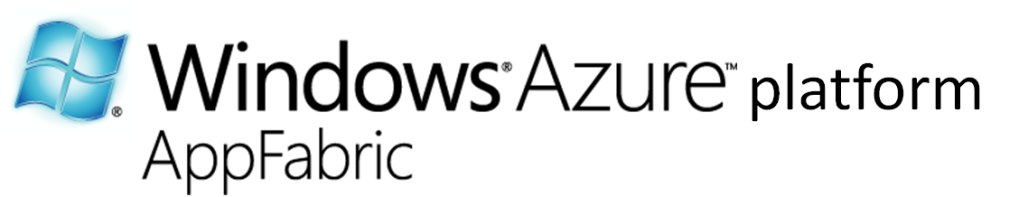
|  |  |
| --- | --- |
| **mslogo-1.jpg** | **PluralsightColorLogoLgTrans.png** |



An Introduction to Windows Azure platform AppFabric for Developers

Extending .NET Applications to the Windows Azure Platform

Aaron Skonnard, Pluralsight

Keith Brown, Pluralsight

November 2009

*All information and code samples are based on the November 2009 CTP release.*

Contents

[Abstract 4](#_Toc245876608)

[An Overview of the Windows Azure Platform 4](#_Toc245876609)

[Windows Azure 5](#_Toc245876610)

[Compute 5](#_Toc245876611)

[Storage 6](#_Toc245876612)

[Service Bus and Access Control 6](#_Toc245876613)

[SQL Azure 8](#_Toc245876614)

[Service Bus & Access Control 8](#_Toc245876615)

[Overview of Service Bus and Access Control 8](#_Toc245876616)

[Getting Started with the Service Bus & Access Control 10](#_Toc245876617)

[Creating your First Service Namespace 13](#_Toc245876618)

[Samples in the Service Bus and Access Control SDK 15](#_Toc245876619)

[Service Bus 15](#_Toc245876620)

[Relayed Connectivity 16](#_Toc245876621)

[Direct Connectivity 17](#_Toc245876622)

[Relay Addresses 18](#_Toc245876623)

[Service Registry 19](#_Toc245876624)

[Access Control 19](#_Toc245876625)

[Message Buffers 19](#_Toc245876626)

[Integration with WCF 20](#_Toc245876627)

[A Simple Service Bus Example 21](#_Toc245876628)

[Access Control 26](#_Toc245876629)

[Claims-based Identity 26](#_Toc245876630)

[Benefits of Claims-based identity 27](#_Toc245876631)

[Support for REST Web Services 27](#_Toc245876632)

[Where can I use Access Control Today? 27](#_Toc245876633)

[Hello Access Control 28](#_Toc245876634)

[Simplicity of Client Interaction 29](#_Toc245876635)

[Simplifying Authorization for REST Web Services on any Platform 30](#_Toc245876636)

[Configuring Access Control 30](#_Toc245876637)

[Cross Realm Federation 31](#_Toc245876638)

[The Beauty of Claims, Manifested 31](#_Toc245876639)

[Bringing it all Together 31](#_Toc245876640)

[Summary 32](#_Toc245876641)

[Additional Resources 33](#_Toc245876642)

[About the Authors 33](#_Toc245876643)

[Acknowledgements 33](#_Toc245876644)

# Abstract

This whitepaper provides a quick introduction to Windows Azure Platform AppFabric, which includes Service Bus and Access Control. Windows® Azure™ platform AppFabric provides common building blocks required by .NET applications when extending their functionality to the cloud, and specifically, the Windows Azure platform. In this overview paper, we’ll introduce you to these core service offerings, discuss how they fit together, and show you how to quickly get started taking advantage of them. For more in-depth coverage of each service, see the additional resources listed at the end of this paper.

*Note: The Service Bus and Access Control services that were once collectively known as the .NET Services now run directly within Windows Azure. Windows Azure now provides secure connectivity natively via Service Bus and Access Control, in much the same way that it also provides compute and storage as a cloud service. To reflect this change, these capabilities are now branded Windows Azure platform AppFabric, and you will see these changes take effect as we transition from a CTP to a business.*

# An Overview of the Windows Azure Platform

The Windows Azure platform is poised to radically change the way Microsoft architects and developers think about building and managing applications. The Windows Azure 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.

The Windows Azure platform portal shown in Figure 1 is your starting point for all things Azure. You can browse to it at <http://windowsazure.com>. From here, you can create Windows Azure accounts, download the developer SDKs, and access a variety of valuable learning resources.



Figure 1: Windows Azure Platform

The Windows Azure platform consists of the Windows Azure cloud-based operating system, which provides the core compute and storage capabilities required by cloud-based applications as well as some constituent services – specifically the Service Bus and Access Control – that provide other key connectivity and security-related features. The Windows Azure platform also comes with a cloud-based relational database called SQL Azure™, allowing you to move your on-premises relational databases and logic to the cloud. These features and services offer a valuable cloud-based development fabric.

You can take advantage of the new Windows Azure 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 is an operating system as a service – you can think of it as Windows in the cloud. It provides a cloud computing *fabric*, hosted within Microsoft data centers, for creating, deploying, managing, and distributing applications and services on the Internet (see Figure 2).

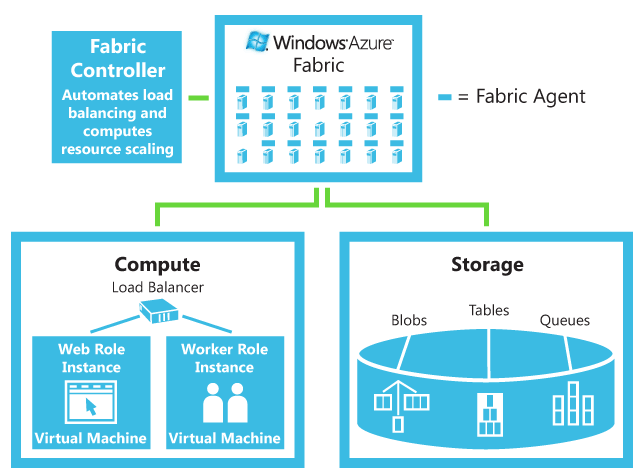


Figure 2: Windows Azure Fabric

The Windows Azure fabric provides two main areas of functionality: *compute* (e.g., executing an application) and *storage* (e.g., storing data on disk), the foundational building blocks for all cloud applications. In addition to these core services, Windows Azure also comes with *Service Bus* and *Access Control* capabilities, which make it easier to extend your .NET applications into the cloud.

### Compute

The compute service offered by Windows Azure makes it possible to “execute” your applications in the cloud. The compute service provides you with a way to run your applications on a Windows Server running in a virtual machine hosted in Microsoft data center. When you deploy and application to Windows Azure, you’re deploying it to execute within this type of highly-scalable environment.

The value is in how Windows Azure provides these core operating system capabilities without theoretical limits. From an application perspective, 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.

### Storage

It’s important to note that the Windows Azure storage services are designed to be very simple and highly scalable. They provide fundamental services for BLOB storage, queue storage, and simple table storage. You interact with these services through a simple REST API based on HTTP requests. You manipulate data in the storage services through traditional POST, PUT, and DELETE requests, and your retrieve information from the storage services using simple GET requests. This approach makes it possible for anyone to integrate with the storage services, regardless of their platform.

It’s also important to note that the Windows Azure storage services are not relational and you don’t query them using SQL. If you need the richer capabilities of a relational database, turn to SQL Azure.

### Service Bus and Access Control

The Service Bus and Access Control features make it easier to extend the reach of your .NET applications through the Windows Azure platform. Today they provide key functionality related to bi-directional application connectivity and federated claims-based access control, both of which are extremely important to applications attempting the migration to the Windows Azure platform.

The primary feature of the Service Bus is to “relay” messages from clients through the Windows Azure cloud to your software running on-premises, by-passing any firewalls, NATs, or other network obstacles that might be in the way (see Figure 3). In addition to relaying messages, the Service Bus can also help negotiate direct connections between apps. The primary feature of Access Control is to provide a claims-based access control mechanism for applications to build on in the cloud (see Figure 4). This makes federation much easier to tackle, allowing your apps to trust identities provided by other systems.

These services come with a .NET developer SDK that simplifies integrating these services into your on-premises .NET applications. The SDK provides seamless integration with Windows Communication Foundation (WCF) and other Microsoft technologies to build on existing skill sets as much as possible.

Although these services have been designed to provide a first-class .NET developer experience, it’s important to note that they each provide interfaces based on industry standard protocols, making it possible for applications running on any platform to integrate with them through REST, SOAP, and WS-\* protocols. There are already SDKs for Java and Rubyavailable for download today.

These services combined with the underlying Windows Azure platform services offer a powerful cloud-based development fabric for developers to build on today.

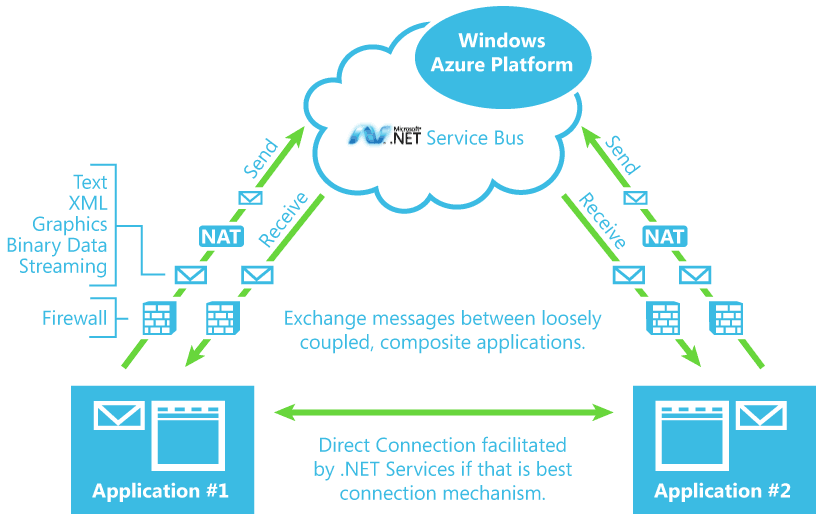


Figure 3: Service Bus

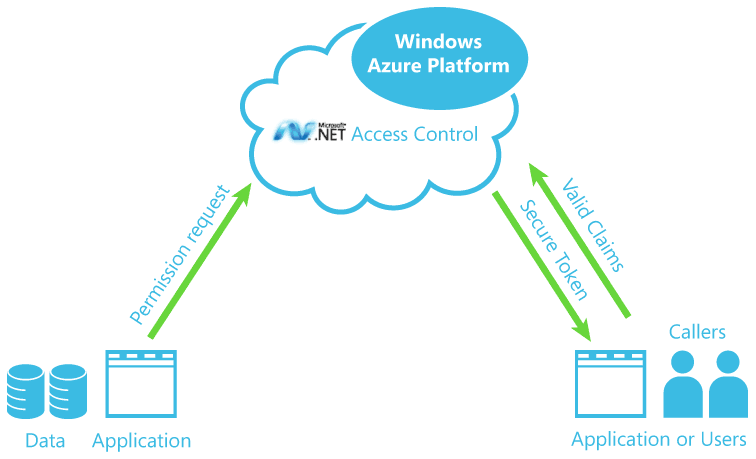


Figure 4: Access Control

## SQL Azure

Microsoft® SQL Azure™ Database is a cloud-based relational database service built on SQL Server® technologies. It provides a highly available, scalable, multi-tenant database service hosted by Microsoft in the cloud. SQL Azure Database helps to ease provisioning and deployment of multiple databases. Developers do not have to install, setup, patch, or manage any software. High availability and fault tolerance is built-in and no physical administration is required. SQL Azure Database supports Transact-SQL (T-SQL). Customers can use existing knowledge in T-SQL development and a familiar relational data model for symmetry with existing on-premises databases. SQL Azure Database can help reduce costs by integrating with existing toolsets and providing symmetry with on-premises and cloud databases.

SQL Azure gives you 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. With SQL Azure, you will be able to provision logical servers and database instances in minutes and begin working with them using the same tools and technologies you use today. This is possible because SQL Azure is built on the Tabular Data Stream (TDS) protocol, the same protocol used to communicate with SQL Server instances running on-premise. Hence, developers can use any TDS-compatible tool or technology when working with their SQL Azure instances running in the cloud.

In order to take advantage of SQL Azure, most applications will only need to update their connection strings to point to SQL Azure and the rest of the application logic can remain largely unchanged.

Now that we’ve provided a quick overview of the Windows Azure platform, we’re ready to dive into the Service Bus and Access Control. The rest of this whitepaper focuses on these service offerings. For more information on the Windows Azure platform, check out the list of resources at the end of this paper.

# Service Bus & Access Control

If you’re a .NET developer moving towards the Windows Azure platform, you’ll want to spend some time becoming familiar with Service Bus and Access Control (the Windows Azure platform AppFabric). The reason is simple: these services offer some of the key developer building blocks you’ll need when building cloud applications targeting Windows Azure. Throughout this whitepaper we’ll provide a quick introduction to each of these key feature areas.

## Overview of Service Bus and Access Control

Today the Service Bus and Access Control provide core functionality related to secure application connectivity and federated access control as described here:

* 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.
* Access Control: provides claims-based access control in the cloud. It includes a claims transformation engine that federates with identity providers like ADFS v2 (Active Directory Federation Services).

Each of these services is available using open protocols and standards, including REST, SOAP, Atom/AtomPub, which means developers on any platform can easily integrate with them.

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

By using the SDK, developers are able to build on their existing .NET experience by taking advantage of the new extensions found in the SDK (e.g., new WCF “relay” bindings). In addition to the SDK, you can find Java and Ruby SDKs available from Microsoft partners today.

Microsoft also provides a Web portal for managing and configuring your Service Bus and Access Control solutions (see Figure 5). You can browse to the developer portal from the Windows Azure platform portal or by browsing to http://netservices.azure.com directly. From here, you can download the SDK, login to manage your account, and access a variety of learning resources.

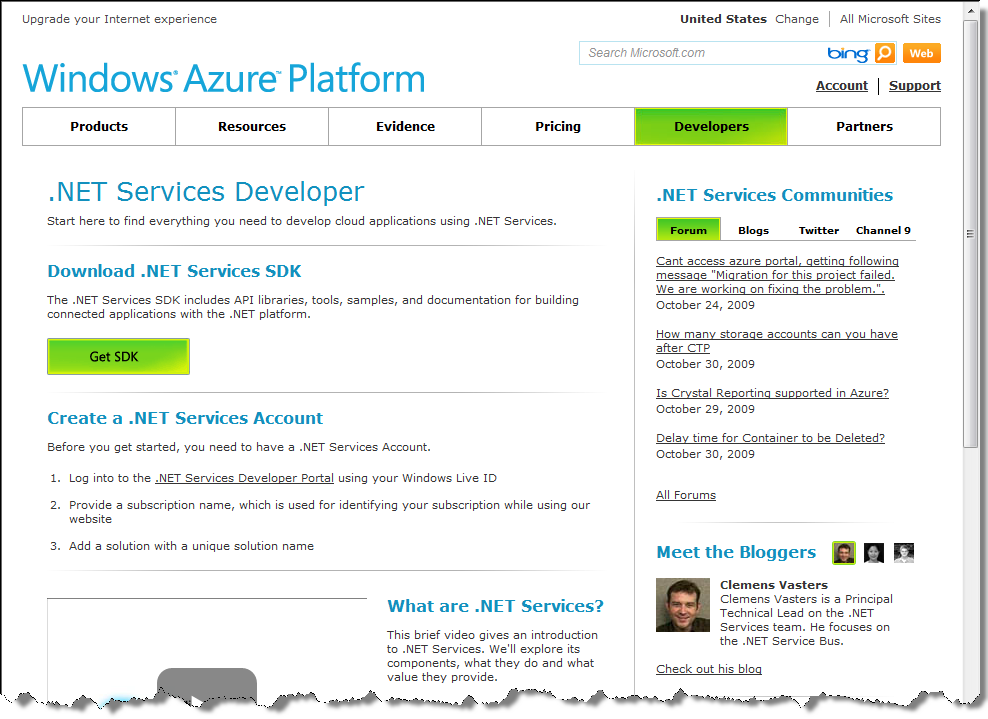


Figure 5: Service Bus & Access Control Developer Portal

## Getting Started with the Service Bus & Access Control

To get started with the Service Bus and Access Control, first browse to the Windows Azure platform portal at <http://www.microsoft.com/windowsazure/> and press the “Learn More” link. This will take you to a “Getting Started” page that provides important links for registering and downloading the various SDKs you’ll need. In order to get started, follow the instructions on this page.

First, download the Service Bus and Access Control SDK and run the setup program as illustrated in Figure 6. Once installed, you’ll have the new assemblies you’ll need to begin taking advantage of the various Service Bus and Access Control features. Next, you must register for an account before you’ll be able to begin using the services.

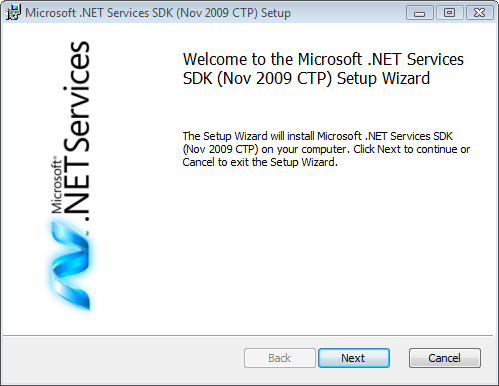


Figure 6: Running the Service Bus and Access Control SDK Setup

In order to create an account, log in to the developer portal using your Windows Live ID (WLID). Doing so will prompt you to login with a WLID as illustrated in Figure 7.

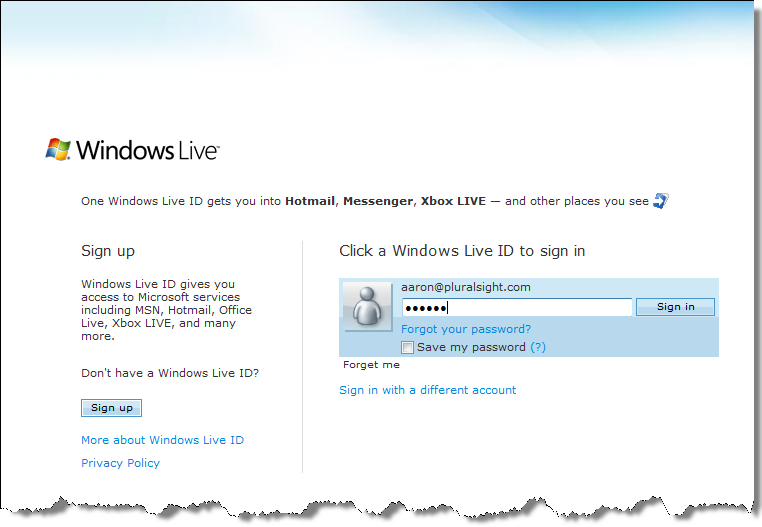


Figure 7: Login to the Developer Portal

Once you’ve successfully logged in, you’ll be asked to “sign-up” for a new account by creating a new project (see Figure 8). A “project” is basically a container for all of your “service namespaces” used by the Service Bus and Access Control services.

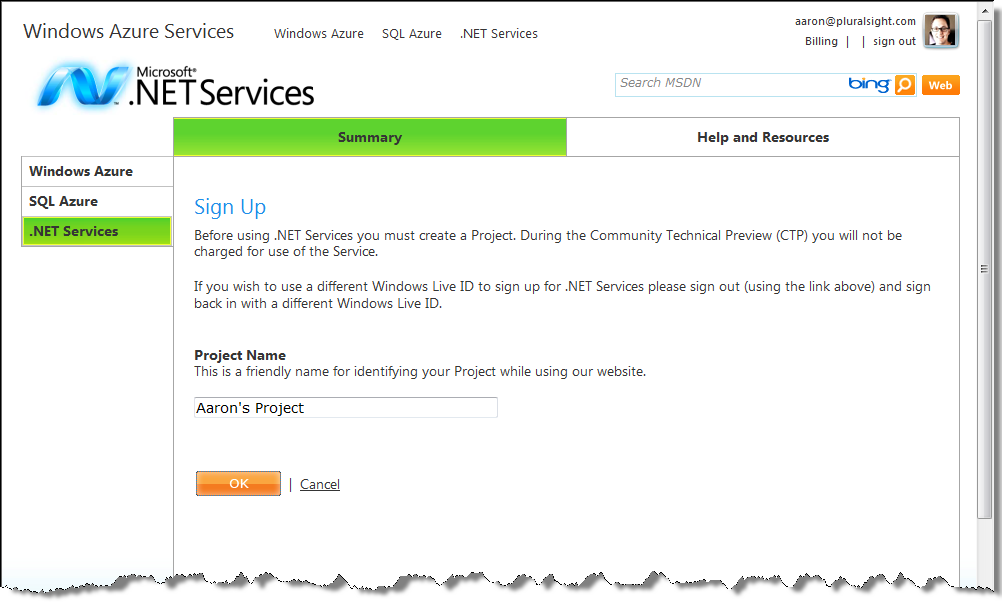


Figure 8: Creating a New Project

In the future, whenever you login with your WLID, you will see all of the projects associated with your account listed within the portal (see Figure 9).

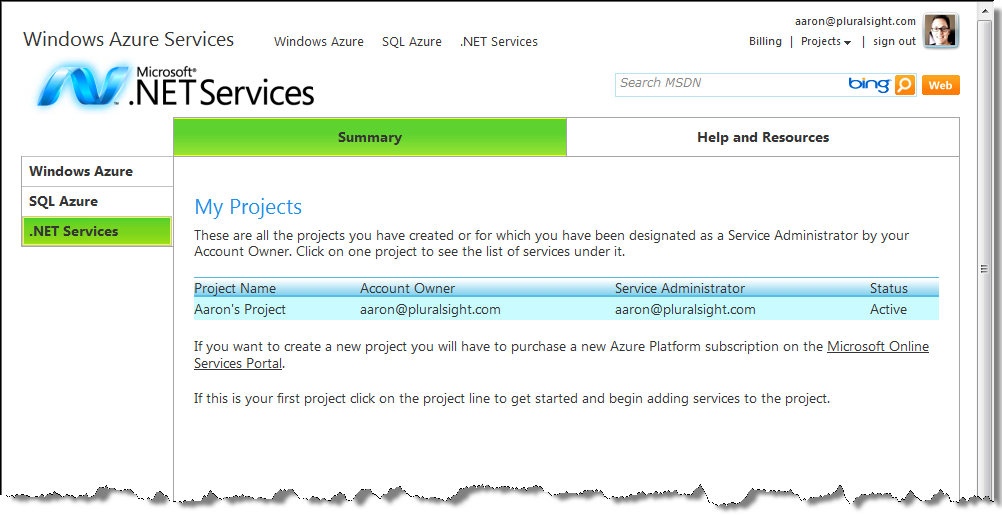


Figure 9: My Projects

If you had multiple projects associated with my account, you would see them all listed here. When you click on one of the projects in the list, you’ll be taken to a project-specific page that displays the project details and allows you to add new service namespaces (see Figure 10).

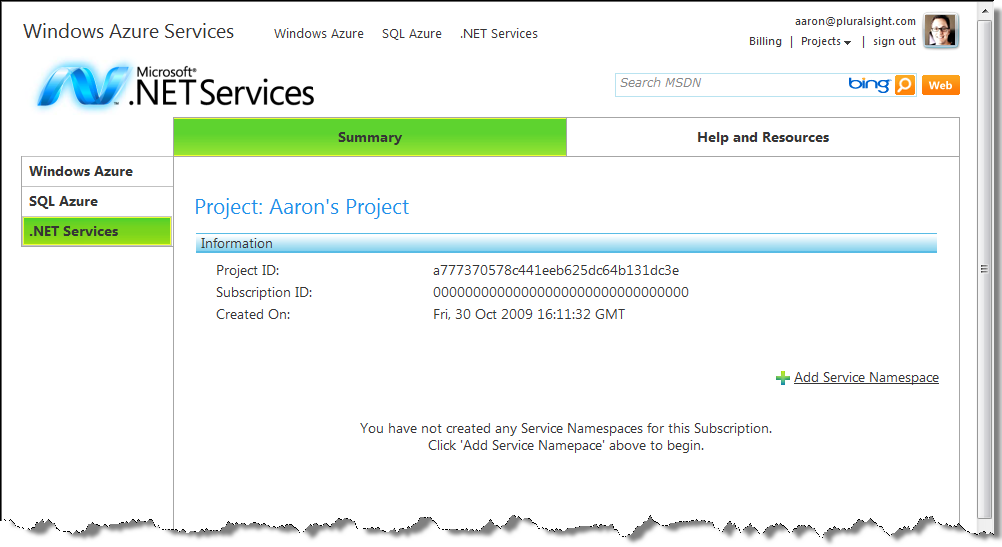


Figure 10: Project Details

## Creating your First Service Namespace

You can think of the *service namespace* as a container for a particular set of Service Bus endpoints and Access Control rules. In order to create a new service namespace, click “Add Service Namespace” shown in Figure 10. This will take you to the Create New Service Namespace page shown in Figure 11.

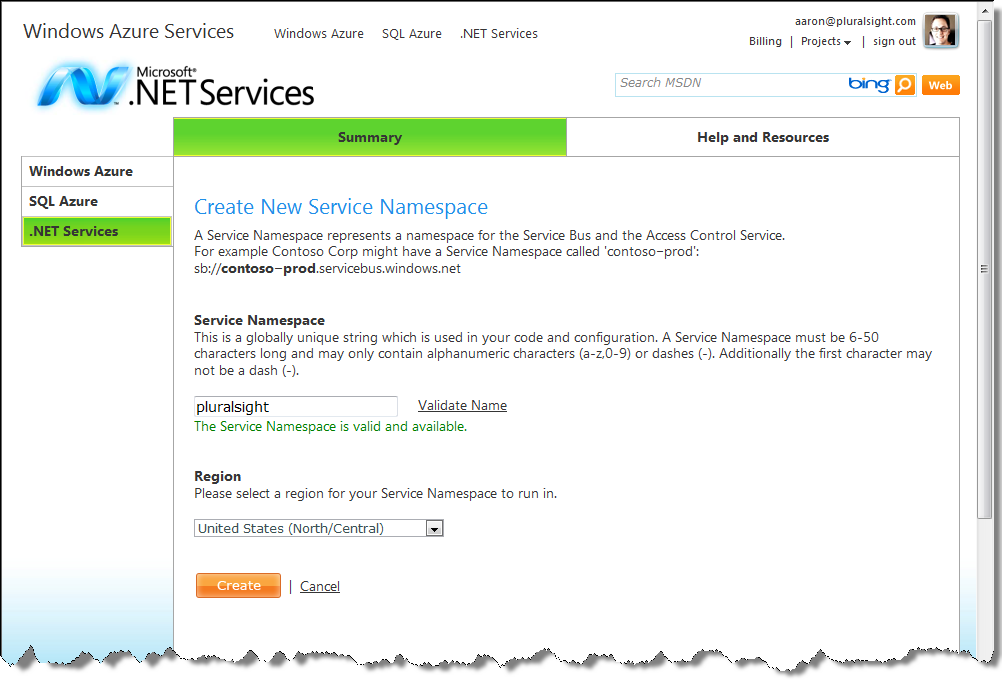


Figure 11: Creating a New Service Namespace

You’ll need to enter a unique service namespace in the provided text box – you can press “Validate Name” to ensure the name you’ve entered is indeed unique before continuing.[[1]](#footnote-1) Then simply press “Create”. At that point, the Windows Azure infrastructure will begin provisioning and activating your new service namespace (see Figure 12), and it will be associated with your project.

Once the infrastructure is finished activating your new service namespace, it will show a status of “Active” in the list (see Figure 13). At that point, you can begin using your service namespace.

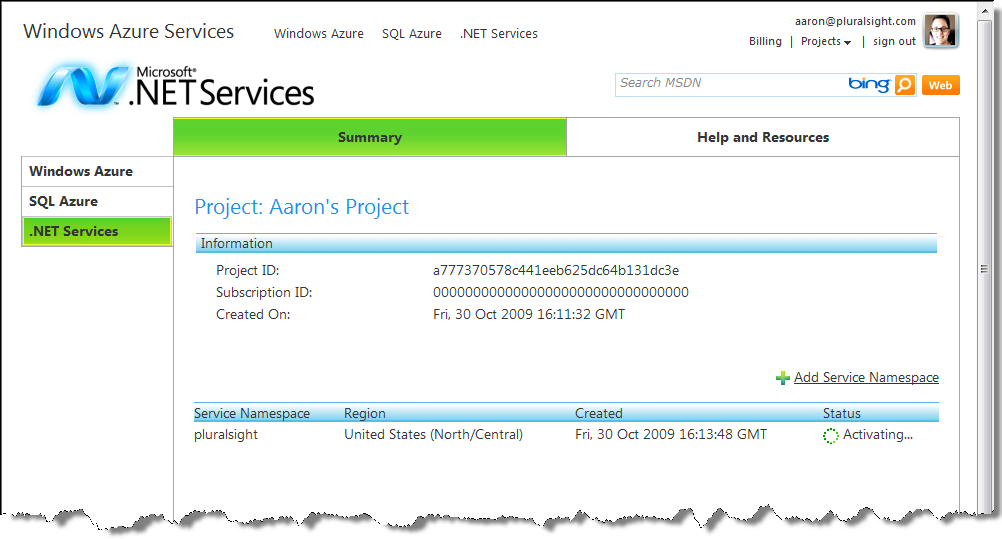


Figure 12: Activating a New Service Namespace

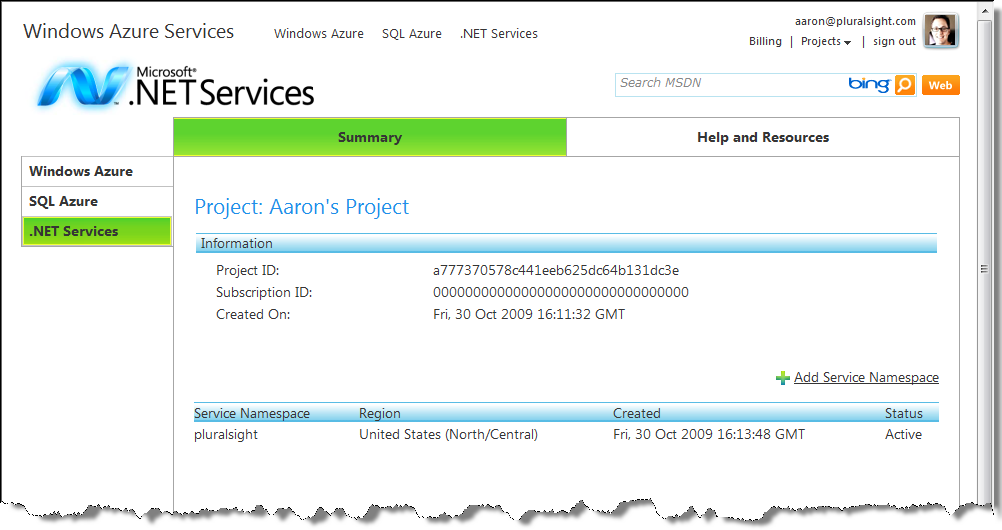


Figure 13: Service Namespace “Active”

You can click on the service namespace (in the list shown in Figure 13) to browse to the service namespace details page. From here, you can delete the service namespace, and you can view all of the important details around the service namespace. You can view items such as the management key, the Service Bus endpoints, the issuer name & key, and the various Access Control endpoints. This information will be used when you begin writing your integration logic.

Now that you have the SDK installed, and you’ve successfully created a Service Bus and Access Control account, a new project, and a new service namespace, you’re ready to begin taking advantage of the Service Bus and Access Control features in your applications.

## Samples in the Service Bus and Access Control SDK

The Service Bus and Access Control SDK comes with a number of interesting samples that illustrate how to leverage the various capabilities offered by the Service Bus and Access Control. You’ll find the samples in the SDK installation directory. Most of these samples have been designed to ask the user to specify the service namespace dynamically. Some will ask you to supply your credentials via the console window while others will require you to enter them in the application configuration file ahead of time. The samples make it easy to explore the features offered by these services so be sure to check them out.

# 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 Service Bus is focused on making the ESB pattern a reality at Internet scope as part of the Windows Azure platform. The 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 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 14).[[2]](#footnote-2)

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 Windows Azure platform), and with a variety of desktop, RIA[[3]](#footnote-3), 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.

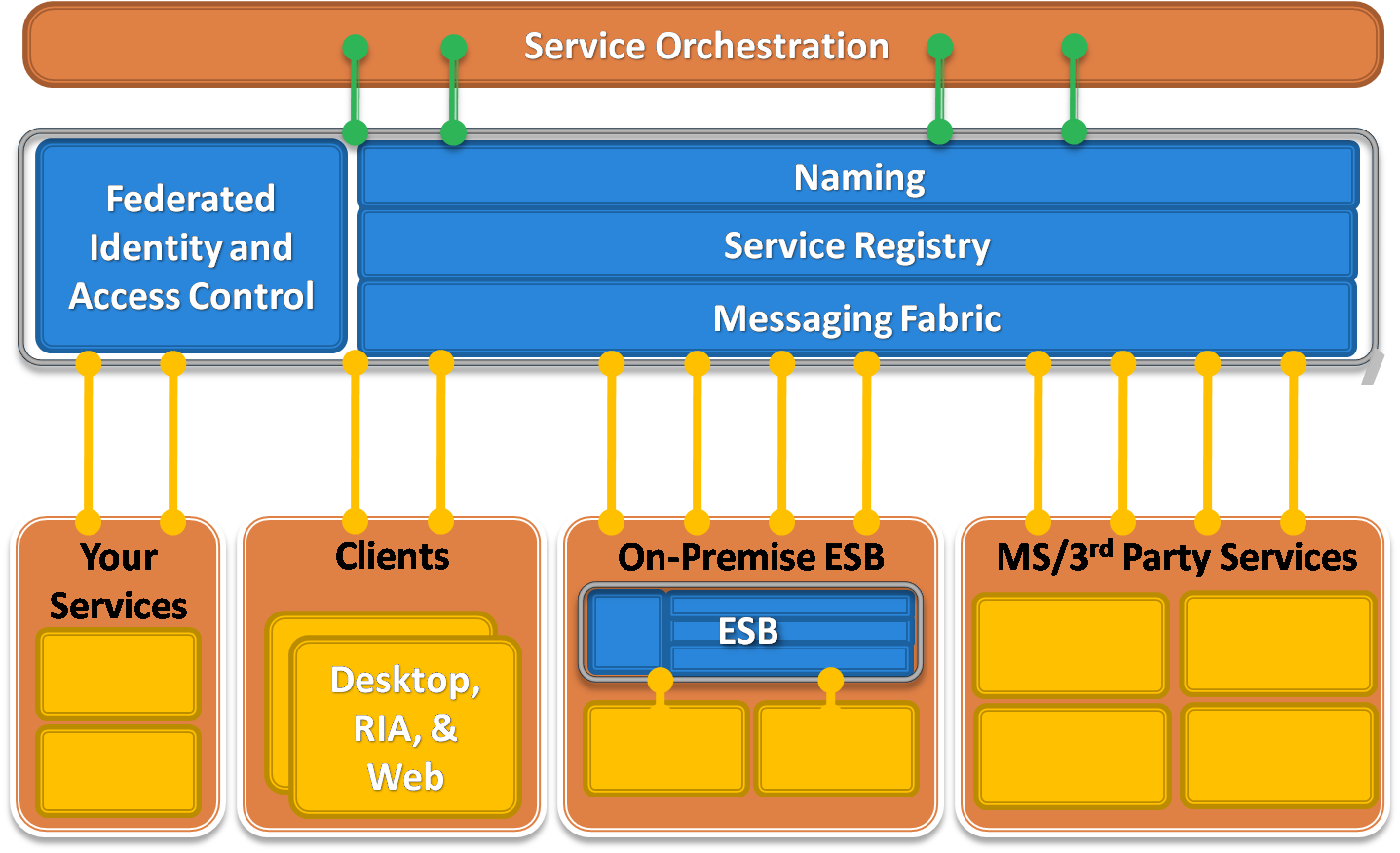


Figure 14: The Internet Service Bus

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 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 such as 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 connectivity logic (see Figure 15).

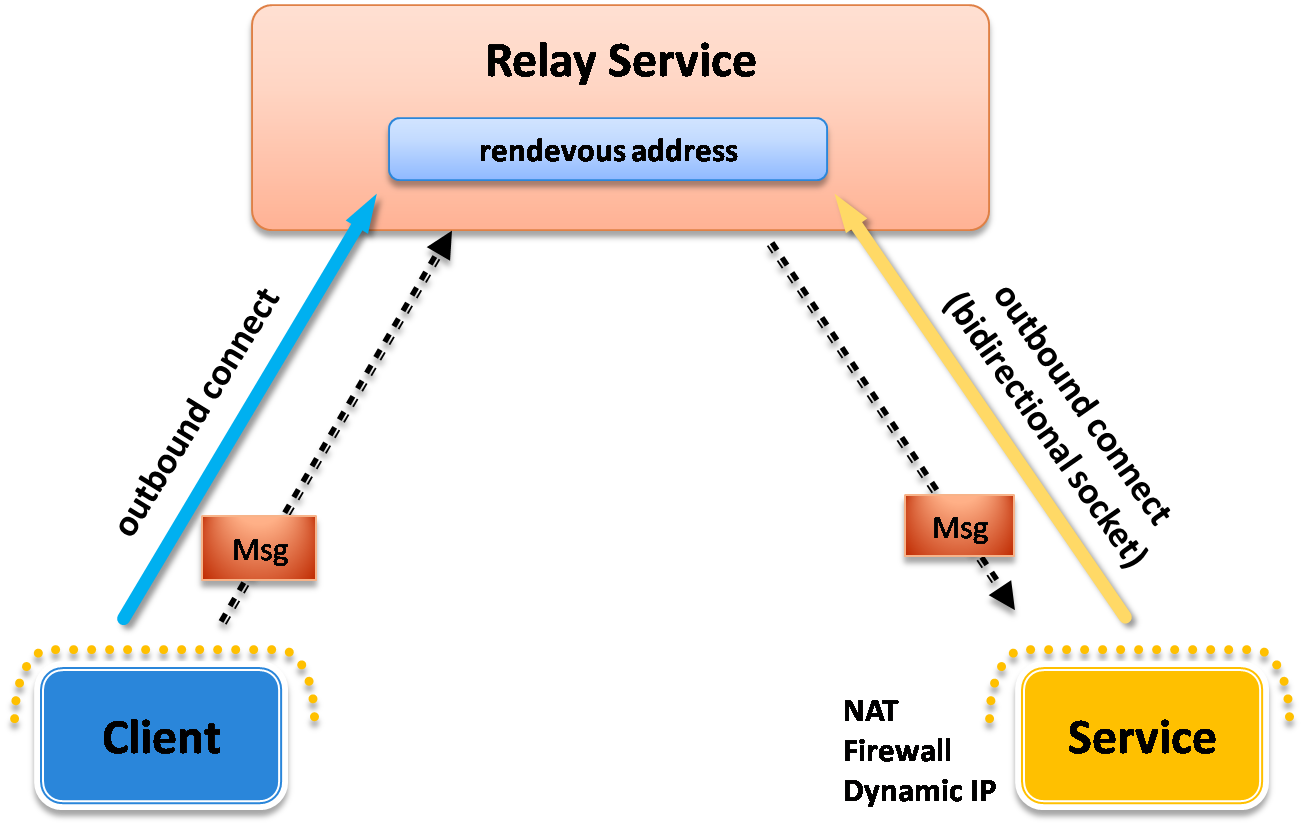


Figure 15: Using the Relay Service

Here’s how it works: the on-premises service connects to the relay service through an outbound port and creates a bidirectional socket for communication tied to a particular *rendezvous address*. The client can then communicate with the on-premises service by sending messages to the relay service targeting the rendezvous address. The relay service will then “relay” messages to the on-premises service through the bidirectional socket already in place. The client does not need a direct connection to the on-premises service nor does it need to know where it resides. The on-premises service doesn’t need any inbound ports open on the firewall. This is how most instant messaging applications work today.

When using the Service Bus relay, you can take advantage of one-way messaging, request-response, publish-subscribe (multicast), and even asynchronous or “buffered” messaging.

For one-way messaging, a single on-premises service registers with the relay to “listen” for messages on a particular rendezvous address. Clients can then “send” messages to the Service Bus address to have the messages “relayed” to the registered service. The relay service also makes it easy to implement publish/subscribe architectures by allowing multiple services to “listen” on the same Service Bus rendezvous address. In that case, when a client transmits a message to that address, the relay will distribute the message to all registered services, providing multicast capabilities through the relay.

Both clients and services 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 client interoperability or when you need to ensure you’re able to communicate through firewalls.

## Direct Connectivity

In addition to relayed communications, the Service Bus also provides a capability for establishing direct connectivity between clients and services in order to improve performance and throughput. Clients and services still communicate with the relay through a common rendezvous address but then the relay 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 prediction algorithm based on probing information from the client and service. 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 next. It can then provide that information to the client/service 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 16).

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.

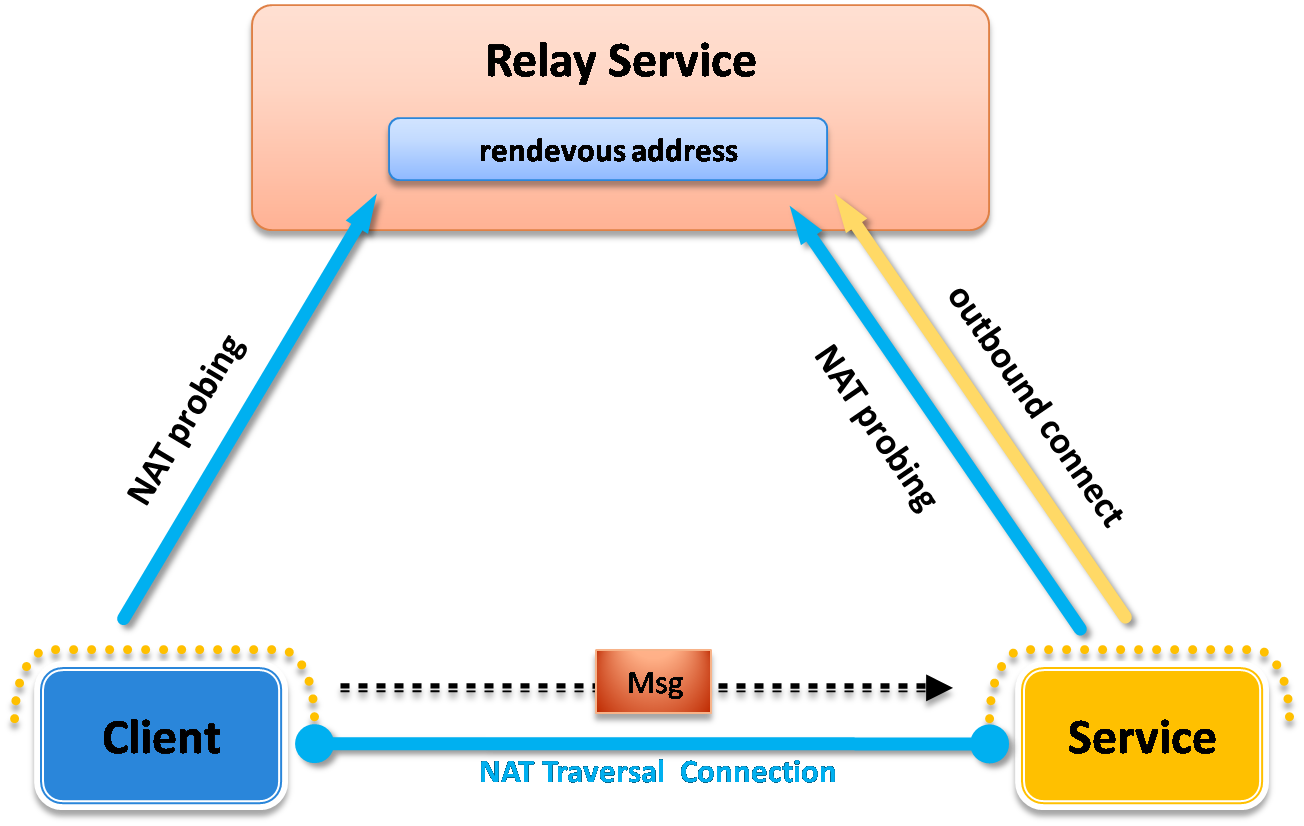


Figure 16: Establishing a Direct Connection

## Relay Addresses

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

sb://{namespace}.servicebus.windows.net/{name1}/{name2}/...

When using an HTTP based address, simply replace the “sb” protocol scheme with “http” or “https”:

http://{namespace}.servicebus.windows.net/{name1}/{name2}/...

Notice that you specify your service namespace as part of the address in order to distinguish your endpoints from those used by other users on the Service Bus. After the domain name that includes your service namespace, you have complete control to specify whatever relative URI you’d like to use.

## Service Registry

The Service Bus provides a service registry for publishing and discovering service endpoint references within a solution. The Service Bus automatically publishes your “public” endpoints within your service registry. Then, others can discover your endpoints by browsing to the base address for the service namespace where they can retrieve an Atom feed containing the endpoint information.

In order to make sure an endpoint appears in your service registry, you must configure the endpoint to be publicly discoverable. You can accomplish this by associating a ServiceRegistrySettings behavior with your endpoints configured with a DiscoveryMode set to DiscoveryType.Public.

## Access Control

The Service Bus implements access control through a claims-based security model and a trust relationship with Access Control discussed in more detail below.

The relay looks for a security token issued by Access Control to determine whether a particular client or service should be allowed to “send” or “listen” on a particular rendezvous address. Before a service can listen on an address, it must acquire a security token containing the “listen” claim. Clients must also acquire a security token containing a “send” claim. Clients and services will need to supply credentials to Access Control in order to acquire a security token for the Service Bus relay.

Access Control supports several different types of credentials including shared secrets, Simple Web Tokens (SWT), and SAML tokens, which may be selected via WCF configuration. Since the relay has a trust relationship with Access Control, it’s able to read the tokens that it issues and process the claims.

The various WCF “relay” bindings found in the SDK automate this behavior I just described, making it really easy for WCF developers to take advantage of the Service Bus and Access Control. Developers simply provide credentials through a WCF behavior. Then the WCF “relay” binding takes care of acquiring a token from Access Control and supplying that token to the Service Bus. This happens behind the scenes within the channel layer so WCF developers are ultimately shielded from the interaction.

Later in the whitepaper we’ll show you how you can integrate directly with Access Control when you want to build similar access control logic into your own applications (similar to the Service Bus).

## Message Buffers

The Service Bus focuses primarily on acting as a relay between two parties. This core function requires on-premises WCF services to “listen” on the relay service before clients can begin communicating with them. But what if the on-premises application isn’t capable of using the WCF “relay” bindings capable of creating these “listener” components? This will be the situation whenever the on-premises applications are running on platforms unable to use the SDK (e.g., Silverlight, Flash, Ruby, Perl, etc).

The Service Bus provides message buffers to support interoperability with platforms unable to consume the SDK and the WCF “relay” bindings found within it. A client sends messages to the message buffer and an on-premises application retrieves messages from the same message buffer – it’s probably better to call these applications “message producers” and “message consumers” respectively.

The difference is the message consumer application doesn’t require a WCF-based connection to the relay service – all of the “receive” logic happens over HTTP(S) requests – which greatly simplifies interoperability. The message consumer simply polls for messages using an HTTP-based REST API.

Message buffers, as their name suggests, provide a data structure with FIFO semantics for “buffering” messages in transit to an on-premises application. The message consumer controls the lifetime of the message buffer. When the application creates a new message buffer it specifies how long a message buffer should live after it stops polling for messages. You do this through what’s called a *message buffer policy* (see the ExpiresAfter property). Every time the on-premises application polls for messages, the message buffer expiration clock automatically resets, extending the lifetime of the message buffer.

But once the on-premises application stops polling for messages, the message buffer will eventually go away and message producers will no longer be able to send messages to it.

You register a new message buffer within your service namespace by picking a name for the message buffer, defining a message buffer policy, and then sending a request to create the message buffer at that location using the supplied policy. Once created, message producers can send messages to your message buffer and the message consumer can retrieve messages from it using simple HTTP(S) requests.

## Integration with WCF

The primary programming model for working with the Service Bus on the .NET platform is WCF. The . SDK comes with a set of new WCF “relay” 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 Service Bus bindings.

Figure 17 lists all of the Service Bus WCF bindings and the standard WCF bindings they correspond to. The most commonly used WCF bindings, such as BasicHttpBinding, WebHttpBinding, WS2007HttpBinding, and NetTcpBinding, all have a corresponding 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 |
| WS2007HttpBinding | WS2007HttpRelayBinding |
| NetTcpBinding | NetTcpRelayBinding |
| N/A | NetOnewayRelayBinding |
| N/A | NetEventRelayBinding |

Figure 17: WCF Relay Bindings

In order to use the 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 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 Windows Azure platform, it provides theoretically unlimited scale-out possibilities as well.

## A Simple Service Bus Example

The following is an example of how you can configure an existing WCF application to take advantage of the Service Bus. You can start with the following simple WCF service implementation:

[ServiceContract]

public interface IHelloServiceBus

{

[OperationContract]

string SayHello(string name);

}

public class HelloServiceBus : IHelloServiceBus

{

public string SayHello(string name)

{

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

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

return greeting;

}

}

You 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("\*\*\*\* Service \*\*\*\*");

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

host.Open();

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

Console.ReadLine();

host.Close();

}

}

Use 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="HelloServiceBus">

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

binding="netTcpBinding"

contract="IHelloServiceBus" />

</service>

</services>

</system.serviceModel>

</configuration>

Next, 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 “DirectEndpoint”) will be read from the client’s application configuration file:

class Program

{

static void Main(string[] args)

{

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

Console.WriteLine("Press <Enter> to run client.");

Console.ReadLine();

ChannelFactory<IHelloServiceBus> channelFactory =

new ChannelFactory<IHelloServiceBus>("DirectEndpoint");

IHelloServiceBus channel = channelFactory.CreateChannel();

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

Console.WriteLine(response);

channelFactory.Close();

}

}

And finally, the client’s application configuration file must 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="IHelloService"

name="DirectEndpoint" />

</client>

</system.serviceModel>

</configuration>

If you run the two applications, you’ll see “Hello 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 Service Bus as a relay between the client and service applications. First you must reconfigure the service host to listen on the Service Bus, and then reconfigure the client to send messages through the Service Bus.

You can reconfigure the service to listen on the Service Bus by simply changing the binding from NetTcpBinding to NetTcpRelayBinding. The NetTcpRelayBinding knows how to bootstrap the underlying WCF channel infrastructure in order to listen on the Service Bus. When using this binding, you also must specify a valid Service Bus address for the endpoint. Since you have a service namespace of “pluralsight", you can use an address of sb://pluralsight.servicebus.windows.net/helloservicebus for the “helloservicebus” endpoint. Note that the protocol scheme is changed from “net.tcp” to “sb”, a special TCP-based protocol scheme used by the Service Bus – all Service Bus endpoints, expect HTTP endpoints, must use the “sb” protocol scheme.

When connecting to the Service Bus, you must prove that you are permitted to listen within the “pluralsight” service namespace. This is accomplished through Access Control. Here’s how it works: the Service Bus application provides credentials to Access Control; then, Access Control authenticates those credentials and issues a token that is presented to the Service Bus. The token issued by Access Control indicates whether or not you have permission to “listen” on, “send” to, or “manage” the service namespace. If your credentials prove that you own “pluralsight”, the issued token will indicate that you have full control over that service namespace.

Access Control supports several different types of client credentials today, the simplest being a “shared secret” credential. You can supply these credentials to Access Control via code or configuration.

In the following example, you supply “shared secret” credentials through the <transportClientEndpointBehavior> behavior in configuration. When you create a service namespace, you are automatically assigned a shared secret to use with the Service Bus – navigate to the service namespace page in the portal and you’ll find it listed under “Default Issuer Name” and “Default Issuer Key”. These are the values you want to use in the <sharedSecret> element below.

<configuration>

<system.serviceModel>

<services>

<service name="HelloServiceBus">

<endpoint address=

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

behaviorConfiguration="sharedSecretClientCredentials"

binding="netTcpRelayBinding"

contract="IHelloServiceBus" />

</service>

</services>

<behaviors>

<endpointBehaviors>

<behavior name="sharedSecretClientCredentials">

<transportClientEndpointBehavior credentialType="SharedSecret">

<clientCredentials>

<sharedSecret issuerName="[Enter your issuer name]"

issuerSecret="[Enter your issuer key]" />

</clientCredentials>

</transportClientEndpointBehavior>

</behavior>

</endpointBehaviors>

</behaviors>

</system.serviceModel>

</configuration>

When the WCF service host opens with this configuration, it first sends the “shared secret” to Access Control to acquire a token for “listening” on the Service Bus. It will then establish a TCP connection with the relay service and present the token it acquired. Assuming you are permitted to listen on this address (for example, the token contains the necessary “listen” claim), the Service Bus will create a listener for relaying messages to our on-premises WCF service, thereby connecting your WCF service to the cloud.

Reconfiguring the client application is very similar. First, you must change the endpoint to use the NetTcpRelayBinding and the same Service Bus address on which you configured your service to listen (again, notice the “sb” protocol scheme). You must also configure the client with credentials. Clients must prove that they are permitted to send messages to a particular address on the Service Bus by acquiring a token from Access Control. This example uses the same “shared secret” used to configure the service application. The following shows the complete client-side configuration:

<configuration>

<system.serviceModel>

<client>

<endpoint address=

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

binding="netTcpRelayBinding"

contract="IHelloServiceBus"

behaviorConfiguration="sharedSecretClientCredentials"

name="RelayEndpoint" />

</client>

<behaviors>

<endpointBehaviors>

<behavior name="sharedSecretClientCredentials">

<transportClientEndpointBehavior credentialType="SharedSecret">

<clientCredentials>

<sharedSecret issuerName="[Enter your issuer name]"

issuerSecret="[Enter your issuer key]" />

</clientCredentials>

</transportClientEndpointBehavior>

</behavior>

</endpointBehaviors>

</behaviors>

</system.serviceModel>

</configuration>

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





Figure 18: HelloServiceBus Sample in Action

*For a deeper look at the Service Bus and some additional code samples, see A Developer’s Guide to the Windows Azure platform Service Bus (see Additional Resources).*

# Access Control

The identity solution that the industry has been moving toward over the last decade is based on claims[[4]](#footnote-4). 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

Access Control is a cloud-based service that does exactly that. Instead of writing your own custom user account and role database, you can let Access Control orchestrate the authentication and authorization of your users. With a single, simple code base in your service, you can authorize access to both enterprise clients as well as simple clients.

Enterprise clients can leverage ADFS v2 to allow users to authenticate using their Active Directory logon credentials. Simple clients can establish a shared secret (essentially an extremely long password) to authenticate directly with Access Control. In the future you'll see support for more and more existing identity systems such as Facebook Connect, Google Accounts, Windows Live ID, as well as other enterprise identity providers.

That bears repeating. With a single, simple code base in your REST Web Service and clients, over time as Access Control evolves, you'll be able to authorize access to more and more users without having to change your server-side code base. This is a huge step forward for application security.

When your service uses Access Control, the user must obtain a security token from Access Control in order to log in to your service. This token is similar to a signed email message from Access Control to your service with a set of claims about the user's identity. Access Control won't issue a token unless the user first proves her identity either by authenticating with Access Control directly or by presenting a security token from another trusted issuer (ADFS for example) that has authenticated her. Thus, by the time the user presents a token to your service, assuming you can validate the token (more on that later), you can trust the claims in the token and get to work processing the user's request (see Figure 19).



Figure 19: User Presents Claims

## Benefits of Claims-based identity

Under this model, single sign-on is easier to achieve, and your service is **no longer responsible** for:

* 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
* Delegation of authentication (a.k.a. federation) with other security realms

This model allows your service to make identity-related decisions based on claims about the user made by a trusted issuer such as Access Control. 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.

## Support for REST Web Services

Access Control will initially support REST web services, and in future versions will also support SOAP web services and browser-based web applications that want to take advantage of single sign on. Access Control will expose two token-exchange endpoints for clients using REST:

* REST with symmetric key
* REST with SAML extension

The first uses a very simple token format that can be easily understood by an application without having to link in any special code. The second allows the use of SAML tokens that makes it possible to federate with identity providers such as ADFS v2 that issue SAML tokens.

## Where can I use Access Control Today?

Over the past decade there have been many initiatives to bring claims-based identity and single sign on to browser-based web applications and SOAP Web Services. But REST Web Services, which are quickly gaining momentum, lack this fundamental support for identity. For this reason, the Access Control team reprioritized their work to first address this need. The first release of Access Control will issue simple tokens that can be easily requested and consumed by clients and REST Web Services on any platform, without the need for libraries such as WIF (Windows Identity Foundation) or WCF. Indeed, the service is equally usable from any language or platform.

With its initial release, Access Control will directly authenticate simple clients with a symmetric key (similar to the familiar user name and password) and broker authentication for enterprise clients who use ADFS v2 (Active Directory Federation Services). In a future release, Access Control will implement single-sign on for browser-based web applications as well as SOAP Web Services, and will accept tokens from many other identity frameworks such as Facebook Connect, Google Accounts, Windows Live ID, and others.

In short, Access Control will first provide basic identity services for REST Web Services, but in the future, Access Control will evolve to become a cloud-based identity clearinghouse all types of clients, applications, and services.

## Hello Access Control

Here’s a high level view of how a claims-based system works, illustrating the simplest possible case, a client, an issuer (Access Control), and a service. Figure 20 shows a simple interaction.



Figure 20: Requesting and Sending Claims

The client sends an HTTPS POST to Access Control that includes three things: the user's name, password, and the base URI of the service that the client wishes to access. Access Control checks to see if an account is registered for the user, and if so, validates the password.

If all is well, Access Control then runs a set of rules that take as input the user's name and the base URI of the web service it plans to access. The output of these rules is a set of claims that Access Control bundles up into a token, signs, and sends back to the client.

The client then makes a request to the web service, including the token in the HTTP Authorization header. The web service checks for this header, pulls out the token, validates its signature, and pulls out the claims that Access Control issued. It uses these claims to discover the user's name, roles, and whatever other claims you've configured in your Access Control rule set.

## Simplicity of Client Interaction

Access Control supports a REST-based token request protocol called WRAP (Web Resource Authorization Protocol), which makes it very easy to request tokens using standard REST techniques. Figure 21 shows how simple this can be. Because "bearer tokens" are used, the client doesn’t need any cryptographic libraries, rather it only needs to support HTTPS as it requests tokens from Access Control.

private static string GetTokenFromACS(string issuerKeySuppliedByCaller)

{

// request a token from ACS

WebClient client = new WebClient();

client.BaseAddress = string.Format("https://{0}.{1}",

serviceNamespace, acsHostName);

NameValueCollection values = new NameValueCollection();

values.Add("wrap\_name", "my-issuer");

values.Add("wrap\_password", issuerKeySuppliedByCaller);

values.Add("applies\_to", "http://localhost/ACSGettingStarted/");

byte[] responseBytes = client.UploadValues("WRAPv0.8", "POST", values);

string response = Encoding.UTF8.GetString(responseBytes);

return response

.Split('&')

.Single(value => value.StartsWith("wrap\_token=",

StringComparison.OrdinalIgnoreCase))

.Split('=')[1];

}

Figure 21: Simple Token Request from Access Control

Note how the request is as simple as posting a form to Access Control with a name, password, and the URI that scopes the protected resource where the token is needed for access. Once the token is returned, the client can send it directly to the service, and may continue to use that token until it expires (tokens typically last for a work day or less), reducing round-trips to the issuer.

After retrieving the token, it's trivial to send it along with a standard request to the protected resource. The token must be sent in the Authorization header, as shown in Figure 22.

private static string SendMessageToService(string token, string valueToReverse)

{

WebClient client = new WebClient();

client.BaseAddress = "http://localhost/ACSGettingStarted/Default.aspx";

string headerValue = "WRAPv0.8" + " " + HttpUtility.UrlDecode(token);

client.Headers.Add("Authorization", headerValue);

NameValueCollection values = new NameValueCollection();

values = new NameValueCollection();

values.Add("string\_to\_reverse", valueToReverse);

byte[] serviceResponseBytes = client.UploadValues(string.Empty, values);

return Encoding.UTF8.GetString(serviceResponseBytes);

}

Figure 22: Sending a Token to a Service

When the service receives the token, it validates the signature to ensure it came from Access Control and hasn't been tampered with. It checks to see if the token has expired, and if it was issued for the correct resource. Once satisfied that the token is valid, the service can use the claims inside the token to authorize access to the resource and possibly to personalize the user's experience.

## Simplifying Authorization for REST Web Services on any Platform

The token format that Access Control issues is called Simple Web Token (SWT), and can be parsed and validated without needing any complicated libraries (you don't need WCF, WIF, or even the .NET Framework). As long as you can compute a hashed message authentication code (HMACSHA256), you're good to go. This can be done using the .NET Framework in about seven lines of code, and most other platforms have similar classes that make this easy.

If you can parse a query string you can parse this query format. Key value pairs are delimited by the '&' character, just as you'd expect, and values are URL-encoded. This allows you to send any type of claim you want, anything you can represent in a string can be sent in a claim, and processed by any service running on any platform or technology.

## Configuring Access Control

Access Control includes a management service that exposes an easy to use REST interface, as well as a command line tool, Acm.exe, that will help you get started quickly.

With these tools, you can manage the rules that Access Control uses to issue claims and the keys used to sign security tokens. You can manage trust relationships with identity providers, as well as name/password pairs for simple systems like the Hello Access Control example above.

## Cross Realm Federation

Figure 23 shows a more interesting scenario, where Access Control delegates responsibility for authentication to a corporate identity provider, Active Directory Federation Services (ADFS v2). This is more typical of how Access Control is designed to be used, as it grows to support more and more identity providers over time.



Figure 23: Access Control and ADFS - Cross-Realm Federation

## The Beauty of Claims, Manifested

The beauty of claims-based identity and Access Control is that *it allows your service to be loosely coupled to identity*. Indeed, if you consider all of the ways you can configure Access Control to allow users to access a service, from federating with ADFS v2, another Access Control provider, or some other provider of Simple Web Tokens, *none of these changes require you to redesign, recode, recompile, or even reconfigure your service*.

Also, consider how this system scales under load: *validating tokens is cheap*. Your service doesn't need to do any I/O to validate a token, I/O which can get expensive if you're hosted in the cloud. No user database lookups. No heavy crypto - HMACSHA256 is quite fast to compute. If your service gets hit with a bunch of unauthorized requests, it doesn't cost much to reject them quickly and to proceed with processing legitimate requests.

*For a deeper look at Access Control and some additional code samples, see A Developer’s Guide to Windows Azure platform Access Control (see Additional Resources).*

# Bringing it all Together

The Service Bus and Access Control constituents of the Windows Azure platform provide key building block services you’ll need when building cloud-based or cloud-aware applications. With the 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 Service Bus relay. The valuable connectivity fabric offered by the Service Bus accommodates a variety of messaging patterns including one-way messaging, request/response messaging, publish/subscribe (multicast), “buffered” messaging (message buffers), and others, all while overcoming network obstacles (firewalls, NATs, etc) found at Internet scope.

The Service Bus relies on Access Control for securing access to the relay. Access Control makes it possible to leverage a modern claims-based authentication and authorization model without requiring you to build this complex infrastructure yourself. The Service Bus trusts the claims produced by Access Control, which it can then process to determine if clients and services should be allowed to “send to” or “listen on” a particular Service Bus address. Clients and services must present credentials to Access Control in order to acquire a security token for the Service Bus. They can provide a variety of different credentials. Once authenticated, Access Control will issue the authorization token for the Service Bus relay. This is just one example of how the Access Control can be used in a particular application.

You can also use Access Control in conjunction with your own applications and services. Instead of writing your own custom user account and role database, you can let Access Control orchestrate the authentication and authorization of your users. With a single, simple code base in your service, you can authorize access to both enterprise clients as well as simple clients, through a variety of partners.

Enterprise clients can leverage ADFS v2 to allow users to authenticate using their Active Directory logon credentials. Simple clients can establish a shared secret with Access Control (essentially a very long password) to authenticate directly with Access Control at runtime. And as we mentioned earlier, in the future you'll see support for more and more identity providers such as Facebook Connect, Google Accounts, Windows Live ID, and others. Over time as Access Control evolves, you'll be able to authorize access to more and more users without having to recode your service.

Overall, the Service Bus and Access Control features provide valuable building blocks for cloud applications targeting the Windows Azure platform. There’s a great deal of synergy between them, and it’s likely you’ll come into contact with each feature when building a complete cloud-based application.

# Summary

The Windows Azure platform represents a comprehensive Microsoft strategy designed to make it easy for Microsoft developers to realize the opportunities inherent in cloud computing. The Service Bus and Access Control offer a key part of the platform strategy, designed specifically to aid .NET developers in making the move to the cloud. These services provide cloud-centric building blocks and infrastructure in the areas of secure application connectivity and federated access control. For more information on the Service Bus & Access Control, check out the in-depth whitepaper on each topic found below.

# Additional Resources

We’ve provided links to several resources below that will further your education on the Service Bus.

* An Introduction to Windows Azure platform AppFabric for Developers (*this paper*)
  + <http://go.microsoft.com/fwlink/?LinkID=150833>
* A Developer’s Guide to Service Bus in Windows Azure platform AppFabric
  + <http://go.microsoft.com/fwlink/?LinkID=150834>
* A Developer’s Guide to Access Control in Windows Azure platform AppFabric
  + <http://go.microsoft.com/fwlink/?LinkID=150835>
* Windows Azure platform
  + <http://www.microsoft.com/windowsazure/>
* Service Bus and Access Control portal
  + <http://netservices.azure.com/>

# About the Authors

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>.

Keith Brown is a cofounder of Pluralsight, a premier Microsoft .NET training provider offering both instructor-led and online training courses. Keith has authored books on Windows security, and spent eight years as the security columnist for MSJ and MSDN magazines. Keith has spent over a decade developing courses, speaking at conferences, and teaching developers about security. You can reach him at <http://www.pluralsight.com/keith> or at <http://twitter.com/keithbrown42>.

# Acknowledgements

Thanks to John Shewchuk and Dennis Pilarinos for their early vision that led to what’s now the Service Bus and Access Control. In addition, thanks to Justin Smith, Clemens Vasters, Anjan Das, David Wortendyke, Scot Gellock, Xuehai Zhang, Brandon Watson, and Derek Pai for their time and helpful feedback throughout the process.

1. The service namespace must be globally unique across all accounts and it must be at least 6 characters long. You may have to get a bit creative to find a service namespace domain that hasn’t already been used by someone else. [↑](#footnote-ref-1)
2. This terminology was used in the BizTalk Services documentation but is no longer an official term used by Microsoft. [↑](#footnote-ref-2)
3. RIA = Rich Internet Application [↑](#footnote-ref-3)
4. In the SAML world (Security Assertion Markup Language), the term *security assertion* is used instead of *claim*, but it essentially means the same thing. [↑](#footnote-ref-4)