update a particular node with a new value. Set the InXml property to “<SayHello><input> </input></SayHello>” (do not use quotes when entering these values). Notice that the “input” element is currently empty. In order to populate that element, set the XPathExpression property to “/SayHello/input” to indicate the target node. Finally, set the InNewValue property to “Hello from cloud workflow”. The activity configuration should resemble that shown in Figure 11.

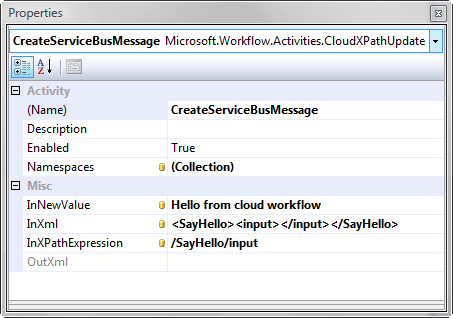


Figure 11: Configuration of the CloudXPathUpdate Activity

Next add a CloudServiceBusSend activity to the end of the workflow. This activity will send the message created by the previous activity. First, set the Body property using activity binding so the value is read from the OutXml property of the CloudXPathUpdate activity previously configured. Configure the activity as shown in Figure 12 by setting the Action property to “urn:IHelloService/SayHello” and the ConnectionMode to “MultiCast”. Finally, set the URL property to the correct address where your service is listening on the .NET Service Bus: “sb://{solution-name}.servicebus.windows.net/services/Hello”. This activity now has all the information it needs to correctly construct a message and send it to the endpoint on the .NET Service Bus. Your completed workflow should look similar to the one shown in Figure 13.

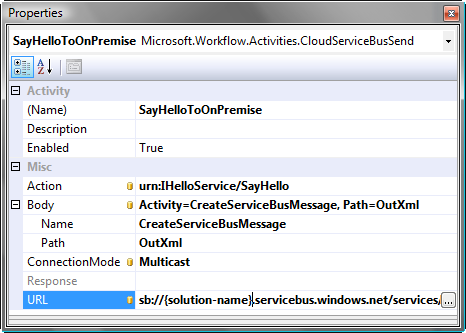


Figure 12: Configuration of CloudServiceBusSend Activity

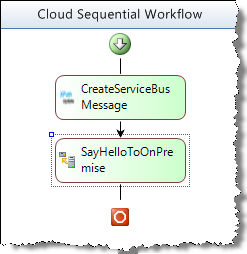


Figure 13: Completed Workflow

In this simple example we are only exercising two of the activities that are available for use in the cloud. Notice also that activity binding, or binding properties on one activity to those on another, is not only supported but actually required, as there is no code behind where developers often “connect the dots”. Workflows must be truly declarative, and activity binding is the way to pass data in a declarative workflow model.

In order to test the solution, make sure the console application hosting the service is the startup project and press CTRL+F5 (or select Debug | Start Without Debugging). Once the service has registered its endpoint, right-click on the workflow design surface for the cloud workflow in Visual Studio and select the “Deploy Workflow” menu item. Selecting this menu item brings up a publishing dialog that will prompt for solution information and credentials as shown in Figure 14. Enter your solution name and password and click the Deploy & Run button. After the solution correctly deploys and starts, you should see your message appear in the console application hosting your .NET Service Bus connected service. You have successfully built, deployed and run a workflow in the cloud, and that workflow has used the .NET Service Bus to communicate with a service hosted in your environment.

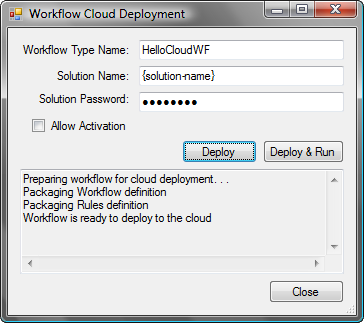


Figure 14: Visual Studio Cloud Workflow Deployment Dialog

In addition to the Visual Studio integration, the .NET Workflow Service provides a Web service interface for management tasks, including deploying workflow types and managing instances. The SDK comes with a client library for integrating with the management interface through traditional .NET code. Additionally, the .NET Services portal can be used from any Web browser to define and edit workflow types as well as manage workflow instances. The combination of the portal, Visual Studio tooling, and client API provide for a reasonable v1 developer experience that can be extended and enhanced over time.

## Cloud Activities

When building cloud-based workflows, the main area of learning is how to use the activities specifically defined for use in the cloud. Each activity has its own unique uses and configurations. The following information is intended to provide detail on what each activity can do and how to properly use it.

### CloudHttpSend

The CloudHttpSend activity allows a workflow to send a request to a specified URL and return both a status code and the response text. This activity can be used to call RESTful services or other HTTP endpoints to retrieve data that can be used in the workflow. The CloudHttpSend activity provides the following properties, some of which are bindable, as indicated by the asterisks.

|  |  |  |
| --- | --- | --- |
| Property | Purpose | Notes |
| URL\* | The address to invoke |  |
| Method | The HTTP method to use | *Any string value is allowed, but the designer provides the most common options.* |
| Request\* | The request body to post | *Ignored when Method = GET* |
| RequestContentType\* | Specifies the content type for the data in the Request property | *Ignored when Method = GET* |
| Response | String containing the response body |  |
| StatusCode | The HTTP status code returned from the web server |  |
| RequestHeaders\* | Allow for sending custom HTTP headers with the request |  |
| ResponseHeaders\* | A collection of headers populated during the request and available after the activity completes execution |  |
| Parameters\* | A collection of query string parameters. | *When using parameters, the parameters are added to the Uri in name/value pairs. Parameters are only included when the Method is GET, DELETE or HEAD.* |

*\* = Dependency (bindable) property*

For the collections, headers and parameters, the designer allows you to specify names for the items in the collection. Each item then becomes available as a property in the property grid to be set statically, or bound to another property in the workflow. For example, to add a query string parameter named ID, with a value of “12345” you click the ellipses to edit the Parameters collection and add a parameter named “ID” as shown in Figure 15.

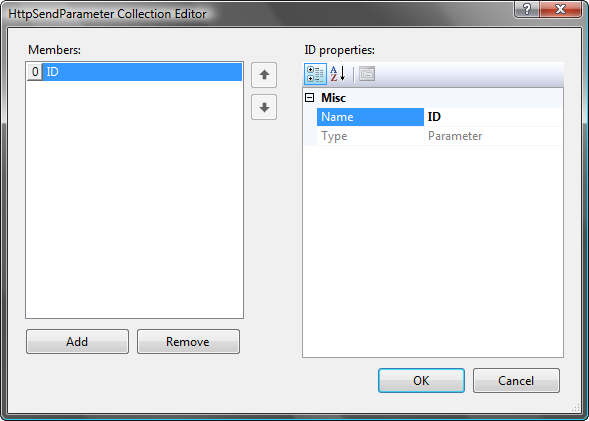


Figure 15: Parameter dialog

Once you have added the parameter definition, the property grid updates to dynamically include the named parameter as a configurable property as shown in Figure 16.

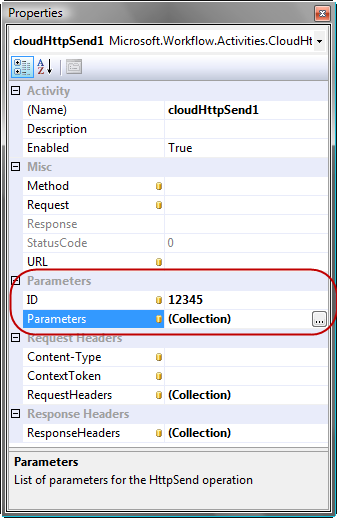


Figure 16: Dynamic parameters on CloudHttpSend activity

This same process of adding named parameters applies to the request and response headers as well. Using these properties enables the passing of values, through activity binding, between requests and responses. For example, you can use the CloudHttpSend activity to post data to a web address. If the service you are invoking sets response headers, you can bind those response header values to request headers on another CloudHttpSend activity to make further requests to the service. This type of interaction is typical in Http message exchanges and these properties enable those scenarios when using workflow to coordinate the messaging.

Because the CloudHttpSend activity supports any HTTP verb, it also enables many scenarios for interacting with RESTful services. Workflows have full capabilities to craft HTTP messages with the correct query string parameters, headers, HTTP verb and body, to create just about any kind of request and be able to retrieve all relevant response data, not just the body.

For example, to POST some data to a REST service, you configure the activity with the URL, method, and payload, at minimum. In Figure 17 you can see an example of the activity being used to send data to a REST endpoint to create a new product. The Request property contains the payload, which it retrieves from another activity through data binding, and the Method has been set to POST. The URL points to the appropriate URL for the service to enable posting data. Notice also that for this particular request, the Content-Type header needed to be changed. This particular header is automatically added to the collection and appears as a property. Other headers can likewise be added.

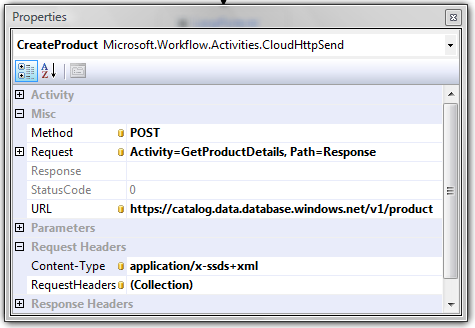


Figure 17: Configuring the CloudHttpSend to POST to a RESTful service

### CloudHttpReceive

The CloudHttpReceive activity is one of two activities, the CloudServiceBusReceive being the other, that enable a workflow to receive data. The CloudHttpReceive is unique, however, in its ability to enable a workflow to be created and started with a message.

If a running workflow needs to have data pushed to it this activity can be used to accept direct HTTP messages posted to a specific URL whereas the CloudServiceBusReceive can listen to an endpoint on the .NET Service Bus. In order to

|  |  |
| --- | --- |
| Property | Purpose |
| Response\* | String containing the response body to be sent to the caller. |
| Request | A string containing the body of data sent to the HTTP endpoint. |
| StatusCode\* | The status code to return to the caller upon completion of the activity. Default value is 200 (OK). |
| CanCreateInstance | Indicates that a properly formed HTTP request can start a new instance of the workflow. |

*\* = Dependency (bindable) property*

Upon execution, this activity interacts with a runtime service loaded into the runtime by the hosting environment and subscribes to be notified when data is sent directly to it. In order to send data to this activity in a running workflow instance, a specific URL format is required which provides the hosting environment the information necessary to route the message correctly to the activity instance, and can be created using the CreateInstanceQueueUri method on the WorkflowClientConfig class.

http://{SolutionName}.workflow.windows.net/workflowsHttp/workflows/{WorkflowTypeName}/instances/{InstanceId}/{ActivityName}

A client application that wants to send data to the workflow can POST data to the unique URL identifying the workflow instance. Figure 18 shows how to write the .NET code for sending an HTTP request to this type of URI, which enables us to communicate with the activity in the running workflow.

string SendHttpRequest(string typeName, string instanceId,

string queueName, string requestMessageBody)

{

Uri serviceUri = WorkflowClientConfig.CreateInstanceQueueUri(

this.solutionName, typeName, instanceId, queueName);

HttpWebRequest request = WebRequest.Create(

serviceUri) as HttpWebRequest;

request.Headers.Add(WorkflowClientConfig.IdentityTokenHeader,

this.authenticationToken);

byte[] requestBodyBytes = Encoding.UTF8.GetBytes(requestMessageBody);

request.ContentLength = requestBodyBytes.Length;

request.ContentType = "text/xml";

request.Method = "POST";

Stream requestStream = request.GetRequestStream();

try {

requestStream.Write(requestBodyBytes, 0,

requestBodyBytes.Length);

}

finally {

requestStream.Close();

}

HttpWebResponse response =

request.GetResponse() as HttpWebResponse;

Stream stream = response.GetResponseStream();

using (StreamReader reader = new StreamReader(stream)) {

string contents = reader.ReadToEnd();

return contents;

}

}

Figure 18: Posting Data to a CloudHttpReceive Activity

The data in the URL contains the information the hosting environment needs to send data to the waiting activity. To understand how the communication architecture works in WF and how by using the instance id and the activity name (queue name) the host can send data to the correct activity, see “Workflow Communications” in Additional Resources.

This sample uses the basic HttpWebRequest and HttpWebResponse classes found in the System.Net namespace to manage posting the data and reading the response. The one unique element of this sample is the inclusion of the custom header. In order to post a message, the user must be authenticated, and because this is not a SOAP request, the authentication token is sent in the HTTP header. shows sample code that can be used to retrieve an authentication token from the .NET Access Control Service by passing the solution name and password in the query string. The token is contained in the response body and once read out, can be used to invoke operations on the workflow.

string GetAuthenticationToken(string solutionName, string password)

{

string tokenUrl = string.Format(

"https://{0}/issuetoken.aspx?u={1}&p={2}",

WorkflowClientConfig.StsHostName, solutionName, password);

HttpWebRequest tokenRequest =

(HttpWebRequest)WebRequest.Create(tokenUrl);

HttpWebResponse tokenResponse =

(HttpWebResponse)tokenRequest.GetResponse();

byte[] tokenBody = new byte[500];

int tokenBodyLength = tokenResponse.GetResponseStream().Read(

tokenBody, 0, 500);

tokenResponse.Close();

return Encoding.UTF8.GetString(tokenBody, 0, tokenBodyLength);

}

Figure 19: Get Authentication Token for HTTP POST

The HTTP endpoint is secured using the .NET Access Control Service. By default, the .NET Access Control Service defines a scope named http://{solutionName}.workflow.windows.net/workflowsHttp in the “workflow” solution which contains a single rule. That rule provides access to the Send operation to the account that owns {solutionName}. You can add other rules to the scope specifying specific input claims that must be met, and setting the output claim of type “Action” to “Send” enabling other users or services to send data to your workflows. The .NET Access Control Service checks the caller for the incoming claims and if they match the rule, presents the outgoing claim (for the Send action) to the listener. Only when that claim is present will the listener accept the message and send it to the activity.

This example shows sending data to a running workflow; it is also possible to start a workflow by sending data in a similar fashion but with a few notable differences. First, the URI that messages are posted to does not include the instance ID or the *instances* path information as shown here.

http://{SolutionName}.workflow.windows.net/workflowsHttp/workflows/{WorkflowTypeName}/{ActivityName}

As with messages intended for a specific instance, the WorkflowClientConfig class provides a method, CreateActivationQueueUri, that will create the correct URI for you given the placeholders of {SolutionName}, {WorkflowTypeName} and {ActivityName}[[4]](#footnote-4).

Before a workflow instance can be created and started using this method, both the workflow type and the CloudHttpReceive activity must be configured correctly. The CloudHttpReceive activity must have the CanCreateInstance property set to "True" while the workflow type must have the AllowActivation property enabled. Each tool for deploying or updating workflows provides a means to set the AllowActivation property on the workflow type. There are checkboxes in the Visual Studio deploy dialog (see ) and in the web portal discussed later (see ). The API for managing workflows through code provides method overloads that accept a boolean value indicating whether the workflow type allows activation as shown in .

WorkflowClient workflowClient = new WorkflowClient(this.solutionName);

workflowClient.CredentialType = TransportClientCredentialType.UserNamePassword;

workflowClient.Credentials.UserName.UserName = this.solutionName;

workflowClient.Credentials.UserName.Password = this.password;

workflowClient.Open();

**workflowClient.CreateWorkflowType(this.workflowTypeName, this.workflowXoml,**

**this.workflowRules, true);**

workflowClient.Close();

Figure : Enabling activation with the WorkflowClient API

Once the workflow has been properly enabled for activation a message can be posted to the URI as before. However, because this message will cause a workflow instance to be created and run, the request will need to include an additional header to specify the identity under which the workflow instance should operate. This becomes especially important when the workflow instance attempts to interact with the .NET Service Bus through either the CloudServiceBusSend or CloudServiceBusReceive activities. shows the code to create an start a new workflow instance with repetitive code from removed. Notice that the URI created is specifically an activation URI rather than an instance URI as we are activating a new instance. Also, the WorkflowTokenHeader is added to the headers of the request in order to provide the identity under which the workflow instance should execute. In this example the same identity token is used to authenticate when sending the message and to identify the execution identity, but different tokens could be used for each header.

private void CreateAndStartWorkflow(string xmlRequest, string token)

{

**Uri serviceUri = WorkflowClientConfig.CreateActivationQueueUri(**

**Constants.SolutionName, WF\_TYPE\_NAME, QUEUE\_NAME);**

HttpWebRequest request =

WebRequest.Create(serviceUri) as HttpWebRequest;

request.Headers.Add(WorkflowClientConfig.IdentityTokenHeader, token);

**request.Headers.Add(WorkflowClientConfig.WorkflowTokenHeader, token);**

byte[] requestBodyBytes = Encoding.UTF8.GetBytes(xmlRequest);

request.ContentLength = requestBodyBytes.Length;

request.ContentType = "text/xml";

request.Method = "POST";

Stream requestStream = request.GetRequestStream();

try{

requestStream.Write(requestBodyBytes, 0, requestBodyBytes.Length);

}

finally{requestStream.Close();}

//response omitted

//. . .

Figure : Creating a workflow instance with a message

### CloudXPathRead

The CloudXPathRead activity provides the ability to extract information from an HTTP payload or a .NET Service Bus message. As such, this activity is primarily used with the CloudHttpSend, CloudHttpReceive, CloudServiceBusReceive and CloudServiceBusSend activities.

|  |  |  |
| --- | --- | --- |
| Property | Purpose | Notes |
| InXml\* | String containing the source XML document or fragment. | *Almost always bound to a communication activity.* |
| InXPathExpression\* | A string value containing the XPath expression to be evaluated. |  |
| OutReadValue | A string containing the result of evaluating the XPath expression. | Contains the resulting value of the query, which is generally simple content. |
| Namespaces\* | A collection of namespace prefix mappings. | *Enables using namespaces, through the mapped prefix, in the InXPathExpression.* |

*\* = Dependency (bindable) property*

The CloudXPathRead activity evaluates the XPath expression against the input XML and provides the value of the located node. The expression will be evaluated to find a single node, and the value of that node, if found, will be returned. Thus this activity cannot be used to query a document and retrieve a block of XML. The primary use case for this activity is to inspect a payload and then surface the value so that it can be used in conditions on the IfElse activity or the While activity.

For example, Figure 22 shows a CloudXPathRead activity consuming a message from a CloudHttpReceive activity in a workflow. The InXml property on the CloudXPathRead activity is bound to the Request property on the CloudHttpReceive activity. The InXPathExpression property is configured with a static XPath expression indicating the path to the element in the XML which should be extracted.

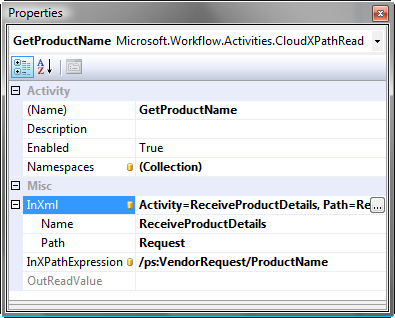


Figure 22: CloudXPathRead and CloudHttpReceive Activities

When the GetProductName (CloudXPathRead) activity executes, the XPath expression is evaluated to select a single node in the input XML document. The value of the node selected is used to populate the OutReadValue property on the activity. In this example, a client application starts the workflow and posts XML in the following format.

<ps:VendorRequest xmlns:ps="http://www.pluralsight.com/azure/services/">

<ProductName>Chai Tea</ProductName>

<ProductID>1</ProductID>

<Amount>250</Amount>

</ps:VendorRequest>

The XML is posted to a URL in the following format where the workflow type is “VendorWorkflow” and uses the name of the CloudHttpReceive activity: “ReceiveProductDetails”.

http://{SolutionName}.workflow.windows.net/workflowsHttp/workflows/VendorWorkflow /instances/{InstanceId}/ReceiveProductDetails

Notice in Figure 22 that the XPathExpression uses the “ps” prefix. In order for this syntax to work, the Namespaces collection must have a mapping from this prefix to the actual namespace. Figure 23 shows the dialog box to for configuring XML Namespace Item objects. The Name property refers to the prefix you will use in your XPath to refer to the namespace, while the Uri property represents the actual namespace uri the prefix onto which the prefix maps.

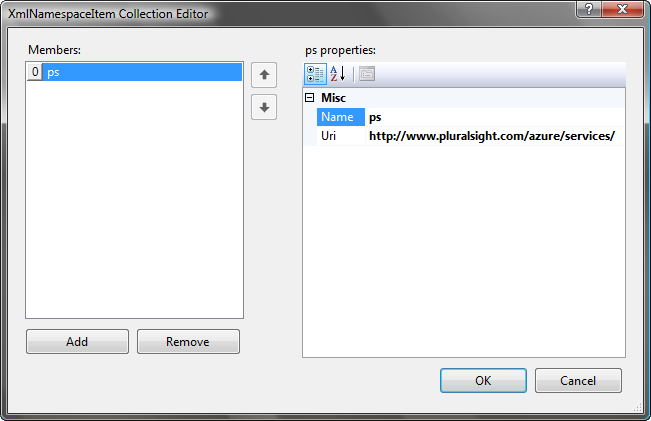


Figure 23: Configuring namespace prefixes

### CloudXPathUpdate

The CloudXPathUpdate activity is the counterpart to the CloudXPathRead activity and provides the ability to set the value of an XML element in a document. The most likely scenario for using this activity is when you have an XML template that contains most of the information needed to construct an outbound message, but you need to insert specific values into the XML at runtime.

|  |  |  |
| --- | --- | --- |
| Property | Purpose | Notes |
| InXml\* | String containing the source XML document or fragment. | *Almost always bound to a communication activity or another CloudXPathUpdate*. |
| InXPathExpression\* | A string value containing the XPath expression to be evaluated. |  |
| InNewValue\* | A string containing the value to be inserted into the node identified by the InXPathExpression. |  |
| OutXml | The result of inserting the new value in the input document. |  |
| Namespaces\* | A collection of mappings between namespace prefixes and namespace URIs. | Use to identify the namespace identified by prefixes used in the InXPathExpression property. |

*\* = Dependency (bindable) property*

This activity will generally be used to create a message that can be used by the CloudServiceBusSend or the CloudHttpSend activities. Each of those activities has the ability to bind their outgoing message body to another activity, and using the OutXml property on the CloudXPathUpdate will be common. Note also that the CloudXPathUpdate activities can be chained together to update multiple values in an XML document. You are not limited to only setting a single value in the document. To accomplish this, each activity in the chain would bind its InXml property to the OutXml property of the previous activity.

As an example, Figure 24 shows a CloudXPath activity being used to create a message that will be sent to a .NET Service Bus endpoint. The InXml property is bound to the OutXml property of a previous CloudXPathUpdate, thus chaining them together in order to make multiple updates to an XML document. The InXPathExpression property indicates the location in the XML document where the new value should be inserted. The InNewValue property is bound to a CloudXPathRead activity that occurs earlier in the workflow; thereby copying a value from the XML sent to the workflow and using it in an outgoing message.

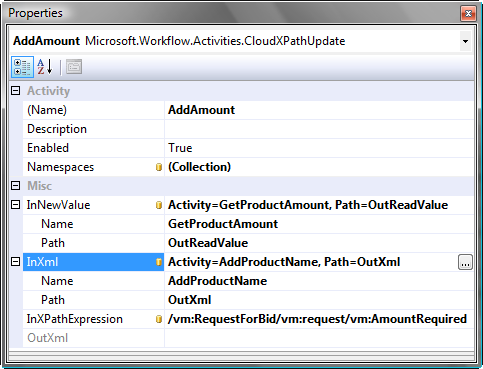


Figure 24: Configuring the CloudXPathUpdate Activity

Notice that the XPath expression references Xml namespaces by their prefix. The namespaces collection has entries for each prefix used, just as the CloudXPathRead activity did.

### CloudServiceBusSend

The CloudServiceBusSend activity allows a workflow to send a message to a .NET Service Bus endpoint, which allows for more advanced network traversal scenarios. It supports the following properties.

|  |  |  |
| --- | --- | --- |
| Property | Purpose | Notes |
| URL\* | The address of the .NET Service Bus endpoint |  |
| Action\* | The SOAP action to apply to the message. | *The action often drives the operation invoked on a service.* |
| Body\* | A string representing the request body to send in the SOAP message. |  |
| ConnectionMode\* | Specifies the .NET Service Bus connection mode used to send the message. | *Currently supports Tcp and Multicast* |
| Response | String containing the response body | *Only when ConnectionMode = Tcp* |

*\* = Dependency (bindable) property*

The CloudServiceBusSend allows for more advanced communications with support for using TCP to do request/response style interactions using the firewall traversal capabilities of the .NET Service Bus or by using the multicast support to direct messages to all authorized listeners on a .NET Service Bus endpoint. This activity can be used to send messages as part of a service orchestration that integrates with many partners, or it can be used to invoke service operations on-premise as part of a hybrid solution.

When using a connection mode of “Tcp”, the client channel created to communicate with the .NET Service Bus uses the NetTcpRelayBinding to establish communication. This is one of several new bindings provided in the .NET Services SDK for communicating with the Service Bus. Using this connection mode, the activity can invoke a method on Service Bus endpoint and receive a response.

When using a connection mode of “Multicast”, the client channel uses the NetEventRelayBinding which supports sending one way messages to a single endpoint, and having that message broadcast to multiple subscribers. In this scenario, the activity does not receive a response directly as the operation is strictly one way. In order to receive a response, the workflow would have to use CloudHttpReceive.

The example in Figure 25 shows a CloudServiceBusSend activity configured to send a message to a Multicast endpoint on the .NET Service Bus. The Body property is bound to the OutXml property of the preceding CloudXPathUpdate activity allowing the content of the message to be manipulated at runtime before the message is sent. The Action property specifies the SOAP action to be applied to the message when it is created and corresponds to the Action [[5]](#footnote-5)specified on the OperationContract on the method.

Specifying the address, along with the other details, provides the activity enough information to create a WCF client channel and create a SOAP message with the correct Action and Body. By choosing the Multicast option for ConnectionMode, I have indicated that the NetEventRelayBinding should be used. For more information on using WCF to communicate through the .NET Service Bus, read the companion article in this series found in the Additional Resources section.

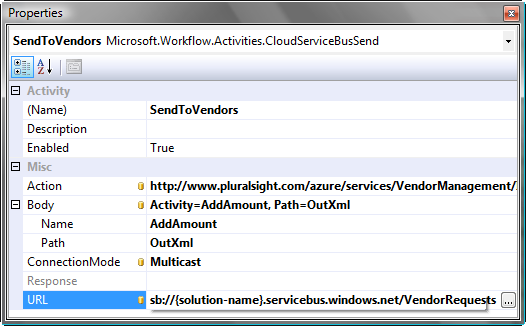


Figure 25: CloudServiceBusSend Activity

### CloudServiceBusReceive

The CloudServiceBusReceive activity enables a workflow to listen on a ServiceBus endpoint and wait for messages to arrive. Unlike the CloudHttpReceive activity, this activity cannot activate a workflow; the workflow must be running so the activity can register itself on the endpoint.

|  |  |  |
| --- | --- | --- |
| Property | Purpose | Notes |
| SubscriptionAddress\* | The address of the .NET ServiceBus endpoint on which the activity should register. | *Currently, this endpoint must have a router policy defined in order to be supported by this activity.* |
| Message | The message received on the endpoint, represented as a string. | *If the message is SOAP message, the string will contain the entire SOAP message with the envelope and all headers.* |

*\* = Dependency (bindable) property*

The CloudServiceBusReceive activity opens many opportunities to define automated responses to messages being sent through the .NET Service Bus. You can start a workflow, then have it listen on the queue in a loop and each time a message arrives, the service responds. For example, after sending several requests out to vendors using the CloudServiceBusSend activity in Multicast mode as shown previously, I can add a CloudServiceBusReceive activity to my workflow to accept the replies from the vendors. The setup of the activity itself is very simple because I only need to supply an address on which the activity should register as shown in .

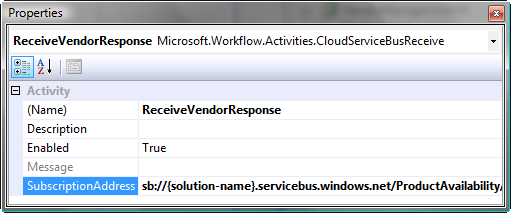


Figure :Configuring the CloudServiceBusReceive activity

The currently supported pattern for enabling the endpoint is to define a router policy [[6]](#footnote-6)at that endpoint enabling messages sent to that endpoint to be forwarded to any subscribing applications, including your workflow instances. shows code for creating a router on a named relative endpoint. Once the endpoint is created, clients can send messages to it, and the workflow instance will receive those messages and continue processing.

private void CreateServiceBusRouter(string SBServiceName)

{

TransportClientEndpointBehavior transportClientEndpointBehavior = null;

RouterPolicy routerPolicy = null;

Uri SBUri = ServiceBusEnvironment.CreateServiceUri("sb",

Constants.SolutionName, SBServiceName);

transportClientEndpointBehavior = new

TransportClientEndpointBehavior();

transportClientEndpointBehavior.CredentialType =

TransportClientCredentialType.UserNamePassword;

transportClientEndpointBehavior.Credentials.UserName.UserName =

Constants.SolutionName;

transportClientEndpointBehavior.Credentials.UserName.Password =

Constants.Password;

routerPolicy = new RouterPolicy();

routerPolicy.ExpirationInstant = DateTime.UtcNow.AddHours(1);

routerPolicy.MaxSubscribers = 5;

routerClient = RouterManagementClient.CreateRouter(

transportClientEndpointBehavior, SBUri, routerPolicy);

}

Figure : Creating a router policy on a .NET Service Bus endpoint

The message body received by the activity is made available as a property that can be consumed through binding. In the current example, once I receive a message from a vendor, I use the CloudXPathRead activity extract important information to make decisions about how to proceed. shows the configuration of a CloudXPathRead activity that extracts the available amount of product as sent by the vendor. Based on this extraction I can configure IfElse activity conditions to follow different paths in my workflow, or create new messages to send to an on premise service that can finalize the purchase process with the vendor.

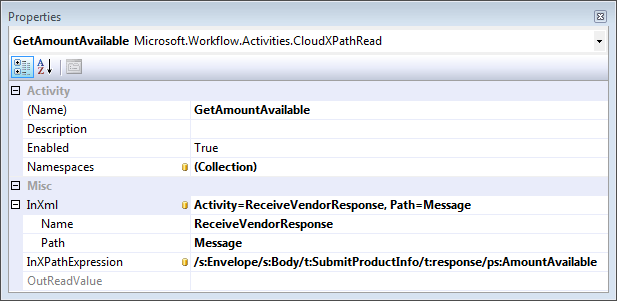


Figure : Reading data from a .NET Service Bus message

### CloudDelay

The CloudDelay activity is slightly different from the Delay activity found in the base activity library in that is uses a separate timer service. In WF, the responsibility of managing timers is put on the persistence service and requires polling of the database periodically to look for expired timers. In order to meet the demand of an internet scale workflow host, a custom timer service was developed and is used by the CloudDelay activity to manage a time based subscription. Other than these implementation details, the activity can be used in the same way as the standard Delay activity.

# Managing Cloud Workflows

Once a workflow has been developed it must be deployed and managed; especially when the workflows are deployed into the cloud. The .NET Workflow Service currently provides a SOAP web service which allows an authenticated user to manage workflow types and instances. This same web service is used by several tools to provide higher level management interfaces.

The Azure Services Platform Web portal provides a Web based management user-interface which uses the Web service to manage instances and types. The simple right-click deployment option available in Visual Studio also uses the Web service to deploy the type and run an instance if that option is chosen. Finally, in the Microsoft .NET Services SDK, the WorkflowClient class is a pre-built client proxy class which can be used to manage cloud based workflows. In the future, the management capabilities will be expanded to fully support current and emerging Web standards including a RESTful interface.

## Azure Management Portal

The Azure Service Platform management portal provides tools for managing many different aspects of the platform including the Microsoft .NET Services. For managing workflows, the two primary functions provided on the portal are creating and editing workflow types and managing running instances. Figure 29 shows the landing page for managing types in a new solution where no types yet exist.

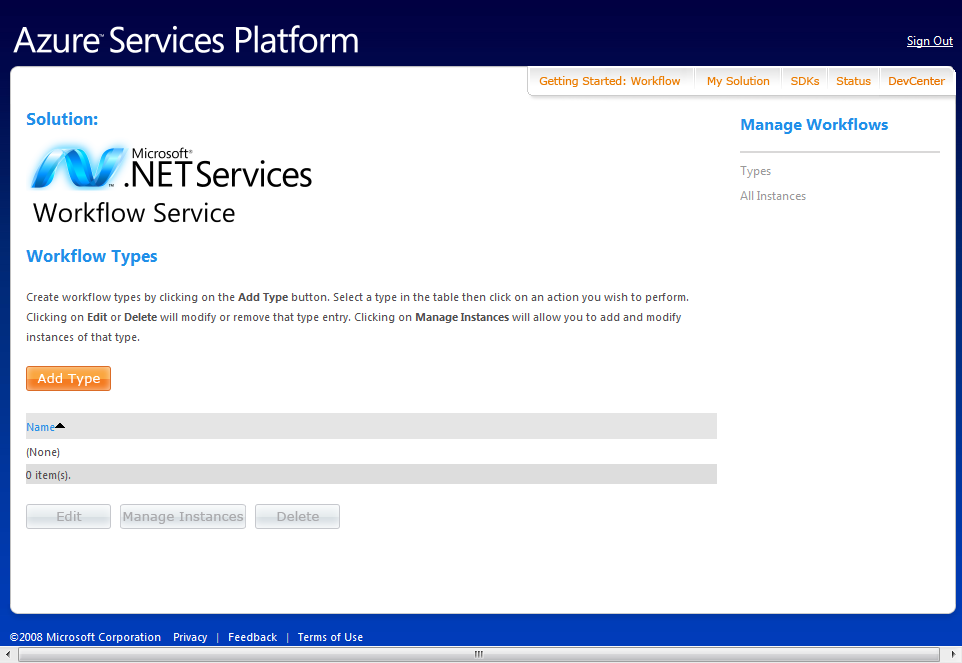


Figure 29: Workflow Management Portal

To create a type, you simply click the Add Type button and then you’re presented with a page containing text boxes for entering the workflow XAML and rules XML and saving them in your solution. The type will be validated as you save it to make sure there are no obvious issues. If you have created the workflow using the Visual Studio tools, then the validation should have already enforced the rules. Figure 30 shows the simple interface for adding or editing a type using the web portal. The Allow Activation check box allows you to specify that instances of this workflow type can be created and started upon receiving an HTTP message. This works in conjunction with the CloudHttpReceive activity and its CanCreateInstance property. Both must be enabled for that scenario to work.

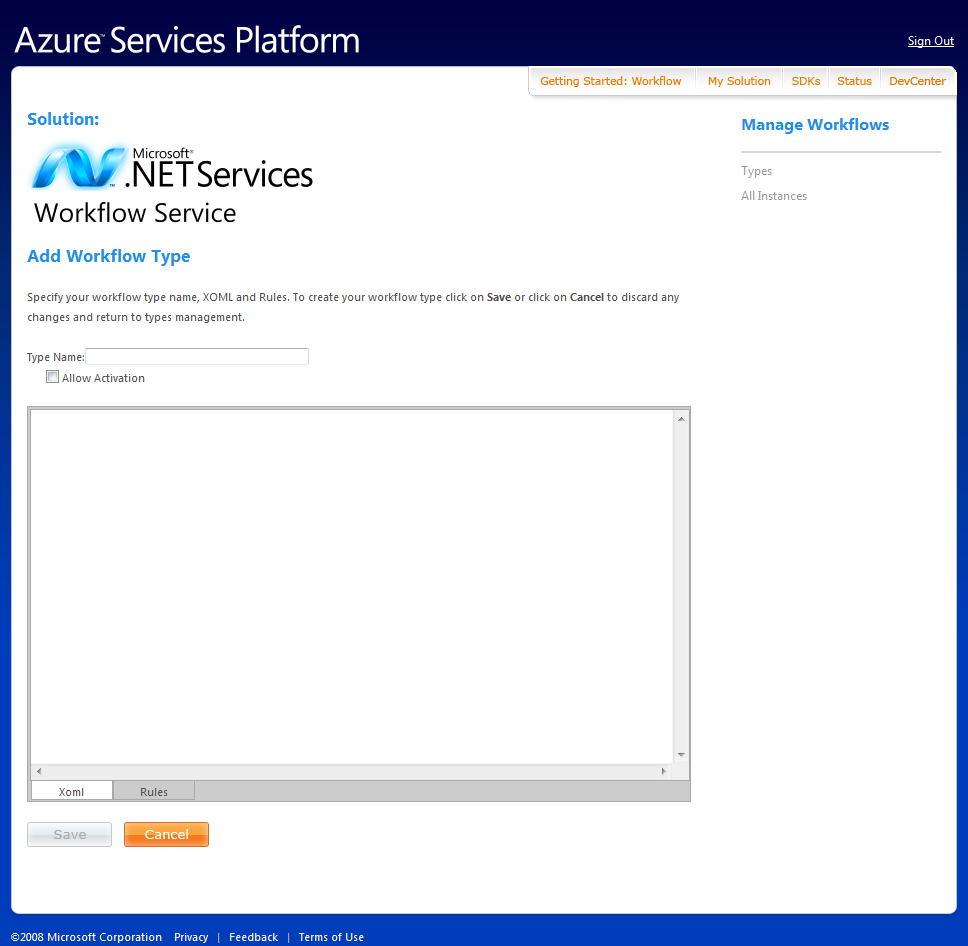


Figure 30: Add/Edit Workflow Type Page

Once a type has been defined, you can select the type and manage the instances. Instance management allows for starting, terminating, suspending and resuming workflow instances.

Basic tracking information about each workflow instance is stored until an administrator deletes the workflow instance. The information tracked includes current status, timestamps, and information on the security context under which the instance is running. You can look for exception messages for terminated workflows on the details page. Figure 31 shows the instance management page while Figure 32 shows the details page.

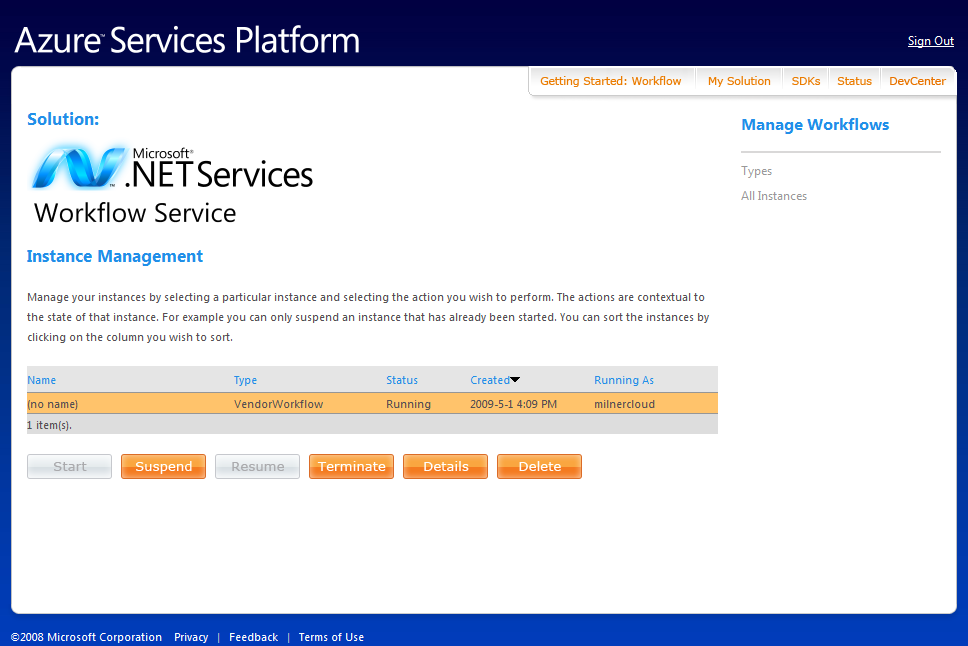


Figure 31: Instance Management Page

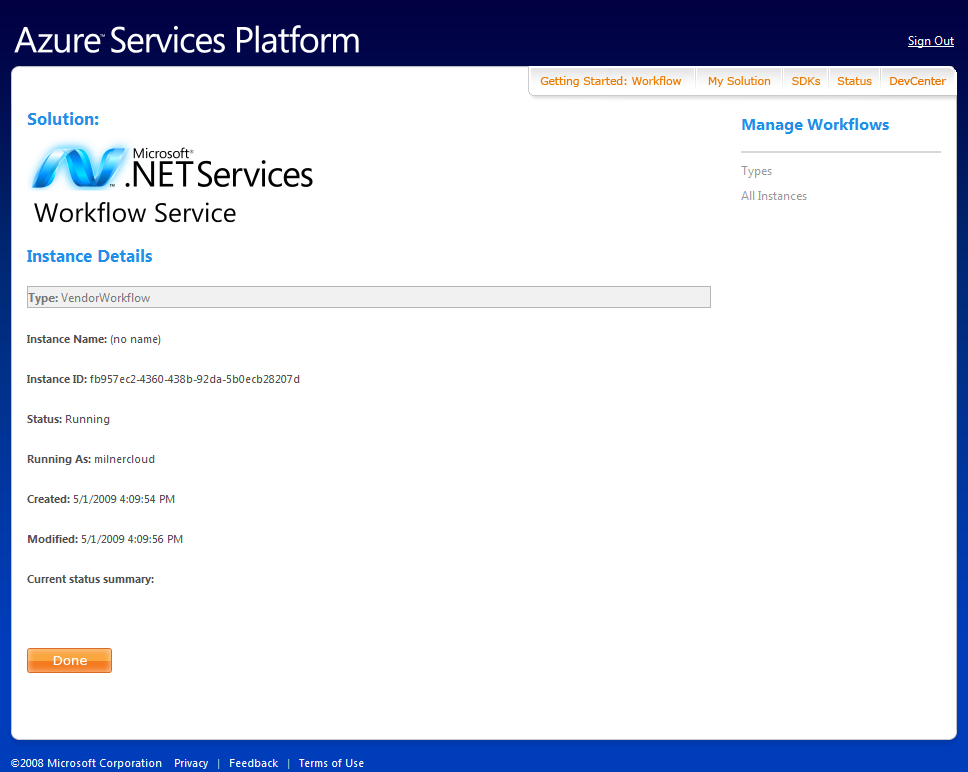


Figure 32: Instance Details Page

The goal of the portal is to provide a management interface that can be accessed from anywhere and provide the functionality required to be able to modify workflow definitions or manage specific instances. Most developers will do the initial deployment of their workflow in Visual Studio and make simple changes using the portal. A more complete solution might include a rich client application to allow editing/updating of workflows from any connected system using the web service interface.

## Custom Workflow Management using WorkflowClient

The WorklfowClient class is essentially a pre-built proxy for interacting with the SOAP management services. This proxy provides operations for creating, deleting and updating workflow types. For instances, the service supports create, start, suspend, terminate, resume, abort, and delete.

The web service API provides all the operations required to build the editing and management of workflow instances into an on-premise application. For example, it provides operations to query the service and retrieve a list of types for a given user, as well as operations to retrieve the workflow definition and update the workflow. As Figure 33 shows, you can use these operations together to retrieve a workflow type, make changes to it, and then update the type information in the cloud. New instances created for this type would then use the new definition.

WorkflowClient client = new WorkflowClient(solutionName);

client.CredentialType =

TransportClientCredentialType.UserNamePassword;

client.Credentials.UserName.UserName = solutionName;

client.Credentials.UserName.Password = password;

client.Open();

WorkflowTypeDescription typeInfo =

client.GetWorkflowType(workflowTypeName);

string xaml = typeInfo.Xoml;

string rules = typeInfo.Rules;

//Update type with custom logic or editor

client.UpdateWorkflowType(workflowTypeName, xaml, rules);

client.Close();

Figure 33: Retrieve and Update Workflow Type Information

Managing running instances begins with being able to create an instance based on the type. In order to create an instance of a workflow, additional data is required. The type information and username of the account are the obvious bits of data, but when a workflow is running, it needs to run as a particular user.

Because of this, the CreateWorkflowInstance operation takes an additional parameter for a security token. The security token is acquired with an HTTP GET operation to the authorization service, as shown earlier with the CloudHttpReceive activity and represents the user identity under which the workflow will execute. This identity will determine what resources the workflow has access to, most importantly what .NET Service Bus endpoints can be invoked with the CloudServiceBusSend activity.

Figure 34 shows an example of creating a workflow instance passing in a security token for the identity. Notice that once the workflow has been created, it can also be started with another service call. The return value from the create operation is the instance identifier for the workflow instance.

WorkflowClient client = new WorkflowClient(solutionName);

client.CredentialType = TransportClientCredentialType.UserNamePassword;

client.Credentials.UserName.UserName = solutionName;

client.Credentials.UserName.Password = password;

client.Open();

client.CreateWorkflowType(workflowTypeName, workflowXoml, workflowRules);

//Create a workflow instance

string instanceId = client.CreateWorkflowInstance(this.workflowTypeName,

authenticationToken);

//start a workflow instance

client.StartWorkflowInstance(workflowTypeName, instanceId);

client.close();

Figure 34 - Create Workflow Instance

Monitoring running workflows is probably one of the most important capabilities required for a hosted workflow service. The web service provides several different query operations to get a list of workflow identifiers or a list of WorkflowInstanceInfo objects.

The WorkflowInstanceInfo class provides the same summary details as seen in the portal interface including timestamps, status, error messages, etc. The likely use of these methods is to get a list of workflows from the service, display the list, and upon selection, show the details.

Figure 35 shows one example of using these operations to allow a user to get instance information for a given type. Here the user is shown the list of types and chooses a particular type, then the list of summary information for the instances of that type are retrieved and shown to the user.

WorkflowClient client = new WorkflowClient(solutionName);

client.CredentialType = TransportClientCredentialType.UserNamePassword;

client.Credentials.UserName.UserName = solutionName;

client.Credentials.UserName.Password = password;

client.Open();

//display type names

string[] typeNames = client.GetWorkflowTypeNames();

Console.WriteLine("Currently deployed types:");

for (int typeIndex=0;typeIndex < typeNames.Length; typeIndex++)

{

Console.WriteLine("{0}) {1}", typeIndex, typeNames[typeIndex]);

}

//select the type

Console.Write("Select a type:");

string inputValue = Console.ReadLine();

int selectedIndex = int.Parse(inputValue);

string selectedType = typeNames[selectedIndex];

//display the list of instances

WorkflowInstanceInfo[] infos =

client.GetAllWorkflowInstanceInfoByType(selectedType);

foreach (WorkflowInstanceInfo info in infos)

{

Console.WriteLine("Summary details for {0}", info.InstanceName);

Console.WriteLine("\tCurrent status: {0}", info.Status);

Console.WriteLine("\tDate Created: {0}", info.DateCreated);

}

client.Close();

Figure 35 - Interactive Instance Browsing

The web service programming interface provides the logic necessary to manage the types and instances, and two user interfaces have been created already: the Azure portal and the Visual Studio deployment tool. The encapsulation of the logic into the service enables custom tools to integrate the workflow management features with little effort, and with the complexity managed on the servers.

# Summary

The .NET Workflow Service builds on the previous investments in the .NET Framework as well as the current investments in the Azure Services Platform. It provides the platform for developing and managing workflows that coordinate messages between the .NET Service Bus, on premise applications and cloud-deployed services and applications.

Future releases will continue to expand the capabilities through new activities and tooling, building on the investments in the foundational components and tighten the integration with the .NET Service Bus. The primary goal of the Microsoft .NET Workflow Service is to become a universally accessible, and highly scalable host for WF workflows. In future releases, more of the activities included in the framework will be made available for execution in the cloud. As WF and the entire .NET Framework evolves, the Microsoft .NET Workflow Service will evolve right along with it, providing support for new features, taking advantage of advances in declarative programming, and continuing to leverage the investment developers have made in learning WF.

# Additional Resources

We’ve provided links to several resources below that will further your education on the suite of Microsoft® .NET Services and the .NET Workflow Service in particular.

## Microsoft® .NET Services Whitepaper Series

* An Introduction to Microsoft .NET Services for Developers
  + <http://go.microsoft.com/?linkid=9638347>
* A Developer’s Guide to the Microsoft® .NET Service Bus
  + <http://go.microsoft.com/?linkid=9638348>
* A Developer’s Guide to the Microsoft® .NET Access Control Service
  + <http://go.microsoft.com/?linkid=9638349>
* A Developer’s Guide to the Microsoft .NET Workflow Service (*this paper*)
  + <http://go.microsoft.com/?linkid=9638350>

## .NET Workflow Service Resources

* Azure Services Platform
  + <http://www.microsoft.com/azure/services.mspx>
* Register for Azure Services
  + <http://www.microsoft.com/azure/register.mspx>
* Microsoft® .NET Services
  + <http://www.microsoft.com/azure/netservices.mspx>

## WF Resources

* Introduction to Windows Workflow Foundation Rules Engine
  + <http://msdn.microsoft.com/en-us/library/aa480193.aspx>
* Loading Workflow Models in WF
  + <http://msdn.microsoft.com/en-us/magazine/cc507645.aspx>
* Windows Workflow Foundation Activity Documentation
  + <http://msdn.microsoft.com/en-us/library/ms733615.aspx>
* Workflow Communications
  + <http://msdn.microsoft.com/en-us/magazine/cc163365.aspx>

# About the Author

Matt is a member of the technical staff at Pluralsight, where he focuses on connected systems technologies (WCF, Windows WF, BizTalk, "Dublin" and the Azure Services Platform). Matt is also an independent consultant specializing in Microsoft .NET application design and development. As a writer Matt has contributed to several journals and magazines including MSDN Magazine where he currently authors the workflow content for the Foundations column. Matt regularly shares his love of technology by speaking at local, regional and international conferences such as Tech Ed. Microsoft has recognized Matt as an MVP for his community contributions around connected systems technology.

# Acknowledgements

This paper would not have been complete without the vision of Moustafa Khalil Ahmed and his excellent insights into the current and future value of the .NET Workflow Service. Thanks also to Dan Smiley, David Wortendyke, and Scot Gellock for their technical feedback and quality control.

1. Microsoft® .NET Services is the new, more appropriate name, for the original *BizTalk Services* initiative. [↑](#footnote-ref-1)
2. For more information on Microsoft® .NET Services, the .NET Access Control Service, and the Service Bus, see the accompanying papers in the Microsoft .NET Services Whitepaper Series referenced at the end of this paper. [↑](#footnote-ref-2)
3. In the March 2009 CTP release, only the WF 3.5 sequential workflow model is supported. [↑](#footnote-ref-3)
4. The WorkflowClientConfig methods refer to the ActivityName as QueueName, but the string used in the URI must match the name of the waiting activity, so ActivityName is used here. [↑](#footnote-ref-4)
5. In many cases developers do not specify the Action property in the OperationContract and choose to let WCF provide a default value using the service namespace, contract name, and the method name. [↑](#footnote-ref-5)
6. For more information on router policy in the .NET Service Bus, see A Developer’s Guide to the Microsoft® .NET Service Bus in the Additional Resources section. [↑](#footnote-ref-6)