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

An Introduction to Microsoft® .NET Services for Developers

The .NET framework for the cloud

Aaron Skonnard, Pluralsight

May 2009

*All information and code samples are based on the March 2009 CTP release of .NET Services.*

Contents

[Abstract 4](#_Toc228292237)

[An Overview of the Azure Services Platform 4](#_Toc228292238)

[Windows® Azure™ 4](#_Toc228292239)

[Microsoft® .NET Services 5](#_Toc228292240)

[Microsoft® SQL Services 5](#_Toc228292241)

[Live Services 6](#_Toc228292242)

[Additional Service Offerings 6](#_Toc228292243)

[Building on the Azure™ Services Platform 7](#_Toc228292244)

[Introducing Microsoft® .NET Services 8](#_Toc228292245)

[Overview of .NET Services 8](#_Toc228292246)

[Getting Started with .NET Services 9](#_Toc228292247)

[Creating your First Solution 11](#_Toc228292248)

[Managing your Solution Credentials 14](#_Toc228292249)

[Samples in the .NET Services SDK 14](#_Toc228292250)

[Microsoft® .NET Service Bus 15](#_Toc228292251)

[Relayed Connectivity 16](#_Toc228292252)

[Direct Connectivity 17](#_Toc228292253)

[Relay Addresses and Access Control 18](#_Toc228292254)

[Service Registry 19](#_Toc228292255)

[Queues and Routers 19](#_Toc228292256)

[Integration with WCF 20](#_Toc228292257)

[A Simple .NET Service Bus Example 21](#_Toc228292258)

[Microsoft®.NET Access Control Service 26](#_Toc228292259)

[Claims-based Identity 26](#_Toc228292260)

[Benefits of Single Sign-on 27](#_Toc228292261)

[Administration Portal 27](#_Toc228292262)

[A Simple .NET Access Control Service Example 28](#_Toc228292263)

[Microsoft® Workflow Service 32](#_Toc228292264)

[A Cloud-based Workflow Host 33](#_Toc228292265)

[Cloud Hosting and Persistence 33](#_Toc228292266)

[Designing Workflows for the Cloud 33](#_Toc228292267)

[Cloud Activities 34](#_Toc228292268)

[A Simple .NET Workflow Service Example 35](#_Toc228292269)

[Bringing it all Together 38](#_Toc228292270)

[Summary 40](#_Toc228292271)

[Additional Resources 40](#_Toc228292272)

[Microsoft® .NET Services Whitepaper Series 40](#_Toc228292273)

[Microsoft® .NET Services Resources 40](#_Toc228292274)

[About the Author 41](#_Toc228292275)

[Acknowledgements 41](#_Toc228292276)

# Abstract

This whitepaper is the first in a series of whitepapers dedicated to Microsoft® .NET Services, a core part of the Azure™ Services Platform. Microsoft® .NET Services provides a set of developer-oriented building block services commonly required by cloud-based and cloud-aware applications. This overview paper introduces Microsoft® .NET Services, each of its building block services, and how they fit together. For more in-depth coverage of each service, see the other papers in this series (see Additional Resources).

# An Overview of the Azure Services Platform

The Azure™ Services Platform is poised to radically change the way Microsoft architects and developers think about building and managing applications. The Azure™ Services Platform (see Figure 1) provides an Internet-based cloud computing environment for running applications and storing data in Microsoft data centers around the world. In many ways, you can think of it as Windows® in the cloud.



Figure : Azure Services Platform

The Azure™ Services Platform consists of the Windows®Azure™ cloud-based operating system, as the foundation, and several layered building block service offerings as illustrated in Figure 1. You can take advantage of this new Microsoft cloud computing platform to host entirely new applications or individual services that enhance existing on-premises software investments. The choice is entirely yours.

## Windows® Azure™

Windows®Azure™ provides a cloud computing fabric, hosted within Microsoft data centers, for creating, deploying, managing, and distributing (scaling) applications and services on the Internet. Windows®Azure™ provides two main areas of functionality: computation (e.g., executing an application) and storage (e.g., storing data on disk). The value is in how Windows® Azure™ provides these foundational capabilities theoretical without limits. Scale-out is simply a matter of configuration. From a business perspective, Windows® Azure™ shields you from many of the costly IT complexities related to provisioning, configuring, and managing physical servers and the software running on them.

It’s important to note that the Windows® Azure™ storage services are designed to be very simple and highly scalable. It provides fundamental services for BLOB storage, queue storage, and simple table storage, but it doesn’t provide the capabilities of a relational database (e.g., query, search, reporting, or analytics). If you need those richer relational capabilities, you’ll want to turn to Microsoft® SQL Services.

As shown in Figure 1, there are several service offerings that run on the Windows® Azure™ foundation including Microsoft® .NET Services, Live Services, Microsoft® SQL Services, and others. While the focus of this whitepaper is specifically Microsoft® .NET Services, it is helpful to describe each of these service offerings in a little more detail and how they all fit together within the Azure™ Services Platform.

## Microsoft® .NET Services

Microsoft® .NET Services provides a set of *.NET developer-oriented* *services* and a software development kit (SDK) for building .NET applications to run in the cloud. Today it provides functionality related primarily to application connectivity, access control, and workflow hosting.[[1]](#footnote-1) Today, the three services it provides include the Microsoft® .NET Service Bus, the Microsoft® .NET Access Control Service, and the Microsoft® .NET Workflow Service. In some ways, you can think of Microsoft® .NET Services as the new .NET framework for building cloud applications, but it’s an entirely service-based development fabric.

One of the reasons it’s called Microsoft® *.NET* Services is because it has been designed and optimized to provide a first-class *.NET* developer experience. The Microsoft® .NET Services SDK makes working with these cloud-based services feel just like writing any other .NET application. The SDK provides integration with Windows Communication Foundation (WCF) and Windows Workflow Foundation (WF), which allows .NET developers to build on their existing skills in those key areas. In the end, Microsoft® .NET Services provides a .NET-centric development experience when building applications for the cloud.

Although Microsoft® .NET Services has been designed to provide a first-class .NET developer experience, it’s important to note that it’s based on industry standard protocols, making it possible for any service platform to integrate with it through standard REST, SOAP, and WS-\* techniques. As an example of that, there are already Java and Ruby SDKs for Microsoft® .NET Services available for download today.

## Microsoft® SQL Services

Microsoft® SQL Services provides a set of *data-oriented services* designed to extend the capabilities SQL Server into the cloud as part of the Azure™ Services Platform. Microsoft® SQL Services is actually the brand name for the family of SQL-related services. The first service included within this brand is called Microsoft® SQL Data Services (SDS), which offers full relational database capabilities as a service offering within Azure™ Services Platform. More data-centric services are likely to come in the future.

SDS gives you all the features of a relational database but as a service running in the cloud. This includes tables, stored procedures, triggers, views, indexes, and compatibility with Visual Studio .NET, ADO.NET, and ODBC. Developers will be able to provision logical servers and database instances in the cloud and begin working with them using the same tools and technologies they use today. This is possible because Micrososft® SQL Services supports the Tabular Data Stream (TDS) protocol, the same protocol used by SQL Server running on-premise. Hence, developers can use any TDS-compatible tool or technology when working with their Microsoft® SQL Services instances running in the cloud. In the end, most developers will just need to update their connection strings to point to their Microsoft® SQL Services databases.Live Services

Live Services provides a set of *user-centric services* focused primarily on social applications and experiences, along with a programming framework that makes them easy to program against. More specifically, the Live Services brand consists of Mesh Services, Identity Services, Directory Services, User-Data Storage Services, Communication and Presence Services, Search Services, and Geospatial Services.

The Live Framework provides a standards-based development framework for interacting with all of the Live Services through a consistent protocol/interface. The Live Framework embraces REST, Atom, and AtomPub, thereby making it possible for anyone to integrate with Live Services via common HTTP/XML programming techniques. It also comes with a friendly client SDK for .NET developers and a rich client runtime that provides built-in mesh synchronization capabilities as well as online/offline support.

Ultimately, Live Services makes it possible for you to build rich *mash-up* applications that leverage the data found within the Windows Live platform, actively used by more than 400 million people today, in ways that allow rich data synchronization across devices, applications, and business partners.

## Additional Service Offerings

In addition to these core service offerings (.NET, SQL, and Live), Microsoft is also actively building some domain-specific service offerings. One such offering is Microsoft® SharePoint Services, which will provide a set of SharePoint building block services that you can incorporate into your own applications. Another example is Microsoft® Dynamics CRM Services, a set of Microsoft® Dynamics CRM building block services hosted in the cloud. In both cases, developers will be able to continue writing code in Visual Studio and have the resulting logic hosted on the Azure™ Services Platform. Both offerings are still a work in progress so stay tuned for the initial CTP release of each technology.

It’s important to understand that Microsoft® SharePoint Services and Microsoft® Dynamics CRM Services are *service* offerings; they are not end-user applications. They provide capabilities in each of their respective areas that developers can use to incorporate those features into their applications.

## Building on the Azure™ Services Platform

Ultimately, the Azure™ Services Platform is a *platform* for developers to build applications on. One of the most practical aspects of the Azure™ Services Platform is that it builds on the skills Microsoft developers already have. Developers writing code for the Azure™ Services Platform can write traditional .NET code using Visual Studio, and they can leverage their experience with Windows Communication Foundation, Windows Workflow Foundation, Windows SharePoint, Windows Live, and SQL Server.

If you’re a Microsoft .NET developer today, the Azure™ developer experience should feel very familiar to you. The differences revolve around how you deploy, host, scale, and manage your new cloud-based software investments. The fact that the Azure™ Services Platform successfully builds on the .NET ecosystem already in place gives Microsoft a strong competitive advantage in cloud computing.

If you’re building a comprehensive cloud application targeting the Azure™ Services Platform, it’s likely that you’ll need to leverage many, if not all, of the service offerings described above. You might use Microsoft® .NET Services for connectivity and security, Microsoft® SQL Services for data storage and retrieval, and Live Services for mesh-based synchronization features. There is great value in how these different service offerings can be composed with one another into complete applications. See Figure 2 for a detailed view of the Azure™ Services Platform and the service offerings we’ve covered.



Figure : Detailed View of the Azure Services Platform

Microsoft itself has been busy building end-user applications that take advantage of these services on the Azure™ Services Platform. These applications include Windows Live, Microsoft® Office Live, Microsoft® Exchange Online, Microsoft® SharePoint Online, and Microsoft® Dynamics CRM Online (refer back to Figure 1) . These are all examples of *end-user applications* designed for immediate use.

As you can see, Microsoft is moving towards cloud computing with a comprehensive strategy that will make it easy for .NET developers to participate in this new era of software development.

# Introducing Microsoft® .NET Services

As a Microsoft .NET developer moving towards the Azure™ Services Platform, you’ll want to spend some time becoming familiar with the Microsoft® .NET Services offering, which now I’ll simply refer to as “.NET Services” from now on. The reason is simple: .NET Services provides the key building blocks you’ll need when building cloud-based and cloud-aware applications for the Azure™ Services Platform. Throughout the rest of this whitepaper we’ll be drilling into the .NET Services box shown in Figure 2.

## Overview of .NET Services

The services found within the .NET Services brand provide valuable cloud-specific *infrastructure* that will ultimately make your job easier when building cloud applications.

Today .NET Services provides core functionality related to application connectivity, access control, and workflow-based message orchestration. It will evolve to provide even more cloud-based features and infrastructure over time. Today, the .NET Services brand includes the following building block services:

* Microsoft® .NET Service Bus: provides network infrastructure for connecting applications over the Internet, using a variety of different messaging patterns, in a way that’s capable of traversing firewalls and NAT devices without forfeiting the security afforded by these devices.
* Microsoft® .NET Access Control Service: provides claims-based access control in the cloud. It includes a claims transformation engine that federates with identity providers like Active Directory and Windows Live ID (WLID).
* Microsoft® .NET Workflow Services: provides infrastructure for hosting and managing WF workflows focused primarily on messaging orchestration through the .NET Service Bus. It comes with new WF activities and tools for hosting and managing workflow instances.

You can think of these new services as your cloud-centric .NET service framework. Each of these services is available using open protocols and standards, including things like REST, SOAP, Atom/AtomPub, and WS-\*, which means developers on any platform can integrate with these services.

However, in an effort to make things as natural as possible for .NET developers, Microsoft has also provided a *.NET Services SDK* that provides a first-class .NET developer experience that hides many of the complexities that you would otherwise experience when working with them directly.

By using the .NET Services SDK, developers are able to build on their existing .NET experience, specifically in the areas of WCF and WF, by taking advantage of the new framework extensions found in the SDK (e.g., new bindings, channels, and activities). The SDK also includes Visual Studio tool support for integrating with the Azure™ Services Portal. In addition to the .NET Services SDK, you can find Java and Ruby SDKs available from Microsoft partners today (see Additional Resources for links).

## Getting Started with .NET Services

To get started with .NET Services, first browse to the Azure™ Services Platform portal at <http://azure.com> and press the “Try It Now” link. This will take you to the “Register for Azure Services” page shown in Figure 3. This page provides important links for downloading the various SDKs, accessing additional resources, and browsing to Microsoft Connect where you can register for an invitation code.

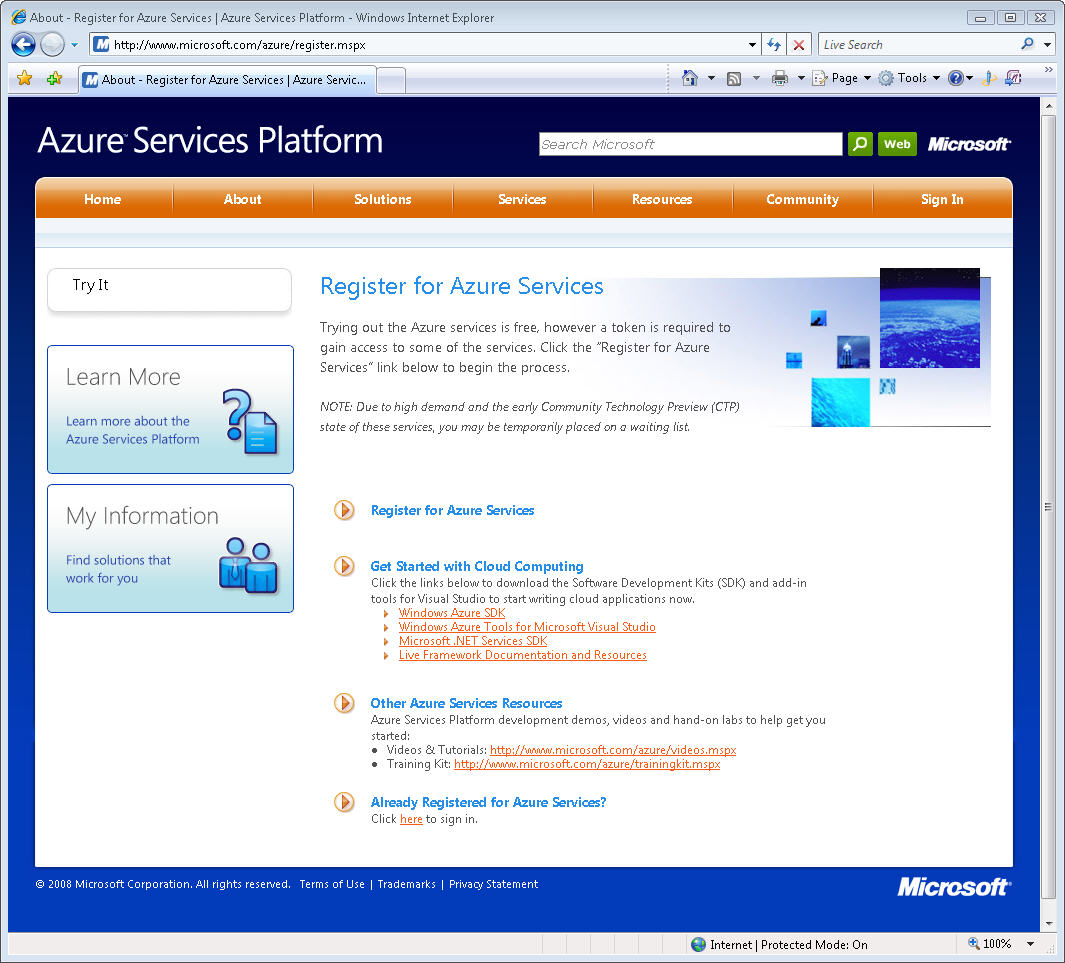


Figure : The Azure™ Services Platform Portal

Next you’ll want to download the .NET Services SDK. Notice that there are a few SDKs specifically for Windows® Azure™ development, another for .NET Services development, and others for SQL Data Services and the Live Framework. For the purposes of this whitepaper series, you only need to download and install the .NET Services SDK to replicate the various samples we’ll be showing you.

Once you’ve downloaded the .NET Services SDK, simply run the setup program as illustrated in Figure 4, and then you’ll have the new .NET assemblies you’ll need along with some Visual Studio add-ins that make it easy to begin taking advantage of the various .NET Services features. As you begin your experience with .NET Services, be sure to check out the other resources that you can browse to from this page (demos, videos, hands-on labs, etc), designed to enrich your learning experience. You can download the SDK without an account but to actually use the services, you’ll need to register.

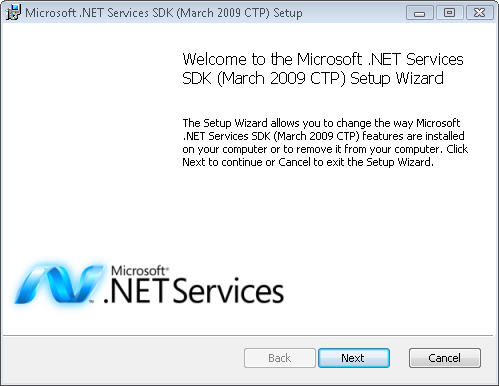


Figure : Running the .NET Services SDK Setup

In order to register for an Azure Services account, click on the “Register for Services” link shown above. This will require you to login with a Windows Live ID (WLID) and then it will take you to the Microsoft Connect site where you’ll need to fill out an Azure Services CTP registration form. Once you’ve successfully registered for the Azure Services CTP, you’ll see the page shown in Figure 5.

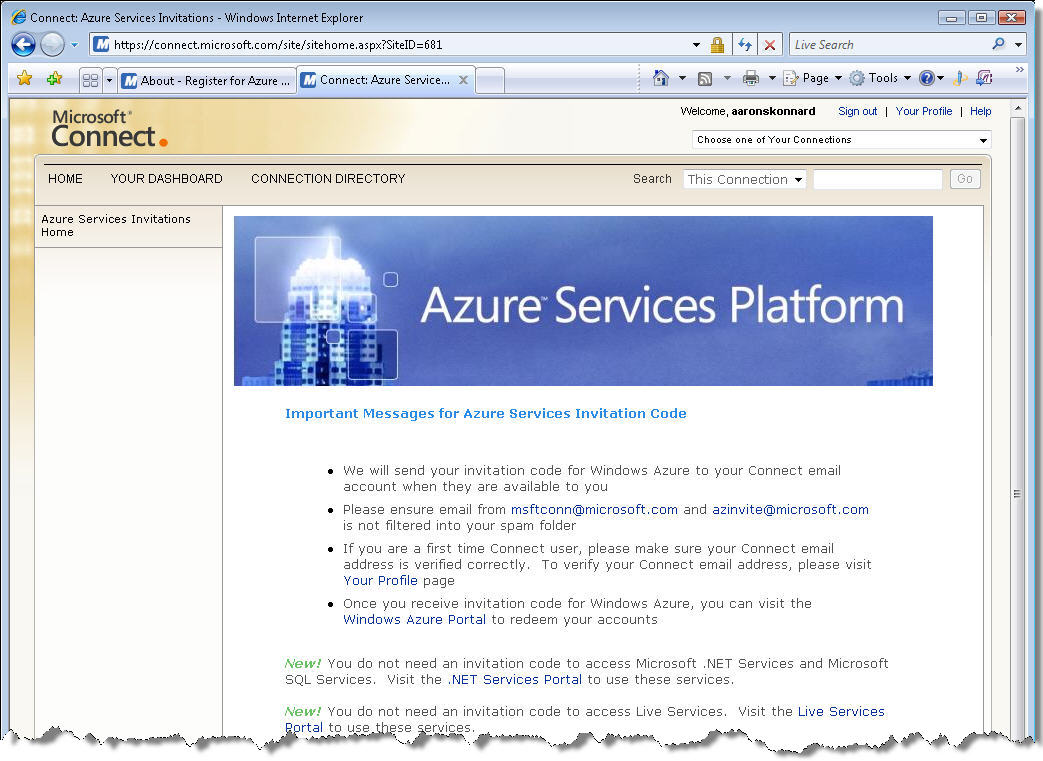


Figure 5: Registering for the Azure Services Platform on Microsoft Connect

At this point, you can return to the .NET Services sign-up page (click on the “.NET Services Portal” link shown in Figure 5) and you’ll be taken to the page shown in Figure 6.

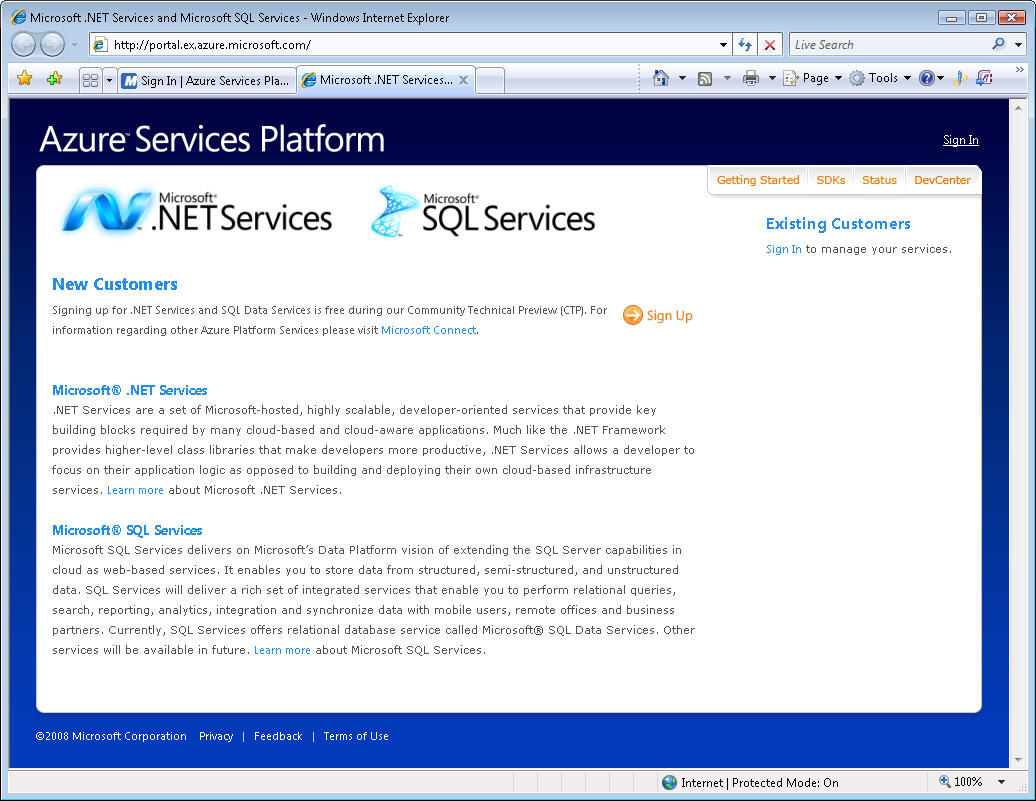


Figure 6: .NET Services Sign-up Page

Now click on “Sign Up” and then you’ll be able to create a .NET Services solution. Note: as of the March 2009 CTP release, you no longer need an invitation code in order to create a .NET Services solution.

## Creating your First Solution

shows the “create solution” page. In order to create a solution, simply enter a unique solution name[[2]](#footnote-2), accept the terms of use, and press “Create Solution”. Then your new .NET Services solution will be provisioned and associated with your WLID. In the future, whenever you sign into the Azure™ Services Platform portal, you’ll be able to manage all of the solutions associated with your WLID.

After your new solution has been created, you’ll see a page like the one shown in Figure 7 that provides you with a solution password that you’ll want to record for future use. Your solution name and password serve as credentials for accessing the various .NET Services.

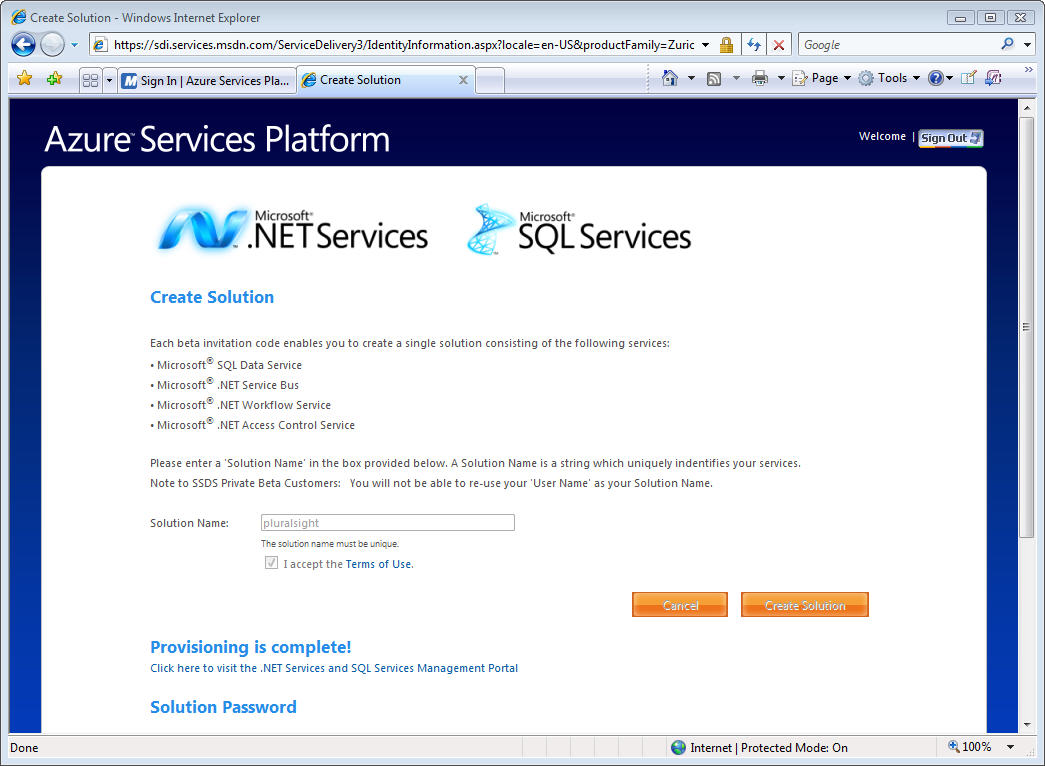


Figure : Completing the Solution Provisioning Process

Once you’ve successfully created a solution, you can begin managing and using your solution from within the Azure™ Services Platform portal. You’ll see a “My Solutions” menu on the right side of the portal after logging in with your WLID (see Figure 8). You can manage a specific solution by simply selecting it from the My Solutions menu, and then you’ll see the page shown in Figure 9.

A *solution* is basically a top-level container for managing your various .NET Services assets.[[3]](#footnote-3) For example, it’s a container for your .NET Service Bus endpoints, your .NET Workflow Service types and instances, and your .NET Access Control Service identities and claims transformation rules. One of the most important things you’ll want to manage after creating your solution are the solution credentials.

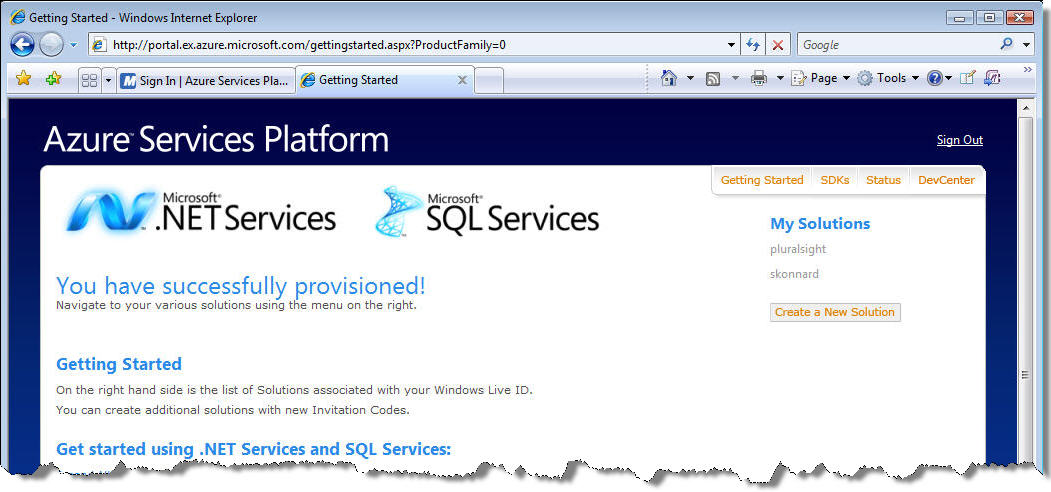


Figure : Managing your Solutions via “My Solutions”

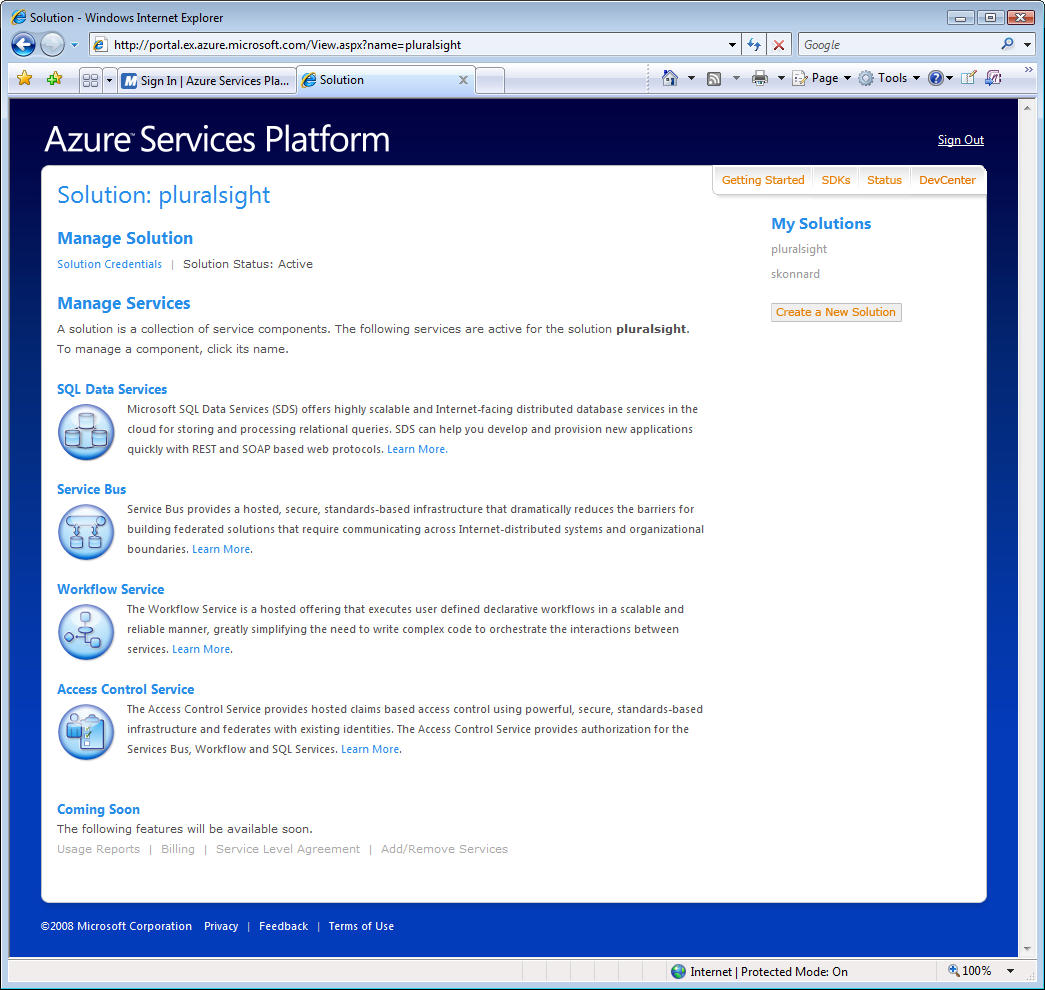


Figure : Managing an Individual Solution

## Managing your Solution Credentials

You can change the solution password provided to you during the provisioning process from the “Credential Management” page (simply click on “Solution Credentials” in Figure 9). From this page, you can also configure the Windows CardSpace information cards associated with your solution, and any certificates you want to associate with your solution as well (see Figure 10).

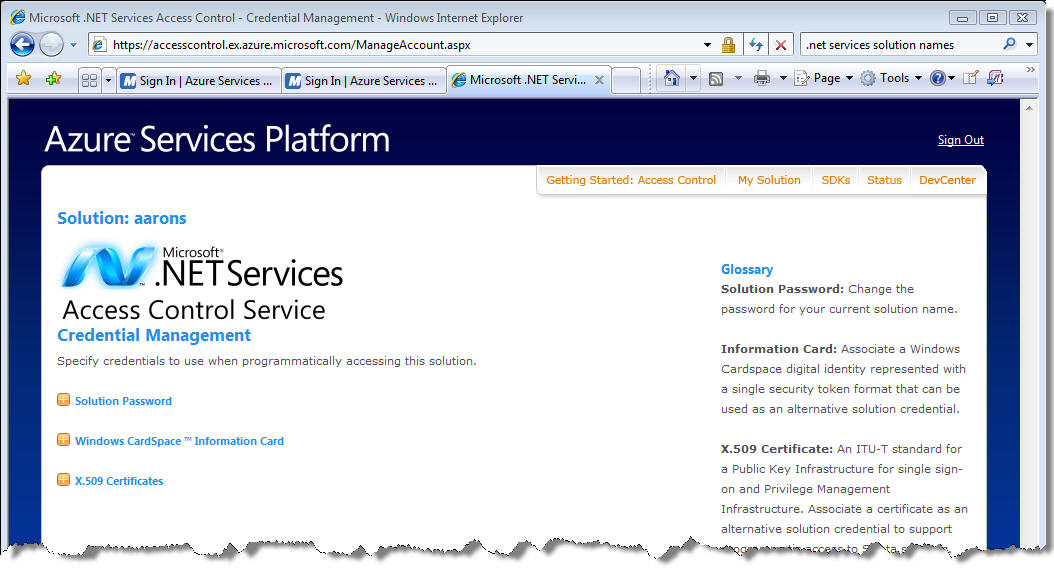


Figure : Managing your Solution Credentials

In the case of Windows CardSpace and certificates, this page will ask you to select the card/certificate you’d like to associate, and then it will upload the information to your solution account. From that point on, you’ll be able to use the specified card/certificate credentials in conjunction with your account.

## Samples in the .NET Services SDK

The .NET Services SDK comes with a number of interesting samples that illustrate how to leverage the various capabilities offered by .NET Services. Once you’ve installed the SDK, you’ll find the samples in the following directory: C:\Program Files\Microsoft .NET Services SDK (March 2009 CTP)\Samples.

All of these samples have been designed to ask the user to specify the solution credentials dynamically. Some of the samples will ask you to specify your solution name and password while others will ask you to choose a Windows CardSpace information card. In the latter case, you’ll need to specify an information card that you’ve previously associated with your solution account.

Now that you know how to get started with the .NET Services SDK and how to provision a .NET Services solution account, we’re ready to dig into each functional area of the .NET Services in more detail.

# Microsoft® .NET Service Bus

One of the most common needs in large-scale distributed applications is application connectivity. In fact, application integration is usually one of the most costly and troublesome areas of IT. Today it’s common for many organizations to use an enterprise service bus (ESB) solution to address these challenges.

The .NET Service Bus is a core part of the .NET Services offering focused on making the ESB pattern a reality at Internet scope as part of the Azure™ Services Platform. The .NET Service Bus provides many of the same architectural characteristics found in typical ESB solutions, including things like identity and access control, naming, a service registry, and a common messaging fabric. The primary difference is one of scope. In the case of the .NET Service Bus, the components must be designed to operate in the cloud, at a global Internet scope, and in a highly scalable and federated manner. This is precisely why Microsoft has referred to this particular service offering as the *Internet Service Bus* in the past (see Figure 11).[[4]](#footnote-4)

An Internet Service Bus would make it possible to integrate your on-premises ESB product with your own services running in the cloud, with a variety of 3rd party services provided by Microsoft or other vendors (such as those offered within the Azure™ Service Platform), and with a variety of desktop, RIA[[5]](#footnote-5), and Web applications that may be running in satellite locations outside of the corporate firewall.

In order to make this possible, the implementation must provide federated solutions based on open Internet standards and a rich messaging fabric capable of bidirectional communication at Internet scope.

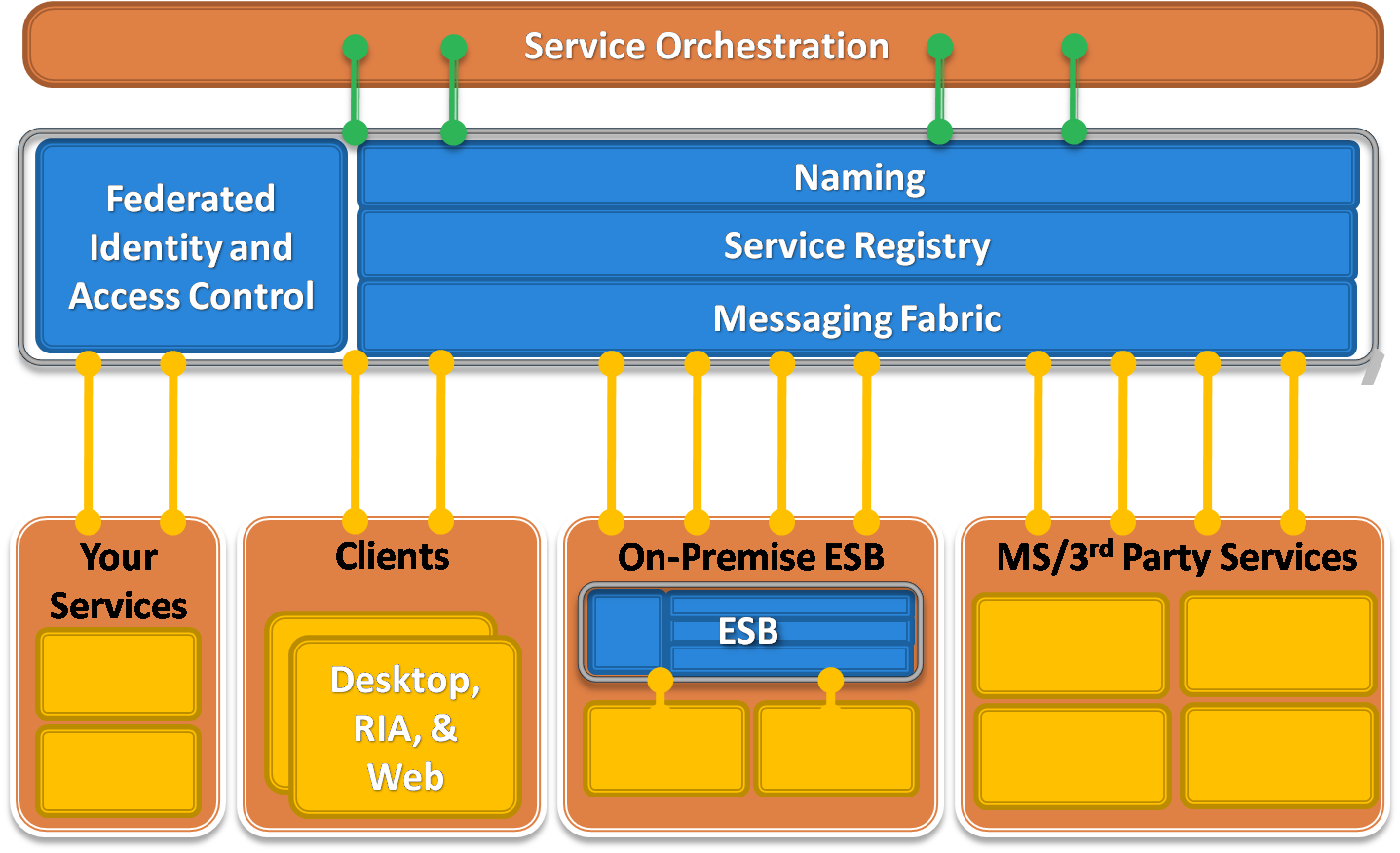


Figure : The Internet Service Bus

Tackling bidirectional communication at Internet scope is not trivial due to some of today’s networking realities. This is primarily due to the network barriers introduced by firewall and NAT devices, which make it difficult to communicate with nodes sitting behind such layers. Imagine a situation where a sales person is traveling and she’s using your application on a wireless network in random hotel somewhere in the world. How would you locate and initiate communication with her device in that scenario?

Companies often deal with these connectivity challenges by opening inbound firewall ports (much to their system administrator’s dismay) or by using different workarounds like dynamic DNS, NAT port mappings, or UPnP, all of which are brittle, difficult to manage, and susceptible to security threats. As more and more applications are requiring this type of bidirectional communication, we’re experiencing a growing tension here, and this tension is precisely what the .NET Service Bus aims to reduce.

## Relayed Connectivity

Despite these connectivity challenges, some of today’s most popular Internet applications are inherently bidirectional. Consider things like instant messaging, online multiplayer games, and peer-to-peer file sharing applications that use protocols like BitTorrent, which accounts for a large percentage of all Internet traffic today. These applications have written the low-level networking logic to traverse firewalls and NAT devices, and to create direct peer-to-peer connections when possible. They typically accomplish this through a central *relay service* that provides the network traversal logic (see Figure 12).

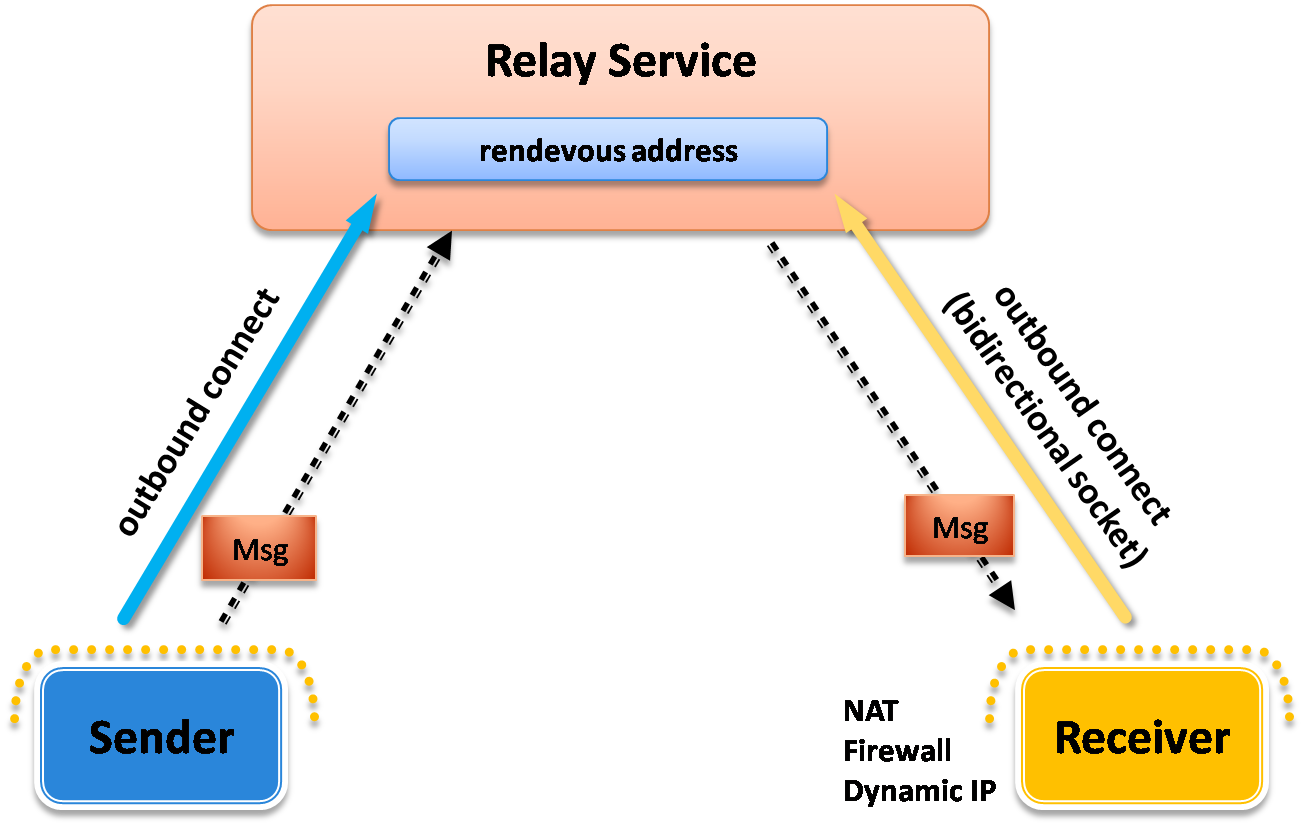


Figure : Using the Relay Service

The relay makes it possible for a sender to communicate with a receiver through a shared *rendezvous address*. The receiver connects to the relay through an outbound port and creates a bidirectional socket for communication, specifying the rendezvous address it wants to “listen” on. The sender can then “send” a message to the relay using the same rendezvous address, at which point the relay can transmit the message to the receiver registered on the same address; the message is transmitted through the bidirectional socket already in place. The receiver doesn’t need to have any inbound ports open on the firewall to make this work. This type of relay service sits at the heart of the .NET Service Bus.

When using the .NET Service Bus relay, you can take advantage of one-way messaging, request/response, publish/subscribe (multicast), and even asynchronous/disconnected messaging.

For one-way messaging, a single receiver registers with the relay to “listen” for messages on a particular rendezvous address within the .NET Service Bus. Senders can then “send” messages to the .NET Service Bus address to have the messages “relayed” to the registered receiver. Using one-way communication through the relay offers a more aggressive network traversal mode because it’s then possible for both the sender and the receiver to use HTTP (e.g., the receiver can poll for messages from the relay via HTTP). Request/response messaging is very similar, only you’ll typically use TCP on the receiver end.

The relay service makes it easy to implement publish/subscribe architectures by allowing multiple receivers to register on the same .NET Service Bus rendezvous address. In that case, when a sender transmits a message to that rendezvous address, the relay service will distribute the transmitted message to all currently registered receivers, thereby providing multicast capabilities through the relay.

Both senders and receivers can communicate with the relay using either TCP or HTTP. You typically use the former when you care more about performance and throughput and the latter when you care more about interoperability or when you need to ensure you’re able to communicate through firewalls.

## Direct Connectivity

In addition to relayed communications, the .NET Service Bus also provides a capability for establishing direct connectivity between senders and receivers in order to improve performance and throughput. Senders and receivers still communicate with the relay through a common rendezvous address but then it tries to help them connect directly to one another in order to avoid future relayed transmissions.

The way the relay accomplishes this is through a mutual port predication algorithm based on probing information from the sender and receiver. The relay service looks at this probing information and does its best to predict what ports are going to be open on their respective NAT devices. It can then provide that information to the sender/receiver so that they can attempt to establish a direct connection with one another. If the relay service predicts correctly, the connection will succeed, otherwise it can try again until it decides to give up and to stick with the relayed connection (see Figure 13).

This approach is similar to the approach used by many of today’s instant messaging applications when transferring files between users. Next time you use that feature, pay attention to the initial file transfer speed and whether or not it significantly speeds up at some point during the process. If you notice a significant boost in transfer speed, you just witnessed the upgrade to a direct connection.

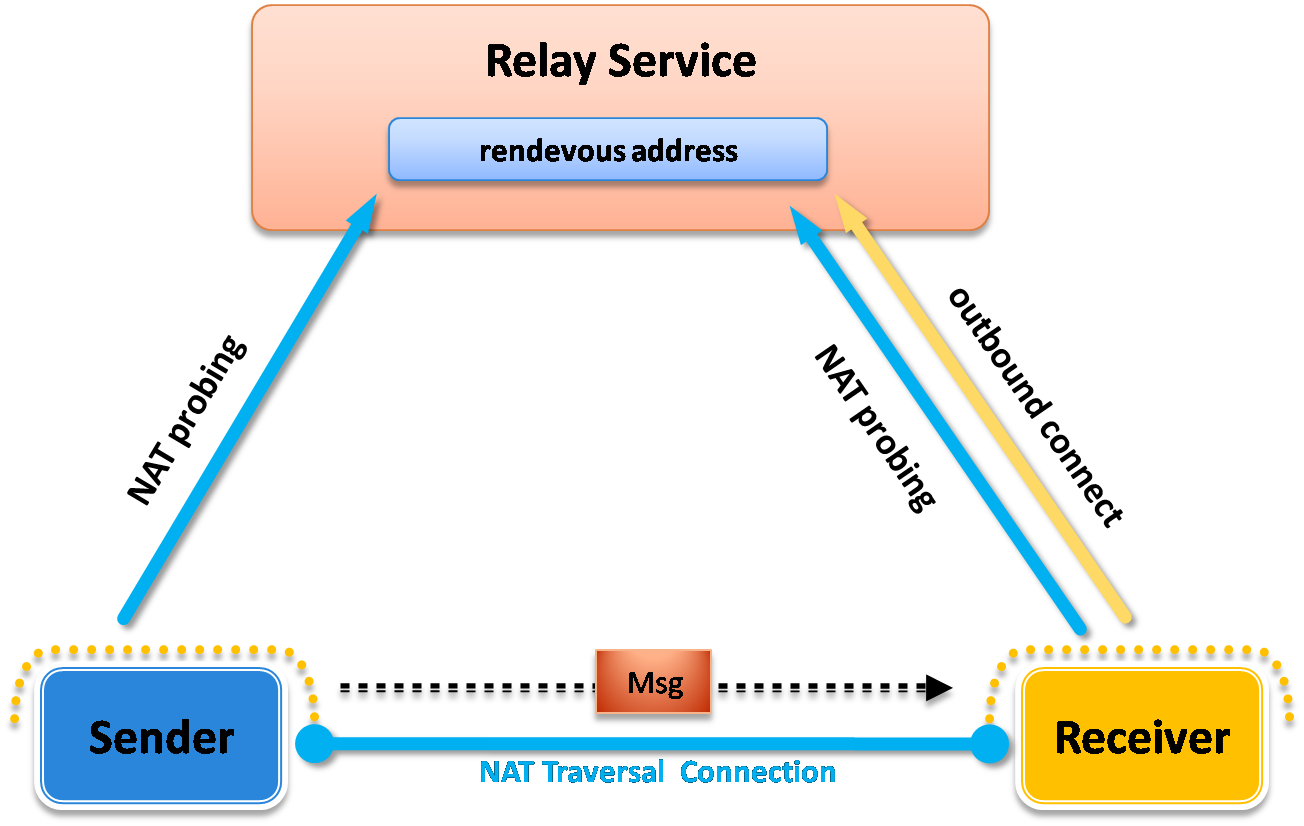


Figure : Establishing a Direct Connection

## Relay Addresses and Access Control

One of the most important things to understand when using the .NET Service Bus is how to structure the rendezvous addresses for your .NET Service Bus endpoints. When you expose a TCP-based endpoint, you’ll need to structure your addresses as follows:

sb://{solution}.servicebus.windows.net/{user-defined}

When you expose an HTTP based address, simply replace the “sb” protocol scheme with “http”:

http://{solution}.servicebus.windows.net/{user-defined}

Notice that you specify your solution name as part of the address in order to distinguish your endpoints from those used by other solutions throughout the .NET Service Bus. After the solution name, you have complete control to specify whatever URI name hierarchy you’d like to use.

If you’re logged into the Azure™ Services Platform portal, you can browse to the solution management page (see Figure 9) for one of your solutions and then browse to the Service Bus management section. This page provides documentation showing you how to structure the addresses for your solution.

The relay provides access control capabilities through a claims-based security model and a trust relationship with the .NET Access Control Service. The relay looks for a security token issued by the .NET Access Control Service to determine whether a particular sender or receiver should be allowed to “send” or “listen” on a particular .NET Service Bus rendezvous address.

Before a receiver can listen on a particular address, it must acquire a security token from the .NET Access Control Service containing the “#Listen” claim. It will be required to provide credentials to the .NET Access Control Service in order to acquire such a token. Senders must also acquire a security token from the .NET Access Control Service containing a “#Send” claim.

Senders and receivers will need to supply credentials to the .NET Access Control Service in order to acquire a security token for the .NET Service Bus relay. The .NET Access Control Service supports different types of credentials as we’ll discuss shortly. Since the relay has a trust relationship with the .NET Access Control Service, it’s able to read the security tokens that it issues and process the claims.

## Service Registry

The .NET Service Bus provides a service registry for publishing and discovering service endpoint references within a solution. It allows you to publish endpoint references as either simple URI’s or as official WS-Addressing endpoint references. Others can then discover a solution’s endpoint references by browsing to the solution’s base address and retrieving an Atom feed that contains the information.[[6]](#footnote-6)

There are two ways to publish endpoint references into the service registry. The most common approach is to let WCF take care of it for you when you register listeners with the relay service. The relay service automatically populates the service registry with endpoint references as you create listeners through the WCF bindings (and corresponding channels) that come with the .NET Services SDK. You can also manually publish endpoint references into the service registry through an AtomPub interface.

## Queues and Routers

The .NET Service Bus acts primarily as a transient relay between two parties. This core function always requires listeners to be actively listening on the .NET Service Bus before senders can begin sending messages to them. But what if senders and receivers can’t always be running at the same time? Or what if they can’t process messages at the same speed due to bandwidth or other resource constraints?

One of the most commonly requested features during the prior .NET Service Bus CTP’s has been for a durable queuing mechanism that would allow developers to implement asynchronous or disconnected communication patterns through the .NET Service Bus messaging fabric.

In order to make this possible, the March 2009 CTP release introduces two new messaging primitives – *queues* and *routers* – that make it possible to support a variety of additional communication requirements commonly required by distributed applications today. These new messaging primitives represent a shift in the way Microsoft thinks about the .NET Services Bus and the road ahead. They are long-lived (durable) and can exist independent of traditional .NET Service Bus listeners. In other words queues/routers can exist even when no active listeners are present.

Queues, as their name suggests, provide a durable data structure with FIFO semantics. You can create a queue within your .NET Service Bus namespace just like you can create listeners. You simply pick a name for the queue within your .NET Service Bus solution namespace, define a *queue policy*, and issue a request to the .NET Service Bus to create the queue at that specific name.

Once created, senders can send messages to the queue and receivers can read messages from the queue, but the key is that senders and receivers don’t have to be operating at the same time. Senders can send messages to queues using either HTTP(S) or net.tcp but receivers can only read messages from queues using HTTP(S). Also, queues support the notion of *peeks with locks*, which makes it possible to ensure that messages aren’t lost upon application failure. A few general limitations to be aware of: queues only support messages up to 64 KB in size and they do not support streaming.

Routers, on the other hand, are responsible for routing messages to one or more subscribers. You create routers just like you create queues. You pick a name for the router within your .NET Service Bus solution, define a *router policy*, and issue a request to create the router at that name. One thing that the router policy specifies is how distribution should occur – routers can distribute to all subscribers (multicast) or to a single subscriber using a round robin load-balancing algorithm. Once created, one or more receivers (including queues) can create *subscriptions* with the router to receive messages from them.

Once you’ve created queues and routers within a .NET Service Bus solution, they will begin to appear within the solution’s service registry, assuming you’ve configured the queue/router policy to allow for public discoverability. This allows consumers to find these constructs dynamically at runtime.

One of the most interesting things about queues and routers is how they compose with one another. Queues can subscribe to routers, just like services can. This provides long-lived durable storage for the routed messages. Once messages are routed to a queue, they can sit there until a consuming application comes to retrieve them, which could be long after the sender goes away. Figure 14 shows an example of how queues and routers can be composed to implement these different messaging scenarios.

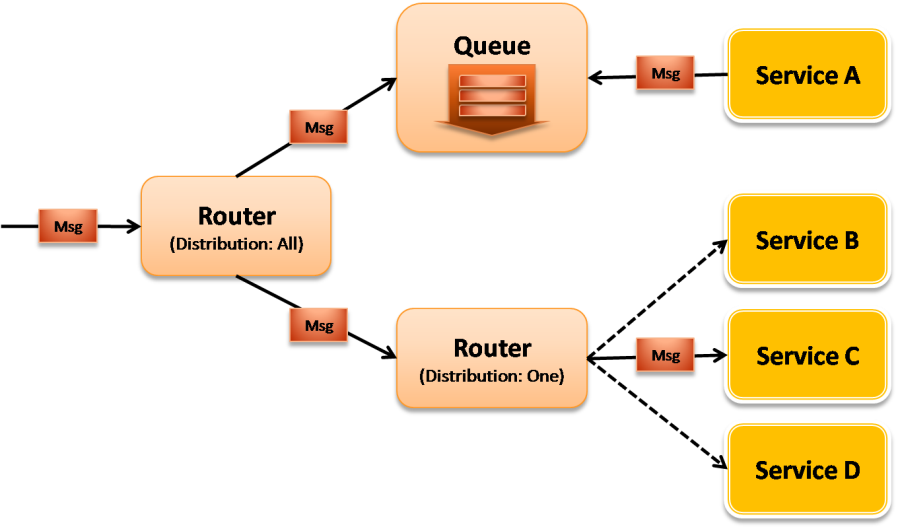


Figure 14: Composing Queues and Routers

In this example, the front-end router has a router distribution policy of ‘All’, which means it will multicast incoming messages to all subscribers. This router has two subscribers: a queue and another router. The queue simply provides durable backing for the incoming messages until a service (Service A) periodically comes along to pick them up directly from the queue. Notice how the secondary router is configured with a router distribution policy of ‘One’, which means it will load balance incoming messages across all subscribes (Service B, C, and D) one at a time. It could also easily multicast the messages to all subscribers by simply changing the distribution policy to ‘All’.

## Integration with WCF

The primary programming model for working with the .NET Service Bus on the .NET platform is WCF. The .NET Services SDK comes with a set of new WCF bindings that automate the integration between your WCF services and clients with the relay service. In most cases, all you need to do is replace the current WCF binding that you’re using with one of the .NET Service Bus bindings.

Figure 15 lists all of the .NET Service Bus WCF bindings and the standard WCF bindings they correspond to. The most commonly used WCF bindings, such as BasicHttpBinding, WebHttpBinding, WSHttpBinding, and NetTcpBinding, all have a corresponding .NET Service Bus binding with a very similar name (just insert “Relay” before “Binding”). There are only a few relay-specific bindings – NetOneWayRelayBinding and NetEventRelayBinding – that don’t have a corresponding binding in WCF.

|  |  |
| --- | --- |
| Standard WCF Binding | Equivalent Relay Binding |
| BasicHttpBinding | BasicHttpRelayBinding |
| WebHttpBinding | WebHttpRelayBinding |
| WSHttpBinding | WSHttpRelayBinding |
| WS2007HttpBinding | WS2007HttpRelayBinding |
| WSHttpContextBinding | WSHttpRelayContextBinding |
| WS2007HttpFederationBinding | WS2007HttpRelayFederationBinding |
| NetTcpBinding | NetTcpRelayBinding |
| NetTcpContextBinding | NetTcpRelayContextBinding |
| N/A | NetOnewayRelayBinding |
| N/A | NetEventRelayBinding |

Figure : WCF Relay Bindings

In order to use the .NET Service Bus, you specify the appropriate “relay” binding (that provides the connectivity and messaging semantics you’re after) and a relay address when defining the WCF endpoint. Then when you open the ServiceHost (for a receiver) or a ChannelFactory (for a client), the WCF infrastructure takes care of communicating with the relay behind the scenes as describe above.

Despite this WCF integration, it’s important to remember the .NET Service Bus is based on open Internet standards, making it possible to connect applications across a variety of platforms. And since it’s built on the Azure™ Services Platform, it provides theoretically unlimited scale-out possibilities as well.

## A Simple .NET Service Bus Example

Let me show you a quick example of how you can configure a simple WCF application to take advantage of the .NET Service Bus. We’ll start with the following simple WCF service contract and implementation:

[ServiceContract]

public interface IHelloServiceBus

{

[OperationContract]

string SayHello(string name);

}

class HelloServiceBus : IHelloServiceBus

{

public string SayHello(string name)

{

string greeting = string.Format("Hello {0}!", name);

Console.WriteLine("Returning: {0}", greeting);

return greeting;

}

}

We’ll host this service in the following console application that reads the service configuration details from the application configuration file:

class Program

{

static void Main(string[] args)

{

Console.WriteLine("\*\*\*\* Receiver \*\*\*\*");

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

host.Open();

Console.WriteLine("Press [Enter] to exit");

Console.ReadLine();

host.Close();

}

}

And we’ll start by using the following endpoint definition in the application configuration file. Notice how the endpoint uses NetTcpBinding and a local address of net.tcp://localhost:8080/helloservicebus:

<configuration>

<system.serviceModel>

<services>

<service name="Microsoft.ServiceBus.Samples.HelloServiceBus">

<endpoint address="net.tcp://localhost:8080/helloservicebus"

binding="netTcpBinding"

contract="Microsoft.ServiceBus.Samples.IHelloServiceBus" />

</service>

</services>

</system.serviceModel>

</configuration>

Next, we can write a client application that invokes the service. The following code shows how to do this using the same IHelloServiceBus contract definition. It also assumes that the endpoint details (for “RelayEndpoint”) will be read from the client’s application configuration file:

class Program

{

static void Main(string[] args)

{

Console.WriteLine("\*\*\*\* Sender \*\*\*\*");

Console.WriteLine("Press <Enter> to start sending messages.");

Console.ReadLine();

ChannelFactory<IHelloServiceBus> channelFactory =

new ChannelFactory<IHelloServiceBus>("RelayEndpoint");

IHelloServiceBus channel = channelFactory.CreateChannel();

string response = channel.SayHello(".NET Service Bus");

Console.WriteLine(response);

channelFactory.Close();

}

}

And finally, the client’s application configuration file needs to have the equivalent endpoint definition in order to communicate with the service via the TCP endpoint it exposes:

<configuration>

<system.serviceModel>

<client>

<endpoint address="net.tcp://localhost:8080/helloservicebus"

binding="netTcpBinding"

contract="Microsoft.ServiceBus.Samples.IHelloService"

name="RelayEndpoint" />

</client>

</system.serviceModel>

</configuration>

If you run the two applications, you’ll see “Hello .NET Service Bus!” displayed in both console windows. In this case there’s a direct TCP connection between the client and the service applications.

Now let’s look at what it takes to introduce the .NET Service Bus as a relay between the client and service applications. First we’ll need to reconfigure the service host to listen on the .NET Service Bus, and then we’ll need to reconfigure the client to send messages through the .NET Service Bus.

We can reconfigure the service to listen on the .NET Service Bus by simply changing the binding from NetTcpBinding to NetTcpRelayBinding. When doing so, we’ll also need to specify a valid .NET Service Bus rendevous address for the endpoint. Since I have a solution named “pluralsight", I can use an address of sb://pluralsight.servicebus.windows.net/helloservicebus.

I also need to provide credentials in order to prove to the relay service that I’m allowed to listen within that solution address space. These credentials will be presented to the .NET Access Control Service in order to receive a token for the .NET Service Bus. There are several ways you can do this but I’ll simply provide a username and password for this example. You can specify the username and password in code or configuration. I’m using the latter approach through the <transportClientEndpointBehavior> element. The following shows the complete configuration for the .NET Service Bus-enabled service:

<configuration>

<system.serviceModel>

<services>

<service name="Microsoft.ServiceBus.Samples.HelloServiceBus">

<endpoint address=

"sb://pluralsight.servicebus.windows.net/helloservicebus"

behaviorConfiguration="default"

binding="netTcpRelayBinding"

contract="Microsoft.ServiceBus.Samples.IHelloServiceBus" />

</service>

</services>

<behaviors>

<endpointBehaviors>

<behavior name="default">

<transportClientEndpointBehavior credentialType="UserNamePassword">

<clientCredentials>

<userNamePassword

userName="[solution-name]" password="[solution-password]" />

</clientCredentials>

</transportClientEndpointBehavior>

</behavior>

</endpointBehaviors>

</behaviors>

</system.serviceModel>

</configuration>

When the WCF service host opens with this configuration, it will first send the client credentials to the .NET Access Control Service and acquire a security token for listening on the .NET Service Bus. It will then establish a TCP connection with the relay service and present the security token it acquired. Assuming the service is allowed to listen on that address (meaning the token contains the necessary claim), the relay service will create a listener for relaying messages to our local WCF service. At this point if you browse to the base HTTP address for the solution, you’ll find this endpoint in the service registry.

Reconfiguring the client application is similar. First, we need to change the endpoint to use the NetTcpRelayBinding and the same rendezvous address that we configured our service to listen on. We also need to configure the client with credentials. As with the service, clients must also prove that they are allowed to send messages to a particular .NET Service Bus address by acquiring a token from the .NET Access Control Service. The following shows the complete client configuration:

<configuration>

<system.serviceModel>

<client>

<endpoint address=

"sb://pluralsight.servicebus.windows.net/helloservicebus"

binding="netTcpRelayBinding"

contract="Microsoft.ServiceBus.Samples.IHelloServiceBus"

behaviorConfiguration="default"

name="RelayEndpoint" />

</client>

<behaviors>

<endpointBehaviors>

<behavior name="default">

<transportClientEndpointBehavior credentialType="UserNamePassword">

<clientCredentials>

<userNamePassword

userName="[solution-name]" password="[solution-password]" />

</clientCredentials>

</transportClientEndpointBehavior>

</behavior>

</endpointBehaviors>

</behaviors>

</system.serviceModel>

</configuration>

With these changes in place, we can run the service host application followed by the client application, and we’ll see the same result as before only this time the communication was relayed through the .NET Service Bus (see Figure 16), making it possible to traverse a variety of network obstacles.

|  |  |
| --- | --- |
| HelloSender.jpg | HelloReceiver.jpg |
|  |  |

Figure : HelloServiceBus Sample in Action

*For a deeper look at the .NET Service Bus and some additional code samples, see the Developer’s Guide to the Microsoft .NET Service Bus in this series (see Additional Resources).*

# Microsoft®.NET Access Control Service

The identity solution that Microsoft has been moving toward over the last few years is based on the idea of claims. A claims-based identity model allows the common features of authentication and authorization to be factored out of applications and centralized into external services written and maintained by subject matter experts in security and identity, which is beneficial to all involved.

## Claims-based Identity

The Microsoft® .NET Access Control Service is a cloud-based service that does exactly that. Instead of writing your own custom user account and role database, you can let the .NET Access Control Service orchestrate the authentication and authorization of your users. The .NET Access Control Service relies on existing user account stores such as Windows Live ID, Active Directory, as well as any other store that supports the standard federation protocols. This makes it natural to achieve single sign on across applications. It also centralizes authentication and access control logic, simplifying your applications.

When you build a claims-aware application, the user presents her identity to your application as a set of *claims*. One claim could be the user’s name; another might be her email address. These claims are supplied by an issuing authority that knows how to authenticate the user and where to find her attributes. The client application, which might be a browser or a rich client, transparently works with the authority to discover these claims and pass them along to your application (see Figure 18).



Figure : Using Claims-based Identity with Web Services

The end result is that your application receives all of the identity details it needs to know about the user in a set of claims. And the claims are signed to give you cryptographic assurance of their origin.

## Benefits of Single Sign-on

Under a claims-based identity model, single sign-on is easier to achieve, and your application is *no longer responsible* for any of the following security concerns:

* Authenticating users
* Storing user accounts and passwords
* Calling to enterprise directories to look up user identity details
* Integrating with identity systems from other platforms or companies

This type of model allows your application to make identity-related decisions based on claims supplied by the user. This could be anything from simple application personalization with the user’s first name, to authorizing the user to access higher valued features and resources in your application.

## Administration Portal

The .NET Access Control Service brings claims-based identity to life within the Azure™ Services Platform. A big part of the .NET Access Control Service is its administration system.

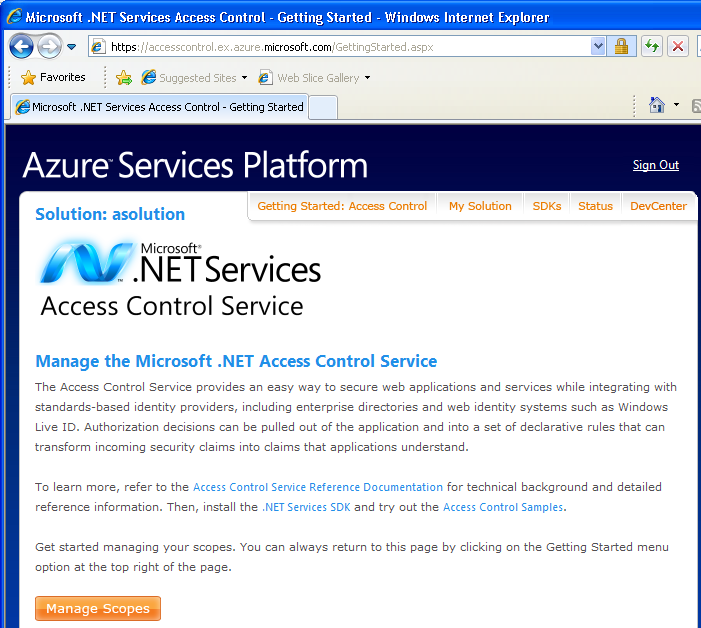


Figure : The ACS portal

The .NET Access Control Service provides an administration portal (see Figure 18) within the Azure™ Services Portal[[7]](#footnote-7). This is where you configure the rules that determine how it will issue claims for various users, and ultimately answer the question, “What can you do?”

The ACS portal is a great way to explore, learn, and get started with ACS. And for relatively simple applications, it might be the only tool you need. But for non-trivial systems with hundreds or thousands of users and possibly a similar number of rules, the portal becomes unwieldy and a programmatic interface is preferable. This is why ACS also exposes an AtomPub interface for programmatic administration. AtomPub is a RESTful protocol that standardizes basic CRUD (Create, Retrieve, Update, and Delete) operations for managing remote resources. This opens up a whole new realm of flexibility.

## A Simple .NET Access Control Service Example

The best way to get your head around ACS is to see it work. So in this section, I’ll walk you through one of the samples packaged with the .NET Services SDK, called the UserNamePasswordCalculatorService.

This sample uses ACS to secure a simple WCF service that offers four operations: Add, Subtract, Multiply, and Divide. The calculator service in this example doesn’t care about the user’s name, email address, or other personal information. All this service cares about is that the user making an Add request (for example) is allowed to add. I’ll show you how to configure a solution in ACS to control access to these operations. In order to get this sample working, you first need to have a “solution” in ACS. I’ll talk more about what a solution is later on, but for now think of it as your own personal security token issuer in the cloud. I will use my own personal solution as an example; its name is “asolution”.

The calculator service is an application that relies on claims issued by ACS to make security decisions. So this service, like any other claims-based application, needs to have its own X.509 certificate that ACS can use for encrypting the security tokens that the service will receive. The sample comes with a simple test certificate called localhost.cer, and that’s what I’ll use to get started.

The Utils subdirectory of this example contains the certificate, along with a batch file (installcerts.bat) that installs it into the local machine’s personal certificate store. I ran that batch file to install the localhost.pfx file on the machine where the service will run, because the service will need access to the private key in order to decrypt tokens the tokens that ACS will produce.

Next I need to tell ACS about the Calculator application, so it knows what to do when a client shows up asking for a security token for that app. The three things ACS needs to know about the application are:

1. It’s name (a URI that clients will use to identify the app)
2. It’s certificate (ACS needs the public key to encrypt tokens it issues)
3. The rules ACS should to issue claims

So I surfed to accesscontrol.windows.net and signed in, then I selected my solution (“asolution”). This allowed me to add an application into my solution, as shown in Figure 19. Note that this is where I specified the application’s name (“Application URL”) and its certificate.

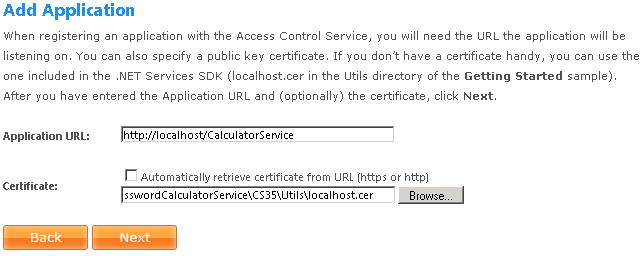


Figure : Adding an Application to an ACS Solution

I pressed Next to take me to the rules page, where I configured a few very simple rules for the application. The client program that comes with the Calculator service expects a user name and password: that’s how it authenticates with ACS. And as you’ll learn later in this paper, each solution in ACS comes with a password so that you can use the solution name as a user name and the password to authenticate to ACS. So when I set up my rules, I told ACS to grant permission to add, subtract, multiply, and divide only if the user provided the correct user name and password for the solution.

Figure 20 shows the resulting set of rules.

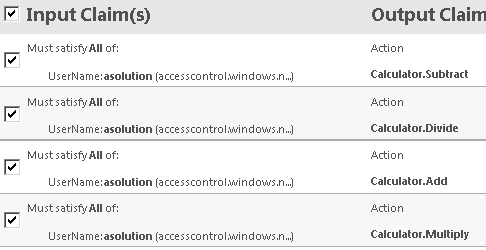


Figure : Rules for the Calculator Application

At its heart, ACS is a claims transformation engine, and you can see that by looking at the rules in Figure 9. The rules I specify all look for one incoming claim: a “UserName” claim with a value of “asolution” issued by ACS. The only way a user can get this claim from ACS is by supplying the solution name and its corresponding password when asking ACS for a security token. These four rules ensure that any user that proves knowledge of the password for my solution will get four “Action” claims. An “Action” claim contains a string naming the action, which can be anything: “foo”, “1234”, or “Calculator.Divide”.

So how are these claims used in the Calculator service? The service simply looks at the incoming security token presented by the client, and ensures that the corresponding action is present (see Figure 21).

public class CalculatorService : ICalculator

{

public double Add(double n1, double n2)

{

AccessControlHelper.DemandActionClaim("Calculator.Add");

return n1 + n2;

}

public double Subtract(double n1, double n2)

{

AccessControlHelper.DemandActionClaim("Calculator.Subtract");

return n1 - n2;

}

public double Multiply(double n1, double n2)

{

AccessControlHelper.DemandActionClaim("Calculator.Multiply");

return n1 \* n2;

}

public double Divide(double n1, double n2)

{

AccessControlHelper.DemandActionClaim("Calculator.Divide");

return n1 / n2;

}

}

Figure : Code for the Calculator Service

I’ll show the code for the helper method, DemandActionClaim shortly, but I hope you get the idea from Figure 21 that all we’re doing is looking for an action claim with a particular string value before performing the operation. The helper method throws an “access denied” exception if it doesn’t find the specified claim. Ultimately, if you can prove you know the password for the solution, you will be allowed to add, subtract, multiply, or divide. If you don’t, you will be denied access to all of these operations.

Figure 22 shows the helper method, DemandActionClaim. You may not be aware of this, but WCF was built with claims in mind. See how the OperationContext gives access to a set of claim sets? Each claim set represents a security token. The code here loops through and finds the token (there’s only one in this case, issued by ACS). It then looks for a claim of a particular type (the hardcoded URI that ends in /action identifies an Action claim) and ensures that it has the value specified by the caller (“Calculator.Add”, for example) and that it’s issued by ACS[[8]](#footnote-8). If these requirements aren’t met, the helper method faults with “Access denied.”

public static void DemandActionClaim(string claimValue)

{

foreach (ClaimSet claimSet in OperationContext.Current

.ServiceSecurityContext

.AuthorizationContext

.ClaimSets)

{

foreach (Claim claim in claimSet)

{

if (AccessControlHelper.CheckClaim(claim.ClaimType,

claim.Resource.ToString(),

"http://docs.oasis-open.org/wsfed/authorization/200706/claims/action",

claimValue))

{

if (AccessControlHelper.IsIssuedByIbn(claimSet))

{

return;

}

}

}

}

throw new FaultException("Access denied.");

}

Figure : Code for DemandActionClaim helper method

Figure 23 shows a run of the client with these rules set up. I made sure I typed in the correct solution name and password so that access would be granted. If I’d typed in an incorrect password, I’d have gotten an exception complaining that I couldn’t be authenticated by ACS.

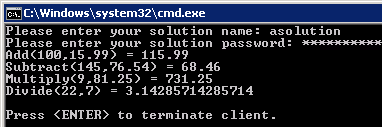


Figure : Running the client with all four Action claims

Once I got the solution running, I went back to the rules and disabled Calculator.Divide by simply unchecking its box (see Figure 20). Then I ran the client again. This time, it got only three of the four Action claims, and was denied access to the Divide operation (Figure 24).

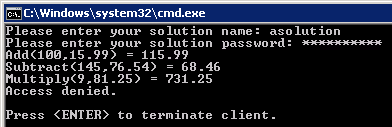


Figure : Running the client after disabling one of the Action claims

Next, I tried typing in another solution name and password to ACS. Figure 25 shows the result.

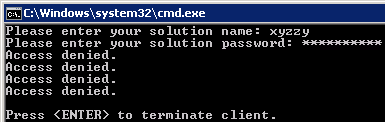


Figure : Running the client with a different user name

As you can see, I was able to specify the user name and password for another solution (xyzzy), but recall from Figure 20 that my rules look specifically for a UserName claim with a value of “asolution”, not “xyzzy”. So I ended up with a token with no Action claims in it, and wasn’t able to access any of the operations in the Calculator service (e.g., the DemandActionClaim authorization logic failed in this case).

Finally, I tried specifying “asolution” with an incorrect password, to simulate an attacker who might try to use the application without knowing the correct password. The result? WCF threw an exception that indicated that the caller could not be authenticated. So ACS didn’t even try to issue a token in this case, because it couldn’t verify the user name and password that I supplied.

If you examine this SDK sample further, you’ll see that it uses the claims programming model built into WCF today (System.IdentityModel). It’s also possible to pull in the Geneva Framework and use its programming model instead. The Geneva Framework is a newer approach to programming claims-based applications, which tends to be more flexible and feature rich, and more accurately represents the future of claims-based programming in Microsoft .NET going forward[[9]](#footnote-9).

*For a deeper look at the .NET Access Control Service and some additional code samples, see the Developer’s Guide to the Microsoft .NET Access Control Service in this series (see Additional Resources).*

# Microsoft® Workflow Service

One of the biggest challenges in building large-scale distributed applications is figuring out how to model complex messaging interactions. The Microsoft .NET Workflow Service allows you to design message orchestration logic with WF, and it provides a hosted, scalable environment for running and managing these WF workflow instances in the cloud, relieving your from having to build your own WF host.

## A Cloud-based Workflow Host

The .NET Workflow Service is part of the Azure™ Services Platform and it integrates with the .NET Service Bus and the .NET Access Control service to provide secure coordination of message interactions. The .NET Workflow Service also provides management tools for creating and managing workflow types and instances and a Web service API for situations where you may wish to build your own tools.

Because the hosting environment is built on the Windows® Azure™ platform, it’s 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 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.

As you move towards cloud computing, the .NET Workflow Services provides a simplified approach for coordinating complex .NET Service Bus interactions in the composite “cloud” solutions you build.

## Cloud Hosting and Persistence

Building a host for WF workflows means making decisions about what features the environment will support, and how best to make the environment secure, scalable and stable. Today, the .NET Workflow Service is built on .NET Framework 3.5 and the WF activities and components in that framework release. However, in order to provide the best possible experience, Microsoft has added a few custom activities and 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-premises solutions, the use of a custom persistence provider is transparent to workflow developers. It feels just like any other WF environment.

## Designing Workflows for the Cloud

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 .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 .NET Workflow Service will support them as well. In addition, if custom steps are required, on-premises workflow capabilities can be combined with cloud-hosted workflows using the .NET Service Bus.

The .NET Services SDK comes with a new project template for building cloud workflows, a set of new cloud activities, and a client API for remotely deploying and managing workflows hosted in the cloud.

## Cloud Activities

When writing cloud workflows, you have to be careful about which WF activities you use (the designer will only show you what’s allowed). You’re allowed to use some of the basic WF control flow activities including IfElse, While, Sequence, Suspend, Terminate, and FaultHandler. In addition to these basic control flow activities, you can also use the CancellationHandlerActivity and FaultHandlersActivity in cloud workflows to provide the ability to model both exception handling and cancelation logic. Note that these activities are not usually added to a model directly, instead the designer for composite activities adds them automatically when the view is switched on the activity.

Aside from these basic activities, you’re not allowed to use any of the other WF activities nor are you allowed to use custom activities. You can only use the new cloud-specific activities included in the .NET Services SDK. Figure 26 describes the new cloud-specific activities that ship with the .NET Services SDK.

Since the primary purpose of the .NET Workflow Service is to simplify message orchestration, these activities are primarily focused on sending, receiving, and processing messages. You can send/receive messages using traditional HTTP requests or through the .NET Service Bus. These activities will appear in the Visual Studio toolbox when you use the sequential cloud workflow project template.

|  |  |
| --- | --- |
| 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 |
| CloudServiceBusSend | Send a message to a specific endpoint on the ServiceBus |
| CloudServiceBusReceive | Receives messages from an 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 |

Figure : Cloud-specific Activities

## A Simple .NET Workflow Service Example

Let’s walk through a simple example that illustrates the steps involved in creating and hosting a workflow in the cloud using the .NET Workflow Service. We’ll design the workflow to communicate with a WCF service running on-premises through 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-premises service. To this project add an interface and a class to define a simple service contract and implementation as shown in Figure 27. 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 : Simple On-premises 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 28.[[10]](#footnote-10)

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

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

new NetEventRelayBinding(),

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

TransportClientEndpointBehavior tb = new TransportClientEndpointBehavior();

tb.CredentialType = TransportClientCredentialType.UserNamePassword;

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

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

ep.Behaviors.Add(tb);

host.Open();

Console.WriteLine("Host is listening");

Console.ReadLine();

host.Close();

Figure : Hosting the On-premises Service

The next step is to create the cloud-based workflow in Visual Studio. To do this, add a new project to the solution and choose the new CloudSequentialWorkflow[[11]](#footnote-11) 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 different ways to create a business process. The process for designing a cloud workflow is really no different than any other WF workflow, 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. 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 29.

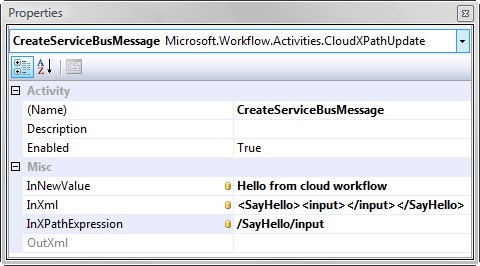


Figure : Configuration of the CloudXPathUpdate Activity

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 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}.servicebus.windows.net/Hello”.

This activity now has all the data it needs to correctly construct a message and send it to the endpoint on the .NET Service Bus. Once you’re done, the completed workflow should look similar to Figure 30.

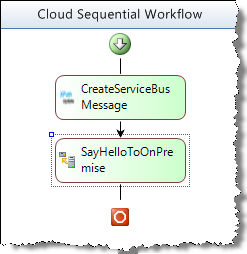


Figure : Completed Workflow

In this simple example we’re only exercising two of the activities that are available for use in the cloud. Developers can use several composite activities to create fault handling logic and cancellation steps just as they would in WF. 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 file to use. 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 31.

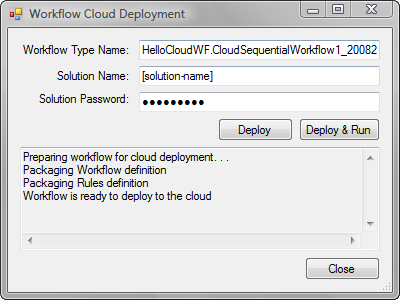


Figure : Visual Studio Cloud Workflow Deployment Dialog

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. Congrats on successfully running your first workflow in the cloud!

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 developer experience that can be extended and enhanced over time.

*For a deeper look at the .NET Workflow Service and some additional code samples, see the Developer’s Guide to the Microsoft .NET Workflow Service in this series (see Additional Resources).*

# Bringing it all Together

.NET Services provides the key building block services you’ll need when building cloud-based or cloud-aware applications on the Azure™ Services Platform. With the .NET Service Bus, you can connect your existing on-premises applications with the new investments you’re building for the cloud. Those cloud assets will be able to communicate with your on-premises services through the network traversal capabilities provided by through the .NET Service Bus relay. The valuable connectivity fabric offered by the .NET Service Bus accommodates a variety of messaging patterns including one-way messaging, request/response messaging, publish/subscribe (multicast), asynchronous messaging (queues), and others, all while overcoming network obstacles (firewalls, NATs, etc) found at Internet scope.

The .NET Service Bus relies on the .NET Access Control Service for securing access to the relay. The .NET Access Control Service makes it possible to leverage a modern claims-based identity security model without requiring you to build complex security infrastructure yourself. The .NET Service Bus trusts the claims produced by the .NET Access Control Service, which it can then process to determine if senders or receivers should be allowed to send or listen on a particular .NET Service Bus address. Senders and receivers must present credentials to the .NET Access Control service in order to acquire a security token for the .NET Service Bus. They can provide solution credentials, WLIDs, or Active Directory identities for the .NET Access Control Service to authenticate (this is all configurable through the portal). Once authenticated, the .NET Access Control Service will issue the security token for the .NET Service Bus relay. This is just one example of how the .NET Access Control Service can be used.

You can also use the .NET Access Control Service in conjunction with your own services. You’d follow a very similar model to the one just described. First you establish a trust relationship between your service and the .NET Access Control Service (by supplying a certificate). Then you configure the .NET Access Control Service with the types of credentials it should trust, and some declarative rules for producing the output claims required by your application. Since the output claims will be encrypted with your certificate, only your service will be able to process the resulting security token. Then you can write code within your service to process the incoming claims provided by the .NET Access Control Service (just like the .NET Service Bus relay does). Ultimately, this provides a very flexible security model based on claims-based identity. It allows you to centralize authentication and claims transformation logic while federating identities from a variety of different sources (WLID, Active Directory, business partners, etc).

Finally, the .NET Workflow Service makes it possible to securely coordinate messaging interactions at Internet scope through the .NET Service Bus and .NET Access Control Service. You can design message-oriented workflows using the Visual Studio workflow designer. The SDK comes with a new set of cloud-based activities that you can use to build cloud-based workflow definitions. Then you can upload the XAML workflow definition (using a Visual Studio add-in) to the Azure™ Services Platform for hosting and execution. As your workflow instances run, they can communicate directly with other services found on the Internet and with the services on your private networks through the .NET Service Bus fabric.

The graphical workflow design experience offered by WF makes building cloud-based composite services a simpler proposition for .NET developers. The .NET Workflow Service is integrated with the .NET Service Bus and the .NET Access Control Service, and makes them easier to compose into business processes.

Overall, the family of .NET Services provides the basic developer building blocks for cloud applications targeting the Azure™ Services Platform. There’s a great deal of synergy and integration between them, and it’s likely you’ll come into contact with each one when building a complete cloud-based application.

# Summary

The Azure™ Services Platform represents a comprehensive Microsoft strategy designed to make it easy for Microsoft developers to realize the opportunities inherent in cloud computing. Microsoft® .NET Services is a key part of that platform, designed specifically to aid .NET developers in making the move. .NET Services provides cloud-centric building blocks and infrastructure in the areas of application connectivity, access control, and workflow hosting and management. These building blocks will become central to the “cloud” experience for .NET developers in the years ahead. For more information on the .NET Service Bus, the .NET Access Control Service, and the .NET Workflow Service, check out the in-depth whitepaper on each topic found within this series of .NET Services whitepapers.

# Additional Resources

We’ve provided links to several resources below that will further your education on the suite of Microsoft® .NET Services and each of its key service offerings described in this paper.

## Microsoft® .NET Services Whitepaper Series

* An Introduction to Microsoft .NET Services for Developers (*this paper*)
  + <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
  + <http://go.microsoft.com/?linkid=9638350>

## Microsoft® .NET Services Resources

* Azure Services Platform
  + <http://www.microsoft.com/azure/services.mspx>
* Microsoft® .NET Services
  + <http://www.microsoft.com/azure/netservices.mspx>
* Java SDK for Microsoft .NET Services
  + <http://www.jdotnetservices.com/>
* Ruby SDK for .NET Services
  + <http://www.dotnetservicesruby.com/>

# About the Author

Aaron Skonnard is a cofounder of Pluralsight, a premier Microsoft .NET training provider offering both instructor-led and online training courses. Aaron is the author of numerous books, articles, and whitepapers, as well as Pluralsight’s Azure Services, REST, WCF, and BizTalk courses. Aaron has spent years developing courses, speaking at conferences, and teaching developers throughout the world. You can reach him at <http://pluralsight.com/aaron> or at <http://twitter.com/skonnard>.

# Acknowledgements

Thanks to Keith Brown and Matt Milner, my coauthors on this series of .NET Services whitepapers. They are primarily responsible for the access control and workflow content found in this paper.

Thanks to John Shewchuk and Dennis Pilarinos for their early vision that led to Microsoft® .NET Services, which has helped to reshape the future of .NET development in the cloud. In addition, thanks to Anjan Das, David Wortendyke, and Scot Gellock for their specific feedback while authoring this paper.

1. The functionality of Microsoft® .NET Services is likely to grow to provide even more .NET Framework features in the cloud. [↑](#footnote-ref-1)
2. The solution name must be globally unique across all .NET services users and it must be at least 6 characters long. You may have to get a bit creative to find a solution name that hasn’t already been used by someone else. [↑](#footnote-ref-2)
3. This is why the solution name must be unique across all users. [↑](#footnote-ref-3)
4. This terminology was used in the BizTalk Services documentation but is no longer an official term used by Microsoft. [↑](#footnote-ref-4)
5. RIA = Rich Internet Application [↑](#footnote-ref-5)
6. This can be done programmatically by simply issuing an HTTP GET request to the solution’s base address. [↑](#footnote-ref-6)
7. The .NET Access Control Service also includes SOAP and REST endpoints for programmatic administration, as well as a set of .NET classes that simplify calling these endpoints. So you can build your own administration console if you don’t like the one ACS provides, or if you want problem domain-specific customizations. [↑](#footnote-ref-7)
8. “Ibn” apparently was an old internal code name for ACS, so don’t let the IsIssuedByIbn helper method confuse you. [↑](#footnote-ref-8)
9. The only reason the ACS team used the WCF programming model in the calculator sample was due to shipping considerations (they weren’t allowed to redistribute the Geneva Framework at the time the SDK shipped). [↑](#footnote-ref-9)
10. Be sure to use your own solution name and password if you’re following along with this example. [↑](#footnote-ref-10)
11. In the November 2008 CTP release, only the sequential workflow model is supported. In future releases, other root workflow types likely will be added to the list of supported workflow types. [↑](#footnote-ref-11)