Windows Azure Prescriptive Guidance

Larry Franks Sidney Higa Suren Machiraju
Christian Martinez Valery Mizonov Walter Myers III
Rama Ramani Jason Roth Mark Simms Paolo Salvatori
Adam Skewgar Ralph Squillace Patrick Wickline Trace Young

Guide

Microsoft

Windows Azure Prescriptive Guidance

Larry Franks, Sidney Higa, Suren Machiraju, Christian Martinez, Valery Mizonov, Walter Myers III, Rama Ramani, Jason Roth, Mark Simms, Paolo Salvatori, Adam Skewgar, Ralph Squillace, Patrick Wickline, Trace Young

Reviewers: Thiago Almeida, Jaime Alva Bravo, Monilee Atkinson, Brad Calder, Janice Choi, Brian Goldfarb, Sidney Higa, Michael Hyatt, YousefKhalidi, Christian Martinez, Paulette McKay, Shont Miller, Valery Mizonov, Sasha Nosov, James Podgorski, VladRomanenko, Ralph Squillace, Brian Swan, Tina Stewart, Michael Thomassy, Steve Wilkins, Steve Young

Summary: Windows Azure Prescriptive Guidance provides you with some of the best practices and recommendations for working with the Windows Azure platform. Categories cover planning, designing, developing, and managing a variety of different types of Windows Azure applications, including mobile, enterprise, and consumer-based applications. It also provides guidance for developers using non-.NET applications, libraries, and stacks with Windows Azure.

Category: Guide
Applies to: Windows Azure , Windows Azure SQL Database, Windows Azure Cloud Services, and Enterprise Integration Applications
Source: MSDN Library (link to source content)
E-book publication date: May 2012
422 pages

Microsoft

Contents

Welcome to the Windows Azure Prescriptive Guidance

Windows Azure Prescriptive Guidance provides you with some of the best practices and recommendations for working with the Windows Azure platform. Categories cover planning, designing, developing, and managing a variety of different types of Windows Azure applications, including mobile, enterprise and consumer-based applications. It also provides guidance for developers using non-.NET applications, libraries, and stacks with Windows Azure.

The scenarios described here are based on direct customer input to the Windows Azure Customer Advisory Team (CAT) and the developer documentation team. These teams will continue to collect new guidance over the coming weeks and months. The most current topics, including any updates to the topics in this book, are located at the Windows Azure Development Guidance. If there is a scenario or topic that you need best practices for, please contact us at azuredocs@microsoft.com.

Planning and Designing Windows Azure Applications

This section contains articles that provide guidance about planning and designing either Windows Azure applications themselves or their provisioning and management.

Is Your Application a Good Fit for Windows Azure?

Author: Jason Roth

Reviewers: Paulette McKay, Ralph Squillace, Sidney Higa, Brian Swan

If you're considering using Windows Azure to host an application, you might wonder if your application or business requirements are best served by the platform. This topic attempts to answer this question by:

- **Looking at the benefits Windows Azure provides to your application**
- Applying the strengths of the platform to common scenarios
- Rejecting scenarios that do not leverage the strengths of the platform
- Examining some common architecture and development considerations

The intent is to provide a framework for thinking about your application and how it relates to the capabilities of Windows Azure. In many cases, links to additional resources are provided to improve your ability to analyze your application and make a decision on how to move to the cloud.

Understand the Benefits of Windows Azure

Before you can determine if your application is well-suited for Windows Azure, you must first understand some of the main benefits of the platform. A complete list of benefits can be found in the Windows Azure documentation and many articles and videos about Windows Azure. One excellent paper on this subject is Cloud Optimization – Expanding Capabilities, while Aligning Computing and Business Needs.

There are several benefits to having hardware and infrastructure resources managed for you. Let's look at a few of these benefits at a high level before we discuss scenarios that take advantage of these features.

Resource Management

When you deploy your application and services to the cloud, Windows Azure provides the necessary virtual machines, network bandwidth, and other infrastructure resources. If machines go down for hardware updates or due to unexpected failures, new virtual machines are automatically located for your application.

Because you only pay for what you use, you can start off with a smaller investment rather than incurring the typical upfront costs required for an on-premises deployment. This can be especially useful for small companies. In an on-premises scenario, these organizations might not have the data center space, IT skills, or hardware skills necessary to successfully deploy their applications. The automatic infrastructure services provided by Windows Azure offer a low barrier of entry for application deployment and management.

Dynamic Scaling

Dynamic scaling refers to the capability to both scale out and scale back your application depending on resource requirements. This is also referred to as *elastic scale*. Before describing how this works, you should understand the basic architecture of a Windows Azure application. In Windows Azure, you create roles that work together to implement your application logic. For example, one web role could host the ASP.NET front-end of your application, and one or more worker roles could perform necessary background tasks. Each role is hosted on one or more virtual machines, called role instances, in the Windows Azure data center. Requests are load balanced across these instances. For more information about roles, see the paper The Windows Azure Programming Model.

If resource demands increase, new role instances running your application code can be provisioned to handle the load. When demand decreases, these instances can be removed so that you don't have to pay for unnecessary computing power. This is much different from an on-premises deployment where hardware must be over-provisioned to anticipate peak demands. This scaling does not happen automatically, but it is easily achieved through either the web portal or the Service Management API. The paper Dynamically Scaling an Application demonstrates one way to automatically scale Windows Azure applications. There is also an Autoscaling Application Block created by the Microsoft Patterns and Practices team.

If your application requires fluctuating or unpredictable demands for computing resources, Windows Azure allows you to easily adjust your resource utilization to match the load.

High Availability and Durability

Windows Azure provides a platform for highly available applications that can reliably store and access backend data through storage services or SQL Azure.

First Windows Azure ensures high availability of your compute resources when you have multiple instances of each role. Role instances are automatically monitored, so it is able to respond quickly to hardware restarts or failures by automatically deploying a role to a new instance.

Second, Windows Azure ensures high availability and durability for data stored through one of the storage services. Windows Azure storage services replicate all data to at least three different servers. Similarly, SQL Azure replicates all data to guarantee availability and durability.

Other Windows Azure services provide similar high availability guarantees. For more information, see the Windows Azure SLA.

Target Scenarios that Leverage the Strengths of Windows Azure

With an understanding of the strengths of the Windows Azure platform, you can begin to look at the scenarios that are best suited for the cloud. The following sections discuss several of these patterns and how Windows Azure is ideally suited for certain workloads and goals. The video Windows Azure Design Patterns explains many of the scenarios below and provides a good overview of the Windows Azure platform.

Tip

Although there is a focus on application scenarios here, understand that you can choose to use individual services of Windows Azure. For example, if you find that using blob

storage solves an application problem, it is possible that the rest of your application remains outside of the Cloud. This is called a *hybrid application* and is discussed later in this topic.

Highly Available Services

Windows Azure is well-suited to hosting highly available services. Consider an online store deployed in Windows Azure. Because an online store is a revenue generator, it is critical that it stay running. This is accomplished by the service monitoring and automatic instance management performed in the Windows Azure data center. The online store must also stay responsive to customer demand. This is accomplished by the elastic scaling ability of Windows Azure. During peak shopping times, new instances can come online to handle the increased usage. In addition, the online store must not lose orders or fail to completely process placed orders. Windows Azure storage and SQL Azure both provide highly available and durable storage options to hold the order details and state throughout the order lifecycle.

Periodic Workloads

Another good fit for Windows Azure is some form of an "on and off" workload. Some applications do not need to run continuously. One simple example of this is a demo or utility application that you want to make available only for several days or weeks. Windows Azure allows you to easily create, deploy, and share that application with the world. But once its purpose is accomplished, you can remove the application and you are only charged for the time it was deployed.

Note

Note: You must remove the deployment, not just suspend the application, to avoid charges for compute time.

Also consider a large company that runs complex data analysis of sales numbers at the end of each month. Although processing-intensive, the total time required to complete the analysis is at most two days. In an on-premises scenario, the servers required for this work would be underutilized for the majority of the month. In Windows Azure, the business would only pay for the time that the analysis application is running in the cloud. And assuming the architecture of the application is designed for parallel processing, the scale out features of Windows Azure could enable the company to create large numbers of worker role instances to complete more complex work in less time. In this example, you should use code or scripting to automatically deploy the application at the appropriate time each month.

Unpredictable Growth

All businesses have a goal of rapid and sustainable growth. But growth is very hard to handle in the traditional on-premises model. If the expected growth does not materialize, you've spent money maintaining underutilized hardware and infrastructure. But if growth happens more quickly than expected, you might be unable to handle the load, resulting in lost business and poor customer experience. For small companies, there might not even be enough initial capital to prepare for or keep up with rapid growth.

Windows Azure is ideal for handling this situation. Consider a small sports news web site that makes money from advertising. The amount of revenue is directly proportional to the amount of traffic that the site generates. In this example, initial capital for the venture is limited, and they do not have the money required to setup and run their own data center. By designing the web site to run on Windows Azure, they can easily deploy their solution as an ASP.NET application that uses a backend SQL Azure database for relational data and blob storage for pictures and videos. If the popularity of the web site grows dramatically, they can increase the number of web role instances for their front-end or increase the size of the SQL Azure database. The blob storage has built-in scalability features within Windows Azure. If business decreases, they can remove any unnecessary instances. Because their revenue is proportional to the traffic on the site, Windows Azure helps them to start small, grow fast, and reduce risk.

With Windows Azure, you have complete control to determine how aggressively to manage your computing costs. You could decide to use the Service Management API or the Autoscaling Application Block to create an automatic scaling engine that creates and removes instances based on custom rules. You could choose to vary the number of instances based on a predetermined amount, such as four instances during business hours versus two instances during non-business hours. Or you could keep the number of instances constant and only increase them manually through the web portal as demand increases over time. Windows Azure gives you the flexibility to make the decisions that are right for your business.

Workload Spikes

This is another workload pattern that requires elastic scale. Consider the previous example of a sports news web site. Even as their business is steadily growing, there is still the possibility of temporary spikes or bursts of activity. For example, if they are referenced by another popular news outlet, the numbers of visitors to their site could dramatically increase in a single day. In a more predictable scenario, major sporting events and sports championships will result in more activity on their site.

An alternative example is a service that processes daily reports at the end of the day. When the business day closes, each office sends in a report which is processed at the company headquarters. Since the process needs to be active only a few hours each day, it is also a candidate for elastic scaling and deployment.

Windows Azure is well suited for temporarily scaling out an application to handle spikes in load and then scaling back again after the event has passed.

Infrastructure Offloading

As demonstrated in the previous examples, many of the most common cloud scenarios take advantage of the elastic scale of Windows Azure. However, even applications with steady workload patterns can realize cost savings in Windows Azure. It is expensive to manage your own data center, especially when you consider the cost of energy, people-skills, hardware, software licensing, and facilities. It is also hard to understand how costs are tied to individual applications. In Windows Azure, the goal is to reduce total costs as well as to make those costs more transparent. The paper, Cloud Optimization – Expanding Capabilities, while Aligning Computing and Business Needs, does a great job of explaining typical on-premises hosting costs

and how these can be reduced with Windows Azure. Windows Azure also provides a pricing calculator for understanding specific costs and a TCO (Total Cost of Ownership) calculator for estimating the overall cost reduction that could occur by adopting Windows Azure. For links to these calculator tools and other pricing information, see the Windows Azure web site.

Scenarios that Do Not Require the Capabilities of Windows Azure

Not all applications should be moved to the cloud. Only applications that benefit from Windows Azure features should be moved to the cloud.

A good example of this would be a personal blog website intended for friends and family. A site like this might contain articles and photographs. Although you could use Windows Azure for this project, there are several reasons why Windows Azure is not the best choice. First, even though the site might only receive a few hits a day, one role instance would have to be continuously running to handles those few requests (note that two instances would be required to achieve the Windows Azure SLA for compute). In Windows Azure, the cost is based on the amount of time that each role instance has been deployed (this is known in Windows Azure nomenclature as *compute time*); suspending an application does not suspend the consumption of (and charge for) compute time. Even if this site responded to only one hit during the day, it would still be charged for 24 hours of compute time. In a sense, this is rented space on the virtual machine that is running your code. So, at the time of writing this topic, even one extra small instance of a web role would cost $30 a month. And if 20 GB of pictures were stored in blob storage, that storage plus transactions and bandwidth could add another $6 to the cost. The monthly cost of hosting this type of site on Windows Azure is higher than the cost of a simple web hosting solution from a third party. Most importantly, this type of web site does not require resource management, dynamic scaling, high availability, and durability.

Windows Azure allows you to choose only the options that are suited to your business' needs. For example, you might find instances in which certain data cannot be hosted in the cloud for legal or regulatory reasons. In these cases, you might consider a hybrid solution, where only certain data or specific parts of your application that are not as sensitive and need to be highly available are hosted in Windows Azure.

There are other scenarios that are not well-suited to Windows Azure. By understanding the strengths of Windows Azure, you can recognize applications or parts of an application that will not leverage these strengths. You can then more successfully develop the overall solution that most effectively utilizes Windows Azure capabilities.

Evaluate Architecture and Development

Of course, evaluating a move to Windows Azure involves more than just knowing that your application or business goals are well-suited for the cloud. It is also important to evaluate architectural and development characteristics of your existing or new application. A quick way to start this analysis is to use the Microsoft Assessment Tool (MAT) for Windows Azure. This tool asks questions to determine the types of issues you might have in moving to Windows Azure. Next to each question is a link called "See full consideration", which provides additional information about that specific area in Windows Azure. These questions and the additional

information can help to identify potential changes to the design of your existing or new application in the cloud.

In addition to the MAT tool, you should have a solid understanding of the basics of the Windows Azure platform. This includes an understanding of common design patterns for the platform. Start by reviewing the Windows Azure videos or reading some of the introductory white papers, such as The Windows Azure Programming Model. Then review the different services available in Windows Azure and consider how they could factor into your solution. For an overview of the Windows Azure services, see the MSDN documentation.

It is beyond the scope of this paper to cover all of the possible considerations and mitigations for Windows Azure solutions. However, the following table lists four common design considerations along with links to additional resources.

Area	Description
Hybrid Solutions	It can be difficult to move complex legacy applications to Windows Azure. There are also sometimes regulatory concerns with storing certain types of data in the cloud. However, it is possible to create hybrid solutions that connect services hosted by Windows Azure with on-premises applications and data. There are multiple Windows Azure technologies that support this capability, including **Service Bus**, **Access Control Service**, and Windows Azure Connect. For a good video on this subject from October 2010, see Connecting Cloud & On-Premises Apps with the Windows Azure Platform. For hybrid architecture guidance based on real-world customer implementations, see Hybrid Reference Implementation Using BizTalk Server, Windows Azure, Service Bus and SQL Azure.
State Management	If you are moving an existing application to Windows Azure, one of the biggest considerations is state management. Many on-premises applications store state locally on the hard drive. Other features, such as the default ASP.NET session state, use the memory of the local machine for state management. Although your roles have access to their virtual machine's local drive space and memory, Windows Azure load balances all requests across all role instances. In addition, your role instance could be taken down and moved at any time (for example, when the machine running the role instance requires an update). This dynamic management of running role instances is important for the scalability and availability features of Windows Azure. Consequently, application code in the cloud must be designed to store data and state remotely using services such as Windows Azure storage or SQL Azure. For more information about storage options, see the resources in the Store and Access Data section of the Windows Azure web site.

Area	Description
Storage Requirements	SQL Azure is the relational database solution in Windows Azure. If you currently use SQL Server, the transition to SQL Azure should be easier. If you are migrating from another type of database system, there are SQL Server Migration Assistants that can help with this process. For more information on migrating data to SQL Azure, see Data Migration to SQL Azure: Tools and Techniques. Also consider Windows Azure storage for durable, highly available, and scalable data storage. One good design pattern is to effectively combine the use of SQL Azure and Windows Azure storage tables, queues, and blobs. A common example is to use SQL Azure to store a pointer to a blob in Windows Azure storage rather than storing the large binary object in the database itself. This is both efficient and cost-effective. For a discussion of storage options, see the article on Data Storage Offerings on the Windows Azure Platform.
Interoperability	The easiest application to design or port to Windows Azure is a .NET application. The Windows Azure SDK and tools for Visual Studio greatly simplify the process of creating Windows Azure applications. But what if you are using open source software or third-party development languages and tools? The Windows Azure SDK uses a REST API that is interoperable with many other languages. Of course, there are some challenges to address depending on your technology. For some technologies, you can choose to use a stub .NET project in Visual Studio and overload the Run method for your role. Microsoft provides Windows Azure SDKs for Java and Node.js that you can use to develop and deploy applications. There are also community-created SDKs that interact with Windows Azure. A great resource in this area is the Ineroperability Bridges and Labs Center. Deploying projects that use open source software can also be a challenge. For example, the following blog post discusses options for deploying Ruby applications on Windows Azure: http://blogs.msdn.com/b/silverlining/archive/2011/08/29/deploying-ruby-java-python-and-node-js-applications-to-windows-azure.aspx. The important point is that Windows Azure is accessible from a variety of languages, so you should look into the options for your particular language of choice before determining whether the application is a good candidate for Windows Azure.

Beyond these issues, you can learn a lot about potential development challenges and solutions by reviewing content on migrating applications to Windows Azure. The Patterns and Practices group at Microsoft published the following guidance on migration: Moving Applications to the Cloud on the Microsoft Windows Azure Platform. You can find additional resources on migration from the Windows Azure web site: Migrate Services and Data.

Summary

Windows Azure offers a platform for creating and managing highly scalable and available services. You pay only for the resources that you require and then scale them up and down at any time. And you don't have to own the hardware or supporting infrastructure to do this. If your business can leverage the platform to increase agility, lower costs, or lower risk, then Windows Azure is a good fit for your application. After making this determination, you can then look at specific architecture and development options for using the platform. This includes decisions about new development, migration, or hybrid scenarios. At the end of this analysis, you should have the necessary information to make an informed decision about how to most effectively use Windows Azure to reach your business goals.

Designing Multitenant Applications on Windows Azure

Authors: Suren Machiraju and Ralph Squillace

Reviewers: Christian Martinez, James Podgorski, Valery Mizonov, and Michael Thomassy

This paper describes the approaches necessary to design multitenant applications (typically ISV applications that provide services for other organizations) for Windows Azure that are more efficient – that is, either less expensive to run or build, and/or more performant, robust, or scalable. The paper first describes the general principles of multitenant applications, the structure and behavior of the Windows Azure platform that is relevant to building and running multitenant applications, and then focuses on specific areas of multitenancy, especially as they apply to Windows Azure: application and data resources, as well as the provisioning and management of those resources. In each area of focus, this paper describes the security, cost, and scalability issues that you must balance to create a successful multitenant application on Windows Azure.

What is a Multitenant Application?

Wikipedia defines multitenancy in the following way.

> **Important**
> If you already know what we're talking about in this article, go ahead and start reading the Windows Azure Multitenant Service Overview to get straight to the guidance. You can return to this section later if there's something you didn't quite understand.

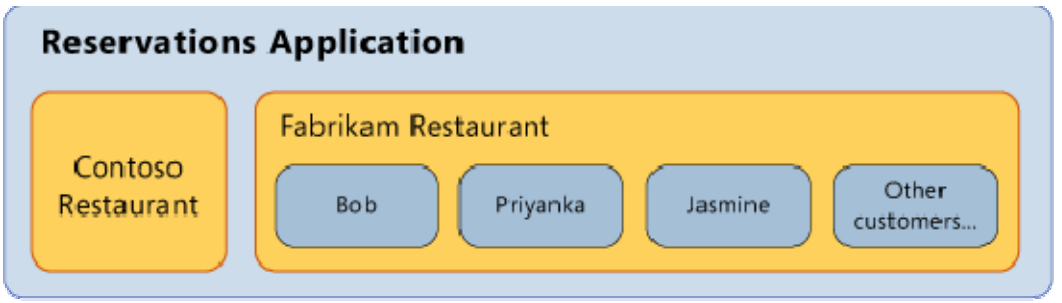
Wikipedia.org

What does this mean, simply put? Well, an example of this might be a restaurant reservation application. Restaurants themselves do not have the IT infrastructure to build a full-fledged internet reservation site that connects with the front desk in nearly real time to make or cancel reservations. (In most cases, restaurants hire other companies to build their static websites!) So you might create a single application that enables individual restaurants to sign up, create an account, customize the look of the web pages with their logos and colors, upload their menu, connect the web application with the computer at the front desk, handle reservation requests from other reservation applications, and so on.

In this example, each restaurant sees an application that they use, and that – once logged in – is only theirs. Their customers see only the web site of the restaurant, and for all they know, the restaurant built everything itself. (It's this added value to the hungry customer that the restaurant is paying the ISV for!)This might look something like the following diagram, in which the Reservations Application supports multiple tenants – in this case, the Contoso and Fabrikiam Restaurants. In this diagram, Bob, Priyanka, Jasmine, and other customers are pondering whether to make a reservation or not, perhaps by looking at the menu.

It might look like the following diagram.

The Reservations Application diagram.



Architecturally speaking, it is possible to build such an application by hosting a brand new instance of it for each customer with special customizations for them; perhaps each customer gets its own virtual machine (VM). This is common enough, but it has many problems for the ISV, almost all of which appear once your application becomes reasonably popular (we'll have a bit more to say on this later, but suffice it to say that this article does not focus on this approach, which is often referred to as a single-tenant, multi-instance design).

If you want your application to become popular among restaurants – and perhaps eventually among any company needing generic reservation support – you'll want to build one application that shares resources among tenants (in this case, restaurants) such that your maintenance costs and resource costs are dramatically reduced, resulting in better performance and lower cost as you increase workloads. Building the application this way is what Wikipedia refers to as a multitenant, single-instance application.

The Application Instance

It is worth noting that "instance" in this sense refers to the entire logical application, and not a VM, or an .exe, a process, or anything like that. In fact, the Reservations application diagrammed above is composed of many different external and internal "services" – components that communicate through web services.

Such web applications are typically built to work with "web farming" – in which groups of servers handle multiple instances of important components of the application by storing any application state data in remote storage while executing, so that if any one instance of an application vanishes for any reason, another instance can either recover or complete the process. In short, modern, hugely scalable web applications are decomposed into process components and storage components that, along with communication and security components, together make up a single, outwardly facing "instance" of the overall application.

The Rise of Multitenant Services

Multitenancy on Windows Azure means decomposing applications into stateless services and storage and managing data flow by identity where necessary: State is only introduced in those services that must deal directly with the tenants' identities. This follows the general service oriented application principle of Service Statelessness. For example, top-level services like web pages and public facing web services, or the security services themselves, of necessity must handle security claims and tokens directly. Other components and services that are part of the

larger application, however, should strive to be reusable by any other portion of the application acting on behalf of any tenant.

Tenants and Their Customers Do Not Care About Multitenancy

This section is here to emphasize one thing: Tenant companies and their customers do not care how your application is built. They care ONLY about how the application works: its robustness, performance, feature set, and cost to them. If they care about other things, they're just nosy geeks. (We shall concede there are indeed cases in which you may have to demonstrate through an audit of some kind that you really ARE building and running things correctly for security and especially legal or regulatory reasons. But these are only the exceptions that prove the rule; those audits are for compliance, not architecture per se.)

Windows Azure Multitenant Service Overview

The key problem designing a multitenant system is deciding upon the right balance between providing tenant isolation (on the one hand) and the cost of providing such dedicated resources. Said a different way, you must figure out just how much of the resources can be shared amongst your tenants so that your solution works properly but is as cost efficient as possible. Typically, the more a resource can be shared amongst many tenants, the more cost efficient that solution is – so long as it still satisfies the performance and security requirements.

Note

If at any time you're not clear about what multitenancy is or why it's important for Windows Azure applications, read What is a Multitenant Application?to get some context; then you can return here.

Within that larger problem, the major considerations center around:

- Security - Isolating stored data, authentication & authorization mechanisms
- Scaling - Auto-scaling platform compute, scaling platform storage
- Management and monitoring
- Provisioning tenant resources
- Metering and billing

Observe that devising a multitenant architecture is an involved process, with many moving parts and this article does not attempt to address every detail, but rather put together a perspective on the major considerations with as many commonly helpful tips and pointers to the details as possible. With those points in mind, let's start to explore the architecture from a high level.

High Level Architecture

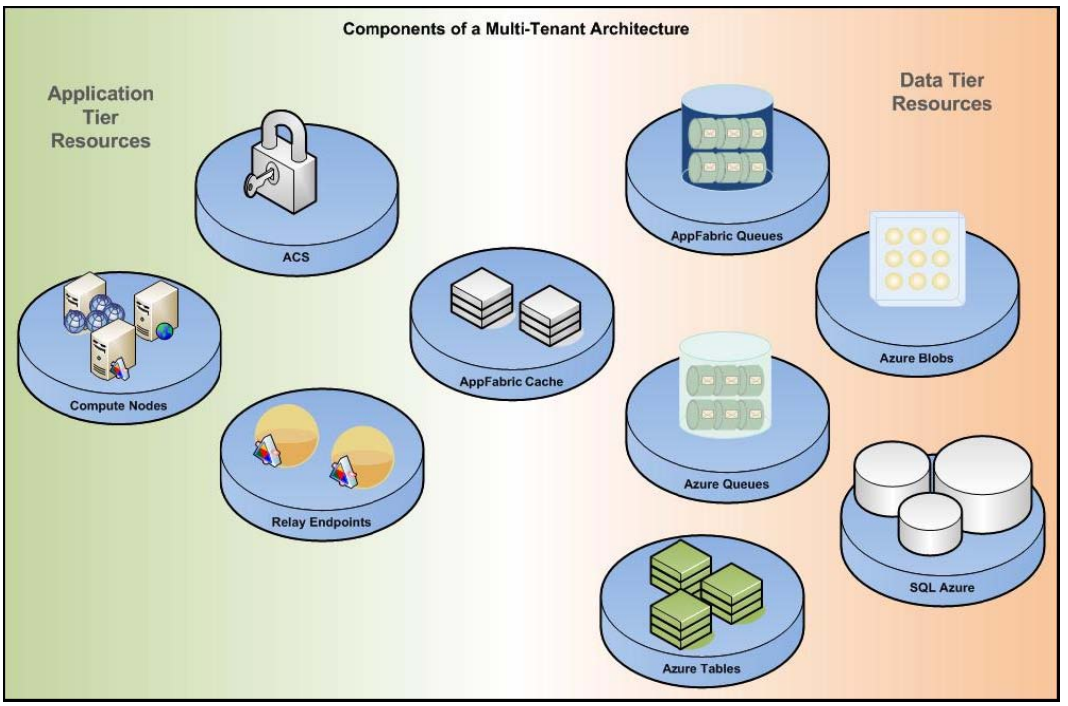
One approach to supporting multiple distinct customers is to reject the multitenant route completely and allocate resources for each customer on demand. When you use this approach – called single tenant -- you are automating the provisioning of dedicated resources for customers. If you imagine a spectrum of distributed application architectures, this approach represents completely tenant-dedicated resources on one side, with completely shared resources on the other. This paper attempts to describe how to design distributed applications that are closer to the

completely shared portion of this design spectrum: distributed applications with shared (or multitenant) application and data resources.

When you build a multitenant application, you need to weigh the costs of resources against the isolation you want to provide each customer. It's easy to think that you will need to consider which application and data resources are shared between customers and which are allocated exclusively to customers, but it's important to realize that being multitenant is not an all-or-nothing proposition when it comes to resource sharing. In a service-oriented application, you are able to – and most likely will -- mix and match which resources are shared between customers and which resources are exclusive to individual customers depending on your requirements. It's worth emphasizing this again: You can decide which application services can be shared, and which cannot be – and to what extent they are shared. This gives you a tremendous amount of design flexibility. Use it.

Which Resources Can Be Shared?

The figure below illustrates the Windows Azure resources under consideration, and we'll explore some of the decisions behind each of these as motivation. One helpful way to look at all the resources in the diagram (and multitenant architectures in general) is to take each component and define where it sits within the application or data tier.



In this diagram, Windows Azure compute and the Access Control Service are clearly application layer resources, as are relay endpoints, which serve to connect computing instances. Tables, Blobs, and SQL Azure are clearly data layer resources. Caching, however, along with queues, often serves as temporary storage that enables certain application designs and communication patterns, and as a result can be seen as both storage or as application layer resources, depending upon their usage.

Furthermore, exactly which approach you can take in any one case depends upon how many features you intend to expose to your tenant. For example, if you want to allow your tenant to offer a tenant-specific identity to its clients, you may want to have a general – and multitenant-safe – username/password security token service (STS). But you might also enable your tenant to implement one, and federate with Access Control Service instead. Another example is whether you intend to allow your tenant's customers to upload and execute code in the tenant's web or worker role instances. Doing so may well require a particular multitenant solution that mixes and matches multitenant and single-tenant role instances.

With these ideas in mind, let's consider how to successfully share the resource. Although we'll discuss in this and other topics the general approaches you can take to supporting multiple tenants in one resource from the point of view of an application, we'll also discuss how to provision and manage those resources, and how to use Azure to do so. Provisioning and managing resources that are shared does not pose great problems, but in some cases it's worth a closer look.

Application Resources

Designing a multitenant application means making sure that to the extent possible compute, data, and other application components handle multiple tenant identities and potentially each tenant's customer identities as well. This section discusses the issues involved in various approaches using Windows Azure and suggests the best approaches to handling them in your application design.

Compute

Let's begin by looking at the application resources provided by Windows Azure web and worker roles, and how they should be used in a multitenant solution to host web services, web sites, and other general processing.

The clearest use of Windows Azure web and worker roles in a multitenant application is to provide additional (or to reduce!) compute resources on demand in the service of the tenants, with the notion that a single role instance serves multiple tenants.

Segment Web site Tenants by Host Headers

In the most coarse-grained solution, the approach is to configure each web role instance to support multiple web sites and web applications, perhaps one for each tenant. This can be accomplished by modifying the Sites element in the Service Definition. Each website can either be bound to the same endpoint and use the HTTP/1.1 Host header to isolate them or they can be bound to their own endpoints. Note, however, that binding each web site to its own endpoint doesn't scale per tenant since Windows Azure limits you to 5 input endpoints per role instance).

If you use HTTP/1.1 Host headers to support multiple tenants, the tenants (and their customers) simply visit a different a URL (such as www.fabrikam.com for one and www.contoso.com for the other tenant) and get the website for the desired tenant, even though they are in fact to talking to a single endpoint within a single web role instance. This approach allows you to isolate different web site files by tenant with minimal effort. As an additional benefit, this approach enables you to isolate tenants by application pool because by default each web site gets its own application pool.

Note, however, that because the URLs that provide the HTTP/1.1 Host headers are defined in the domain name system (DNS), you will need to configure the DNS with your domain provider and point it to your *.cloudapp.net address; remember that it may take 24 hours for your new tenant URL to be fully discoverable on the internet. For a great example of how one instance can host multiple websites differentiated by host headers, check out the Windows Azure Accelerator for Web Roles.

SSL Communication

One cautionary note to using the HTTP/1.1 Host headers approach is that Windows Azure does not enforce a security boundary between these websites; this means that SSL certificates used by one website are useable from the other websites. To complicate matters, IIS 7 does not support having multiple websites with their own certificate if only their host headers are different.

Important

It is possible to have multiple SSL Web sites that use unique server certificates if each Web site uses a separate IP address or port for its HTTPS binding. As in IIS 6.0, IIS 7.0 does not support having multiple Web sites with their own server certificates if the HTTPS bindings for those Web sites are on the same IP address/port and differentiate only by using host headers. This is because the host header information is not available when the SSL negotiation occurs. Because of this, the only way to have multiple Web sites on the same IP address that use host headers is to configure all of those Web sites to use the same SSL certificate with a wildcard CN. For more information, see http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/596b9108-b1a7-494d-885d-f8941b07554c.mspx?mfr=true.

Therefore, if you are using SSL certificates, you may need to take a different approach. One is to use a single certificate with a wildcard CN (of the form *.yourdomain.com) or a Unified Communications Certificate (where you list all the subdomains explicitly). These types of certificates are available from certificate authorities like GoDaddy and Thawte and are straightforward to create. Each tenant would then access resources over SSL using URL's like https://fabrikam.yourdomain.com or https://contoso.yourdomain.com.

The alternative is to have each website live on a dedicated web role and only the one tenant certificate. As this is not a multitenant approach, it does not scale as well and therefore absorbs more compute resources as well as management overhead. It nevertheless must sometimes be done.

Segment Website Tenants by Query Parameters

Another approach is to use a single website to serve resources for multiple tenants. If your website is built using ASP.NET MVC, you could create a separate MVC area for each tenant. Alternatively, you share the default area between them and instead have the controllers render differently based on query parameters (such as a tenant id). If you want to provide process isolation by tenant, then you need to define virtual applications within the website. Effectively, each tenant gets a virtual application underneath the website. This is also accomplished by editing the Sites element within the Service Definition, but instead of adding a new Site for each tenant, within your main Site element you add one VirtualApplication element for each tenant.

Web Services in Worker Roles

For fully multitenant solutions, worker roles hosting web services behave the same way as their relatives hosted in a web role—all tenants access the same service endpoints.

Worker Roles are also used to provide background processing. In the fully multitenant scenario, the approach you take is to have a worker role whose RoleEntryPoint.Run method loop processes work items agnostic of which tenant created it -- hence you are sharing the compute resources across all tenants.

Storage

Multitenant applications must handle how the data they store and use is isolated and yet performs well. This section discusses data storage technologies in Windows Azure, including SQL Azure databases; Windows Azure blobs, tables, and queues; Service Bus queues and topics, and the Caching service in a multitenant application.

SQL Azure storage is clearly an important location in acting as the storage for an application, but there use in a multitenant architecture extends to acting as storage for provisioning new tenants and for storing management data.

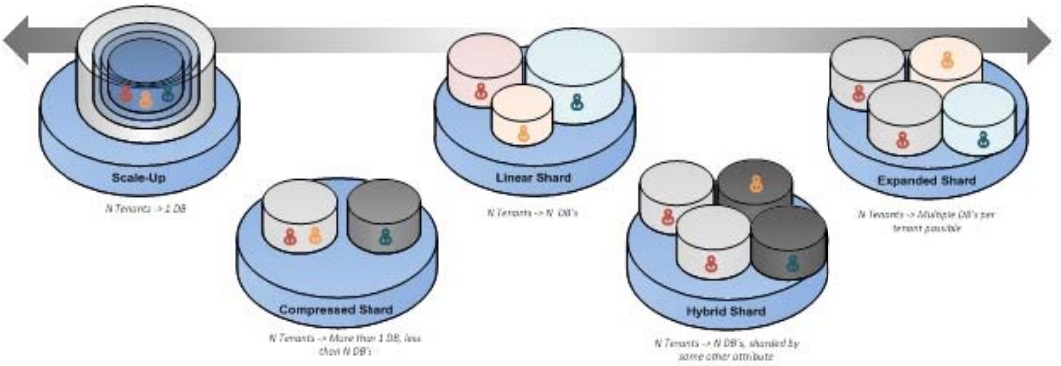
Using SQL Azure for Application Resources

In this scenario application data for multiple tenants are stored within a single logical database. While there are obvious cost benefits to delivering a multitenant database (realized by splitting the cost across all tenants), these come with the tradeoff of increased complexity in dealing with various requirements specific to this scenario:

- Adequately isolating data and preventing one tenant from inadvertently or maliciously accessing another tenant's data.
- Maintaining reasonable query performance for all tenants, in addition to limiting the effect of a single, especially resource-hungry tenant.
- Handling the rapidly increasing size of a shared database, which may exceed the maximum 50 GB database size that SQL Azure currently supports.

Conceptually, there is a performance benefit gained when distributing tenant data and query processing across multiple databases—the question becomes, what architectures allow you to leverage this for your solution? Before we answer that question, it's is helpful to look at the architectures on a spectrum. On one end of this spectrum is the traditional scale-up, in which you simply throw bigger hardware at the problem. As with the single-tenant architectural approach, we won't focus on scaling-up except where it helps understanding, because it is too limiting for multitenant solutions. We focus on the various scale-out approaches using sharding techniques:

- Linear Shard
- Expanded Shard
- Compressed Shard and
- Hybrid Linear/Expanded Shard architectures.



All of the preceding four useful sharding techniques add databases to the architecture to support scale-out, and for each there is some logical separation between the data stored (often referred to as a data domain or data federation) that defines the shard. These data domains can be defined using certain key values (such as a customer id) or an attribute (such as the year). For the sake of simplicity in this article, and in the context of multitenant architectures, we take the perspective that the data domain is the tenant.

Returning to the spectrum diagram, on one end of the spectrum we have the traditional scale-Up model. In this approach all tenants share the same SQL Azure database. In other words we have multiple tenants per database. In the middle of the spectrum diagram, above, we have the linear shard architecture, in which each tenant gets its own SQL Azure database. On the far end of the spectrum, each tenant may get multiple SQL Azure databases, which is referred to as the expanded shard approach.

The other two approaches address in-between scenarios. In a compressed shard architecture there are fewer SQL Azure databases than tenants, but more than one SQL Azure database. Similarly, scenarios where there are multiple data domains with which sharding decisions are made (such as a region and tenant), then we have Hybrid Linear/Expanded Shard architecture where each region gets a particular expanded shard each shard having multiple databases, and a particular tenant's data is spread across multiple databases in that expanded shard. In other words, the architecture scales out linearly by region, but in an expanded fashion by tenant.These sharding approaches necessitate a need for a layer of abstraction, like the Enzo SQL Shard Library, that knows how to navigate the sharding approach implemented, and in particular assists with fanning out queries to all shards and guiding data modifications operations to the appropriate shard.SQL Azure Federations coming out the second half of 2011 promises to simplify the implementation of sharding architectures implemented on SQL Azure, but let's consider what steps you can take to implement these architectures today.

Securing Multitenant Data in SQL Azure

We begin by looking at how to secure your tenants data. In its core form, managing security at the database level for SQL Azure is almost identical to that of an on-premise SQL Server, amounting to creating security principals and using them to control access to specific schema objects (schemas, tables, views, stored procedures, and so on). Then you create the appropriate per-tenant SQL Server logins, Database Users, and Database Roles, and apply permissions to schema objects using these security principals. One approach is to define one schema per tenant – security applied to schemas restrict access to just that tenant.

Another approach is to store all tenant data within the same set of tables, separating them within the tables by some key (such as a tenant id column), thereby creating a logical separation. In this approach logic in the business tier is responsible for correctly filtering data accesses by tenant (albeit SQL Azure Federations can also perform this task). The crux of this approach is enforcing that all requests for data must go through this logic in order to limit access to the correct tenants; OData (using WCF Data Services that use Query and Change Interceptors) is a great technology with which to implement this approach. There are even some frameworks, like Enzo (discussed later) that relieve you from having to implement this logic.

With this in mind, let's apply this to the scale-out architectures previously described. Security in the Linear Shard architecture can be pretty simple: each tenant gets its own database, and as such gets its own dedicated logins and users specifically for that shard. In all of the other sharding architectures, security needs to be more granular and relies on a combination of permissions defined at the schema object level and enforcement at the business tier.

Performance Considerations

Next, let's turn our focus to performance. SQL Azure maintains three replicas of any SQL Azure database. While data for each database is replicated between one primary and two secondary replicas, all reads and writes take place on the primary replica- the secondary replicas only help out with query processing in a fail over scenario. The SQL Azure fabric determines the location of the primary and two secondary replicas. If a machine hosting your primary replica is under heavy load (due to your own tenants or other tenants), the SQL Azure fabric may switch the primary for a secondary on another machine that has a lighter work load. This switch happens quickly, but does result in the disconnection of active sessions—a situation which your multitenant application must be prepared to handle.

With this in mind, if per-tenant performance is a top priority, then you should consider a sharding architecture that allocates one or more databases per tenant (such as the Linear, Expanded or Hybrid architectures). However, if you have a lot of tenants for which you allocate a database this way, you run into a few obstacles. Obviously, each database you create has a cost, but the obstacle to be aware of is the limit of 149 databases per SQL Azure Server, which can be raised if you call Cloud Support Services (CSS). For multitenant scenarios with lots of databases, your solution also needs to be able to allocate SQL Azure Servers as this limit is approached.

Using Windows Azure Tables for Application Resources

Windows Azure Tables are secured using a storage account name and key, which can only be allocated globally for all storage resources. This means that to provide isolated storage of per-tenant data the service provider will have to create a storage account for each tenant. This can be performed through the Windows Azure Service Management API. There is no additional cost to allocating new storage accounts and there is a performance benefit to be gained by doing so because different resources can be applied to different accounts. That said, there are quotas governing the number of storage accounts a subscription can have, which can be adjusted by contacting Cloud Services Support (CSS).

If your application needs to store multiple tenants' worth of data within a single account, you have to create your own abstraction to filter the data returned to a given tenant. One approach is to create one table per tenant, and authorize access at a per table grain. This approach is beneficial in that you get to leverage the full partition key and row key to quickly access the data, with a

fairly simple logic for routing queries to the correct table. The downside is the extra complexity in that you have to route queries to the appropriate table based on the tenant. Another approach is to partition horizontally by tenant within the table, where your data access layer only returns data having a certain partition key to the client. This eliminates the need to juggle routing queries to the appropriate table, but limits you to a single partition key value, which in turn can affect your performance on very large per-tenant datasets.

Using Windows Azure Blobs for Application Resources

When it comes to the storage needs of the application, Windows Azure Blobs offer a great deal for multitenant applications. You should create public blob containers for read-only access to shared data like website images, graphics and tenant logos. Private blob containers (only accessible by the tenants or the tenants and the system) would be used for application data like documents or per-tenant configuration files. Outside of anonymous access, they key approach is to be sure to specify a container-level access policy for containers or blobs secured using Shared Access Signatures for various actions such as blob read/write, block list, properties, or metadata, deleting a blob, leasing a blob and enumerating blobs within a container. By specifying a container level access policy, you can the ability to adjust permissions without having to issue new URL's for the resources protected with shared access signatures.

Windows Azure Queues for Application Resources

Windows Azure queues are commonly used to drive processing on behalf of tenants, but may also be used to distribute work required for provisioning or management.

Here, the consideration is whether a given queue manages items belonging to multiple tenants, or if each queue is dedicated to a single tenant. The Windows Azure Storage model works at the storage account level only, therefore if queues need to be isolated by tenant, they need to be created under separate storage accounts just as for tables. Moreover, in the case of a single queue, it's not possible to define permissions restricting use to only put or only receive messages—once a tenant has access with the storage account the tenant can do everything allowed by the Azure Queue API, even delete the shared queue! In short, the queues should not be exposed to the tenant, but rather managed by the service automatically.

Service Bus Queues for Application Resources

For tenant specific application functionality that pushes work to a shared a service, you can use a single queue where each tenant sender only has permissions (as derived from claims issued from ACS) to push to that queue, while only the receivers from the service have permission to pull from the queue the data coming from multiple tenants. The reverse direction of this is possible with AF Queues, you can have system services push messages destined for specific tenant receivers across a single queue. You can also multicast these messages by using the Subscription functionality of AF Queues. Finally, you can use Session Messages to drive customer specific work to customer-specific receivers, albeit only pushing through a single, shared queue.

While these approaches minimize the number of queues you have to manage (which to a certain extent reduces your overall system complexity), it can limit the throughput of enqueue or dequeue operations because the load is not partitioned amongst more resources (as would be the case when using multiple queues). Also, in the case of multiple tenant senders, you must be careful not to be at the mercy of a single tenant who floods the queue and effectively starves other

tenants of downstream processing. With AF Queues you can defer messages, so it is also possible to detect such a flood coming from a single tenant, defer those messages temporarily to process messages from other tenants and later return to processing them.

Cache Service for Application Resources

Within a multitenant application, the Cache is traditionally used for frequently accessed tenant-specific application data. As it does not provide the ability to set distinct permissions within a single cache, the pattern is to provision a distinct cache per tenant. This implies provisioning a new AppFabric namespace for the tenant and creating the cache within that. At present, there is no management API to automate Cache creation and this still needs to be done by human operator using the Windows Azure Portal.

Connection and Security Services

Windows Azure multitenant applications use other “middleware” services for connectivity and security: Service Bus relay services and the Windows Azure Access Control Service (ACS).

Using ACS for Application Resources

Windows Azure Access Control Service

read about it here

Whether you are building a multitenant system or not, ACS is the way to secure access to the application itself. Using ACS enables your application to be built using one authentication API (the ACS API) that can process security tokens from any identity provider. This means you don’t write separate code each for Facebook, Google, Yahoo, Windows Live, a different third-party identity provider, or Active Directory Federation Server (ADFS). Instead, you only write code for ACS, and let ACS and the identity providers do the heavy lifting for you.

Provisioning Identities in ACS

The details of configuring ACS or implementing your own STS are beyond the scope of this document, but you can get started thinking about such things in Authorization in Claims-Aware Web Applications and Services. To get the fundamentals of the approach used to outsource security from your application via federated security, see this great guide. To get a start on building your own, you might want to get the background on how to build one using Windows Identity Foundation (WIF) from here, but you will likely benefit by starting with a fully fleshed out STS like Thinktecture’s Starter STS, and customizing it to suite your needs.

Within a multitenant application that uses ACS, identities are usually provisioned with the following high-level steps:

- Creating certificates (for example, per tenant)
- ACS namespace provisioning (if isolating by namespace), including the configuration of tenant’s application that is to be secured -- called a relying party (RP) -- and claims transformation rules. This is done using ACS management APIs or the ACS Portal.
- Creating root administrator account for tenant to login with (using API of an identity providing security token service).

Using Service Bus Relay for Application Resources

The services that are exposed as endpoints may belong to the tenant (for example, hosted outside of the system, such as on-premise), or they may be services provisioned specifically for the tenant (because sensitive, tenant-specific data travels across them). In either scenario, handling multiple tenants is really not the issue; enforcing tenant-specific usage is. Access to these endpoints is secured using Access Control Service (ACS), where clients must present an Issuer Name and Key, a SAML token, or a Simple Web Token. This can be configured programmatically using the Service Bus and ACS management APIs.

Note

Remember that Service Bus queues and topics and subscriptions are discussed as storage, though they often serve to handle application data flow in a particular way.

Provisioning Resources

This section discusses the provisioning of resources in a way that supports multitenant applications. As with supporting more than one tenant in application design, designing multitenant provisioning also has several decision points, depending upon the features you intend to enable and what Windows Azure services your application uses.

As with the design of multitenant applications, we'll discuss how compute, storage, and connectivity and security services are used to provision multitenant applications.

Provisioning and Managing Resources Using Azure Roles

A dedicated worker role in multitenant solution is typically used to provision and de-provision per tenant resources (such as when a new tenant signs-up or cancels), collecting metrics for metering use, and managing scale by following a certain schedule or in response to the crossing of thresholds of key performance indicators. This same role may also be used to push out updates and upgrades to the solution.

A web role is typically dedicated to enable the service provider to monitor and manage system resources, view logs, performance counters, and provision manually, and so on.

Using ACS for Provisioning Resources

When provisioning tenant resources protected by ACS, you will need to use the ACS management APIs or portal to create the initial admin" account for newly provisioned tenants.

Using Cache Service for Provisioning Resources

When provisioning tenant resources protected by ACS, you will need to use the ACS management APIs or portal to create the initial admin" account for newly provisioned tenants.

Considerations for Provisioning Storage

One consideration that applies to Windows Azure Tables, blobs, and SQL Azure in a multitenant solution is geo-distribution. Specifically it is identifying data that must be unique system-wide (across all data centers used in the solution) and balancing that against maintaining a performant

user experience. One option is to build a custom replication strategy to bring such shared data near to end users (which naturally has to ensure that new data is unique, perhaps by always inserting to the master data source). The other option is to partition the data, such that the amount and frequency of access of global data is minimized.

Another consideration for Azure Storage in general is the hard limits imposed by Azure, which albeit large are not infinite. When planning your multitenant solution, you should take these scalability limits into account.

Using SQL Azure for Provisioning Resources

In some multitenant scenarios with large amount of data involved, it is best to provision new SQL Azure databases by copying from an existing SQL Azure reference instance. The enhanced provisioning speed provided by this must be weighed against the cost of maintaining an extra SQL Azure database on top of those required by tenants and the system itself.

Provisioning SQL Azure Resources

The options for provisioning new SQL Azure resources for a tenant include:

- Use DDL in scripts or embedded as resources within assemblies
- Create SQL Server 2008 R2 DAC Packages and deploy them using the API's. You can also deploy from a DAC package in Windows Azure blob storage to SQL Azure, as shown in this example.
- Copy from a master reference database
- Use database Import and Export to provision new databases from a file.

Provisioning Windows Azure BLOB Storage

The approach for provisioning BLOB storage is to first create the container(s), then to each apply the policy and create and apply Shared Access Keys to protected containers and blobs.

Using Windows Azure Blobs for Provisioning Resources

When it comes to provisioning compute or pre-initialized storage resources for new tenants, Azure blob storage should be secured using the container level access policy (as described above) to protect the CS Packages, VHD images and other resources.

Management Resources

Finally, the design of a multitenant application must tackle the extremely important task of managing the application, tenants and their services, all the data resources, and any connectivity or security issues they entail. This section discusses common uses of compute, data, and other services to support multitenant applications while running Windows Azure.

Using Windows Azure Roles for Management Resources

The service provider will need a way to monitor and manage the system resources. A web role is typically dedicated to provide the service provider with tools to manage resources, view logs, performance counters, and provision manually, etc.

Using ACS for Management Resources

Most multitenant systems will require a namespace created within ACS that is used to secure system resources, as well as the creation and management of individual namespaces per tenant (such as for using the AF Cache). This is also accomplished using the ACS management namespace.

Using Cache Service for Management Resources

If the service provider exposes certain KPI's or computed statistics to all tenants, it may decide to cache these often requested values. The tenants themselves do not get direct access to the cache, and must go through some intermediary (such as a WCF service) that retains the actual authorization key and URL for access to the Cache.

Using SQL Azure for Management Resources

Examples of these are single, system wide and datacenter agnostic membership/roles database for non-federating tenants or those relying on a custom IP STS configured for use with ACS. For multitenant systems concerned with multiple geographic distributions, the challenge of a centralized system for management data surfaces. To solve these problems you can take the approach previously described for application resources, either by defining your geographical regions as shards in Hybrid Linear/Expanded Shard architecture or a more simple Linear Shard architecture. In both cases, leveraging middle-tier logic to fan out and aggregate results for your monitoring and management queries.

Using Windows Azure Tables for Management Resources

The Windows Azure Diagnostics (WAD) infrastructure by default logs to Windows Azure Tables. When relying on these WAD tables (Trace, Event Logs and Performance Counters) you need to consider just how sensitive the logged data may, who has access to them and ultimately choose if they are isolated (aka provisioned) between customers or shared system-wide. For example, it's unlikely that you would provide all tenants direct access to the diagnostic log tables which aggregates traces from all instances across the solution and might expose one tenant's data to another tenant.

Using Windows Azure Blobs for Management Resources

The canonical management resources stored in blob storage are IIS Logs and crash dumps. IIS Logs are transferred from role instances to blob storage. If your system monitoring relies on the IIS Logs, you will want to ensure that only the system has access to these logs via a container level access policy. If your tenants require some of this data, you will want to perform some post processing on the data (perhaps to ensure that only that tenant's data is included) and then push

the results out to tenants via a tenant specific container. Crash dumps, on the other hand are something only the service provider system should have access to as they assist in troubleshooting the infrastructure and will very likely contain data that spans tenants.

Metering

Within the context of multitenant applications, metering is motivated by a desire to bill charges to tenants that are influenced by usage as well as to collect system-wide KPI's to inform capacity planning and on-going architecture decisions. What is metered? Typically it boils down to these:

- Raw resource consumption: Azure and AppFabric resource use (e.g., compute hours, data storage size, number of messages)
- Specific use of applications features (for example, premium operations charged per use)
- Use by tenants own users

The latter two tend to be driven by application specific requirements, so we will focus on the raw resource consumption that is common to all Azure based service providers. From a raw resource perspective, for most SaaS applications, compute hours is by far the most significant, followed by storage size and to a lesser degree, data transfer out from the data centers (egress) -- particularly as data transfers into Azure datacenters is now free.

So how can you get this data for metering on raw compute or storage? Let's explore some examples.

Compute Hours

Unfortunately there is no API at present to query for this by the service provider's subscription. The best approach is to approximate usage in line with how Windows Azure compute time is billed. For example, for each instance allocated, compute the hours of instance uptime, rounding up to the nearest hour multiplied by the hourly cost for the instance size.

Data Storage

Again, there is no public billing or management API for Windows Azure Storage (Blobs, Tables, and to a lesser extent Queues) or Cache and Service Bus Queues) that provides exact usage, but one can extrapolate by knowing the size of the entities being stored and tracking the number of entities stored by the tenant on average by the tenant over the billing period. SQL Azure database size is the exception; You can determine the database size that for which you will be billed that day by querying sys.database_usage within the master database and aggregating the result over the month to get at the actual cost.

Scaling Compute for Multitenant Solutions

While specific requirements vary, as a general theme when "auto-scaling" is considered the approach amounts to increasing or decreasing the instance count according to some heuristic. This heuristic may depend on some key performance indicator (KPI) derived from sources such as performance logs, IIS logs or even queue depth. Alternately, it may simply be implemented in response to a schedule, such as incrementing for a month end burst typical in accounting applications, and decrementing the instance count at other times.

The key factor to consider here is that scaling the instance count up or down is not instantaneous. Your algorithms should incorporate this in two ways. When scaling up, it's best if you can anticipate the demand-- it doesn't have to be a long range forecast (but on the order of 20 minutes) to allow for your instances to become available. When scaling down, recall that you pay for partial hour used as a complete hour, so it makes economic sense to keep those unneeded instances around for that full hour.

Conclusion and Resources

Frameworks and Samples

Having read through the above, you probably agree that building a multitenant solution is a big investment. From this perspective, starting from a sample or framework is a great idea. Here are some great starting points.

Microsoft Enterprise Library 5.0 Integration Pack for Windows Azure

Microsoft Enterprise Library is a collection of reusable application blocks that address common cross-cutting concerns in enterprise software development. The Microsoft Enterprise Library Integration Pack for Windows Azure is an extension to Microsoft Enterprise Library 5.0 that can be used with the Windows Azure technology platform. It includes the Autoscaling Application Block, Transient Fault Handling Application Block, blob configuration source, protected configuration provider, and learning materials. This application block is a great place to start.

CloudNinja

The CloudNinja Sample available on CodePlex demonstrates the following aspects of multi-tenancy as discussed in this article.

- Multitenant Web Roles
- Dedicated Management/Provisioning Worker Roles
- Linear Sharding in SQL Azure
- Dedicated Windows Azure Tables for Management
- Dedicated Azure Queues for Provisioning
- Dedicated Public/Private Blob Storage
- Multitenant AppFabric Cache
- Time & KPI Based Auto-Scaling
- Metering
- Federated Security with a Custom STS and ACS

Fabrikam Shipping

The Fabrikam Shipping Sample sample available on Codeplex demonstrates many of the aspects of a multitenant SaaS offering.

- Dedicated & Multitenant Web Roles

- Dedicated Management/Provisioning Worker Roles
- Linear Sharding in SQL Azure
- Dedicated Public/Private Blob Storage
- Federated Security with a Custom STS, ADFS, Facebook, Google, Live ID and ACS
- PayPal integration

Enzo SQL Shard Library

The Enzo SQL Shard Library available on CodePlex demonstrates and assists with the numerous SQL Azure sharding techniques discussed in this article.

Lokad Cloud

Lokad Cloud is a feature rich framework that provides lots of the functionality as described in this article and a lot more.

- Auto-scaling
- Strongly typed Azure Blob, Table and Queue storage (they call this an Object to Cloud mapper).
- Task scheduler
- Logging

Azure Auto Scaling

If you're looking for a simple, straightforward example from which to build your own auto-scaling, the Azure Auto Scaling sample available on CodePlex is worth a look.Of course, there are also some commercial offerings available to help with some of the more difficult aspects. In particular, Paraleap AzureWatch can help you auto-scale your application.

Links and References

Multitenant Architecture

- Developing Applications for the Cloud on the Microsoft Windows Azure™ Platform
  - Hosting a Multi-Tenant Application on Windows Azure
  - Building a Scalable, Multi-Tenant Application for Windows Azure
- Architecture Strategies for Catching the Long Tail
- Multi-tenancy in the cloud: Why it matters
- The Force.com Multitenant Architecture: Understanding the Design of Salesforce.com's Internet Application Development Platform
- Patterns For High Availability, Scalability, And Computing Power With Windows Azure

Related Topics

- GeekSpeak: Autoscaling Azure
- Performance-Based Scaling in Windows Azure
- Accounts and Billing in SQL Azure
- Security Resources for Windows Azure
- How to Configure a Web Role for Multiple Web Sites

- How to Configure a Web Role for Multiple Web Sites
- Federations: Building Scalable, Elastic, and Multi-tenant Database Solutions with SQL Azure
- Managing Databases and Logins in SQL Azure
- Inside SQL Azure

Packaging and Deploying an Application to Windows Azure

Authors: Larry Franks, Rama Ramani

This document provides guidance on deploying an application to a Windows Azure hosted service. It also provides guidance on working with other Windows Azure services that your application may use, such as SQL Azure and Windows Azure Storage.

Before deploying an application to Windows Azure, you should have an understanding of the following:

- Differences between the Windows Azure Emulator and Windows Azure, SQL Azure, and Windows Azure Storage
- How to configure:
  - Connection strings for Windows Azure Storage Services and SQL Azure
  - Endpoints
  - Role size
  - Number of instances
- How to create an affinity group
- Microsoft's SLA requirements for hosted services
- Development and Production environments for hosted services
- How to deploy an application using the Windows Azure Management Portal

Differences between Windows Azure and the Windows Azure Emulator

The Windows Azure SDK installs the Windows Azure Emulator, which emulates the Windows Azure hosting and Storage services. Before deploying an application to Windows Azure, you should first perform testing in the Windows Azure Emulator. While the emulator provides an easy way to test a hosted application during development, it cannot fully emulate all aspects of the Windows Azure platform. For example, the connection string used to connect to Windows Azure Storage differs between the Windows Azure Emulator and Windows Azure. Before deploying an application to Windows Azure, you should understand the differences between the emulator and Windows Azure and ensure that your application does not rely on a behavior of the emulator that is not present in the Windows Azure environment.

For more information on the differences between the emulator and the Windows Azure platform, see Overview of the Windows Azure SDK Tools.

SQL Azure and Other Windows Azure Services

While the Windows Azure Emulator provides a local testing solution for hosted services and storage, it does not provide any development equivalent for SQL Azure, the Caching Service, Service Bus, and other services provided by the Windows Azure Platform.

For database design and testing, you can use SQL Server; however there are differences between SQL Server and SQL Azure that you must be aware of. For a comparison, see Compare SQL Server with SQL Azure.

If your solution is developed against SQL Server, you should consider whether you will recreate your databases and associated artifacts in SQL Azure or migrate your SQL Server development environment to SQL Azure. For information on migration options, see Migrating Databases to SQL Azure.

For other services such as Caching or Service Bus, you must develop against the live Windows Azure service.

Pre-Deployment Checklist

The following items should be verified before deploying an application to Windows Azure:

Item to check	Description
Number of instances	At least two instances must be created to meet the Windows Azure Compute Service Level Agreement (SLA) requirements. For more information on Windows Azure SLAs, see Service Level Agreements.
Connection strings	All connection strings should be checked to ensure they do not reference development storage
Virtual machine size	The virtual machine size governs available memory, local storage, processor cores, and bandwidth for your application. For more information, see How to Configure Virtual Machine Sizes.
Endpoints	Endpoints determine the ports used for communications with your hosted services, and whether the port is public or for internal use only. For more information, see How to Enable Role Communication.
Affinity group	To ensure that you are deploying to the correct datacenter, you should consider creating an affinity group for your project and use this when provisioning services or deploying to the Windows Azure platform. If you do not use an affinity group, you may accidentally deploy services to different datacenters, which can impact performance and increase costs. For more information, see How to Create an

Item to check	Description
	Affinity Group.
Certificates	If you wish to enable SSL communications, or remote desktop functionality for your hosted service, you must obtain and deploy a certificate to Windows Azure. For more information, see How to Add a Certificate to the Certificate Store and Using Remote Desktop with Windows Azure.
Co-Administrators	Ensure that the co-administrators for your Windows Azure subscription contain the appropriate people. For more information, see How to Add and Remove Co-Administrators for your Windows Azure Subscription.
Upgrade planning	You should familiarize yourself with the information in the Post-Deployment section of this article before deployment, as part of designing your Windows Azure based solution is creating an upgrade plan.

Deployment

There are three primary methods of deploying an application to Windows Azure. The deployment methods, and the tools used to perform each type of deployment are described in the following table:

Deployment Method	Tool	Requirements
Web based	Windows Azure Management Portal	Browser support for Silverlight
Integrated Development Environment (IDE)	Visual Studio 2010 and the Windows Azure SDK	Visual Studio 2010
Command Line	The Windows Azure SDK.	Command line tools for deployment are provided as part of the Windows Azure SDK

For more information on packaging and deploying an application to Windows Azure, see the following links:

- **.NET Languages:**Deploying Applications in Windows Azure
- **Node.js:**Windows Azure PowerShell for Node.js Cmdlet Reference

- **PHP:**Packaging and Deploying PHP Applications for Windows Azure
- **Java:**Creating a Hello World Application Using the Windows Azure Plugin for Eclipse with Java

Post-Deployment

If you perform a change to an existing deployment, such as updating a service configuration setting, upgrading the application, or updating a certificate, this will cause the application instances to restart. Also, while most changes to a deployment can be made as in place updates to the existing service, some changes may require you to delete and then redeploy the hosted service.

For more information on updating an existing deployment, see Overview of Updating a Windows Azure Service.

For more information on the actions that will cause a restart of the hosted service and how to minimize the impact of these actions, see Improving Application Availability in Windows Azure.

Note

You are charged for deployments even if they are not running. To ensure that you are not charged for resources you are not actively using, ensure that you delete any inactive deployments.

Note

If you are performing tests that involve creating additional instances of your application, verify that the number of instances is reduced to the normal number after tests have completed.

Note

If you have configured your deployment to allow remote desktop connections, ensure that you enable this functionality in the Windows Azure Management Portal only when needed. Also, If you save the RDP file used for the connection to your local system, you may be unable to use it to connect to Windows Azure after updating the deployment. If this is the case, download a new RDP file from the Windows Azure Management Portal.

See Also

Developing Windows Azure Applications

Planning and Designing Windows Azure Applications

Testing, Managing, Monitoring and Optimizing Windows Azure Applications

Best Practices for Running Unmanaged Code in Windows Azure Applications

Writing .NET code for Windows Azure applications is for the most part just like writing .NET code for Windows applications. There are subtle differences to be aware of when writing .NET code for one platform versus the other. This document provides recommendations for running unmanaged/native code in Windows Azure Applications.

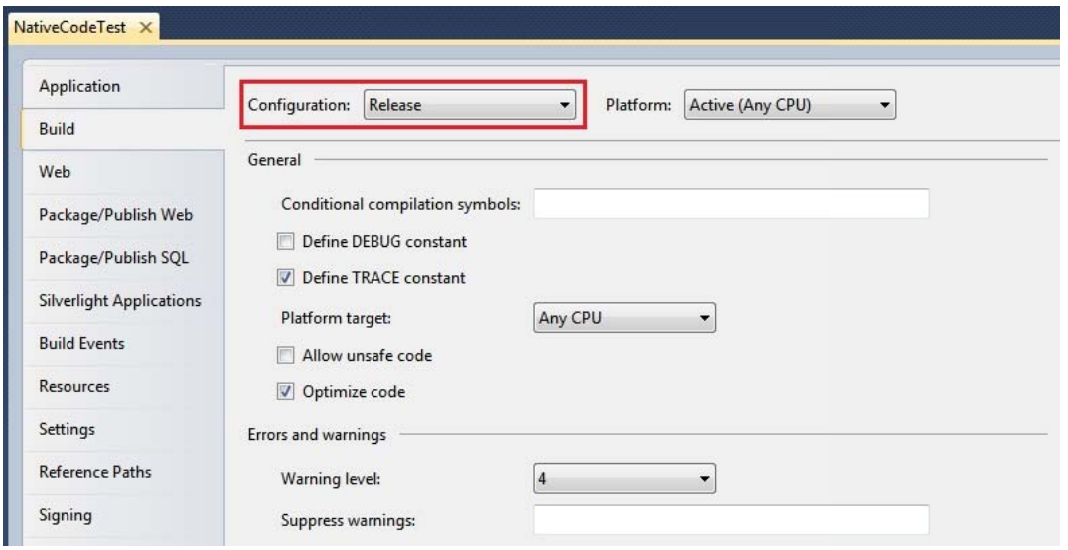
Authors Christian Martinez, Trace Young, and Mark Simms

Best Practices for Running Native Code from Windows Azure Applications

The following sections provide recommendations for ensuring that native code runs correctly when developing Windows Azure applications that call native code.

Verify that the Native Code you deploy with your Windows Azure Application is Compiled in Release Mode

To configure native code to compile in Release mode right-click on the native code project, select **Properties** to display the **Properties** page, and select the **Release** configuration option available from the **Build** tab of the Properties page:



Note

Runtime errors stating that you are missing msvcr100d.dll or msvcp100d.dll indicate that the native code you deployed was compiled in Debug mode. The files msvcr100d.dll and msvcp100d.dll are not redistributable DLLs. For more information about determining which C++ DLL files to redistribute see Determining Which DLLs to Redistribute (http://go.microsoft.com/fwlink/p/?LinkId=236016) in the C++ 2010 documentation.

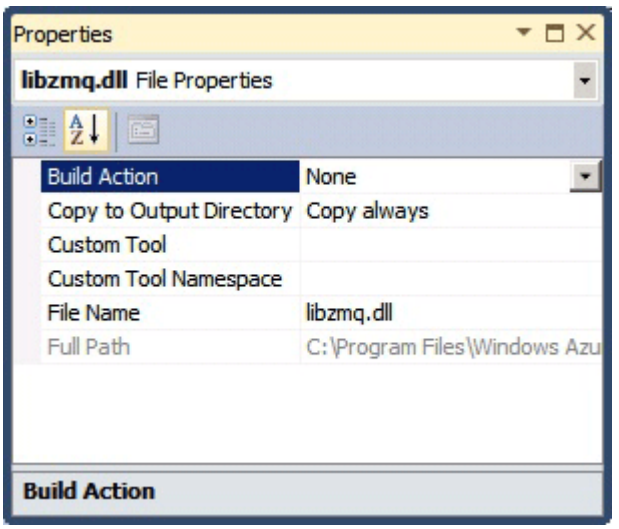
Ensure that Native Code can be Located by your Windows Azure Application when Running in Windows Azure Compute Emulator

Follow these steps to ensure that your Windows Azure Application can locate any referenced native code when running in Windows Azure Compute Emulator:

1. **Set Appropriate Properties for Compiled Native Code Files in Visual Studio**

   Include the compiled native code file as a project item and on the **Properties** dialog for the file set the **Copy to Output Directory** option to **Copy always** and the **Build Action** option to **None**.

   This will copy the compiled native code file into the \bin directory and ensure that your Windows Azure Application can locate the compiled native code file when running in Windows Azure Compute Emulator.



2. **When troubleshooting your RoleEntry implementation in the Windows Azure Compute Emulator it can make it easier to debug issues if you copy the Compiled Native Code File to the devfabric runtime directory**

   - For example, when using Windows Azure SDK version 1.5 or earlier copy the compiled native code file to the following directory:

     ```
     C:\Program Files\Windows Azure SDK\[SDK Version]\bin\devfabric\x64\
     ```

   - Or, when using Windows Azure SDK version 1.6 or later copy the compiled native code file to this directory:

     ```
     C:\Program Files\Windows Azure SDK\[SDK Version]\bin\runtimes\base\x64\
     ```

3. **Ensure that Windows Azure Applications Running in a Web Role Instance can Locate any Referenced Native Code**

   If a Windows Azure Application references native code that is wrapped using C++/CLI and the Windows Azure Application is running in a Web Role instance, use one of the following

methods to ensure that the Windows Azure Application can locate the referenced native code:

Note

The steps below reference the **CppCliWebRole** project provided with the sample code in PinvokeCppCliInAzure.zip, available for download at http://azureunmanagedcode.codeplex.com/. For more information about the sample code see Sample Code: Running Native Code from Windows Azure Applications.

Method 1: Edit the PATH Environment Variable, then Restart Windows Azure Compute Emulator and IIS:

a. Stop and exit Windows Azure Compute Emulator.
b. Edit your PATH environment variable to point to a directory containing the native compiled code.
c. Type `iisreset` from an elevated command prompt.
d. Press **F5** to run the sample code.

Method 2: Use a startup command

In the steps below, the native code is in the file **ExampleNativeCode.dll**.

a. Stop and exit Windows Azure Compute Emulator.
b. Open the indist.cmd file included with the **CppCliWebRole** project.
c. Change the following line:

```
REM copy "%~dps0ExampleNativeCode.dll"
"%windir%\system32\inetsrv"
```

To:

```
copy "%~dps0ExampleNativeCode.dll" "%windir%\system32\inetsrv"
```

d. Save the project.
e. Press **F5** to run the sample code.

Note

The use of a startup command works for actual deployments as well. If you prefer avoiding references to the IIS system directory then you might alternatively create and run a script to:

a. Change the PATH environment variable to point to the directory containing the native compiled code.
b. Restart IIS and any processes that depend on it.

Ensure that the Compiled Native Code is 64-bit

Windows Azure is a 64-bit platform, as are the Windows Azure application (worker and web role) hosts. If you are not using a pure 64-bit development environment and your application references native code, your application will throw errors. One simple test to verify that all of the

native code you are referencing is 64-bit is by testing the native code using console test applications that are hard coded to run as 64 bit applications.

Use the Native Multi-targeting Feature of Visual Studio 2010 to Build Native Libraries with the Visual Studio 2008 Platform Toolset

By default, only the Visual C++ runtime libraries for Visual C++ 2008 are installed on Windows Azure worker and web roles. Therefore, native code compiled against the Visual C++ runtime library for Visual C++ 2010 or Visual C++ 2005 will fail to load in worker and web role instances. If you have both Visual Studio 2008 and Visual Studio 2010 installed on the same computer you can use the native multi-targeting feature of Visual Studio 2010 to build native libraries for your application with the Visual Studio 2008 platform toolset (compiler, linker, headers, and libraries). For more information about using Visual Studio 2010 to build an application with the Visual Studio 2008 platform toolset see C++ Native Multi-Targeting (http://go.microsoft.com/fwlink/p/?LinkId=231170).

Add an Elevated Startup Task to your Worker or Web Role Project to Install the Native Visual C++ Libraries for Visual C++ 2010 on Worker/Web Role Instances

If building native libraries with the Visual Studio 2008 platform toolset is not a viable option, consider adding an elevated startup task to your worker or web role project to copy and install the required version of the runtime libraries. The following steps describe how to create an elevated startup task to copy the 64-bit version of the Microsoft Visual C++ 2010 Redistributable Package to a worker / web role and run the redistributable package to install the Visual C++ libraries for Visual C++ 2010 onto the worker / web role instances:

1. Create a **Startup** folder for your worker or web role project.
2. Copy vcredist_x64.exe (http://go.microsoft.com/fwlink/p/?LinkId=225987) to the Startup folder.
3. Create a startup.cmd file in the Startup folder.
4. Edit startup.cmd and add the following line:

```
"%~dps0vcredist_x64.exe" /q /norestart
```

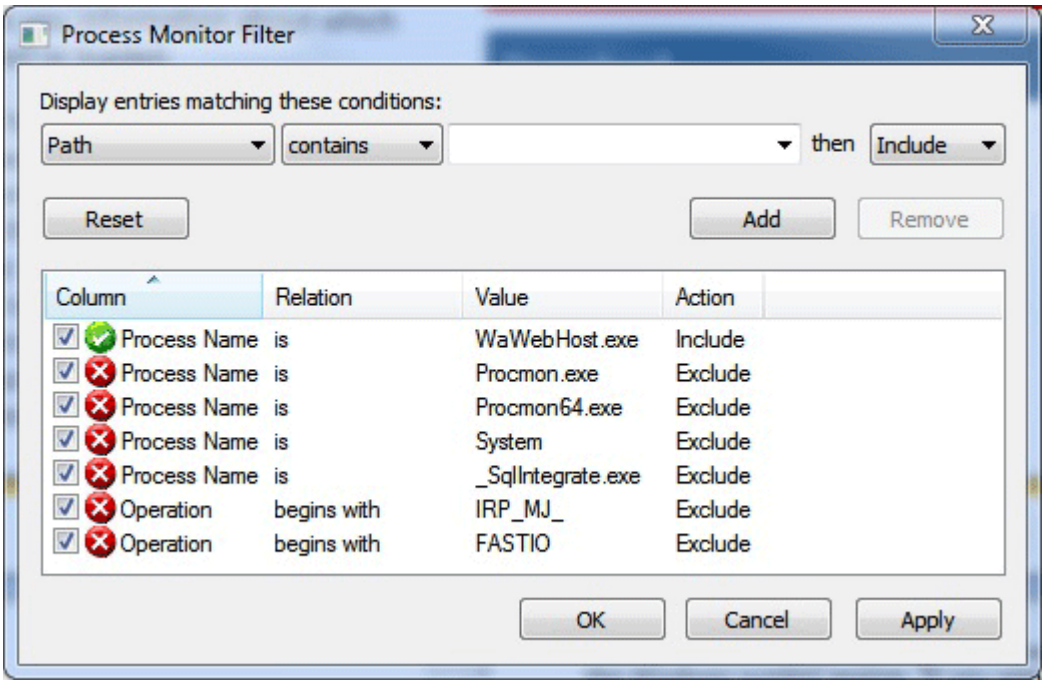
5. Modify the properties of **vcredit_x64.exe** and **startup.cmd** from Visual Studio Solution Explorer. Set the **Build Action** option to **Content** and the **Copy to Output Directory** option to **Copy if Newer**.
6. Edit the **ServiceDefinition.csdef** file for the role by adding the following elevated startup task to execute startup.cmd:

```
< Task commandLine ="Startup\Startup.cmd" executionContext ="elevated" taskType
="simple" />
```

Use Process Monitor to Troubleshoot Problems Loading Files in Windows Azure Compute Emulator

The error message generated when an assembly cannot be loaded by Windows Azure Computer Emulator may be non-intuitive. To troubleshoot problems loading files in a Worker or Web Role host instance use Process Monitor as follows:

1. Download Process Monitor from Process Monitor v2.96 (http://go.microsoft.com/fwlink/p/?LinkID=137175).
2. Launch Process Monitor to troubleshoot problems with Windows Azure Applications loading files when running in Windows Azure Compute Emulator.
3. Set filter parameters as depicted below for the Windows Azure Compute Emulator host. If troubleshooting problems with Windows Azure Applications running in a worker role project change the filter to display entries with the process name **WaWorkerHost.exe** rather than **WaWebHost.exe**.



4. Look for any **NAME_NOT_FOUND** messages. This will help to isolate the missing library file. Once you have determined which file is missing you can narrow the scope of the search by applying a filter to isolate only messages related to that file.

Configure Worker/Web Roles with the .NET Full Trust Level

To enable Windows Azure applications to execute 64-bit native code using P/Invoke, first configure the associated worker or web roles with the .NET Full Trust level. For more information about calling native code from applications running in Windows Azure worker or web roles see the following resources:

- Windows Azure Role Architecture (http://go.microsoft.com/fwlink/p/?LinkId=236012)
- Windows Azure and Native Code, Hands-On Lab (http://go.microsoft.com/fwlink/p/?LinkID=231343)
- Windows Azure and Native Code Hands-On Lab (http://go.microsoft.com/fwlink/p/?LinkId=236013).Provides the contents of the Windows Azure and Native Code, Hands-On Lab in a single Word document.

Sample Code: Running Native Code from Windows Azure Applications

This section describes the sample code in PinvokeCppCliInAzure.zip (http://go.microsoft.com/fwlink/p/?LinkId=236170), available for download at http://azureunmanagedcode.codeplex.com/

Note

When using Windows Azure Compute Emulator from Visual Studio to work through the sample code, it is not necessary to run the portion of the startup command that installs the Visual C++ 2010 runtime libraries, therefore you should comment out the following line in the startup.cmd file:

```
REM "%~dps0vcredist_x64.exe" /q /norestart
```

The sample code contains a project which creates a native DLL called ExampleNativeCode.dll. It has been compiled as a 64 bit library in Release mode as recommended.

Note

Exit and restart Windows Azure Compute Emulator each time you run any of the sample code if any changes are made to the code or to Windows environment variables. This will ensure that any referenced native code is released from memory so that it can be recompiled and that any environment variable changes are picked up by Windows Azure Compute Emulator the next time it runs.

The sample code ExampleNativeCode.dll is wrapped using P/Invoke in a project called **ManagedUsingPinvoke** and wrapped using C++/CLI in a project called **ManagedUsingCppCLI**.

The code in the native DLL is a modified version of the default Win32 project template with exports and contains a single function that answers all known questions in the universe, as long as the answer for said questions is the number 42:

```
EXAMPLENATIVECODE_API int fnExampleNativeCode(void)
{
    return 42;
}
```

The P/Invoke code and the C++/CLI code that utilize this function are very similar:

P/Invoke

```
public class Class1
{
    [DllImport("ExampleNativeCode.dll")]
```

```
    static extern int fnExampleNativeCode();

    public int GetTheAnswerToEverything()
    {
        return fnExampleNativeCode();
    }
}
```

C++/CLI

```
public ref class Class1
{
public:

    int GetTheAnswerToEverything()
    {
        return fnExampleNativeCode();
    }
};
```

Both Worker Role samples (**PInvokeWorkerRole** and **CppCliWorkerRole**) in the solution invoke the code in the same manner:

```
try
{
    Trace.WriteLine(new Class1().GetTheAnswerToEverything());
}
catch (Exception ex)
{
    Trace.WriteLine(ex);
}
```

Both Worker Role samples include the native compiled code and are configured to always copy the native compiled code to the output directory.

When you run press **F5** to execute the code in Windows Azure Compute Emulator you should see the following output:

```
c79eb3f0ab234cb0994e2c5279914089 after the agent is
initialized.
[MonAgentHost] Output: Registered as an event consumer
[MonAgentHost] Output: Agent will exit when parent pro
24672 exits.
[MonAgentHost] Output: Monitoring Agent Started
[Diagnostics]: Starting configuration channel polling
[runtime] Role entrypoint . CALLING   OnStart()
[runtime] Role entrypoint . COMPLETED OnStart()
[runtime] Role entrypoint . CALLING   Run()
Information: $projectname$ entry point called
42
Information: Working
```

Certain considerations apply when calling native code from Windows Azure Applications running in a Web Role instance as compared to running in a Worker Role instance. The **PInvokeWebRole** and **CppCliWebRole** projects contain the following code in a slightly modified version of the default ASP.NET project template:

P/Invoke

```
<p>
    This is PInvoke <br />
<%=Environment.GetEnvironmentVariable("PATH") %><br />
<%=new Class1().GetTheAnswerToEverything() %>
</p>
```

C++/CLI

```
<p>
    This is C++/CLI <br />
<%=Environment.GetEnvironmentVariable("PATH") %><br />
<%=new Class1().GetTheAnswerToEverything() %>
</p>
```

Running the PInvokeWebRole project will result in the expected output similar to the following:

```
This is PInvoke
C:\Program Files\Common Files\Microsoft Shared\Microsoft Online Services;C:\Program Files (x86)\Common Files\Microsoft Shared\Microsoft Online
Services;C:\Program Files\Common Files\Microsoft Shared\Windows Live;C:\Program Files (x86)\Common Files\Microsoft Shared\Windows Live;C:\Ruby192
\bin;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Program Files\System Center
Operations Manager 2007\;C:\Program Files (x86)\Microsoft Application Virtualization Client;C:\Program Files\WIDCOMM\Bluetooth Software\;C:\Program
Files\WIDCOMM\Bluetooth Software\syswow64;C:\Program Files (x86)\Microsoft SQL Server\100\Tools\Binn\;C:\Program Files\Microsoft SQL Server\100
\Tools\Binn\;C:\Program Files\Microsoft SQL Server\100\DTS\Binn\;C:\Program Files (x86)\Microsoft SQL Server\100\Tools\Binn\VSShell\Common7
\IDE\;C:\Program Files (x86)\Microsoft Visual Studio 9.0\Common7\IDE\PrivateAssemblies\;C:\Program Files (x86)\Microsoft SQL Server\100\DTS\Binn\;C:\Program
Files\Microsoft Windows Performance Toolkit\;C:\Program Files (x86)\Windows Live\Shared;C:\Program Files\TortoiseSVN\bin;C:\Program Files (x86)
\Git\cmd;c:\Program Files (x86)\Microsoft ASP.NET\ASP.NET Web Pages\v1.0\;C:\Program Files\SlikSvn\bin\;C:\Program Files\Windows
Imaging\;C:\software\PHPAzure-4.0.3\bin;C:\Program Files (x86)\PHP\v5.3;
42
```

However, running the CppCliWebRole project without modification will result in the following error:

```
Server Error in '/' Application.

Could not load file or assembly 'ManagedUsingCppCli.dll' or one of its dependencies. The specified module could not be found.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.IO.FileNotFoundException: Could not load file or assembly 'ManagedUsingCppCli.dll' or one of its dependencies. The specified module could not be found.

Source Error:
```

Note

This error occurs even with the proper project settings applied to copy the native compiled code to the output directory.

To address this error, use one of the methods described in [Ensure that Windows Azure Applications Running in a Web Role Instance can Locate any Referenced Native Code](). After using one of these methods you should see the expected output for the CppCliWebRole page, similar to the following:

```
This is C++/CLI
C:\Program Files\Common Files\Microsoft Shared\Microsoft Online Services;C:\Program Files (x86)\Common Files\Microsoft Shared\Microsoft Online
Services;C:\Program Files\Common Files\Microsoft Shared\Windows Live;C:\Program Files (x86)\Common Files\Microsoft Shared\Windows Live;C:\Ruby192
\bin;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Program Files\System Center
Operations Manager 2007\;C:\Program Files (x86)\Microsoft Application Virtualization Client;C:\Program Files\WIDCOMM\Bluetooth Software\;C:\Program
Files\WIDCOMM\Bluetooth Software\syswow64;C:\Program Files (x86)\Microsoft SQL Server\100\Tools\Binn\;C:\Program Files\Microsoft SQL Server\100
\Tools\Binn\;C:\Program Files\Microsoft SQL Server\100\DTS\Binn\;C:\Program Files (x86)\Microsoft SQL Server\100\Tools\Binn\VSShell\Common7
\IDE\;C:\Program Files (x86)\Microsoft Visual Studio 9.0\Common7\IDE\PrivateAssemblies\;C:\Program Files (x86)\Microsoft SQL Server\100\DTS\Binn\;C:\Program
Files\Microsoft Windows Performance Toolkit\;C:\Program Files (x86)\Windows Live\Shared;C:\Program Files\TortoiseSVN\bin;C:\Program Files (x86)
\Git\cmd;c:\Program Files (x86)\Microsoft ASP.NET\ASP.NET Web Pages\v1.0\;C:\Program Files\SlikSvn\bin\;C:\Program Files\Windows
Imaging\;C:\software\PHPAzure-4.0.3\bin;C:\Program Files (x86)\PHP\v5.3;
42
```

Other Third Party Software on Windows Azure

This section contains topics that address or describe how to use various types of third-party software on Windows Azure. In many cases, the topics describe how to use a library or software stack; in others, how best to do so.

In This Section

Related Sections

Planning and Designing Windows Azure Applications

Testing, Managing, Monitoring and Optimizing Windows Azure Applications

Node.js Developer Guidance (Windows Azure)

PHP Developer Guidance (Windows Azure)

Developing Windows Azure Applications

Leveraging Node.js' libuv in Windows Azure

Author: Christian Martinez

Node.js is definitely one of the hottest things going right now and you can find blogs about running it in Windows Azure such as this one by Nathan Totten. While that is certainly interesting and cool, the subject of this post was inspired by this post on the Node.js blog by Node's creator Ryan Dahl. More specifically, it was this quote that got me interested:

Since Node is totally non-blocking, libuv turns out to be a rather useful library itself: a BSD-licensed, minimal, high-performance, cross-platform networking library.

I am always interested in cool networking libraries with my current favorite being 0MQ (http://www.zeromq.org/) which is a full featured, multiple MEP supporting, high performance, general purpose messaging library. Libuv is a very different beast from 0MQ and has different goals but both projects share the common bond of being open source and of making performance a first class citizen.

In order to get started on my adventure I downloaded the 5.7 version of the libuv source and attempted to run the vcbuild.bat file. After that tanked, I downloaded python, then subversion and checked out the gyp script and.. anyways these shenanigans could probably be a blog in and of themselves but for now just know that what ended up being generated was a VS 2010 solution that I was then able to tweak to generate a 64 bit static library. Since libuv and Node are highly active projects it's likely that things have changed since I went through my exercise so if you take this on yourself your experience may or may not be different.

Once I had the library, I had to decide how I wanted to deploy things to Azure. Libuv is native C code and comes with a test suite that I could have theoretically deployed using the SDK 1.5ProgramEntryPoint element. But, that would have been boring and probably not seen as too useful by developers who wanted to use the library with their own managed code so what I did instead was develop a prototype in C++/CLI that allowed me to create a server that runs in a WorkerRole and looks like this:

```
public override void Run()
{
     long cnt =0;
     long byteCnt = 0;
     using (var server = new TcpServer())
     {
        var ep = RoleEnvironment.CurrentRoleInstance.InstanceEndpoints["listen"];
        //must assign callback before calling listen
        server.OnBytesReceived += (arg) =>
         {
          Interlocked.Increment(ref cnt);
          Interlocked.Add(ref byteCnt, arg.Length);
          //Trace.WriteLine(cnt);
         };
        server.Listen(ep.IPEndpoint.Address.ToString(),
                          ep.IPEndpoint.Port);
       //if this is successful libuv uv_run blocks exit
      }
}
```

That should look very normal to any managed developer and while all I do in the prototype is increment bytes, and number of calls received you're free to do any interesting thing you choose in the exposed delegate.

So, how did I get from straight C code to a nice managed experience? Well, the first thing I had to do was to decide if I wanted to use P/Invoke or C++/CLI. After perusing the libuv source and realizing the whole thing was callback driven, I decided to go with C++/CLI. While pointer to member function syntax is always a joy to deal with, I still prefer dealing with that to managing the DLLImport and marshalling and mapping one does using P/Invoke. This resource was a great help in figuring out the callback syntax. Essentially what you end up needing is a **Delegate** declaration, a **Delegate** instance variable and a **GCHandle**. My header file ended up looking like this:

```
// LibuvWrapper.h
#pragma once
#include "include\uv.h"
using namespace System;
using namespace System::Collections::Concurrent;
using namespace System::Runtime::InteropServices;
```

```
namespace LibuvWrapper {

    public delegate void BytesReceived(array<Byte>^ stuff);

    public ref class TcpServer sealed
    {
        public:

            TcpServer();
            ~TcpServer();
            !TcpServer();
            void Listen(String^ address, int port);
            //public delegate for passing bits sent
            //from client onto implementation code
            BytesReceived^ OnBytesReceived;
            private:
            //delegate declarations. to convert to native callbacks you
            //need the delegate declaration, delegate instance,
            //gchandle and preferably a typedef
            delegate void AfterRead(uv_stream_t* handle, ssize_t nread,
                                                    uv_buf_t buf);
            delegate void OnConnection(uv_stream_t* server,
                                                    int status);
            delegate void AfterWrite(uv_write_t* req, int status);
            delegate void OnClose(uv_handle_t* peer);
            delegate void OnServerClose(uv_handle_t* handle);

            //callback methods
            void AfterReadImpl(uv_stream_t* handle, ssize_t nread,
                                            uv_buf_t buf);
            void OnConnectionImpl(uv_stream_t* server, int status);
            void AfterWriteImpl(uv_write_t* req, int status);
            void OnCloseImpl(uv_handle_t* peer);
            void OnServerCloseImpl(uv_handle_t* handle);


            //handles and delegate members that will be
            //converted to native callbacks
            AfterRead^ afterReadFunc;
            GCHandle afterReadHandle;
            OnConnection^ onConnectionFunc;
```

```
            GCHandle onConnectionHandle;
            AfterWrite^ afterWriteFunc;
            GCHandle afterWriteHandle;
            OnClose^ onCloseFunc;
            GCHandle onCloseHandle;
            OnServerClose^ onServerCloseFunc;
            GCHandle onServerCloseHandle;


            //libuv members
            uv_tcp_t * server;
            uv_loop_t * loop;
            uv_buf_t * ack;
            //this is the same response sent to all
            //clients as ack. created once and never
            //deleted until object destruction
    };
}
```

Wiring up the delegates looks like this:

```
afterReadFunc = gcnew AfterRead(this, &LibuvWrapper::TcpServer::AfterReadImpl);
afterReadHandle = GCHandle::Alloc(afterReadFunc);

The typedef and passing the callback look like this:
typedef void (__stdcall * AFTERREAD_CB)(uv_stream_t* handle,
                         ssize_t nread, uv_buf_t buf);

...

if(uv_read_start(stream, echo_alloc,static_cast<AFTERREAD_CB>
  (Marshal::GetFunctionPointerForDelegate(afterReadFunc).ToPointer())))
{
    throw gcnew System::Exception(
                 "Error on setting up after read\n");}
```

It is not super pretty but it got the job done! The rest of the internals are available with the posted source.

Once I had a server, I built a simple test client using plain old C# , TcpClient and ran that from another WorkerRole:

```
var ep = RoleEnvironment.Roles["LibuvServer"].Instances.Select(
               i => i.InstanceEndpoints ["listen"]).ToArray();
var cl = new TcpClient();
byte[] msg = Encoding.ASCII.GetBytes(new string('x',msgSize));
```

```
byte[] gbye = Encoding.ASCII.GetBytes("|^|");
byte[] ackBuf = new byte[1];

//it's easy for the client to start before server so wait
//forever until we connect.
//for something more sophisticated AKA "the greatest retry library
//in the history of man" see
//http://code.msdn.microsoft.com/Transient-Fault-Handling-b209151fwhile (!cl.Connected)
{
    try
    {
        cl.Connect(ep[0].IPEndpoint);
    }
    catch (Exception)
    {
        Thread.Sleep(100);
    }

}
using(var stream = cl.GetStream())
{
    var watch = new Stopwatch();
    watch.Start();
    for (int i = 0; i < numTimes ; i++)
    {

    stream.Write(msg, 0, msg.Length);
    stream.Read(ackBuf, 0, ackBuf.Length);
}
watch.Stop();
log = string.Format("Sent {0} messages of size {1} and received  {0} acks in {2}
milliseconds",
                                            numTimes, msgSize,
watch.ElapsedMilliseconds);
Trace.WriteLine(log, "Information");
stream.Write(gbye,0,gbye.Length);
}
```

Next, I added a simple script and Startup Task to my “server” project to ensure the 2010 C runtime was available:

```
"%~dps0vcredist_x64.exe" /q /norestart
```

And finally, I deployed the project and ran it in Windows Azure.

The full source for the project can be found here.

Business Continuity for Windows Azure

Published: March 2012

Authors: Patrick Wickline, Adam Skewgar, Walter Myers III

Reviewers: Michael Hyatt, Brad Calder, Tina Stewart, Steve Young, Shont Miller, Vlad Romanenko, Sasha Nosov, Monilee Atkinson, Yousef Khalidi, Brian Goldfarb

Introduction

IT managers spend considerable time and effort ensuring their data and applications are available when needed. The general approach to business continuity in the cloud is no different than any other environment. The same principles and goals apply, but the implementation and process will be different.

Availability problems can be classified into three broad categories:

- Failure of individual servers, devices, or network connectivity
- Corruption, unwanted modification, or deletion of data
- Widespread loss of facilities

This document explains how you think about and plan for availability in all three of these categories when using Windows Azure.

We start by describing some of the high availability services that Windows Azure provides and then describe some of the common backup and recovery mechanisms you may choose to employ. We then move on to describe the services that Windows Azure exposes to help you build highly-available applications and, finally, discuss strategies for designing applications that can withstand full site outages.

The fundamental concepts of preparedness and recovery are consistent with what you are probably already familiar with. However, Windows Azure mitigates some potential failures for you and provides the infrastructure and capabilities for you to design your own availability and recovery plans at a much lower cost than traditional solutions.

Security and availability are closely related topics. While we touch on security of the platform itself, this document is not intended to provide full coverage of that topic and we will not cover designing secure applications. For more information on these topics, please see the following:

- Windows Azure Security Overview
- Security Best Practices for Developing Windows Azure Applications

A Cloud Platform Designed for High Availability

By deploying your application on Windows Azure you are taking advantage of many fault tolerance and secure infrastructure capabilities that you would otherwise have to design, acquire, implement, and manage. This section covers the things Windows Azure does for you without any additional expense or complexity to ensure that you are building on a platform you can trust.

World Class Data Centers in Multiple Geographies

A basic requirement for business continuity is the availability of well-managed datacenter infrastructure in diverse geographic locations. If individual data centers are not properly managed, the most robust application designs may be undermined by gross failures at the infrastructure level. With Windows Azure, you can take advantage of the extensive experience Microsoft has in designing, managing, and securing world-class data centers in diverse locations around the world.

These are the same data centers that run many of the world's largest online services. These datacenters are designed and constructed with stringent levels of physical security and access control, power redundancy and efficiency, environmental control, and recoverability capabilities. The physical facilities have achieved broad industry compliance, including ISO 27001 and SOC / SSAE 16 / SAS 70 Type II and within the United States, FISMA certification.

To ensure recoverability of the Windows Azure platform core software components, Microsoft has established an Enterprise Business Continuity Program based on Disaster Recovery Institute International (DRII) Professional Practice Statements and Business Continuity Institute (BCI) Good Practice Guidelines. This program also aligns to FISMA and ISO27001 Continuity Control requirements. As part of this methodology, recovery exercises are performed on a regular basis simulating disaster recovery scenarios. In the rare event that a system failure does occur, Microsoft uses an aggressive, root cause analysis process to deeply understand the cause. Implementation of improvements learned from outages is a top priority for the engineering organization. In addition, Microsoft provides post-mortems for every customer impacting incident upon request.

Infrastructure Redundancy and Data Durability

The Windows Azure platform mitigates outages due to failures of individual devices, such as hard drives, network interface adapters, or even entire servers. Data durability for SQL Azure and Windows Azure Storage (blobs, tables, and queues) is facilitated by maintaining multiple copies of all data on different drives located across fully independent physical storage sub-systems. Copies of data are continually scanned to detect and repair bit rot, an often overlooked threat to the integrity of stored data.

Compute availability is maintained by deploying roles on isolated groupings of hardware and network devices known as fault domains. The health of each compute instance is continually monitored and roles are automatically relocated to new fault domains in the event of a failure. By taking advantage of the Windows Azure service model and deploying at least two instances of each role in your application, your application can remain available as individual instances are relocated to new fault domains in the event of a failure within a fault domain. This all happens transparently and without downtime with no need for you to deal with managing the underlying infrastructure.

Furthermore, Windows Azure provides built in network load balancing, automatic OS and service patching, and a deployment model that allows you to upgrade your application without downtime by using upgrade domains, a concept similar to fault domains, which ensures that only a portion of your service is updated at any time.

Data Backup and Recovery

The ability to restore application data in the event of corruption or unwanted modification or deletion is a fundamental requirement for many applications. The following sections discuss recommended approaches for implementing point-in-time backups for Windows Azure Blob and Table storage, and SQL Azure databases.

Blob and Table Storage Backup

While blobs and tables are highly durable, they always represent the current state of the data. Recovery from unwanted modification or deletion of data may require restoring data to a previous state. This can be achieved by taking advantage of the capabilities provided by Windows Azure to store and retain point-in-time copies.

For Windows Azure Blobs, you can perform point-in-time backups using the blob snapshot feature. For each snapshot, you are only charged for the storage required to store the differences within the blob since the last snapshot state. The snapshots are dependent on the existence of the original blob they are based on, so a copy operation to another blob or even another storage account is advisable to ensure that backup data is properly protected against accidental deletion. For Windows Azure Tables, you can make point-in-time copies to a different table or to Windows Azure Blobs. More detailed guidance and examples of performing application-level backups of tables and blobs can be found here:

- Protecting Your Tables Against Application Errors
- Protecting Your Blobs Against Application Errors

SQL Azure Backup

Point-in-time backups for SQL Azure databases are achieved with the SQL Azure Database Copy command. You can use this command to create a transactionally-consistent copy of a database on the same logical database server or to a different server. In either case, the database copy is fully functional and completely independent of the source database. Each copy you create represents a point-in-time recovery option. You can recover the database state completely by renaming the new database with the source database name. Alternatively, you can recover a specific subset of data from the new database by using Transact-SQL queries. For additional details about SQL Azure backup, see Business Continuity in SQL Azure.

Platform Services for Building Highly Available Solutions

On top of a secure and highly redundant infrastructure, there are Windows Azure services that act as building blocks for designing highly available deployments that span multiple data centers. These services can each be used on their own or in combination with each other, third-party services and application-specific logic to achieve the desired balance of availability and recovery goals.

Geo-Replication of Blob and Table Storage

All Windows Azure blob and table storage is replicated between paired data centers hundreds of miles apart within a specific geographic region (e.g., between North Central and South Central in

the United States or between North and West in Europe). Geo-replication is designed to provide additional durability in case there is a major data center disaster. With this first version of geo-replication, Microsoft controls when failover occurs and failover will be limited to major disasters in which the original primary location is deemed unrecoverable in a reasonable amount of time. In the future, Microsoft plans to provide the ability for customers to control failover of their storage accounts on an individual basis. Data is typically replicated within a few minutes, although synchronization interval is not yet covered by an SLA. More information can be found the blog post Introducing Geo-replication for Windows Azure Storage.

SQL Azure Export to Blob Using the SQL Azure Import/Export Service

Geo-replication of SQL Azure data can be achieved by exporting the database to an Azure Storage blob using the SQL Azure Import/Export service. This can be implemented in three ways:

- Export to a blob using storage account in a different data center.
- Export to a blob using storage account in the same data center (and rely on Windows Azure Storage geo-replication to the separate data center).
- Import to your on-premises SQL Server.

For implementation details see the MSDN article Business Continuity in SQL Azure.

Traffic Manager

Windows Azure Traffic Manager (currently a Community Technology Preview) allows you to load balance incoming traffic across multiple hosted Windows Azure services whether they are running in the same datacenter or across different datacenters around the world. By effectively managing traffic, you can ensure high performance, availability, and resiliency of your applications. Traffic routing occurs as a result of policies that you define and that are based on one of the following three criteria:

- **Performance –** traffic is forwarded to the closest hosted service in terms of network latency
- **Round robin –** traffic is distributed equally across all hosted services
- **Failover –** traffic is sent to a primary service and, if this service is not online, to the next available service in a list

Traffic Manager constantly monitors hosted services to ensure they are online and will not route traffic to a service that is unavailable. Using Traffic Manager you can create applications that are designed to respond to requests in multiple geographic locations and that can therefore survive entire site outages. Prior to the general availability of a production version of Traffic Manager, you can implement third-party DNS routing solutions such as those provided by Akamai or Level 3 Communications to build applications that are designed for high-availability. For more information, see Windows Azure Traffic Manager.

Planning for Site Disasters

Windows Azure greatly reduces what you need to worry about to ensure high availability within a single data center. However, even the most well designed data center could be rendered inaccessible in the case of a true disaster. To plan for such disasters, you must think through both

the technical and procedural steps required to provide the level of cross-site availability you require. Much of this is application specific and as the application owner you must make tradeoff decisions between availability and complexity or cost.

Types of Cross-Site Availability

There are some basic steps that all but the most trivial of applications should take to know that they can be deployed in a different data center in the event of a site disaster. For many applications, redeployment from scratch is an acceptable solution. For those that need a quicker and more predictable recovery a second deployment must be ready and waiting in a second data center. For an even smaller set of applications, true multi-site high-availability is required. We will look at each of these classes of applications in order from the least complex and costly to the most complex and costly. Cross-site availability is a lot like insurance – you pay for protection that you hope you will never have to use. No two applications have precisely the same business requirements or technical design points, so these classes of applications are meant as general guidance that should be adapted to your specific needs.

For an application to become available in a secondary data center, three requirements must be met:

1. The customer's application and dependent services must be deployed.
2. The necessary data must be available to the application (typically in the same data center).
3. Any external traffic must be routed to the application.

Each of these three requirements can be accomplished at the time of a disaster or ahead of time, and each can be accomplished manually or automatically. Every application is different, so it's possible there are other application-specific requirements, such as availability of a dependent service. In the rest of this section, we'll describe several common patterns for recovering availability in the case of a disaster.

Redeploy on Disaster

The simplest form of disaster recovery is to redeploy your application and dependent services, such as Windows Azure Cache and Windows Azure Service Bus, when a disaster occurs. Redeployment of applications is accomplished using the same method used when the application was originally created. This can be done manually via the Windows Azure Portal or can be automated using the Windows Azure Service Management interface.

To mitigate data loss, the redeployment strategy should be coupled with data backup or synchronization to make sure the data exists and is usable from a backup storage account or database. Because the data for the new deployment is in a different account, the application must be designed with configurable connection strings. This way, on redeploy, only the application's configuration must be changed.

To meet the third requirement (traffic routing to the new deployment), a custom domain name must be used (not <*name*>.cloudapp.net). In the case of a disaster, the custom domain name can be configured to route to the new application name after the new deployment is completed. A CNAME record is sufficient to accomplish this, but you should be sure you know the process for making the necessary changes with your DNS provider.

Because no compute resources are reserved for recovery this is the least expensive solution. However, low cost comes at the expense of increased risk and increased recovery time. In the event of a large-scale disaster, resources in other data centers are allocated based on real-time availability.

Active/Passive Deployment

The redeployment strategy described previously takes time and has risks. Some customers need faster and more predictable recovery and may want to reserve standby resources in an alternate data center. The active/passive pattern means keeping an always-ready (potentially smaller) deployment of the application in a secondary data center. At the time of a disaster, the secondary deployment can be activated and scaled.

Here, the first requirement (deployment of the app) is taken care of ahead of time. The second requirement (data availability) is typically handled one of the data replication methods discussed above. The third requirement (routing traffic) is handled by the same way as with the redeploy pattern (DNS change at the time of disaster). In this pattern, however, the process can be fully automated using Windows Azure Traffic manager (CTP) or other similar DNS management solutions, which can be configured to reroute traffic to a secondary deployment only when the first deployment stops responding.

When using a warm standby, it is important to carefully think through data consistency implications at the time of the failover. You need to plan for any steps that must be taken before your application is brought online in an alternate location as well as how and when to return to the original deployment when it becomes available again. Because the application is not designed to function *simultaneously* in multiple data centers, returning service to the primary location should follow a similar procedure to the one you follow in the case of a failover.

Active/Active Deployment

The active/active pattern is very similar to the active/passive deployment pattern, but the third requirement (traffic routed to the deployment) is always satisfied because both deployments now handle incoming traffic at all times . As in the active/passive configuration, this can be accomplished with Windows Azure Traffic Manager (CTP) or similar solutions, but now using a performance policy, which routes traffic from a user to whichever data center provides them the lowest latency or a load-balancing policy, which distributes load evenly to each site. This solution is the most difficult to implement, as it requires the application to be designed to handle simultaneous requests across multiple instances of the application that reside in distinct data centers. However, it is more efficient than the active/passive solution in that all compute resources are utilized all the time.

To achieve full, multi-site high availability without downtime, another common pattern is the active/active deployment, so named because two deployments in two data centers are both active. This meets the first requirement (that the app is deployed). In this model, a full copy of the data is in both locations, and they are continually synchronized. This meets the second requirement (that the data is available).

Access Control Service (ACS)

Windows Azure Access Control Service (ACS) is a cloud-based service that provides an easy way of authenticating and authorizing users to gain access to your web applications and services while allowing the features of authentication and authorization to be factored out of your code. Instead of implementing an authentication system with user accounts that are specific to your application, you can let ACS orchestrate the authentication and much of the authorization of your users. ACS integrates with standards-based identity providers, including enterprise directories such as Active Directory, and web identities such as Windows Live ID, Google, Yahoo!, and Facebook.

Restoration of ACS Namespaces in the Event of a Disaster

Being built on the Azure platform, ACS is a highly available system and resilient to failures within a single data center. However in the event of a disaster rendering a full data center inoperable, there is the potential for data loss. Should such a disaster occur, all Windows Azure services will make best efforts to restore you to a pre-disaster state as quickly as possible.

Data Recovery

The ability to restore an ACS subscription is dependent on the version of ACS that you subscribe to. ACS currently consists of two versions (version 1.0 and version 2.0) with different disaster recovery and data restoration capabilities. There are two ways to determine the version of an ACS namespace:

1. See the “ACS Version” column in the Service Bus, Access Control, & Caching section of the Windows Azure portal. This column is present for all Access Control and Service Bus namespaces, the latter of which includes a built-in ACS namespace.
2. You can also determine the ACS version by trying to connect to the ACS 2.0 management portal in a web browser. The URL to the ACS 2.0 management portal is https://<*tenant*>.accesscontol.windows.net/ where <*tenant*> should be replaced by your actual ACS namespace name. If you can successfully access the portal or are prompted to sign in, then the ACS version is 2.0. If you get an HTTP status code 404 or error message “Page cannot be found”, then the ACS version is 1.0.

ACS 1.0 Namespaces

While benefiting from the high availability built into Windows Azure, ACS 1.0 namespaces are not recoverable in the event of a disaster. Should a disaster occur Microsoft will work with ACS 1.0 customers to mitigate the loss of data, however no guarantees are made about the ability to recover from a disaster.

In December 2011, Microsoft officially deprecated ACS version 1.0. All customers with ACS 1.0 namespaces are encouraged to migrate to ACS 2.0 in advance of December 20, 2012 to avoid potential service interruptions. For more information about migrating from ACS 1.0 to ACS 2.0, please see the Guidelines for Migrating an ACS 1.0 Namespace to an ACS 2.0 Namespace.

ACS 2.0 Namespaces

ACS 2.0 takes backups of all namespaces once per day and stores them in a secure offsite location. When ACS operation staff determines there has been an unrecoverable data loss at one of ACS’s regional data centers, ACS may attempt to recover customers’ subscriptions by

restoring the most recent backup. Due to the frequency of backups data loss up to 24 hours may occur.

ACS 2.0 customers concerned about potential for data loss are encouraged to review a set of Windows Azure PowerShell Cmdlets available through the Microsoft hosted Codeplex Open Source repository. These scripts allow administrators to manage their namespaces and to import and extract all relevant data. Through use of these scripts, ACS customers have the ability develop custom backup and restore solutions for a higher level of data consistency than is currently offered by ACS.

Notification

In the event of a disaster, information will be posted at the Windows Azure Service Dashboard describing the current status of all Windows Azure services globally. The dashboard will be updated regularly with information about the disaster. If you want to receive notifications for interruptions to any the services, you can subscribe to the service's RSS feed on the Service Dashboard. In addition, you can contact customer support by visiting the Support options for Windows Azure web page and follow the instructions to get technical support for your service(s).

Summary

Meeting your availability and recovery requirements using Windows Azure is a partnership between you and Microsoft. Windows Azure greatly reduces the things you need to deal with by providing application resiliency and data durability to survive individual server, device and network connection failures. Windows Azure also provides many services for implementing backup and recovery strategies and for designing highly-available applications using world-class datacenters in diverse geographies. Ultimately, only you know your application requirements and architecture well enough to ensure that you have an availability and recovery plan that meets your requirements.

We hope this documents helps you think through the issues and your options. We welcome your feedback on this document and your ideas for improving the business continuity capabilities we provide. Please send your comments to azuredrfb@microsoft.com.

Messaging

Capacity Planning for Service Bus Queues and Topics

Author: Valery Mizonov and Ralph Squillace

This topic describes:

- The often-important difference in queue size limits between Windows Azure queues and Windows Azure Service Bus queues and topics
- How to approximate the correct Service Bus queue capacity when you want to take advantage of Service Bus queue and topic features
- A test run with messages of different sizes to give some idea of the number of messages of that size that can be placed in a Service Bus queue or topic

Queue and Topic Size Limitations in Windows Azure Service Bus

Both Windows Azure queues and Service Bus queues are implementations of queued access storage, but each has a slightly different feature set, which means you may choose one or the other depending on the needs of your particular application. For example, Service Bus has topics and subscriptions, a pub-sub implementation that is often important in business notification scenarios; whereas Windows Azure queues can store as much as 100 terabytes of data when Service Bus queues are currently limited to 5 gigabytes.

The two major areas of difference – maximum message size and maximum queue size -- are shown below:

Comparison	Windows Azure Queues	Service Bus Queues
Maximum Message Size	**64 KB** Note: This includes the roughly 25% overhead of base64 encoding.	**256 KB** Note: This includes both the headers and the body, where the maximum header size is 64 KB.
Maximum Queue Size	**100 TB** Note: Maximum size is limited at the storage account level.	**1, 2, 3, 4, or 5 GB** The size is defined upon creation of a queue or topic.

This last difference – the 5 GB queue and topic size limit for Service Bus, compared to 100 TB limit for Windows Azure queues – can be fairly important, because it means that if you want to use Service Bus features like deadlettering, topics and subscriptions, rules, or actions, you must assess how much Service Bus queue or topic capacity you will need for your application in advance to build the solid application you want to build. It can be easy for a very active application to create a queue or topic with 5 GB of data depending upon the message size, the size of the message headers, and the size of any custom message properties – and how fast those messages are enqueued or sent to a topic.

Finally, with Service Bus queues, you establish the queue size when you create the queue. You cannot resize the queue after it's been created; whereas Windows Azure queues have access to so much storage that thinking about increasing the size of the Windows Azure queue is almost not required.

When a queue or topic exceeds its configured capacity, subsequent requests to enqueue a message result in a Microsoft.ServiceBus.Messaging.QuotaExceededException exception. Your application should provide the adequate handling for this particular exception type. For example, one approach would be to temporarily suspend message delivery to a queue or topic and give the consumers enough time to process the backlog before message publication can be resumed.

It is also important to remember that messages in Service Bus queues are comprised of two parts: a header and a body. The total size of the entire message (header + body) cannot exceed 256 KB. The Service Bus brokered messaging API uses binary XML serialization (not text XML), which reduces the output size of the serialized payload, and while this in turn enables storing messages slightly larger than 256 KB, you must test the reduction in size that you see for your own application.

Calculating the Capacity of Service Bus Queues and Topics

Let's look at the basic algorithm to calculate the size of messages, and therefore what kind of Service Bus queues or topics you might need for your application.

An empty message has a body size of 1024 bytes and default header size of 156 bytes; including other elements, this rounds out to a total of 1635 bytes in a message with no custom headers or properties and an empty message body.

The following formula can be used to estimate the size requirements that the specified number of messages can introduce:

Number Of Messages * (Message Body Size + Message Header Size)

In order to determine the body size, use the BrokeredMessage.Size property. Header size can be trickier, so it depends on how accurate you need to be. The most accurate method is to send the message (or a test message if you need to know the info before sending multiple messages), and then query the queue metadata using the NamespaceManager.GetQueue (or NamespaceManager.GetTopic) method and use the **SizeInBytes** property (of either the QueueDescription or TopicDescription objects) to determine how much header was added.

Topic size requires a slightly different algorithm. The formula to determine how much space a given number of messages consumes in the topic is:

Number Of Messages * (Message Body Size + (Message Header Size * Number Of Subscriptions))

Note that the size of the header is multiplied by the number of subscriptions to the topic, which means that if any custom headers are added to your topic messages, the topic size will increase in line with the number of subscriptions.

For example, if you had a default message with 200 subscriptions, it results in a topic size of 32 KB. However, if you increase the header size to 600 bytes, the topic size will now be 120 KB. Finally, if you add the ACK messages that flow from each subscription receiver, you're adding in

quite a bit. A single message with a 600-byte header matching 200 subscriptions, add in 200 ACK messages (one for each subscription) and you're looking at 568,635 bytes. It's important to have thought about this ahead of time.

Validating Capacity Approximations

You can take the following data into consideration when approximating how many messages you can hold in a single Service Bus queue. This data was captured by a custom utility in order to determine the queue capacity with different message size options.

Sample Message Size	1 GB Queue	2 GB Queue	3 GB Queue	4 GB Queue	5 GB Queue
1 KB	1,041,790	2,059,920	3,128,550	4,186,400	5,238,750
10 KB	102,996	208,358	312,537	416,716	520,895
50 KB	20,857	41,792	62,507	83,343	104,179
100 KB	10,466	20,836	31,254	41,672	52,090
250 KB	4,191	8,334	12,501	16,669	20,836

To rule out the potential impact of binary XML serialization on the test results, all sample messages were initialized with a byte array of the respective size filled in with random values.

See Also

Service Bus Queues

The Developer's Guide to Service Bus

An Introduction to Service Bus Queues

Service Bus Topics and Queues

Queues, Topics, and Subscriptions

Best Practices for Handling Large Messages with Windows Azure Queues

Author: Valery Mizonov

Reviewers: Brad Calder, Larry Franks, Jai Haridas, Sidney Higa, Christian Martinez, Curt Peterson, and Mark Simms

This article is intended to offer a developer-oriented guidance on the implementation of a generics-aware storage abstraction layer for the Windows Azure Queue Service. The problem space addressed by the article is centered on supporting very large messages in Windows Azure queues and overcoming the message size limitation that exists today. Simply put, this blog and the associated code will enable you to utilize Windows Azure Queues without having to engage in the message size bookkeeping imposed by the queue's 64KB limit.

Why Large Messages?

There were wonderful times when "640K ought to be enough for anybody." A few kilobytes could buy a luxurious storage space where a disciplined developer from the past was able to happily put all of her application's data. Today, the amount of data that modern applications need to be able to exchange can vary substantially. Whether it's a tiny HL7 message or multi-megabyte EDI document, modern applications have to deal with all sorts of volumetric characteristics evolving with unpredictable velocity. A business object that was expressed in a multi-byte structure in the last century may easily present itself today as a storage-hungry artifact several times larger than its predecessor thanks to modern serialization and representation techniques and formats.

Handling messages in a given solution architecture without imposing technical limitations on message size is the key to supporting ever-evolving data volumes. Large messages cannot always be avoided. For instance, if a B2B solution is designed to handle EDI traffic, the solution needs to be prepared to receive the EDI documents up to several megabytes. Every tier, service and component in the end-to-end flow needs to accommodate the document size that is being processed. Successfully accepting a 20MB EDI 846 Inventory Advice document via a Web Service but failing to store it in a queue for processing due to the queue's message size constraints would be considered as unpleasant discovery during testing.

Why would someone choose to use a queue for large messages on the Windows Azure platform? What's wrong with other alternatives such as blobs, tables, cloud drives or SQL Azure databases to say the least? Mind you, the queues allow implementing certain types of messaging scenarios that are characterized by asynchronous, loosely-coupled communications between producers and consumers performed in a scalable and reliable fashion. The use of Windows Azure queues decouples different parts of a given solution and offers unique semantics such as FIFO (First In, First Out) and At-Least-Once delivery. Such semantics can be somewhat difficult to implement using the other alternative data exchange mechanisms. Furthermore, the queues are best suited as a volatile store for exchanging data between services, tiers and applications, not as persistent data storage. The respective data exchange requirement can manifest itself in many different forms such as passing messages between components in asynchronous manner, load leveling, or scaling out complex compute workloads. Many of these data exchange patterns are not something that can be straightforward to implement without queues. In summary, the queues are

a crucial capability. Not having to worry about what can and cannot go into a queue is a strong argument for building unified queue-based messaging solutions that can handle any data of any size.

In this article, I'm going to implement a solution that will enable to use a Windows Azure queue for exchanging large messages. I also intend to simplify the way my solution interacts with Windows Azure queues by providing an abstraction layer built on top of APIs found in the StorageClient namespace. This abstraction layer will make it easier to publish and consume instances of the application-specific entities as opposed to having to deal with byte arrays or strings which are the only types supported by the Queue Service API today. I am going to make extensive use of .NET generics, will take advantage of some value-add capabilities such as transparent stream compression and decompression as well as apply some known best practices such as handling intermittent faults in order to improve fault-tolerance of storage operations.

Design Considerations

As things stand today, a message that is larger than 64KB (after it's serialized and encoded) cannot be stored in a Windows Azure queue. The client-side API will return an exception if you attempt to place a message larger than 64KB in a queue. The maximum allowed message size can be determined by inspecting the MaxMessageSize property from the CloudQueueMessage class. As of the writing of this post, the message size limit returned by this property is 65536.

Important

The maximum message size defined in CloudQueueMessage.MaxMessageSize property is not reflective of the maximum allowed payload size. Messages are subject to Base64 encoding when they are transmitted to a queue. The encoded payloads are always larger than their raw data. The Base64 encoding adds 25% overhead on average. As a result, the 64KB size limit effectively prohibits from storing any messages with payload larger than 48KB (75% of 64KB).

Even though it's the limit for a single queue item, it can be deemed prohibitive for certain types of messages, especially those that cannot be broken down into smaller chunks. From a developer perspective, worrying about whether a given message can be accommodated on a queue doesn't help my productivity. At the end of the day, the goal is to get my application data to flow between producers and consumers in the most efficient way, regardless of the data size. While one side calls *Put* (or *Enqueue*) and the other side invokes *Get* (or *Dequeue*) against a queue, the rest should theoretically occur automagically.

Overcoming the message size limitation in Windows Azure queues by employing a smart way of dealing with large messages is the key premises for the technical challenge elaborated in this article. This will come at the cost of some additional craftsmanship. In the modern world of commercial software development, any extra development efforts need to be wisely justified. I am going to justify the additional investments with the following design goals:

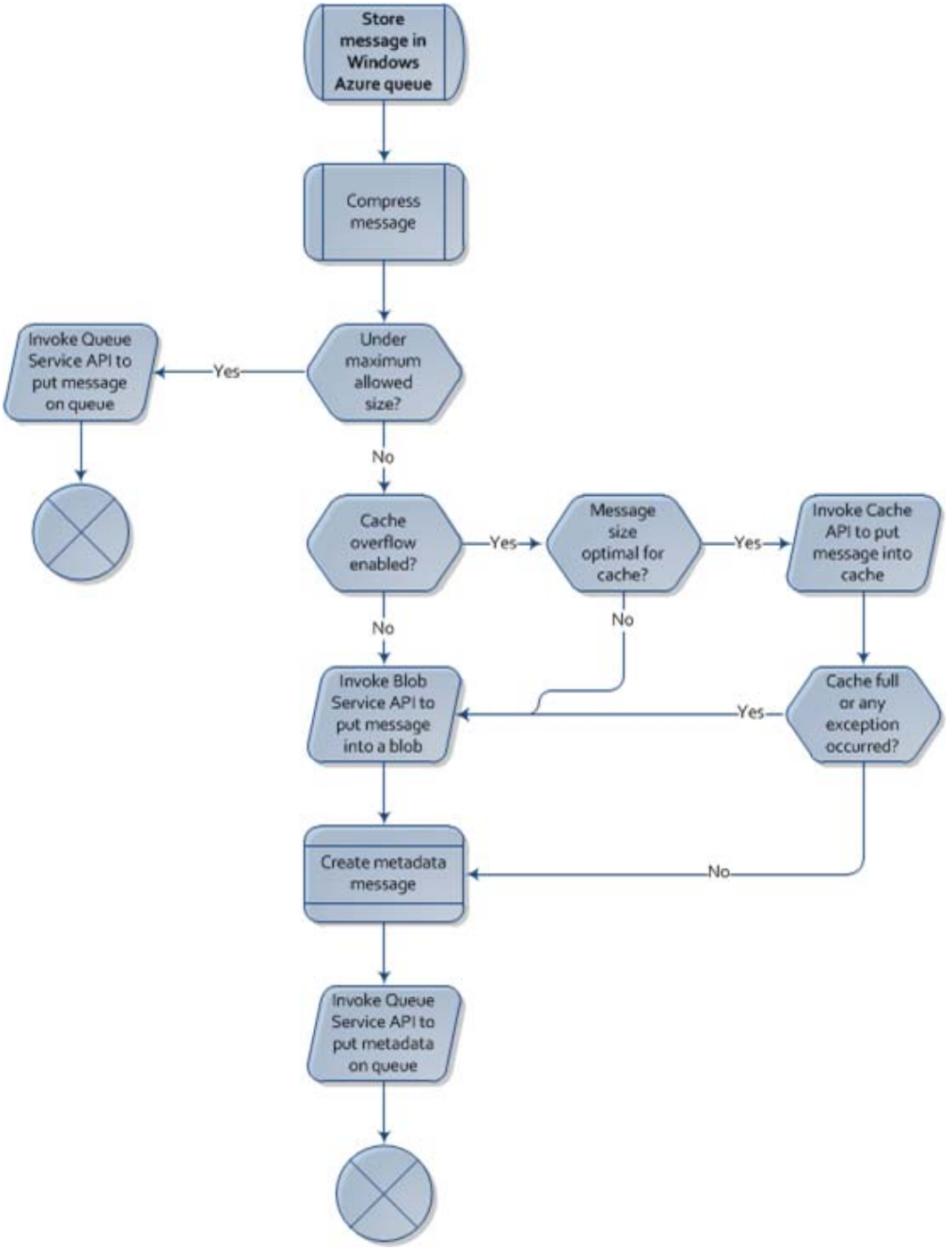
- **Support for very large messages** through eliminating any restrictions imposed by the Queue Service API as it pertains to the message size.
- **Support for user-defined generic objects** when publishing and consuming messages from a Windows Azure queue.

- **Transparent overflow into a configurable message store** either blob container, distributed cache or other type of repository capable of storing large messages.
- **Transparent compression** that is intended to increase cost-efficiency by minimizing the amount of storage space consumed by large messages.
- **Increased reliability** in the form of extensive use of the transient condition handling best practices when performing queue operations.

The foundation for supporting large messages in size-constrained queues will be the following pattern. First, I check if a given message can be accommodated on a Windows Azure queue without performing any extra work. The way to determine whether a message can be safely stored on a queue without violating size constraints will be through a formula which I wrap into a helper function as follows:

```
/// <summary>
/// Provides helper methods to enable cloud application code to invoke common, globally
accessible functions.
/// </summary>
public static class CloudUtility
{
    /// <summary>
    /// Verifies whether or not the specified message size can be accommodated in a
Windows Azure queue.
    /// </summary>
    /// <param name="size">The message size value to be inspected.</param>
    /// <returns>True if the specified size can be accommodated in a Windows Azure queue,
otherwise false.</returns>
    public static bool IsAllowedQueueMessageSize(long size)
    {
        return size >= 0 && size <= (CloudQueueMessage.MaxMessageSize - 1) / 4 * 3;
    }
}
```

If message size is under the enforced limit, I should simply invoke the Queue Service API to enqueue the message "as is." If the message size is in excess of the limitation in question, the data flow becomes quite interesting. The following flowchart visualizes the subsequent steps:



In summary, if a message cannot be accommodated on a queue due to its size, it overflows into a message store capable of storing large messages. A tiny metadata message is then created consisting of a reference to the item in the overflow store. Finally, the metadata message is put on a queue. I always choose to compress a message before asserting its suitability for persistence in a queue. This effectively expands the population of messages that can be queued without incurring the need to go into the overflow store. A good example is an XML document slightly larger than 64KB which, after serialization and compression is performed, becomes a

perfect candidate to be simply put on a queue. You can modify this behavior in case the default compression is not desirable. It can be achieved by providing a custom serializer component elaborated in the next section.

There are several considerations that apply here, mainly from a cost perspective. As it can be noted in the above flowchart, I attempt to determine whether a large message can first overflow into Windows Azure Caching Service (referred herein as *Caching Service* for the sake of brevity). Since the usage of distributed cloud-based caching service is subject to a charge, the cache overflow path should be made optional. This is reflected on the flowchart.

In addition, there may be situations when the message is quite large and therefore is not suitable for being stored in a size-constrained distributed cache. As of the writing of this article, the maximum cache size is 4GB. Therefore, we must take this into consideration and provide a failover path should we exceed cache capacityor quotas. The quotas come with eviction behavior that also needs to be accounted for.

Important

The use of the Windows Azure Caching ServHFice as an overflow store helps reduce latency and eliminate excessive storage transactions when exchanging a large number of messages. It offers a highly available, distributed caching infrastructure capable of replicating and maintaining cached data in memory across multiple cache servers for durability. These benefits can be outweighed by the cache size limitation and costs associated with the service usage. It is therefore important to perform a cost-benefit analysis to assess the pros and cons of introducing the Caching Service as an overflow store in certain scenarios.

Given that the distributed cache storage is limited, it is essential to set out some further rules that will enable the efficient use of the cache. In connection to this, one important recommendation needs to be explicitly called out:

Important

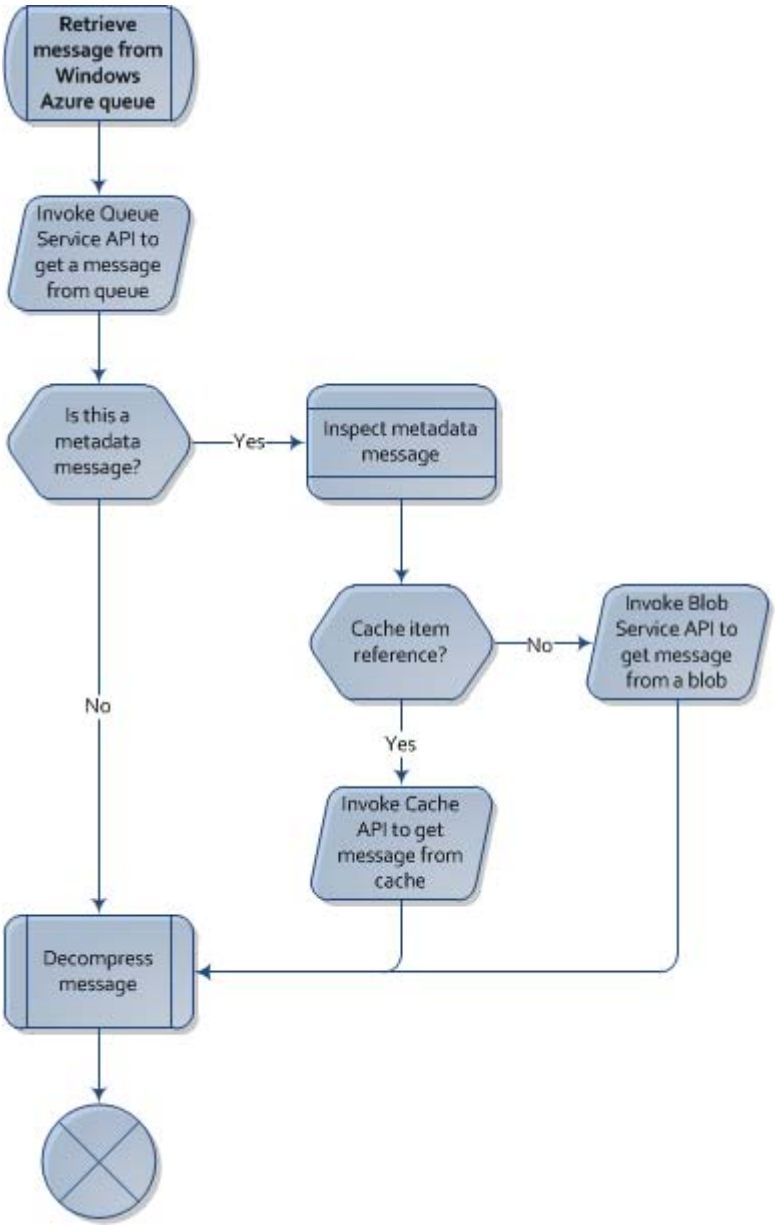
Due to specifics of its eviction behavior, Caching Service does not offer a complete and ultimate guaranteed durability when compared to the Windows Azure Blob Service. When used as an overflow store, Caching Service is best suited when individual messages are volatile in nature and are under 8MB in size. The term “volatile” means that messages are published into and subsequently consumed as quickly as possible. The 8MB recommendation is due to the optimal cache item size that is configured in the Caching Service by default.

I’m going to reflect the above recommendation in the code by providing a helper function that will determine whether or not the specified item size value can be considered as optimal when storing an item of the given size in the cache.

```
public static class CloudUtility
{
    private static readonly long optimalCacheItemSize = 8 * 1024 * 1024;

    /// <summary>
```

```
    /// Determines whether the specified value can be considered as optimal when storing
an item of a given size in the cache.
    /// </summary>
    /// <param name="size">The item size value to be inspected.</param>
    /// <returns>True if the specified size can be considered as optimal, otherwise
false.</returns>
    public static bool IsOptimalCacheItemSize(long size)
    {
        return size >= 0 && size <= optimalCacheItemSize;
    }
}
```

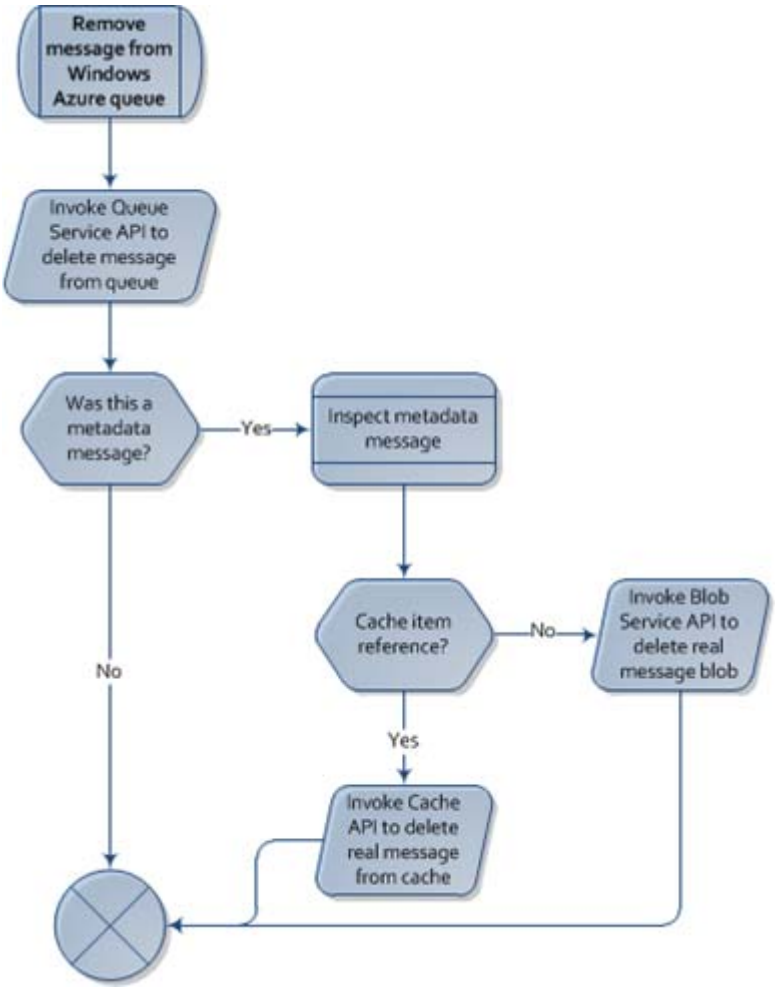
Now that some initial pre-requisites are considered, it’s time to switch over to the consumer side and take a look at the implementation pattern for retrieving large messages from a queue. First, let’s visualize the process flow for the purposes of facilitating overall understanding:



To summarize the above flow, a message of an unknown type is fetched from a queue and compared against a metadata message type. If it is not a metadata message, the flow continues with decompression logic, so that the original message can be correctly reconstructed before being presented to the consumer. By contrast, if it was in fact a metadata message, it is inspected to determine the type of overflow store that was used for storing the actual message. If it is identified as a message stored in the cache, the respective Caching Service API is invoked and the real message will be fetched before being decompressed and returned to the consumer. In case the real message was put into a blob container, the Blob Service API will be targeted to retrieve the real message from the blob entity, decompressed and handed back to the caller.

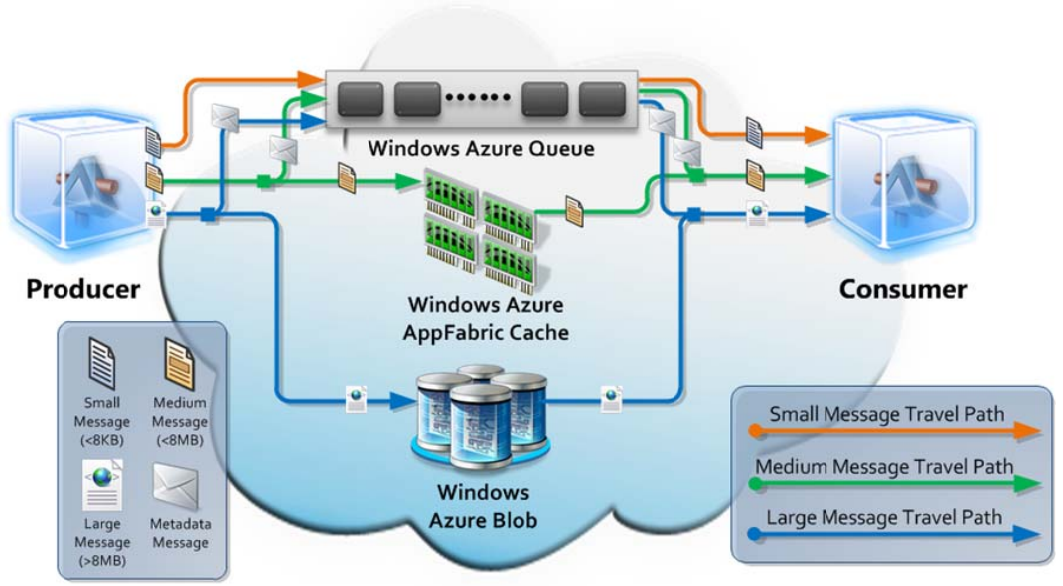
In addition to handling *Enqueue* and *Dequeue* operations for large messages, there is a need to make sure that all overflowed payloads are removed from their respective overflow message

stores upon the consumer's request. To accomplish this, one of the potential implementation patterns is to couple the removal process with the *Delete* operation when it's being invoked for a given message. The visual representation of this operation can be depicted as follows:



Before we start implementing the patterns mentioned above, one last consideration worth making is the definition of a message. What would be considered a message, and what forms will it manifest itself in? Would it be a byte array, a stream of data, a simple type like a string, or a complex application-specific object which the developer implements as part of the solution object model? I truly believe that this is the area where we should not constrain ourselves. Let's just assume that a message is of generic type <T> meaning it's anything the developer wishes to use. You will see that the end implementation will naturally unfold itself around this idea.

Putting all together, the following diagram summarizes all the three possible travel paths which are accounted for in the above design:



At this point, there seem to be enough input to start bringing the technical design to life. From this point onwards, I will switch the focus to the source code required to implement the patterns discussed above.

Technical Implementation

To follow along, download the full sample code from the MSDN Code Gallery. The sample is shipped as part of a larger end-to-end reference implementation which is powered by the patterns discussed in this article. Once downloaded and unzipped, navigate to the **Azure.Services.Framework** project under **Contoso.Cloud.Integration** and expand the **Storage** folder. This location contains all the main code artifacts discussed below.

As noted at the beginning, the original idea was to abstract the way a cloud application interacts with Window Azure queues. I approach this requirement by providing a contract that governs the main operations supported by my custom storage abstraction layer. The programming interface through which the contract surfaces to consumers is shown below. I intentionally omitted a few infrastructure-level functions from the code snippet below such as creation and deletion of queues since these do not add significant value at this time.

```
/// <summary>
/// Defines a generics-aware abstraction layer for Windows Azure Queue storage.
/// </summary>
public interface ICloudQueueStorage : IDisposable
{
    /// <summary>
    /// Puts a single message on a queue.
    /// </summary>
    /// <typeparam name="T">The type of the payload associated with the
message.</typeparam>
```

```
    /// <param name="queueName">The target queue name on which message will be
placed.</param>
    /// <param name="message">The payload to be put into a queue.</param>
    void Put<T>(string queueName, T message);

    /// <summary>
    /// Retrieves a single message from the specified queue and applies the default
visibility timeout.
    /// </summary>
    /// <typeparam name="T">The type of the payload associated with the
message.</typeparam>
    /// <param name="queueName">The name of the source queue.</param>
    /// <returns>An instance of the object that has been fetched from the
queue.</returns>
    T Get<T>(string queueName);

    /// <summary>
    /// Gets a collection of messages from the specified queue and applies the specified
visibility timeout.
    /// </summary>
    /// <typeparam name="T">The type of the payload associated with the
message.</typeparam>
    /// <param name="queueName">The name of the source queue.</param>
    /// <param name="count">The number of messages to retrieve.</param>
    /// <param name="visibilityTimeout">The timeout during which retrieved messages will
remain invisible on the queue.</param>
    /// <returns>The list of messages retrieved from the specified queue.</returns>
    IEnumerable<T> Get<T>(string queueName, int count, TimeSpan visibilityTimeout);

    /// <summary>
    /// Gets a collection of messages from the specified queue and applies the default
visibility timeout.
    /// </summary>
    /// <typeparam name="T">The type of the payload associated with the
message.</typeparam>
    /// <param name="queueName">The name of the source queue.</param>
    /// <param name="count">The number of messages to retrieve.</param>
    /// <returns>The list of messages retrieved from the specified queue.</returns>
    IEnumerable<T> Get<T>(string queueName, int count);

    /// <summary>
    /// Deletes a single message from a queue.
```

```
    /// </summary>
    /// <typeparam name="T">The type of the payload associated with the
message.</typeparam>
    /// <param name="message">The message to be deleted from a queue.</param>
    /// <returns>A flag indicating whether or not the specified message has actually been
deleted.</returns>
    bool Delete<T>(T message);
}
```

There is also a need for one extra contract (interface) which will abstract access to the large-message overflow store. Two components implement the contract, one for each overflow store (blob storage and distributed cache). The contract is comprised of the following operations:

```
/// <summary>
/// Defines a generics-aware abstraction layer for Windows Azure Blob storage.
/// </summary>
public interface ICloudBlobStorage : IDisposable
{
    /// <summary>
    /// Puts a blob into the underlying storage, overwrites if the blob with the same
name already exists.
    /// </summary>
    /// <typeparam name="T">The type of the payload associated with the blob.</typeparam>
    /// <param name="containerName">The target blob container name into which a blob will
be stored.</param>
    /// <param name="blobName">The custom name associated with the blob.</param>
    /// <param name="blob">The blob's payload.</param>
    /// <returns>True if the blob was successfully put into the specified container,
otherwise false.</returns>
    bool Put<T>(string containerName, string blobName, T blob);

    /// <summary>
    /// Puts a blob into the underlying storage. If the blob with the same name already
exists, overwrite behavior can be applied.
    /// </summary>
    /// <typeparam name="T">The type of the payload associated with the blob.</typeparam>
    /// <param name="containerName">The target blob container name into which a blob will
be stored.</param>
    /// <param name="blobName">The custom name associated with the blob.</param>
    /// <param name="blob">The blob's payload.</param>
    /// <param name="overwrite">The flag indicating whether or not overwriting the
existing blob is permitted.</param>
    /// <returns>True if the blob was successfully put into the specified container,
otherwise false.</returns>
```

```
    bool Put<T>(string containerName, string blobName, T blob, bool overwrite);

    /// <summary>
    /// Retrieves a blob by its name from the underlying storage.
    /// </summary>
    /// <typeparam name="T">The type of the payload associated with the blob.</typeparam>
    /// <param name="containerName">The target blob container name from which the blob
will be retrieved.</param>
    /// <param name="blobName">The custom name associated with the blob.</param>
    /// <returns>An instance of <typeparamref name="T"/> or default(T) if the specified
blob was not found.</returns>
    T Get<T>(string containerName, string blobName);

    /// <summary>
    /// Deletes the specified blob.
    /// </summary>
    /// <param name="containerName">The target blob container name from which the blob
will be deleted.</param>
    /// <param name="blobName">The custom name associated with the blob.</param>
    /// <returns>True if the blob was deleted, otherwise false.</returns>
    bool Delete(string containerName, string blobName);
}
```

Both contracts heavily rely on generic type <T>. It enables you to tailor the message type to any .NET type of your choice. I will however have to handle some extreme use cases, namely types that require special treatment, such as streams. I expand on this later on.

Regardless of the message type chosen, one important requirement will apply; the object type that represents a message on a queue must be serializable. All objects passing through the storage abstraction layer are subject to serialization before they land on a queue or overflow store. In my implementation, serialization and deserialization are also coupled with compression and decompression, respectively. This approach increases efficiency from a cost and bandwidth perspective. The cost-related benefit comes from the fact that compressed large messages inherently consume less storage, resulting in a decrease in storage costs. The bandwidth efficiency arises from savings on payload size thanks to compression, which in turn makes payloads smaller on the wire as they flow to and from the Windows Azure storage.

The requirement for serialization and deserialization is declared in a specialized interface. Any component that implements this interface must provide the specific compression, serialization, deserialization, and decompression functionality. An example of this interface is shown:

```
/// <summary>
/// Defines a contract that must be supported by a component which performs serialization
and
/// deserialization of storage objects such as Azure queue items, blobs and table
entries.
```

```
/// </summary>
public interface ICloudStorageEntitySerializer
{
    /// <summary>
    /// Serializes the object to the specified stream.
    /// </summary>
    /// <param name="instance">The object instance to be serialized.</param>
    /// <param name="target">The destination stream into which the serialized object will
be written.</param>
    void Serialize(object instance, Stream target);

    /// <summary>
    /// Deserializes the object from specified data stream.
    /// </summary>
    /// <param name="source">The source stream from which serialized object will be
consumed.</param>
    /// <param name="type">The type of the object that will be deserialized.</param>
    /// <returns>The deserialized object instance.</returns>
    object Deserialize(Stream source, Type type);
}
```

For compression and decompression, I use the DeflateStream component in the .NET Framework. This class represents the Deflate algorithm, an industry standard RFC-compliant algorithm for lossless file compression and decompression. In comparison to the GZipStream class, the former produces more optimal compressed images and generally delivers better performance. By contrast, the **GZipStream** class uses the GZIP data format, which includes a cyclic redundancy check (CRC) value for detecting data corruption. Behind the scenes, the GZIP data format uses the same compression algorithm as the **DeflateStream** class. In summary, GZipStream = DeflateStream + the cost of calculating and storing CRC checksums.

My implementation of the contract is included below. Note that compression algorithms can be easily toggled by replacing **DeflateStream** class with **GZipStream** and vice versa.

```
/// <summary>
/// Provides a default implementation of ICloudStorageEntitySerializer which performs
serialization and
/// deserialization of storage objects such as Azure queue items, blobs and table
entries.
/// </summary>
internal sealed class CloudStorageEntitySerializer : ICloudStorageEntitySerializer
{
    /// <summary>
    /// Serializes the object to the specified stream.
    /// </summary>
    /// <param name="instance">The object instance to be serialized.</param>
```

```
    /// <param name="target">The destination stream into which the serialized object will
be written.</param>
    public void Serialize(object instance, Stream target)
    {
        Guard.ArgumentNotNull(instance, "instance");
        Guard.ArgumentNotNull(target, "target");

        XDocument xmlDocument = null;
        XElement xmlElement = null;
        XmlDocument domDocument = null;
        XmlElement domElement = null;

        if ((xmlElement = (instance as XElement)) != null)
        {
            // Handle XML element serialization using separate technique.
            SerializeXml<XElement>(xmlElement, target, (xml, writer) => {
xml.Save(writer); });
        }
        else if ((xmlDocument = (instance as XDocument)) != null)
        {
            // Handle XML document serialization using separate technique.
            SerializeXml<XDocument>(xmlDocument, target, (xml, writer) => {
xml.Save(writer); });
        }
        else if ((domDocument = (instance as XmlDocument)) != null)
        {
            // Handle XML DOM document serialization using separate technique.
            SerializeXml<XmlDocument>(domDocument, target, (xml, writer) => {
xml.Save(writer); });
        }
        else if ((domElement = (instance as XmlElement)) != null)
        {
            // Handle XML DOM element serialization using separate technique.
            SerializeXml<XmlElement>(domElement, target, (xml, writer) => {
xml.WriteTo(writer); });
        }
        else
        {
            var serializer = GetXmlSerializer(instance.GetType());

            using (var compressedStream = new DeflateStream(target,
CompressionMode.Compress, true))
```

```
            using (var xmlWriter =
XmlDictionaryWriter.CreateBinaryWriter(compressedStream, null, null, false))
            {
                serializer.WriteObject(xmlWriter, instance);
            }
        }
    }

    /// <summary>
    /// Deserializes the object from specified data stream.
    /// </summary>
    /// <param name="source">The source stream from which serialized object will be
consumed.</param>
    /// <param name="type">The type of the object that will be deserialized.</param>
    /// <returns>The deserialized object instance.</returns>
    public object Deserialize(Stream source, Type type)
    {
        Guard.ArgumentNotNull(source, "source");
        Guard.ArgumentNotNull(type, "type");

        if (type == typeof(XElement))
        {
            // Handle XML element deserialization using separate technique.
            return DeserializeXml<XElement>(source, (reader) => { return
XElement.Load(reader); });
        }
        else if (type == typeof(XDocument))
        {
            // Handle XML document deserialization using separate technique.
            return DeserializeXml<XDocument>(source, (reader) => { return
XDocument.Load(reader); });
        }
        else if (type == typeof(XmlDocument))
        {
            // Handle XML DOM document deserialization using separate technique.
            return DeserializeXml<XmlDocument>(source, (reader) => { var xml = new
XmlDocument(); xml.Load(reader); return xml; });
        }
        else if (type == typeof(XmlElement))
        {
            // Handle XML DOM element deserialization using separate technique.
```

```
            return DeserializeXml<XmlElement>(source, (reader) => { var xml = new
XmlDocument(); xml.Load(reader); return xml.DocumentElement; });
        }
        else
        {
            var serializer = GetXmlSerializer(type);

            using (var compressedStream = new DeflateStream(source,
CompressionMode.Decompress, true))
            using (var xmlReader =
XmlDictionaryReader.CreateBinaryReader(compressedStream, XmlDictionaryReaderQuotas.Max))
            {
                return serializer.ReadObject(xmlReader);
            }
        }
    }

    private XmlObjectSerializer GetXmlSerializer(Type type)
    {
        if (FrameworkUtility.GetDeclarativeAttribute<DataContractAttribute>(type) !=
null)
        {
            return new DataContractSerializer(type);
        }
        else
        {
            return new NetDataContractSerializer();
        }
    }

    private void SerializeXml<T>(T instance, Stream target, Action<T, XmlWriter>
serializeAction)
    {
        using (var compressedStream = new DeflateStream(target, CompressionMode.Compress,
true))
        using (var xmlWriter = XmlDictionaryWriter.CreateBinaryWriter(compressedStream,
null, null, false))
        {
            serializeAction(instance, xmlWriter);

            xmlWriter.Flush();
            compressedStream.Flush();
```

```
        }
    }

    private T DeserializeXml<T>(Stream source, Func<XmlReader, T> deserializeAction)
    {
        using (var compressedStream = new DeflateStream(source,
CompressionMode.Decompress, true))
        using (var xmlReader = XmlDictionaryReader.CreateBinaryReader(compressedStream,
XmlDictionaryReaderQuotas.Max))
        {
            return deserializeAction(xmlReader);
        }
    }
}
```

One of the powerful capabilities in the **CloudStorageEntitySerializer** implementation is the ability to apply special treatment when handling XML documents of both flavors: XmlDocument and XDocument. The other area worth highlighting is the optimal serialization and deserialization of the XML data. Here I decided to take advantage of the XmlDictionaryReader and XmlDictionaryWriter classes which are known to .NET developers as a superb choice when it comes to performing efficient serialization and deserialization of XML payloads by using the .NET Binary XML format.

The decision as to the type of overflow message store is the responsibility of the consumer which calls into the custom, storage abstraction layer. Along these lines, I'm going to provide an option to select the desired message store type by adding the following constructors in the type that implements the **ICloudQueueStorage** interface:

```
/// <summary>
/// Provides reliable generics-aware access to the Windows Azure Queue storage.
/// </summary>
public sealed class ReliableCloudQueueStorage : ICloudQueueStorage
{
    private readonly RetryPolicy retryPolicy;
    private readonly CloudQueueClient queueStorage;
    private readonly ICloudStorageEntitySerializer dataSerializer;
    private readonly ICloudBlobStorage overflowStorage;
    private readonly ConcurrentDictionary<object, InflightMessageInfo> inflightMessages;

    /// <summary>
    /// Initializes a new instance of the <see cref="ReliableCloudQueueStorage"/> class
using the specified storage account information,
    /// custom retry policy, custom implementation of <see
cref="ICloudStorageEntitySerializer"/> interface and custom implementation of
    /// the large message overflow store.
```

```
    /// </summary>
    /// <param name="storageAccountInfo">The storage account that is projected through
this component.</param>
    /// <param name="retryPolicy">The specific retry policy that will ensure reliable
access to the underlying storage.</param>
    /// <param name="dataSerializer">The component which performs serialization and
deserialization of storage objects.</param>
    /// <param name="overflowStorage">The component implementing overflow store that will
be used for persisting large messages that
    /// cannot be accommodated in a queue due to message size constraints.</param>
    public ReliableCloudQueueStorage(StorageAccountInfo storageAccountInfo, RetryPolicy
retryPolicy, ICloudStorageEntitySerializer dataSerializer, ICloudBlobStorage
overflowStorage)
    {
        Guard.ArgumentNotNull(storageAccountInfo, "storageAccountInfo");
        Guard.ArgumentNotNull(retryPolicy, "retryPolicy");
        Guard.ArgumentNotNull(dataSerializer, "dataSerializer");
        Guard.ArgumentNotNull(overflowStorage, "overflowStorage");

        this.retryPolicy = retryPolicy;
        this.dataSerializer = dataSerializer;
        this.overflowStorage = overflowStorage;

        CloudStorageAccount storageAccount = new CloudStorageAccount(new
StorageCredentialsAccountAndKey(storageAccountInfo.AccountName,
storageAccountInfo.AccountKey), true);
        this.queueStorage = storageAccount.CreateCloudQueueClient();

        // Configure the Queue storage not to enforce any retry policies since this is
something that we will be dealing ourselves.
        this.queueStorage.RetryPolicy = RetryPolicies.NoRetry();

        this.inflightMessages = new ConcurrentDictionary<object,
InflightMessageInfo>(Environment.ProcessorCount * 4,
InflightMessageQueueInitialCapacity);
    }
}
```

The above constructors are not performing any complex work; they simply initialize the internal members and configure the client component that will be accessing a Windows Azure queue. It's worth noting however that I explicitly tell the queue client not to enforce any retry policy. In order to provide a robust and reliable storage abstraction layer, I need more granular control over transient issues when performing operations against Windows Azure queues. Therefore, there

will be a separate component that recognizes and is able to handle a much larger variety of intermittent faults.

Let's now take a look at the internals of the **ReliableCloudQueueStorage** class which I drafted above. Specifically, let's review its implementation of the *Put* operation since this is the location of the transparent overflow into a large message store.

```
/// <summary>
/// Puts a single message on a queue.
/// </summary>
/// <typeparam name="T">The type of the payload associated with the message.</typeparam>
/// <param name="queueName">The target queue name on which message will be
placed.</param>
/// <param name="message">The payload to be put into a queue.</param>
public void Put<T>(string queueName, T message)
{
    Guard.ArgumentNotNullOrEmptyString(queueName, "queueName");
    Guard.ArgumentNotNull(message, "message");

    // Obtain a reference to the queue by its name. The name will be validated against
compliance with storage resource names.
    var queue =
this.queueStorage.GetQueueReference(CloudUtility.GetSafeContainerName(queueName));

    CloudQueueMessage queueMessage = null;

    // Allocate a memory buffer into which messages will be serialized prior to being put
on a queue.
    using (MemoryStream dataStream = new
MemoryStream(Convert.ToInt32(CloudQueueMessage.MaxMessageSize)))
    {
        // Perform serialization of the message data into the target memory buffer.
        this.dataSerializer.Serialize(message, dataStream);

        // Reset the position in the buffer as we will be reading its content from the
beginning.
        dataStream.Seek(0, SeekOrigin.Begin);

        // First, determine whether the specified message can be accommodated on a queue.
        if (CloudUtility.IsAllowedQueueMessageSize(dataStream.Length))
        {
            queueMessage = new CloudQueueMessage(dataStream.ToArray());
        }
        else
```

```
        {
            // Create an instance of a large queue item metadata message.
            LargeQueueMessageInfo queueMsgInfo = LargeQueueMessageInfo.Create(queueName);

            // Persist the stream of data that represents a large message into the
overflow message store.
            this.overflowStorage.Put<Stream>(queueMsgInfo.ContainerName,
queueMsgInfo.BlobReference, dataStream);

            // Invoke the Put operation recursively to enqueue the metadata message.
            Put<LargeQueueMessageInfo>(queueName, queueMsgInfo);
        }
    }
    // Check if a message is available to be put on a queue.
    if (queueMessage != null)
    {
        Put(queue, queueMessage);
    }
}
```

A new code artifact that has just manifested itself in the snippet above is the **LargeQueueMessageInfo** class. This custom type is ultimately our metadata message that describes the location of a large message. This class is marked as internal as it's not intended to be visible to anyone outside the storage abstraction layer implementation. The class is defined as follows:

```
/// <summary>
/// Implements an object holding metadata related to a large message which is stored in
/// the overflow message store such as Windows Azure blob container.
/// </summary>
[DataContract(Namespace = WellKnownNamespace.DataContracts.General)]
internal sealed class LargeQueueMessageInfo
{
    private const string ContainerNameFormat = "LargeMsgCache-{0}";

    /// <summary>
    /// Returns the name of the blob container holding the large message payload.
    /// </summary>
    [DataMember]
    public string ContainerName { get; private set; }

    /// <summary>
    /// Returns the unique reference to a blob holding the large message payload.
    /// </summary>
```

```
    [DataMember]
    public string BlobReference { get; private set; }

    /// <summary>
    /// The default constructor is inaccessible, the object needs to be instantiated
using its Create method.
    /// </summary>
    private LargeQueueMessageInfo() { }

    /// <summary>
    /// Creates a new instance of the large message metadata object and allocates a
globally unique blob reference.
    /// </summary>
    /// <param name="queueName">The name of the Windows Azure queue on which a reference
to the large message will be stored.</param>
    /// <returns>The instance of the large message metadata object.</returns>
    public static LargeQueueMessageInfo Create(string queueName)
    {
        Guard.ArgumentNotNullOrEmptyString(queueName, "queueName");

        return new LargeQueueMessageInfo() { ContainerName =
String.Format(ContainerNameFormat, queueName), BlobReference =
Guid.NewGuid().ToString("N") };
    }
}
```

Moving forward, I need to implement a large message overflow store that will leverage the Windows Azure Blob Storage service. As I pointed out earlier, this component must support the **ICloudBlobStorage** interface that will be consumed by the **ReliableCloudQueueStorage** component to relay messages into the **ICloudBlobStorage** implementation whenever these cannot be accommodated on a queue due to message size limitation. To set the stage for the next steps, I will include the constructor's implementation only:

```
/// <summary>
/// Implements reliable generics-aware layer for Windows Azure Blob storage.
/// </summary>
public class ReliableCloudBlobStorage : ICloudBlobStorage
{
    private readonly RetryPolicy retryPolicy;
    private readonly CloudBlobClient blobStorage;
    private readonly ICloudStorageEntitySerializer dataSerializer;

    /// <summary>
```

```
    /// Initializes a new instance of the ReliableCloudBlobStorage class using the
specified storage account info, custom retry
    /// policy and custom implementation of ICloudStorageEntitySerializer interface.
    /// </summary>
    /// <param name="storageAccountInfo">The access credentials for Windows Azure storage
account.</param>
    /// <param name="retryPolicy">The custom retry policy that will ensure reliable
access to the underlying storage.</param>
    /// <param name="dataSerializer">The component which performs
serialization/deserialization of storage objects.</param>
    public ReliableCloudBlobStorage(StorageAccountInfo storageAccountInfo, RetryPolicy
retryPolicy, ICloudStorageEntitySerializer dataSerializer)
    {
        Guard.ArgumentNotNull(storageAccountInfo, "storageAccountInfo");
        Guard.ArgumentNotNull(retryPolicy, "retryPolicy");
        Guard.ArgumentNotNull(dataSerializer, "dataSerializer");

        this.retryPolicy = retryPolicy;
        this.dataSerializer = dataSerializer;

        CloudStorageAccount storageAccount = new CloudStorageAccount(new
StorageCredentialsAccountAndKey(storageAccountInfo.AccountName,
storageAccountInfo.AccountKey), true);
        this.blobStorage = storageAccount.CreateCloudBlobClient();

        // Configure the Blob storage not to enforce any retry policies since this is
something that we will be dealing ourselves.
        this.blobStorage.RetryPolicy = RetryPolicies.NoRetry();

        // Disable parallelism in blob upload operations to reduce the impact of multiple
concurrent threads on parallel upload feature.
        this.blobStorage.ParallelOperationThreadCount = 1;
    }
}
```

Earlier in this article, I have shown the implementation of the *Put* operation which ensures that small messages will always be placed on a queue whereas large messages will be transparently routed into the overflow store. For the sake of continuity, let's now review the mechanics behind the counterpart *Put* operation implemented by the overflow store.

```
/// <summary>
/// Puts a blob into the underlying storage, overwrites the existing blob if the blob
with the same name already exists.
/// </summary>
```

```
private bool Put<T>(string containerName, string blobName, T blob, bool overwrite, string
expectedEtag, out string actualEtag)
{
    Guard.ArgumentNotNullOrEmptyString(containerName, "containerName");
    Guard.ArgumentNotNullOrEmptyString(blobName, "blobName");
    Guard.ArgumentNotNull(blob, "blob");

    var callToken = TraceManager.CloudStorageComponent.TraceIn(containerName, blobName,
overwrite, expectedEtag);

    // Verify whether or not the specified blob is already of type Stream.
    Stream blobStream = IsStreamType(blob.GetType()) ? blob as Stream : null;
    Stream blobData = null;
    actualEtag = null;

    try
    {
        // Are we dealing with a stream already? If yes, just use it as is.
        if (blobStream != null)
        {
            blobData = blobStream;
        }
        else
        {
            // The specified blob is something else rather than a Stream, we perform
serialization of T into a new stream instance.
            blobData = new MemoryStream();
            this.dataSerializer.Serialize(blob, blobData);
        }

        var container =
this.blobStorage.GetContainerReference(CloudUtility.GetSafeContainerName(containerName));
        StorageErrorCode lastErrorCode = StorageErrorCode.None;

        Func<string> uploadAction = () =>
        {
            var cloudBlob = container.GetBlobReference(blobName);
            return UploadBlob(cloudBlob, blobData, overwrite, expectedEtag);
        };

        try
        {
```

```
            // First attempt - perform upload and let the UploadBlob method handle any
retry conditions.
            string eTag = uploadAction();

            if (!String.IsNullOrEmpty(eTag))
            {
                actualEtag = eTag;
                return true;
            }
        }
        catch (StorageClientException ex)
        {
            lastErrorCode = ex.ErrorCode;

            if (!(lastErrorCode == StorageErrorCode.ContainerNotFound || lastErrorCode ==
StorageErrorCode.ResourceNotFound || lastErrorCode ==
StorageErrorCode.BlobAlreadyExists))
            {
                // Anything other than "not found" or "already exists" conditions will be
considered as a runtime error.
                throw;
            }
        }

        if (lastErrorCode == StorageErrorCode.ContainerNotFound)
        {
            // Failover action #1: create the target container and try again. This time,
use a retry policy to wrap calls to the
            // UploadBlob method.
            string eTag = this.retryPolicy.ExecuteAction<string>(() =>
            {
                CreateContainer(containerName);
                return uploadAction();
            });

            return !String.IsNullOrEmpty(actualEtag = eTag);
        }

        if (lastErrorCode == StorageErrorCode.BlobAlreadyExists && overwrite)
        {
            // Failover action #2: Overwrite was requested but BlobAlreadyExists has
still been returned.
```

```
            // Delete the original blob and try to upload again.
            string eTag = this.retryPolicy.ExecuteAction<string>(() =>
            {
                var cloudBlob = container.GetBlobReference(blobName);
                cloudBlob.DeleteIfExists();

                return uploadAction();
            });

            return !String.IsNullOrEmpty(actualEtag = eTag);
        }
    }
    finally
    {
        // Only dispose the blob data stream if it was newly created.
        if (blobData != null && null == blobStream)
        {
            blobData.Dispose();
        }

        TraceManager.CloudStorageComponent.TraceOut(callToken, actualEtag);
    }

    return false;
}
```

In summary, the above code takes a blob of type <T> and first checks if this is already a serialized image of a message in the form of a **Stream** object. All large messages that are relayed to the overflow storage by the **ReliableCloudQueueStorage** component will arrive as streams, ready for persistence. Next, the **UploadBlob** action is invoked, which in turn calls into Blob Service Client API, specifically its UploadFromStream operation. If a large message blob fails to upload successfully, the code inspects the error returned by the Blob Service and provides a failover path for 2 conditions: ContainerNotFound and BlobAlreadyExists. In the event that the target blob container is not found, the code will attempt to create the missing container. It performs this action within a retry-aware scope to improve reliability and increase resilience to transient failures. The second failover path is intended to handle a situation where a blob with the same name already exists. The code will remove the existing blob, provided the overwrite behavior is enabled. After removal, the upload of the new blob will be retried. Again, this operation is performed inside a retry-aware scope for increased reliability.

Now that I can store large messages in a blob container, it's time to design another implementation of the **ICloudBlobStorage** interface that will leverage the Windows Azure Caching Service. For consistency, let's start off with its constructors:

```
/// <summary>
```

```
/// Implements reliable generics-aware layer for Windows Azure Caching Service.
/// </summary>
public class ReliableCloudCacheStorage : ICloudBlobStorage
{
    private readonly RetryPolicy retryPolicy;
    private readonly ICloudStorageEntitySerializer dataSerializer;
    private readonly DataCacheFactory cacheFactory;
    private readonly DataCache cache;

    /// <summary>
    /// Initializes a new instance of the ReliableCloudCacheStorage class using the
specified storage account information
    /// custom retry policy and custom implementation of ICloudStorageEntitySerializer
interface.
    /// </summary>
    /// <param name="endpointInfo">The endpoint details for Windows Azure Caching
Service.</param>
    /// <param name="retryPolicy">The custom retry policy that will ensure reliable
access to the Caching Service.</param>
    /// <param name="dataSerializer">The component which performs custom serialization
and deserialization of cache items.</param>
    public ReliableCloudCacheStorage(CachingServiceEndpointInfo endpointInfo, RetryPolicy
retryPolicy, ICloudStorageEntitySerializer dataSerializer)
    {
        Guard.ArgumentNotNull(endpointInfo, "endpointInfo");
        Guard.ArgumentNotNull(retryPolicy, "retryPolicy");
        Guard.ArgumentNotNull(dataSerializer, "dataSerializer");

        this.retryPolicy = retryPolicy;
        this.dataSerializer = dataSerializer;

        var cacheServers = new List<DataCacheServerEndpoint>(1);
        cacheServers.Add(new DataCacheServerEndpoint(endpointInfo.ServiceHostName,
endpointInfo.CachePort));

        var cacheConfig = new DataCacheFactoryConfiguration()
```

```
        {
            Servers = cacheServers,
            MaxConnectionsToServer = 1,
            IsCompressionEnabled = false,
            SecurityProperties = new
DataCacheSecurity(endpointInfo.SecureAuthenticationToken, endpointInfo.SslEnabled),
            // The ReceiveTimeout value has been modified as per recommendations provided
in
            // http://blogs.msdn.com/b/akshar/archive/2011/05/01/azure-appfabric-caching-
errorcode-lt-errca0017-gt-substatus-lt-es0006-gt-what-to-do.aspx
            TransportProperties = new DataCacheTransportProperties() { ReceiveTimeout =
TimeSpan.FromSeconds(45) }
        };

        this.cacheFactory = new DataCacheFactory(cacheConfig);
        this.cache = this.retryPolicy.ExecuteAction<DataCache>(() =>
        {
            return this.cacheFactory.GetDefaultCache();
        });
    }
}
```

If you recall from earlier considerations, one of the key technical design decisions was to take advantage of both the Blob Service and Caching Service for storing large messages. The cache option is mostly suited for transient objects not exceeding the recommended payload size of 8MB. The blob option is essentially for everything else. Overall, this decision introduces the need for a **hybrid overflow store**. The foundation for building a hybrid store is already in the codebase. It's just the matter of marrying the existing artifacts together as follows:

```
/// <summary>
/// Implements reliable generics-aware storage layer combining Windows Azure Blob storage
and
/// Windows Azure Caching Service in a hybrid mode.
/// </summary>
public class ReliableHybridBlobStorage : ICloudBlobStorage
{
    private readonly ICloudBlobStorage blobStorage;
    private readonly ICloudBlobStorage cacheStorage;
    private readonly ICloudStorageEntitySerializer dataSerializer;
    private readonly IList<ICloudBlobStorage> storageList;
```

```
    /// <summary>
    /// Initializes a new instance of the ReliableHybridBlobStorage class using the
specified storage account information, caching
    /// service endpoint, custom retry policies and a custom implementation of
ICloudStorageEntitySerializer interface.
    /// </summary>
    /// <param name="storageAccountInfo">The access credentials for Windows Azure storage
account.</param>
    /// <param name="storageRetryPolicy">The custom retry policy that will ensure
reliable access to the underlying blob storage.</param>
    /// <param name="cacheEndpointInfo">The endpoint details for Windows Azure Caching
Service.</param>
    /// <param name="cacheRetryPolicy">The custom retry policy that will ensure reliable
access to the Caching Service.</param>
    /// <param name="dataSerializer">The component which performs serialization and
deserialization of storage objects.</param>
    public ReliableHybridBlobStorage(StorageAccountInfo storageAccountInfo, RetryPolicy
storageRetryPolicy, CachingServiceEndpointInfo cacheEndpointInfo, RetryPolicy
cacheRetryPolicy, ICloudStorageEntitySerializer dataSerializer)
    {
        Guard.ArgumentNotNull(storageAccountInfo, "storageAccountInfo");
        Guard.ArgumentNotNull(storageRetryPolicy, "storageRetryPolicy");
        Guard.ArgumentNotNull(cacheEndpointInfo, "cacheEndpointInfo");
        Guard.ArgumentNotNull(cacheRetryPolicy, "cacheRetryPolicy");
        Guard.ArgumentNotNull(dataSerializer, "dataSerializer");

        this.dataSerializer = dataSerializer;
        this.storageList = new List<ICloudBlobStorage>(2);

        this.storageList.Add(this.cacheStorage = new
ReliableCloudCacheStorage(cacheEndpointInfo, cacheRetryPolicy, dataSerializer));
        this.storageList.Add(this.blobStorage = new
ReliableCloudBlobStorage(storageAccountInfo, storageRetryPolicy, dataSerializer));
    }
}
```

At this point, I conclude the saga by including one more code snippet showing the implementation of the *Put* operation in the hybrid overflow store.

```
/// <summary>
/// Puts a blob into the underlying storage. If the blob with the same name already
exists, overwrite behavior can be customized.
```

```
/// </summary>
/// <typeparam name="T">The type of the payload associated with the blob.</typeparam>
/// <param name="containerName">The target blob container name into which a blob will be
stored.</param>
/// <param name="blobName">The custom name associated with the blob.</param>
/// <param name="blob">The blob's payload.</param>
/// <param name="overwrite">The flag indicating whether or not overwriting the existing
blob is permitted.</param>
/// <returns>True if the blob was successfully put into the specified container,
otherwise false.</returns>
public bool Put<T>(string containerName, string blobName, T blob, bool overwrite)
{
    Guard.ArgumentNotNull(blob, "blob");

    bool success = false;
    Stream blobData = null;
    bool treatBlobAsStream = false;

    try
    {
        // Are we dealing with a stream already? If yes, just use it as is.
        if (IsStreamType(blob.GetType()))
        {
            blobData = blob as Stream;
            treatBlobAsStream = true;
        }
        else
        {
            // The specified item type is something else rather than a Stream, we perform
serialization of T into a new stream instance.
            blobData = new MemoryStream();

            this.dataSerializer.Serialize(blob, blobData);
            blobData.Seek(0, SeekOrigin.Begin);
        }

        try
        {
            // First, make an attempt to store the blob in the distributed cache.
            // Only use cache if blob size is optimal for this type of storage.
            if (CloudUtility.IsOptimalCacheItemSize(blobData.Length))
            {
```

```
                success = this.cacheStorage.Put<Stream>(containerName, blobName,
blobData, overwrite);
            }
        }
        finally
        {
            if (!success)
            {
                // The cache option was unsuccessful, fail over to the blob storage as
per design decision.
                success = this.blobStorage.Put<Stream>(containerName, blobName, blobData,
overwrite);
            }
        }
    }
    finally
    {
        if (!treatBlobAsStream && blobData != null)
        {
            // Only dispose the blob data stream if it was newly created.
            blobData.Dispose();
        }
    }

    return success;
}
```

This article would be considered incomplete if I failed to provide some examples of how the storage abstraction layer discussed above can be consumed from a client application. I will combine these examples with a test application that will also validate the technical implementation.

Validation

In order to prove that large messages can successfully pass back and forth through the newly implemented storage abstraction layer, a very simple console application was put together. In the first step, it takes a sample XML document of 90MB in size and puts it on a Windows Azure queue. In the second step, it consumes a message from the queue. The message should indeed be the original XML document which is written back to the disk under a different name to be able to compare the file size and its content. In between these steps, the application enters a pause mode during which you can explore the content of the queue and respective message overflow store such as cache or blob container. The source code for the test application is provided below.

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

```
using System.Configuration;
using System.Xml.Linq;

using Contoso.Cloud.Integration.Framework;
using Contoso.Cloud.Integration.Framework.Configuration;
using Contoso.Cloud.Integration.Azure.Services.Framework.Storage;

namespace LargeQueueMessageTest
{
    class Program
    {
        static void Main(string[] args)
        {
            // Check if command line arguments were in fact supplied.
            if (null == args || args.Length == 0) return;

            // Read storage account and caching configuration sections.
            var cacheServiceSettings =
ConfigurationManager.GetSection("CachingServiceConfiguration") as
CachingServiceConfigurationSettings;
            var storageAccountSettings =
ConfigurationManager.GetSection("StorageAccountConfiguration") as
StorageAccountConfigurationSettings;

            // Retrieve cache endpoint and specific storage account definitions.
            var cacheServiceEndpoint =
cacheServiceSettings.Endpoints.Get(cacheServiceSettings.DefaultEndpoint);
            var queueStorageAccount =
storageAccountSettings.Accounts.Get(storageAccountSettings.DefaultQueueStorage);
            var blobStorageAccount =
storageAccountSettings.Accounts.Get(storageAccountSettings.DefaultBlobStorage);

            PrintInfo("Using storage account definition: {0}",
queueStorageAccount.AccountName);
            PrintInfo("Using caching service endpoint name: {0}",
cacheServiceEndpoint.Name);

            string fileName = args[0], queueName = "LargeMessageQueue";
            string newFileName = String.Format("{0}_Copy{1}",
Path.GetFileNameWithoutExtension(fileName), Path.GetExtension(fileName));

            long fileSize = -1, newFileSize = -1;
```

```
            try
            {
                // Load the specified file into XML DOM.
                XDocument largeXmlDoc = XDocument.Load(fileName);

                // Instantiate the large message overflow store and use it to instantiate
a queue storage abstraction component.
                using (var overflowStorage = new
ReliableHybridBlobStorage(blobStorageAccount, cacheServiceEndpoint))
                using (var queueStorage = new
ReliableCloudQueueStorage(queueStorageAccount, overflowStorage))
                {
                    PrintInfo("\nAttempting to store a message of {0} bytes in size on a
Windows Azure queue", fileSize = (new FileInfo(fileName)).Length);

                    // Enqueue the XML document. The document's size doesn't really
matter any more.
                    queueStorage.Put<XDocument>(queueName, largeXmlDoc);

                    PrintSuccess("The message has been succcessfully placed into a
queue.");
                    PrintWaitMsg("\nYou can now inspect the content of the {0} queue and
respective blob container...", queueName);

                    // Dequeue a message from the queue which is expected to be our
original XML document.
                    XDocument docFromQueue = queueStorage.Get<XDocument>(queueName);

                    // Save it under a new name.
                    docFromQueue.Save(newFileName);

                    // Delete the message. Should remove the metadata message from the
queue as well as blob holding the message data.
                    queueStorage.Delete<XDocument>(docFromQueue);

                    PrintInfo("\nThe message retrieved from the queue is {0} bytes in
size.", newFileSize = (new FileInfo(newFileName)).Length);

                    // Perform very basic file size-based comparison. In the reality, we
should have checked the document structurally.
                    if (fileSize > 0 && newFileSize > 0 && fileSize == newFileSize)
```

```
                    {
                        PrintSuccess("Test passed. This is expected behavior in any code
written by CAT.");
                    }
                    else
                    {
                        PrintError("Test failed. This should have never happened in the
code written by CAT.");
                    }
                }
            }
            catch (Exception ex)
            {
                PrintError("ERROR: {0}", ExceptionTextFormatter.Format(ex));
            }
            finally
            {
                Console.ReadLine();
            }
        }

        private static void PrintInfo(string format, params object[] parameters)
        {
            Console.ForegroundColor = ConsoleColor.White;
            Console.WriteLine(format, parameters);
            Console.ResetColor();
        }

        private static void PrintSuccess(string format, params object[] parameters)
        {
            Console.ForegroundColor = ConsoleColor.Green;
            Console.WriteLine(format, parameters);
            Console.ResetColor();
        }

        private static void PrintError(string format, params object[] parameters)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine(format, parameters);
            Console.ResetColor();
        }
```

```
        private static void PrintWaitMsg(string format, params object[] parameters)
        {
            Console.ForegroundColor = ConsoleColor.Gray;
            Console.WriteLine(format, parameters);
            Console.ResetColor();
            Console.ReadLine();
        }
    }
}
```

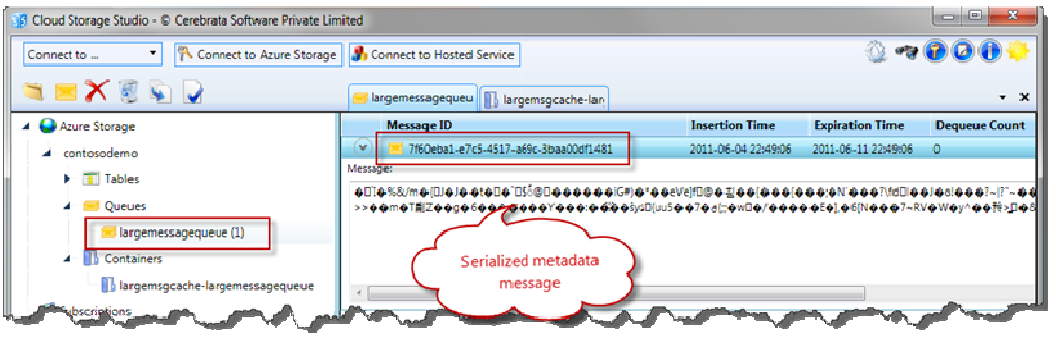
For the sake of completeness, below is the application configuration file that was used during testing. If you are going to try the test application out, please make sure you modify your copy of *app.config* and add the actual storage account credentials and caching service endpoint information.

```
<?xml version="1.0"?>
<configuration>
<configSections>
<section name="CachingServiceConfiguration"
type="Contoso.Cloud.Integration.Framework.Configuration.CachingServiceConfigurationSettin
gs, Contoso.Cloud.Integration.Framework, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=23eafc3765008062"/>
<section name="StorageAccountConfiguration"
type="Contoso.Cloud.Integration.Framework.Configuration.StorageAccountConfigurationSettin
gs, Contoso.Cloud.Integration.Framework, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=23eafc3765008062"/>
</configSections>

<CachingServiceConfiguration defaultEndpoint="YOUR-CACHE-NAMESPACE-GOES-HERE">
<add name="YOUR-CACHE-NAMESPACE-GOES-HERE" authToken="YOUR-CACHE-SECURITYTOKEN-GOES-
HERE"/>
</CachingServiceConfiguration>

<StorageAccountConfiguration defaultBlobStorage="My Azure Storage"
defaultQueueStorage="My Azure Storage">
<add name="My Azure Storage" accountName="YOUR-STORAGE-ACCOUNT-NAME-GOES-HERE"
accountKey="YOUR-STORAGE-ACCOUNT-KEY-GOES-HERE"/>
</StorageAccountConfiguration>
</configuration>
```

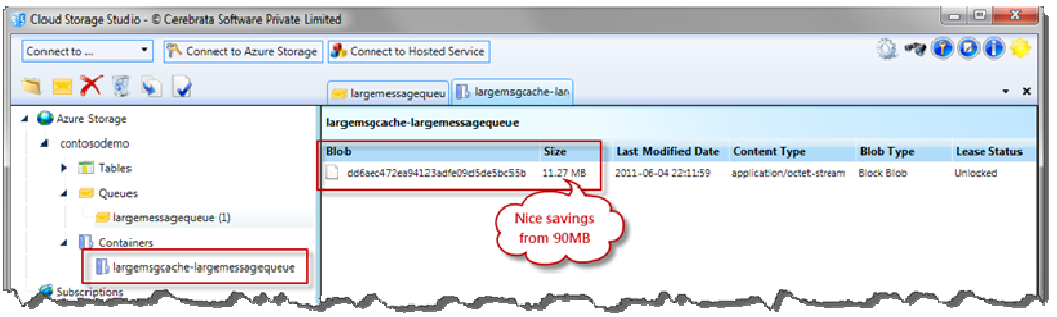
Provided the test application has been successfully compiled and executed, the output similar to the following is expected to appear in the console windows:

```
C:\Windows\system32\cmd.exe - DoMagic.exe LargeMessage.xml

d:\Projects\LargeMsgTest>DoMagic.exe LargeMessage.xml
Using storage account definition: contosodemo
Using caching service endpoint name: contosocache

Attempting to store a message of 99514312 bytes in size on a Windows Azure queue
The message has been succcessfully placed into a queue.

You can now inspect the content of the LargeMessageQueue queue and respective blob co
ntainer...

The message retrieved from the queue is 99514312 bytes in size.
Test passed. This is expected behavior in any code written by CAT.
```

If you peek into the storage account used by the test application, the following message will appear on the queue:

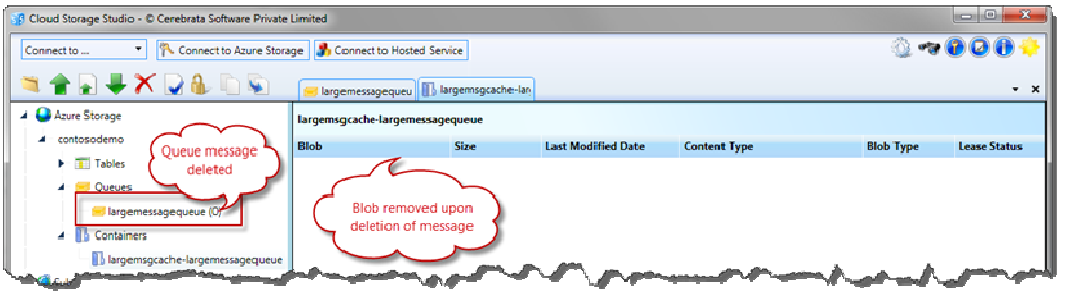


Since the test message was large enough to overflow directly into the blob storage, the following screenshot depicts the expected content inside the respective blob container while the test application is paused:



Note how the original 90MB XML document used in my test became a 11MB blob. This reflects the **87% savings on storage and bandwidth** which was the result of applying XML binary serialization. Given the target class of scenarios, XML binary serialization + compression is the first and best choice.

Once the test application proceeds with deletion of the queue message, the metadata message is expected to be removed along with the blob holding the message data as shown on the screenshot below:



The example shown above reflects a simplistic view on the lifecycle of a large message. It is intended to highlight the fundamentals of the storage abstraction layer such as large message routing into blob store, transparent compression, automatic removal of both message parts. I guess, it's now the right time to jump to a conclusion.

Conclusion

As we have seen, the use of Windows Azure Queues can be extended to support messages larger than 64KB by leveraging the Windows Azure Caching Service and Windows Azure Blob Service without adding any additional technical restrictions on the client. In fact, I have shown that with a little extra work you can enhance the messaging experience for the client by providing quality of life improvements such as:

- **Transparent message compression** to reduce storage costs and save bandwidth into/out of the datacenter.
- **Transparent, easily customizable overflow** of large messages to Cache or Blob storage.
- **Generics support** that allows you to easily store any object type.
- **Automatic handling of transient conditions** for improved reliability.

As I mentioned earlier, while this solution can use both distributed cache and blob store for overflow storage, the use of Windows Azure Caching Service incurs additional costs. You should carefully evaluate the storage requirements of your project and perform a cost analysis based on projected number of messages and message size before deciding to enable overflow using cache.

While this solution provides an easy to use means of supporting large messages on Windows Azure queues, there is always room for improvement. Some examples of value-add features that are not incorporated in this solution and which you may wish to add are:

- The ability to configure the type of large message overflow store in the application configuration.
- The additional custom serializers in case the default one does not meet your performance goals or functional needs (for instance, you don't need the default compression).
- An item in the blob's metadata acting as a breadcrumb allowing you to scan through your blob storage and quickly find out if you have any orphaned large message blobs (zombies).
- A "garbage collector" component that will ensure timely removal of any orphaned blobs from the overflow message store (in case queues are also accessed by components other than storage abstraction layer implemented here).

The accompanying sample code is available for download from the MSDN Code Gallery. Note that all source code files are governed by the Microsoft Public License as explained in the corresponding legal notices.

Additional Resources/References

For more information on the topic discussed in this article, please refer to the following:

- Data Storage Offerings on the Windows Azure Platform article in the TechNet Wiki Library.
- Windows Azure Storage Architecture Overview post on the Windows Azure Storage Team blog.
- Best Practices for Maximizing Scalability and Cost Effectiveness of Queue-Based Messaging Solutions on Windows Azure topic in the MSDN Library.

Best Practices for Maximizing Scalability and Cost Effectiveness of Queue-Based Messaging Solutions on Windows Azure

Authored by: Valery Mizonov

Reviewed by: Brad Calder, Sidney Higa, Christian Martinez, Steve Marx, Curt Peterson, Paolo Salvatori, and Trace Young

This article offers prescriptive guidance and best practices for building scalable, highly efficient and cost effective queue-based messaging solutions on the Windows Azure platform. The intended audience for this article includes solution architects and developers designing and implementing cloud-based solutions which leverage the Windows Azure platform's queue storage services.

Abstract

A traditional queue-based messaging solution utilizes the concept of a message storage location known as a message queue, which is a repository for data that will be sent to or received from one or more participants, typically via an asynchronous communication mechanism.

The queue-based data exchange represents the foundation of a reliable and highly scalable messaging architecture capable of supporting a range of powerful scenarios in the distributed computing environment. Whether it's high-volume work dispatch or durable messaging, a message queuing technology can step in and provide first-class capabilities to address the different requirements for asynchronous communication at scale.

The purpose of this article is to examine how developers can take advantage of particular design patterns in conjunction with capabilities provided by the Windows Azure platform to build optimized and cost-effective queue-based messaging solutions. The article takes a deeper look at most commonly used approaches to implementing queue-based interactions in Windows Azure solutions, and provides recommendations for improving performance, increasing scalability and reducing operating expense.

The underlying discussion is mixed with relevant best practices, hints and recommendations where appropriate. The scenario described in this article highlights a technical implementation that is based upon a real-world customer project.

Customer Scenario

For the sake of a concrete example, we will generalize a real-world customer scenario as follows.

A SaaS solution provider launches a new billing system implemented as a Windows Azure application servicing the business needs for customer transaction processing at scale. The key premise of the solution is centered upon the ability to offload compute-intensive workload to the cloud and leverage the elasticity of the Windows Azure infrastructure to perform the computationally intensive work.

The on-premises element of the end-to-end architecture consolidates and dispatches large volumes of transactions to a Windows Azure hosted service regularly throughout the day. Volumes vary from a few thousands to hundreds of thousands transactions per submission,

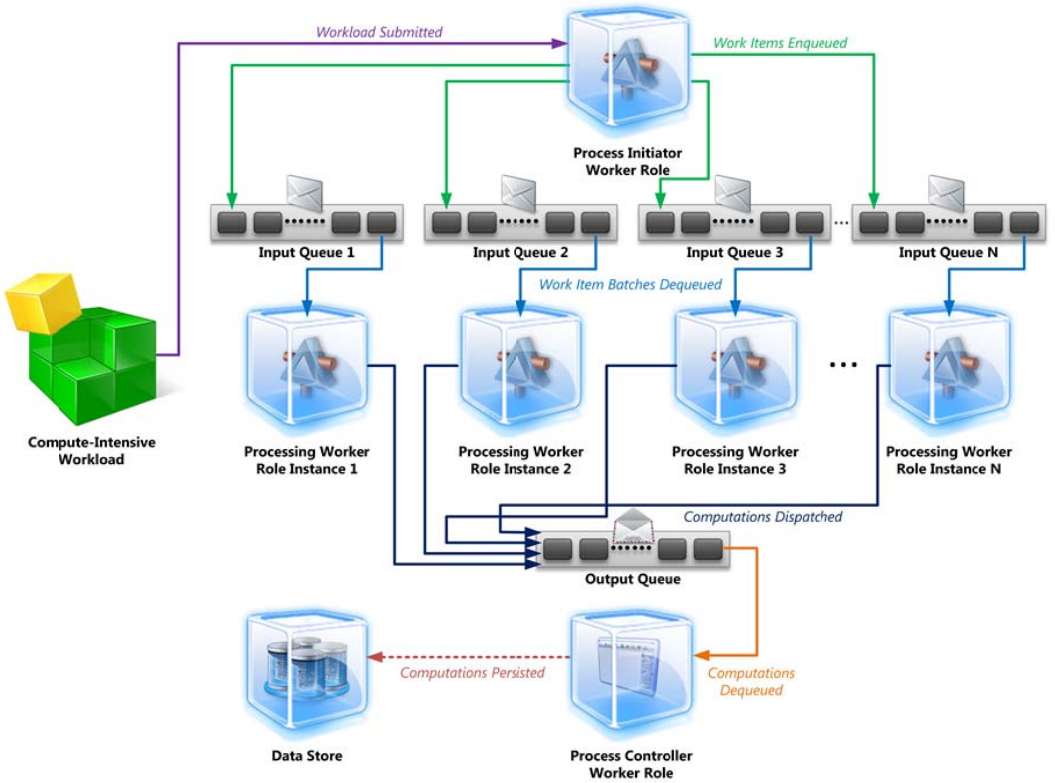
reaching millions of transactions per day. Additionally, assume that the solution must satisfy a SLA-driven requirement for a guaranteed maximum processing latency.

The solution architecture is founded on the distributed map-reduce design pattern and is comprised of a multi-instance worker role-based cloud tier using the Windows Azure queue storage for work dispatch. Transaction batches are received by Process Initiator worker role instance, decomposed (de-batched) into smaller work items and enqueued into a collection of Windows Azure queues for the purposes of load distribution.

Workload processing is handled by multiple instances of the processing worker role fetching work items from queues and passing them through computational procedures. The processing instances employ multi-threaded queue listeners to implement parallel data processing for optimal performance.

The processed work items are routed into a dedicated queue from which these are dequeued by the Process Controller worker role instance, aggregated and persisted into a data store for data mining, reporting and analysis.

The solution architecture can be depicted as follows:



The diagram above depicts a typical architecture for scaling out large or complex compute workloads. The queue-based message exchange pattern adopted by this architecture is also very typical for many other Windows Azure applications and services which need to communicate with each other via queues. This enables taking a canonical approach to examining specific fundamental components involved in a queue-based message exchange.

Queue-Based Messaging Fundamentals

A typical messaging solution that exchanges data between its distributed components using message queues includes *publishers* depositing messages into queues and one or more *subscribers* intended to receive these messages. In most cases, the subscribers, sometimes referred to as *queue listeners*, are implemented as single- or multi-threaded processes, either continuously running or initiated on demand as per a scheduling pattern.

At a higher level, there are two primary dispatch mechanisms used to enable a queue listener to receive messages stored on a queue:

- **Polling (pull-based model):** A listener monitors a queue by checking the queue at regular intervals for new messages. When the queue is empty, the listener continues polling the queue, periodically backing off by entering a sleep state.
- **Triggering (push-based model):** A listener subscribes to an event that is triggered (either by the publisher itself or by a queue service manager) whenever a message arrives on a queue. The listener in turn can initiate message processing thus not having to poll the queue in order to determine whether or not any new work is available.

It is also worth mentioning that there are different flavors of both mechanisms. For instance, polling can be blocking and non-blocking. Blocking keeps a request on hold until a new message appears on a queue (or timeout is encountered) whereas a non-blocking request completes immediately if there is nothing on a queue. With a triggering model, a notification can be pushed to the queue listeners either for every new message, only when the very first message arrives to an empty queue or when queue depth reaches a certain level.

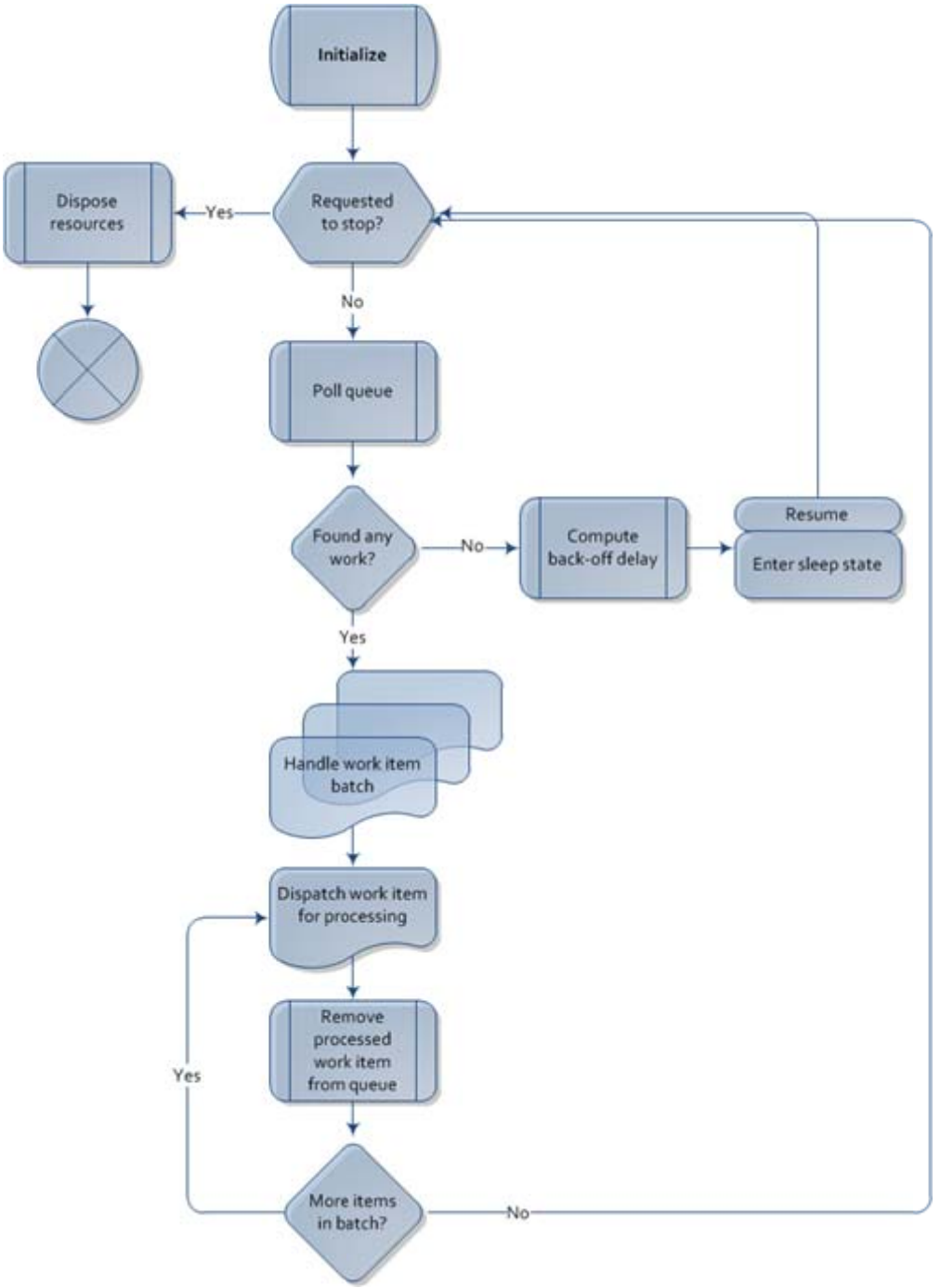
Note

The dequeue operations supported by Windows Azure Queue Service API are non-blocking. This means that the API methods such as GetMessage or GetMessages will return immediately if there is no message found on a queue. By contrast, the Windows Azure Service Bus queues offer blocking receive operations which block the calling thread until a message arrives on a queue or a specified timeout period has elapsed.

The most common approach to implementing queue listeners in Windows Azure solutions today can be summarized as follows:

1. A listener is implemented as an application component that is instantiated and executed as part of a worker role instance.
2. The lifecycle of the queue listener component would often be bound to the run time of the hosting role instance.
3. The main processing logic is comprised of a loop in which messages are dequeued and dispatched for processing.
4. Should no messages be received, the listening thread enters a sleep state the duration of which is often driven by an application-specific back-off algorithm.
5. The receive loop is being executed and a queue is being polled until the listener is notified to exit the loop and terminate.

The following flowchart diagram depicts the logic commonly used when implementing a queue listener with a polling mechanism in Windows Azure applications:



Note

For purposes of this article, more complex design patterns, for example those that require the use of a central queue manager (broker) are not used.

The use of a classic queue listener with a polling mechanism may not be the optimal choice when using Windows Azure queues because the Windows Azure pricing model measures storage transactions in terms of application requests performed against the queue, regardless of if the

queue is empty or not. The purpose of the next sections is to discuss some techniques for maximizing performance and minimizing the cost of queue-based messaging solutions on the Windows Azure platform.

Best Practices for Performance, Scalability & Cost Optimization

In this section we must examine how to improve the relevant design aspects to achieve higher performance, better scalability and cost efficiency.

Perhaps, the easiest way of qualifying an implementation pattern as a "more efficient solution" would be through the design which meets the following goals:

- **Reduces operational expenditures** by removing a significant portion of storage transactions that don't derive any usable work.
- **Eliminates excessive latency** imposed by a polling interval when checking a queue for new messages.
- **Scales up and down dynamically** by adapting processing power to volatile volumes of work.

The implementation pattern should also meet these goals without introducing a level of complexity that effectively outweighs the associated benefits.

Best Practices for Optimizing Storage Transaction Costs

When evaluating the total cost of ownership (TCO) and return on investment (ROI) for a solution deployed on the Windows Azure platform, the volume of storage transactions is one of the main variables in the TCO equation. Reducing the number of transactions against Windows Azure queues decreases the operating costs as it relates to running solutions on Windows Azure.

In the context of a queue-based messaging solution, the volume of storage transactions can be reduced using a combination of the following methods:

1. When putting messages in a queue, **group related messages** into a single larger batch, compress and store the compressed image in a blob storage and use the queue to keep a reference to the blob holding the actual data.
2. When retrieving messages from a queue, **batch multiple messages together** in a single storage transaction. The *GetMessages* method in the Queue Service API enables de-queuing the specified number of messages in a single transaction (see the note below).
3. When checking the presence of work items on a queue, **avoid aggressive polling intervals** and **implement a back-off delay** that increases the time between polling requests if a queue remains continuously empty.
4. **Reduce the number of queue listeners** – when using a pull-based model, use only 1 queue listener per role instance when a queue is empty. To further reduce the number of queue listeners per role instance to zero, use a notification mechanism to instantiate queue listeners when the queue receives work items.
5. If queues remain empty for most of the time, **automatically scale down the number of role instances** and continue to monitor relevant system metrics to determine if and when the application should scale up the number of instances to handle increasing workload.

Most of the above recommendations can be translated into a fairly generic implementation that handles message batches and encapsulates many of the underlying queue/blob storage and thread management operations. Later in this article, we will examine how to do this.

Important

When retrieving messages via the *GetMessages* method, the maximum batch size supported by Queue Service API in a single dequeue operation is limited to 32.

Generally speaking, the cost of Windows Azure queue transactions increases linearly as the number of queue service clients increases, such as when scaling up the number of role instances or increasing the number of dequeue threads. To illustrate the potential cost impact of a solution design that does not take advantage of the above recommendations; we will provide an example backed up by concrete numbers.

The Cost Impact of Inefficient Design

If the solution architect does not implement relevant optimizations, the billing system architecture described above will likely incur excessive operating expenses once the solution is deployed and running on the Windows Azure platform. The reasons for the possible excessive expense are described in this section.

As noted in the scenario definition, the business transaction data arrives at regular intervals. However, let's assume that the solution is busy processing workload just 25% of the time during a standard 8-hour business day. That results in 6 hours (8 hours * 75%) of "idle time" when there may not be any transactions coming through the system. Furthermore, the solution will not receive any data at all during the 16 non-business hours every day.

During the idle period totaling 22 hours, the solution is still performing attempts to dequeue work as it has no explicit knowledge when new data arrives. During this time window, each individual dequeue thread will perform up to 79,200 transactions (22 hours * 60 min * 60 transactions/min) against an input queue, assumed a default polling interval of 1 second.

As previously mentioned, the pricing model in the Windows Azure platform is based upon individual "storage transactions". A storage transaction is a request made by a user application to add, read, update or delete storage data. As of the writing of this whitepaper, storage transactions are billed at a rate of $0.01 for 10,000 transactions (not taking into account any promotional offerings or special pricing arrangements).

Important

When calculating the number of queue transactions, keep in mind that putting a single message on a queue would be counted as 1 transaction, whereas consuming a message is often a 2-step process involving the retrieval followed by a request to remove the message from the queue. As a result, a successful dequeue operation will attract 2 storage transactions. Please note that even if a dequeue request results in no data being retrieved; it still counts as a billable transaction.

The storage transactions generated by a single dequeue thread in the above scenario will add approximately $2.38 (79,200 / 10,000 * $0.01 * 30 days) to a monthly bill. In comparison, 200 dequeue threads (or, alternatively, 1 dequeue thread in 200 worker role instances) will increase the cost by $457.20 per month. That is the cost incurred when the solution was not performing

any computations at all, just checking on the queues to see if any work items are available. The above example is abstract as no one would implement their service this way, which is why it is important to do the optimizations described next.

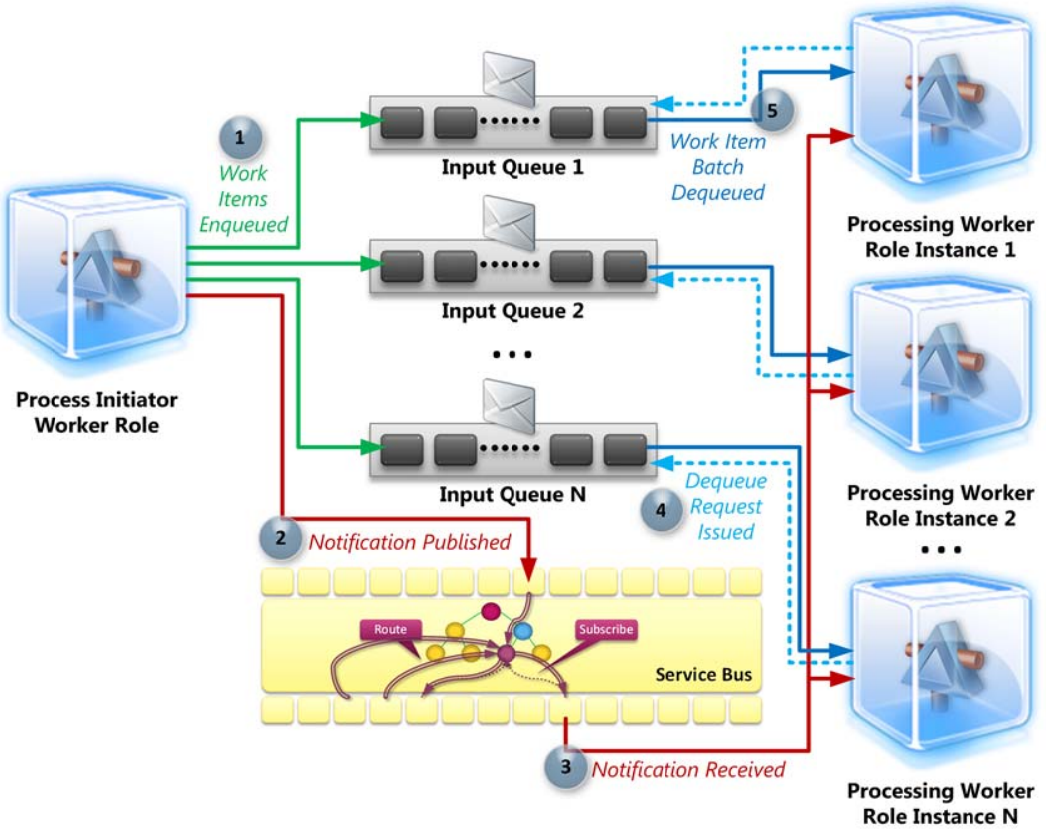
Best Practices for Eliminating Excessive Latency

To optimize performance of queue-based Windows Azure messaging solutions one approach is to use the publish/subscribe messaging layer provided with the Windows Azure Service Bus, as described in this section.

In this approach, developers will need to focus on creating a combination of polling and real-time push-based notifications, enabling the listeners to subscribe to a notification event (trigger) that is raised upon certain conditions to indicate that a new workload is put on a queue. This approach enhances the traditional queue polling loop with a publish/subscribe messaging layer for dispatching notifications.

In a complex distributed system, this approach would necessitate the use of a “message bus” or “message-oriented middleware” to ensure that notifications can be reliably relayed to one or more subscribers in a loosely coupled fashion. Windows Azure Service Bus is a natural choice for addressing messaging requirements between loosely coupled distributed application services running on Windows Azure and running on-premises. It is also a perfect fit for a “message bus” architecture that will enable exchanging notifications between processes involved in queue-based communication.

The processes engaged in a queue-based message exchange could employ the following pattern:



Specifically, and as it relates to the interaction between queue service publishers and subscribers, the same principles that apply to the communication between Windows Azure role instances would meet the majority of requirements for push-based notification message exchange. We have already covered these fundamentals in How to Simplify & Scale Inter-Role Communication Using Windows Azure Service Bus.

Important

The use of the Windows Azure Service Bus is subject to a pricing model that takes into account the volume of messaging operations against a Service Bus messaging entity such as a queue or a topic.

It is therefore important to perform a cost-benefit analysis to assess the pros and cons of introducing the Service Bus into a given architecture. Along those lines, it is worth evaluating whether or not the introduction of the notification dispatch layer based on the Service Bus would, in fact, lead to cost reduction that can justify the investments and additional development efforts.

For more information on the pricing model for Service Bus, please refer to the relevant sections in Windows Azure Platform FAQs.

While the impact on latency is fairly easy to address with a publish/subscribe messaging layer, a further cost reduction could be realized by using dynamic (elastic) scaling, as described in the next section.

Best Practices for Dynamic Scaling

The Windows Azure platform makes it possible for customers to scale up and down faster and easier than ever before. The ability to adapt to volatile workloads and variable traffic is one of the primary value propositions of the cloud platform. This means that “scalability” is no longer an expensive IT vocabulary term, it is now an out-of-the-box feature that can be programmatically enabled on demand in a well-architected cloud solution.

Dynamic scaling is the technical capability of a given solution to adapt to fluctuating workloads by increasing and reducing working capacity and processing power at runtime. The Windows Azure platform natively supports dynamic scaling through the provisioning of a distributed computing infrastructure on which compute hours can be purchased as needed.

It is important to differentiate between the following 2 types of dynamic scaling on the Windows Azure platform:

- **Role instance scaling** refers to adding and removing additional web or worker role instances to handle the point-in-time workload. This often includes changing the instance count in the service configuration. Increasing the instance count will cause Windows Azure runtime to start new instances whereas decreasing the instance count will in turn cause it to shut down running instances.
- **Process (thread) scaling** refers to maintaining sufficient capacity in terms of processing threads in a given role instance by tuning the number of threads up and down depending on the current workload.

Dynamic scaling in a queue-based messaging solution would attract a combination of the following general recommendations:

1. **Monitor key performance indicators** including CPU utilization, queue depth, response times and message processing latency.
2. **Dynamically increase or decrease the number of role instances** to cope with the spikes in workload, either predictable or unpredictable.
3. **Programmatically expand and trim down the number of processing threads** to adapt to variable load conditions handled by a given role instance.
4. **Partition and process fine-grained workloads concurrently** using the Task Parallel Library in the .NET Framework 4.
5. **Maintain a viable capacity** in solutions with highly volatile workload in anticipation of sudden spikes to be able to handle them without the overhead of setting up additional instances.

The Service Management API makes it possible for a Windows Azure hosted service to modify the number of its running role instances by changing deployment configuration at runtime.

Note

The maximum number of Windows Azure small compute instances (or the equivalent number of other sized compute instances in terms of number of cores) in a typical subscription is limited to 20 by default. Any requests for increasing this quota should be raised with the Windows Azure Support team. For more information, see the Windows Azure Platform FAQs

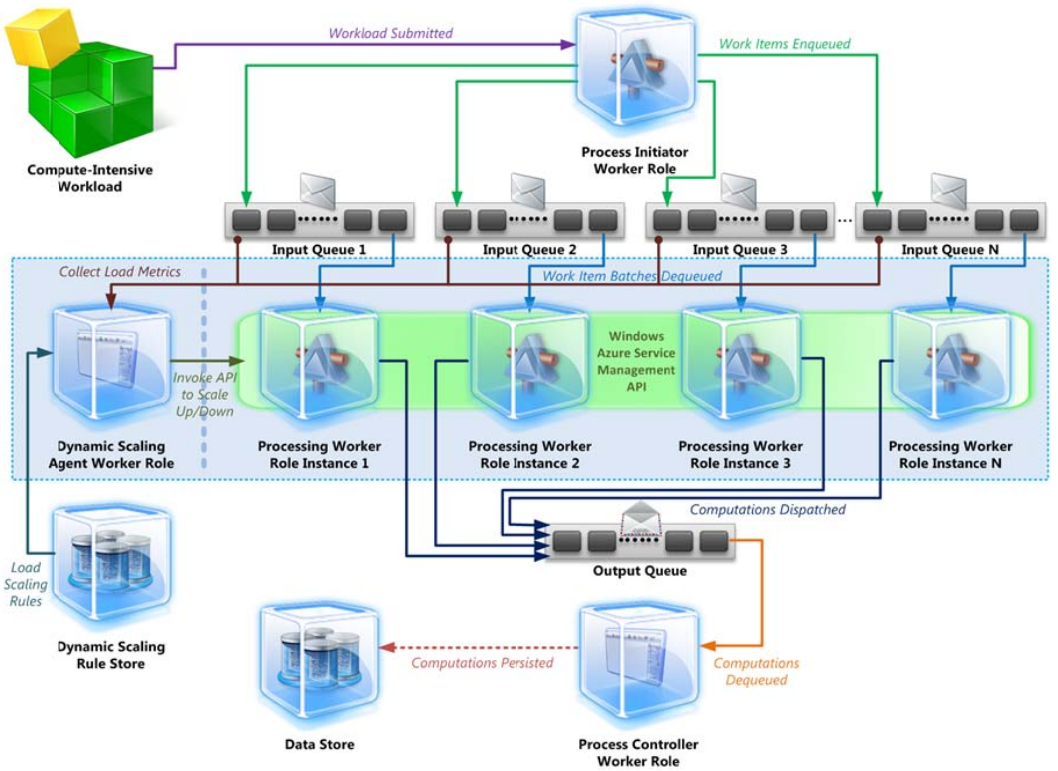
Dynamic scaling of the role instance count may not always be the most appropriate choice for handling load spikes. For instance, a new role instance can take a few seconds to spin up and there are currently no SLA metrics provided with respect to spin-up duration. Instead, a solution

may need to simply increase the number of worker threads to deal with the volatile workload increase. While workload is being processed, the solution will monitor the relevant load metrics and determine whether it needs to dynamically reduce or increase the number of worker processes.

Important

At present, the scalability target for a single Windows Azure queue is “constrained” at 500 transactions/sec. If an application attempts to exceed this target, for example, through performing queue operations from multiple role instance running hundreds of dequeue threads, it may result in HTTP 503 “Server Busy” response from the storage service. When this occurs, the application should implement a retry mechanism using exponential back-off delay algorithm. However, if the HTTP 503 errors are occurring regularly, it is recommended to use multiple queues and implement a partitioning-based strategy to scale across multiple queues.

In most cases, auto-scaling the worker processes is the responsibility of an individual role instance. By contrast, role instance scaling often involves a central element of the solution architecture that is responsible for monitoring performance metrics and taking the appropriate scaling actions. The diagram below depicts a service component called *Dynamic Scaling Agent* that gathers and analyzes load metrics to determine whether it needs to provision new instances or decommission idle instances.



It is worth noting that the scaling agent service can be deployed either as a worker role running on Windows Azure or as an on-premises service. Irrespectively of the deployment topology, the service will be able to access the Windows Azure queues.

To implement a dynamic scaling capability, consider the use of the Microsoft Enterprise Library Autoscaling Application Block that enables automatic scaling behavior in the solutions running on Windows Azure. The Autoscaling Application Block provides all of the functionality needed to define and monitor autoscaling in a Windows Azure application.

Now that we have covered the latency impact, storage transaction costs and dynamic scale requirements, it is a good time to consolidate our recommendations into a technical implementation.

Technical Implementation

In the previous sections, we have examined the key characteristics attributed to a well-designed messaging architecture based on the Windows Azure queue storage queues. We have looked at three main focus areas that help reduce processing latency, optimize storage transaction costs and improve responsiveness to fluctuating workloads.

This section is intended to provide a starting point to assist Windows Azure developers with implementing some of the patterns referenced in this whitepaper from a programming perspective.

Note

This section will focus on building an auto-scalable queue listener that supports both pull-based and push-based models. For advanced techniques in dynamic scaling at the role instance level, please refer to the Enterprise Library Autoscaling Application Block.

In addition, for the sake of brevity, we will only focus on some core functional elements and avoid undesired complexity by omitting much of the supporting infrastructure code from the code samples below. For the purposes of clarification, it's also worth pointing out that the technical implementation discussed below is not the only solution to a given problem space. It is intended to serve the purpose of a starting point upon which developers may derive their own more elegant solutions.

From this point onwards, this whitepaper will focus on the source code required to implement the patterns discussed above.

Building Generic Queue Listener

First, we define a contract that will be implemented by a queue listener component that is hosted by a worker role and listens on a Windows Azure queue.

```
/// Defines a contract that must be implemented by an extension responsible for listening
on a Windows Azure queue.
public interface ICloudQueueServiceWorkerRoleExtension
{
    /// Starts a multi-threaded queue listener that uses the specified number of dequeue
threads.
    void StartListener(int threadCount);
```

```
    /// Returns the current state of the queue listener to determine point-in-time load characteristics.
    CloudQueueListenerInfo QueryState();

    /// Gets or sets the batch size when performing dequeue operation against a Windows Azure queue.
    int DequeueBatchSize { get; set; }

    /// Gets or sets the default interval that defines how long a queue listener will be idle for between polling a queue.
    TimeSpan DequeueInterval { get; set; }

    /// Defines a callback delegate which will be invoked whenever the queue is empty.
    event WorkCompletedDelegate QueueEmpty;
}
```

The *QueueEmpty* event is intended to be used by a host. It provides the mechanism for the host to control the behavior of the queue listener when the queue is empty. The respective event delegate is defined as follows:

```
/// <summary>
/// Defines a callback delegate which will be invoked whenever an unit of work has been completed and the worker is
/// requesting further instructions as to next steps.
/// </summary>
/// <param name="sender">The source of the event.</param>
/// <param name="idleCount">The value indicating how many times the worker has been idle.</param>
/// <param name="delay">Time interval during which the worker is instructed to sleep before performing next unit of work.</param>
/// <returns>A flag indicating that the worker should stop processing any further units of work and must terminate.</returns>
public delegate bool WorkCompletedDelegate(object sender, int idleCount, out TimeSpan delay);
```

Handling queue items is easier if a listener can operate with generics as opposed to using “bare metal” SDK classes such as CloudQueueMessage. Therefore, we define a new interface that will be implemented by a queue listener capable of supporting generics-based access to queues:

```
/// <summary>
/// Defines a contract that must be supported by an extension that implements a generics-aware queue listener.
/// </summary>
/// <typeparam name="T">The type of queue item data that will be handled by the queue listener.</typeparam>
```

```
public interface ICloudQueueListenerExtension<T> : ICloudQueueServiceWorkerRoleExtension,
IObservable<T>
{
}
```

Note that we also enabled the generics-aware listener to push queue items to one or more subscribers through the implementation of the Observer design pattern by leveraging the IObservable<T> interface available in the .NET Framework 4.

We intend to keep a single instance of a component implementing the *ICloudQueueListenerExtension<T>* interface. However, we need to be able to run multiple dequeue threads (worker processes, or tasks for simplicity). Therefore, we add support for multi-threaded dequeue logic in the queue listener component. This is where we take advantage of the Task Parallel Library (TPL). The *StartListener* method will be responsible for spinning up the specified number of dequeue threads as follows:

```
/// <summary>
/// Starts the specified number of dequeue tasks.
/// </summary>
/// <param name="threadCount">The number of dequeue tasks.</param>
public void StartListener(int threadCount)
{
    Guard.ArgumentNotZeroOrNegativeValue(threadCount, "threadCount");

    // The collection of dequeue tasks needs to be reset on each call to this method.
    if (this.dequeueTasks.IsAddingCompleted)
    {
        this.dequeueTasks = new BlockingCollection<Task>(this.dequeueTaskList);
    }

    for (int i = 0; i < threadCount; i++)
    {
        CancellationToken cancellationToken = this.cancellationSignal.Token;
        CloudQueueListenerDequeueTaskState<T> workerState = new
CloudQueueListenerDequeueTaskState<T>(Subscriptions, cancellationToken,
this.queueLocation, this.queueStorage);

        // Start a new dequeue task and register it in the collection of tasks internally
managed by this component.
        this.dequeueTasks.Add(Task.Factory.StartNew(DequeueTaskMain, workerState,
cancellationToken, TaskCreationOptions.LongRunning, TaskScheduler.Default));
    }

    // Mark this collection as not accepting any more additions.
```

```
    this.dequeueTasks.CompleteAdding();
}
```

The *DequeueTaskMain* method implements the functional body of a dequeue thread. Its main operations are the following:

```
/// <summary>
/// Implements a task performing dequeue operations against a given Windows Azure queue.
/// </summary>
/// <param name="state">An object containing data to be used by the task.</param>
private void DequeueTaskMain(object state)
{
    CloudQueueListenerDequeueTaskState<T> workerState =
(CloudQueueListenerDequeueTaskState<T>)state;

    int idleStateCount = 0;
    TimeSpan sleepInterval = DequeueInterval;

    try
    {
        // Run a dequeue task until asked to terminate or until a break condition is
encountered.
        while (workerState.CanRun)
        {
            try
            {
                var queueMessages = from msg in
workerState.QueueStorage.Get<T>(workerState.QueueLocation.QueueName, DequeueBatchSize,
workerState.QueueLocation.VisibilityTimeout).AsParallel() where msg != null select msg;
                int messageCount = 0;

                // Process the dequeued messages concurrently by taking advantage of the
above PLINQ query.
                queueMessages.ForAll((message) =>
                {
                    // Reset the count of idle iterations.
                    idleStateCount = 0;

                    // Notify all subscribers that a new message requires processing.
                    workerState.OnNext(message);

                    // Once successful, remove the processed message from the queue.
                    workerState.QueueStorage.Delete<T>(message);
```

```
                // Increment the number of processed messages.
                messageCount++;
            });

            // Check whether or not we have done any work during this iteration.
            if (0 == messageCount)
            {
                // Increment the number of iterations when we were not doing any work
(e.g. no messages were dequeued).
                idleStateCount++;

                // Call the user-defined delegate informing that no more work is
available.
                if (QueueEmpty != null)
                {
                    // Check if the user-defined delegate has requested a halt to any
further work processing.
                    if (QueueEmpty(this, idleStateCount, out sleepInterval))
                    {
                        // Terminate the dequeue loop if user-defined delegate
advised us to do so.
                        break;
                    }
                }

                // Enter the idle state for the defined interval.
                Thread.Sleep(sleepInterval);
            }
        }
        catch (Exception ex)
        {
            if (ex is OperationCanceledException)
            {
                throw;
            }
            else
            {
                // Offload the responsibility for handling or reporting the error to
the external object.
                workerState.OnError(ex);

                // Sleep for the specified interval to avoid a flood of errors.
```

```
                    Thread.Sleep(sleepInterval);
                }
            }
        }
    }
    finally
    {
        workerState.OnCompleted();
    }
}
```

A couple of points are worth making with respect to the *DequeueTaskMain* method implementation.

First, we are taking advantage of the Parallel LINQ (PLINQ) when dispatching messages for processing. The main advantage of PLINQ here is to speed up message handling by executing the query delegate on separate worker threads on multiple processors in parallel whenever possible.

Note

Since query parallelization is internally managed by PLINQ, there is no guarantee that PLINQ will utilize more than a single core for work parallelization. PLINQ may run a query sequentially if it determines that the overhead of parallelization will slow down the query. In order to benefit from PLINQ, the total work in the query has to be sufficiently large to benefit from the overhead of scheduling the work on the thread pool.

Second, we are not fetching a single message at a time. Instead, we ask the Queue Service API to retrieve a specific number of messages from a queue. This is driven by the *DequeueBatchSize* parameter that is passed to the *Get<T>* method. When we enter the storage abstraction layer implemented as part of the overall solution, this parameter is handed over to the Queue Service API method. In addition, we run a safety check to ensure that the batch size doesn't exceed the maximum size supported by the APIs. This is implemented as follows:

```
/// This class provides reliable generics-aware access to the Windows Azure Queue
storage.
public sealed class ReliableCloudQueueStorage : ICloudQueueStorage
{
    /// The maximum batch size supported by Queue Service API in a single Get operation.
    private const int MaxDequeueMessageCount = 32;

    /// Gets a collection of messages from the specified queue and applies the specified
visibility timeout.
    public IEnumerable<T> Get<T>(string queueName, int count, TimeSpan visibilityTimeout)
    {
        Guard.ArgumentNotNullOrEmptyString(queueName, "queueName");
        Guard.ArgumentNotZeroOrNegativeValue(count, "count");
```

```
        try
        {
            var queue =
this.queueStorage.GetQueueReference(CloudUtility.GetSafeContainerName(queueName));

            IEnumerable<CloudQueueMessage> queueMessages =
this.retryPolicy.ExecuteAction<IEnumerable<CloudQueueMessage>>(() =>
            {
                return queue.GetMessages(Math.Min(count, MaxDequeueMessageCount),
visibilityTimeout);
            });

            // ... There is more code after this point ...
```

And finally, we are not going to run the dequeue task indefinitely. We provisioned an explicit checkpoint implemented as a *QueueEmpty* event which is raised whenever a queue becomes empty. At that point, we consult to a *QueueEmpty* event handler to determine whether or not it permits us to finish the running dequeue task. A well-designed implementation of the *QueueEmpty* event handler allows supporting the “auto scale-down” capability as explained in the following section.

Auto Scaling Down Dequeue Tasks

The purpose of *QueueEmpty* event handler is a two-fold. First, it is responsible for providing feedback to the source dequeue task instructing it to enter a sleep state for a given time interval (as defined in the *delay* output parameter in the event delegate). Secondly, it indicates to the dequeue task whether or not it must gracefully shut itself down (as prescribed by the Boolean return parameter).

The following implementation of the *QueueEmpty* event handler solves the two challenges highlighted earlier in this whitepaper. It calculates a random exponential back-off interval and tells the dequeue task to exponentially increase the delay between queue polling requests. Note that the back-off delay will not exceed 1 second as configured in our solution, since it really isn’t necessary to have a long delay between polling when auto-scaling is implemented well enough. In addition, it queries the queue listener state to determine the number of active dequeue tasks. Should this number be in excess of 1, the event handler advise the originating dequeue task to complete its polling loop provided the back-off interval has also reached its specified maximum. Otherwise, the dequeue task will not be terminated, leaving exactly 1 polling thread running at a time per a single instance of the queue listener. This approach **helps reduce the number of storage transactions and therefore decrease the transaction costs** as explained earlier.

```
private bool HandleQueueEmptyEvent(object sender, int idleCount, out TimeSpan delay)
{
    // The sender is an instance of the ICloudQueueServiceWorkerRoleExtension, we can
safely perform type casting.
```

```
    ICloudQueueServiceWorkerRoleExtension queueService = sender as
ICloudQueueServiceWorkerRoleExtension;

    // Find out which extension is responsible for retrieving the worker role
configuration settings.
    IWorkItemProcessorConfigurationExtension config =
Extensions.Find<IWorkItemProcessorConfigurationExtension>();

    // Get the current state of the queue listener to determine point-in-time load
characteristics.
    CloudQueueListenerInfo queueServiceState = queueService.QueryState();

    // Set up the initial parameters, read configuration settings.
    int deltaBackoffMs = 100;
    int minimumIdleIntervalMs =
Convert.ToInt32(config.Settings.MinimumIdleInterval.TotalMilliseconds);
    int maximumIdleIntervalMs =
Convert.ToInt32(config.Settings.MaximumIdleInterval.TotalMilliseconds);

    // Calculate a new sleep interval value that will follow a random exponential back-
off curve.
    int delta = (int)((Math.Pow(2.0, (double)idleCount) - 1.0) * (new
Random()).Next((int)(deltaBackoffMs * 0.8), (int)(deltaBackoffMs * 1.2)));
    int interval = Math.Min(minimumIdleIntervalMs + delta, maximumIdleIntervalMs);

    // Pass the calculated interval to the dequeue task to enable it to enter into a
sleep state for the specified duration.
    delay = TimeSpan.FromMilliseconds((double)interval);

    // As soon as interval reaches its maximum, tell the source dequeue task that it must
gracefully terminate itself
    // unless this is a last deqeueue task. If so, we are not going to keep it running
and continue polling the queue.
    return delay.TotalMilliseconds >= maximumIdleIntervalMs &&
queueServiceState.ActiveDequeueTasks > 1;
}
```

At a higher level, the above described "dequeue task scale-down" capability can be explained as follows:

1. Whenever there is anything in the queue, the dequeue tasks will ensure that the workload will be processed as soon as possible. There will be no delay between requests to dequeue messages from a queue.

2. As soon as the source queue becomes empty, each dequeue task will raise a *QueueEmpty* event.
3. The *QueueEmpty* event handler will calculate a random exponential back-off delay and instruct the dequeue task to suspend its activity for a given interval.
4. The dequeue tasks will continue polling the source queue at computed intervals until the idle duration exceeds its allowed maximum.
5. Upon reaching the maximum idle interval, and provided that the source queue is still empty, all active dequeue tasks will start gracefully shutting themselves down – this will not occur all at once, since the dequeue tasks are backing off at different points in the back off algorithm.
6. At some point in time, there will be only one active dequeue task waiting for work. As the result, no idle polling transactions will occur against a queue except only by that single task.

To elaborate on the process of collecting point-in-time load characteristics, it is worth mentioning the relevant source code artifacts. First, there is a structure holding the relevant metrics that measure the result of the load that is being applied to the solution. For the purposes of simplicity, we included a small subset of metrics that will be used further in the sample code.

```
/// Implements a structure containing point-in-time load characteristics for a given
queue listener.
public struct CloudQueueListenerInfo
{
    /// Returns the approximate number of items in the Windows Azure queue.
    public int CurrentQueueDepth { get; internal set; }

    /// Returns the number of dequeue tasks that are actively performing work or waiting
for work.
    public int ActiveDequeueTasks { get; internal set; }

    /// Returns the maximum number of dequeue tasks that were active at a time.
    public int TotalDequeueTasks { get; internal set; }
}
```

Secondly, there is a method implemented by a queue listener which returns its load metrics as depicted in the following example:

```
/// Returns the current state of the queue listener to determine point-in-time load
characteristics.
public CloudQueueListenerInfo QueryState()
{
    return new CloudQueueListenerInfo()
    {
        CurrentQueueDepth = this.queueStorage.GetCount(this.queueLocation.QueueName),
        ActiveDequeueTasks = (from task in this.dequeueTasks where task.Status !=
TaskStatus.Canceled && task.Status != TaskStatus.Faulted && task.Status !=
TaskStatus.RanToCompletion select task).Count(),
        TotalDequeueTasks = this.dequeueTasks.Count
```

```
    };
}
```

Auto Scaling Up Dequeue Tasks

In the previous section, we introduced the ability to reduce the number of active dequeue tasks to a single instance in order to minimize the impact of idle transactions on the storage operation costs. In this section, we are going to walk through a contrast example whereby we implement the “auto scale-up” capability to bring the processing power back when it’s needed.

First, we define an event delegate that will help track state transitions from an empty to a non-empty queue for the purposes of triggering relevant actions:

```
/// <summary>
/// Defines a callback delegate which will be invoked whenever new work arrived to a
queue while the queue listener was idle.
/// </summary>
/// <param name="sender">The source of the event.</param>
public delegate void WorkDetectedDelegate(object sender);
```

We then extend the original definition of the *ICloudQueueServiceWorkerRoleExtension* interface to include a new event that will be raised every time a queue listener detects new work items, essentially when queue depth changes from zero to any positive value:

```
public interface ICloudQueueServiceWorkerRoleExtension
{
    // ... The other interface members were omitted for brevity. See the previous code
snippets for reference ...

    // Defines a callback delegate to be invoked whenever a new work has arrived to a
queue while the queue listener was idle.
    event WorkDetectedDelegate QueueWorkDetected;
}
```

Also, we determine the right place in the queue listener’s code where such an event will be raised. We are going to fire the *QueueWorkDetected* event from within the dequeue loop implemented in the *DequeueTaskMain* method which needs to be extended as follows:

```
public class CloudQueueListenerExtension<T> : ICloudQueueListenerExtension<T>
{
    // An instance of the delegate to be invoked whenever a new work has arrived to a
queue while the queue listener was idle.
    public event WorkDetectedDelegate QueueWorkDetected;

    private void DequeueTaskMain(object state)
    {
        CloudQueueListenerDequeueTaskState<T> workerState =
(CloudQueueListenerDequeueTaskState<T>)state;
```

```
        int idleStateCount = 0;
        TimeSpan sleepInterval = DequeueInterval;

        try
        {
            // Run a dequeue task until asked to terminate or until a break condition is
encountered.
            while (workerState.CanRun)
            {
                try
                {
                    var queueMessages = from msg in
workerState.QueueStorage.Get<T>(workerState.QueueLocation.QueueName, DequeueBatchSize,
workerState.QueueLocation.VisibilityTimeout).AsParallel() where msg != null select msg;
                    int messageCount = 0;

                    // Check whether or not work items arrived to a queue while the
listener was idle.
                    if (idleStateCount > 0 && queueMessages.Count() > 0)
                    {
                        if (QueueWorkDetected != null)
                        {
                            QueueWorkDetected(this);
                        }
                    }

                    // ... The rest of the code was omitted for brevity. See the previous
code snippets for reference ...
```

In the last step, we provide a handler for the *QueueWorkDetected* event. The implementation of this event handler will be supplied by a component which instantiates and hosts the queue listener. In our case, it's a worker role. The code responsible for instantiation and implementation of event handler is comprised of the following:

```
public class WorkItemProcessorWorkerRole : RoleEntryPoint
{
    // Called by Windows Azure to initialize the role instance.
    public override sealed bool OnStart()
    {
        // ... There is some code before this point ...

        // Instantiate a queue listener for the input queue.
```

```
        var inputQueueListener = new
CloudQueueListenerExtension<XDocument>(inputQueueLocation);

        // Configure the input queue listener.
        inputQueueListener.QueueEmpty += HandleQueueEmptyEvent;
        inputQueueListener.QueueWorkDetected += HandleQueueWorkDetectedEvent;
        inputQueueListener.DequeueBatchSize =
configSettingsExtension.Settings.DequeueBatchSize;
        inputQueueListener.DequeueInterval =
configSettingsExtension.Settings.MinimumIdleInterval;

        // ... There is more code after this point ...
    }

    // Implements a callback delegate to be invoked whenever a new work has arrived to a
queue while the queue listener was idle.
    private void HandleQueueWorkDetectedEvent(object sender)
    {
        // The sender is an instance of the ICloudQueueServiceWorkerRoleExtension, we can
safely perform type casting.
        ICloudQueueServiceWorkerRoleExtension queueService = sender as
ICloudQueueServiceWorkerRoleExtension;

        // Get the current state of the queue listener to determine point-in-time load
characteristics.
        CloudQueueListenerInfo queueServiceState = queueService.QueryState();

        // Determine the number of queue tasks that would be required to handle the
workload in a queue given its current depth.
        int dequeueTaskCount =
GetOptimalDequeueTaskCount(queueServiceState.CurrentQueueDepth);

        // If the dequeue task count is less than computed above, start as many dequeue
tasks as needed.
        if (queueServiceState.ActiveDequeueTasks < dequeueTaskCount)
        {
            // Start the required number of dequeue tasks.
            queueService.StartListener(dequeueTaskCount -
queueServiceState.ActiveDequeueTasks);
        }
    }       // ... There is more code after this point ...
```

In light of the above example, the *GetOptimalDequeueTaskCount* method is worth taking a deeper look at. This method is responsible for computing the number of dequeue tasks that would be considered optimal for handling the workload in a queue. When invoked, this method should determine (through any appropriate decision-making mechanisms) how much "horsepower" the queue listener needs in order to process the volume of work either awaiting on or expected to arrive to a given queue.

For instance, the developer could take a simplistic approach and embed a set of static rules directly into the GetOptimalDequeueTaskCount method. Using the known throughput and scalability characteristics of the queuing infrastructure, average processing latency, payload size and other relevant inputs, the rule set could take an optimistic view and decide on an optimal dequeue task count.

In the example below, an intentionally over-simplified technique is being used for determining the number of dequeue tasks:

```
/// <summary>
/// Returns the number of queue tasks that would be required to handle the workload in a
queue given its current depth.
/// </summary>
/// <param name="currentDepth">The approximate number of items in the queue.</param>
/// <returns>The optimal number of dequeue tasks.</returns>
private int GetOptimalDequeueTaskCount(int currentDepth)
{
    if (currentDepth < 100) return 10;
    if (currentDepth >= 100 && currentDepth < 1000) return 50;
    if (currentDepth >= 1000) return 100;

    // Return the minimum acceptable count.
    return 1;
}
```

It is worth reiterating that the example code above is not intended to be a “one size fits all” approach. A more ideal solution would be to invoke an externally configurable and manageable rule which performs the necessary computations.

At this point, we have a working prototype of a queue listener capable of automatically scaling itself up and down as per fluctuating workload. Perhaps, as a final touch, it needs to be enriched with the ability to adapt itself to variable load while it’s being processed. This capability can be added by applying the same pattern as it was being followed when adding support for the *QueueWorkDetected* event.

Now, let’s switch focus to another important optimization that will help reduce latency in the queue listeners.

Implementing Publish/Subscribe Layer for Zero-Latency Dequeue

In this section, we are going to enhance the above implementation of a queue listener with a push-based notification mechanism built on top of the Service Bus one-way multicast capability.

The notification mechanism is responsible for triggering an event telling the queue listener to start performing dequeue work. This approach **helps avoid polling the queue to check for new messages and therefore eliminate the associated latency**.

First, we define a trigger event that will be received by our queue listener in case a new workload is deposited into a queue:

```
/// Implements a trigger event indicating that a new workload was put in a queue.
[DataContract(Namespace = WellKnownNamespace.DataContracts.Infrastructure)]
public class CloudQueueWorkDetectedTriggerEvent
{
    /// Returns the name of the storage account on which the queue is located.
    [DataMember]
    public string StorageAccount { get; private set; }

    /// Returns a name of the queue where the payload was put.
    [DataMember]
    public string QueueName { get; private set; }

    /// Returns a size of the queue's payload (e.g. the size of a message or the number
of messages in a batch).
    [DataMember]
    public long PayloadSize { get; private set; }

    // ... The constructor was omitted for brevity ...
}
```

Next, we enable the queue listener implementations to act as subscribers to receive a trigger event. The first step is to define a queue listener as an observer for the *CloudQueueWorkDetectedTriggerEvent* event:

```
/// Defines a contract that must be implemented by an extension responsible for listening
on a Windows Azure queue.
public interface ICloudQueueServiceWorkerRoleExtension :
IObserver<CloudQueueWorkDetectedTriggerEvent>
{
    // ... The body is omitted as it was supplied in previous examples ...
}
```

The second step is to implement the OnNext method defined in the IObserver<T> interface. This method gets called by the provider to notify the observer about a new event:

```
public class CloudQueueListenerExtension<T> : ICloudQueueListenerExtension<T>
{
    // ... There is some code before this point ...

    /// <summary>
```

```
    /// Gets called by the provider to notify this queue listener about a new trigger
event.
    /// </summary>
    /// <param name="e">The trigger event indicating that a new payload was put in a
queue.</param>
    public void OnNext(CloudQueueWorkDetectedTriggerEvent e)
    {
        Guard.ArgumentNotNull(e, "e");

        // Make sure the trigger event is for the queue managed by this listener,
otherwise ignore.
        if (this.queueLocation.StorageAccount == e.StorageAccount &&
this.queueLocation.QueueName == e.QueueName)
        {
            if (QueueWorkDetected != null)
            {
                 QueueWorkDetected(this);
            }
        }
    }

    // ... There is more code after this point ...
}
```

At it can be seen in the above example, we purposefully invoke the same event delegate as it is used in the previous steps. The *QueueWorkDetected* event handler already provides the necessary application logic for instantiating optimal number of dequeue tasks. Therefore, the same event handler is reused when handling the *CloudQueueWorkDetectedTriggerEvent* notification.

As noted in the previous sections, we don't have to maintain a continuously running dequeue task when a push-based notification is employed. Therefore, we can reduce the number of queue tasks per a queue listener instance to zero and use a notification mechanism to instantiate dequeue tasks when the queue receives work items. In order to make sure that we are not running any idle dequeue tasks, the following straightforward modification in the *QueueEmpty* event hander is required:

```
private bool HandleQueueEmptyEvent(object sender, int idleCount, out TimeSpan delay)
{
    // ... There is some code before this point ...

    // As soon as interval reaches its maximum, tell the source dequeue task that it must
gracefully terminate itself.
    return delay.TotalMilliseconds >= maximumIdleIntervalMs;
}
```

In summary, we are no longer detecting whether or not there is a single active dequeue task remaining. The result of the revised *QueueEmpty* event hander only takes into account the fact of exceeding the maximum idle interval upon which all active dequeue tasks will be shut down.

To receive the *CloudQueueWorkDetectedTriggerEvent* notifications, we leverage the publish/subscribe model that is implemented as loosely coupled messaging between Windows Azure role instances. In essence, we hook on the same inter-role communication layer and handle the incoming events as follows:

```
public class InterRoleEventSubscriberExtension : IInterRoleEventSubscriberExtension
{
    // ... Some code here was omitted for brevity. See the corresponding guidance on
Windows Azure CAT team blog for reference ...

    public void OnNext(InterRoleCommunicationEvent e)
    {
        if (this.owner != null && e.Payload != null)
        {
            // ... There is some code before this point ...

            if (e.Payload is CloudQueueWorkDetectedTriggerEvent)
            {
                HandleQueueWorkDetectedTriggerEvent(e.Payload as
CloudQueueWorkDetectedTriggerEvent);
                return;
            }

            // ... There is more code after this point ...
        }
    }

    private void HandleQueueWorkDetectedTriggerEvent(CloudQueueWorkDetectedTriggerEvent
e)
    {
        Guard.ArgumentNotNull(e, "e");

        // Enumerate through registered queue listeners and relay the trigger event to
them.
        foreach (var queueService in
this.owner.Extensions.FindAll<ICloudQueueServiceWorkerRoleExtension>())
        {
            // Pass the trigger event to a given queue listener.
            queueService.OnNext(e);
        }
```

```
    }
}
```

Multicasting a trigger event defined in the *CloudQueueWorkDetectedTriggerEvent* class is the ultimate responsibility of a publisher, namely, the component depositing work items on a queue. This event can be triggered either before the very first work item is enqueued or after last item is put in a queue. In the example below, we publish a trigger event upon completing putting work items into the input queue:

```
public class ProcessInitiatorWorkerRole : RoleEntryPoint
{
    // The instance of the role extension which provides an interface to the inter-role
communication service.
    private volatile IInterRoleCommunicationExtension interRoleCommunicator;

    // ... Some code here was omitted for brevity. See the corresponding guidance on
Windows Azure CAT team blog for reference ...

    private void HandleWorkload()
    {
        // Step 1: Receive compute-intensive workload.
        // ... (code was omitted for brevity) ...

        // Step 2: Enqueue work items into the input queue.
        // ... (code was omitted for brevity) ...

        // Step 3: Notify the respective queue listeners that they should expect work to
arrive.
        // Create a trigger event referencing the queue into which we have just put work
items.
        var trigger = new CloudQueueWorkDetectedTriggerEvent("MyStorageAccount",
"InputQueue");

        // Package the trigger into an inter-role communication event.
        var interRoleEvent = new
InterRoleCommunicationEvent(CloudEnvironment.CurrentRoleInstanceId, trigger);

        // Publish inter-role communication event via the Service Bus one-way multicast.
        interRoleCommunicator.Publish(interRoleEvent);
    }
}
```

Now that we have built a queue listener that is capable of supporting multi-threading, auto-scaling and push-based notifications, it's time to consolidate all recommendations pertaining to the design of queue-based messaging solutions on the Windows Azure platform.

Conclusion

To maximize the efficiency and cost effectiveness of queue-based messaging solutions running on the Windows Azure platform, solution architects and developers should consider the following recommendations.

As a solution architect, you should:

- Provision a queue-based messaging architecture that uses the **Windows Azure queue storage service for high-scale asynchronous communication** between tiers and services in cloud-based or hybrid solutions.
- Recommend **partitioned queuing architecture to scale beyond 500** transactions/sec.
- Understand the fundamentals of Windows Azure pricing model and **optimize solution to lower transaction costs** through a series of best practices and design patterns.
- Consider dynamic scaling requirements by provisioning an architecture that is **adaptive to volatile and fluctuating workloads**.
- Employ the right auto-scaling techniques and approaches to **elastically expand and shrink compute power** to further optimize the operating expense.
- Evaluate the cost-benefit ratio of reducing latency through taking dependency on Windows Azure **Service Bus for real-time push-based notification dispatch**.

As a developer, you should:

- Design a messaging solution that **employs batching when storing and retrieving data** from Windows Azure queues.
- Implement an efficient queue listener service ensuring that queues will be polled by a **maximum of one dequeue thread when empty**.
- **Dynamically scale down the number of worker role instances** when queues remain empty for a prolonged period of time.
- **Implement an application-specific random exponential back-off** algorithm to reduce the effect of idle queue polling on storage transaction costs.
- Adopt the right techniques that **prevent from exceeding the scalability targets for a single queue** when implementing highly multi-threaded multi-instance queue publishers and consumers.
- Employ a **robust retry policy** capable of handling a variety of transient conditions when publishing and consuming data from Windows Azure queues.
- **Use the one-way eventing capability** provided by Windows Azure Service Bus to support push-based notifications in order to reduce latency and improve performance of the queue-based messaging solution.
- Explore the new capabilities of the .NET Framework 4 such as **TPL, PLINQ and Observer pattern to maximize the degree of parallelism**, improve concurrency and simplify the design of multi-threaded services.

The accompanying sample code is available for download from the MSDN Code Gallery. The sample code also includes all the required infrastructure components such as generics-aware abstraction layer for the Windows Azure queue service which were not supplied in the above code snippets. Note that all source code files are governed by the Microsoft Public License as explained in the corresponding legal notices.

Additional Resources/References

For more information on the topic discussed in this whitepaper, please refer to the following:

- Understanding Windows Azure Storage Billing – Bandwidth, Transactions, and Capacity post on the Windows Azure Storage team blog.
- Service Management API article in the MSDN Library.
- About the Service Management API in Windows Azurepost on Neil Mackenzie's blog.
- Windows Azure Service Management CmdLets project on the MSDN CodePlex.
- Windows Azure Storage Abstractions and their Scalability Targets post on the Windows Azure Storage team blog.
- Queue Read/Write Throughput study published by eXtreme Computing Group at Microsoft Research.
- The Transient Fault Handling Framework for Azure Storage, Service Bus & SQL Azure project on the MSDN Code Gallery.
- The Autoscaling Application Block in the MSDN library.
- Windows Azure Storage Transaction - Unveiling the Unforeseen Cost and Tips to Cost Effective Usage post on Wely Lau's blog.

How to Integrate a BizTalk Server Application with Service Bus Queues and Topics

Author: Paolo Salvatori

Reviewers: Ralph Squillace, Thiago Almeida

This article shows how to integrate a BizTalk Server 2010 application with Windows Azure Service Bus queues, topics, and subscriptions to exchange messages with external systems in a reliable, flexible, and scalable manner. Queues and topics, introduced in the September 2011 Windows Azure AppFabric SDK, are the foundation of a new cloud-based messaging and integration infrastructure that provides reliable message queuing and durable publish/subscribe messaging capabilities to both cloud and on-premises applications based on Microsoft and non-Microsoft technologies. .NET applications can use the new messaging functionality from either a brand-new managed API (Microsoft.ServiceBus.Messaging) or via WCF thanks to a new binding (NetMessagingBinding), and any Microsoft or non-Microsoft applications can use a REST style API to access these features.

Microsoft BizTalk Server enables organizations to connect and extend heterogeneous systems across the enterprise and with trading partners. The Service Bus is part of Windows Azure AppFabric and is designed to provide connectivity, queuing, and routing capabilities not only for cloud applications but also for on-premises applications. Using both together enables a significant number of scenarios in which you can build secure, reliable and scalable hybrid solutions that span the cloud and on premises environments:

1. Exchange electronic documents with trading partners.
2. Expose services running on-premises behind firewalls to third parties.
3. Enable communication between spoke branches and a hub back office system.

In this article you will learn how to use WCF in a .NET and BizTalk Server application to execute the following operations:

1. Send messages to a Service Bus topic.
2. Receive messages from a Service Bus queue.
3. Receive messages from a Service Bus subscription.

In this article you will also learn how to translate the explicit and user-defined properties of a BrokeredMessage object into the context properties of a BizTalk message and vice versa. Before describing how to perform these actions, I will start with a brief introduction of the elements that compose the solution:

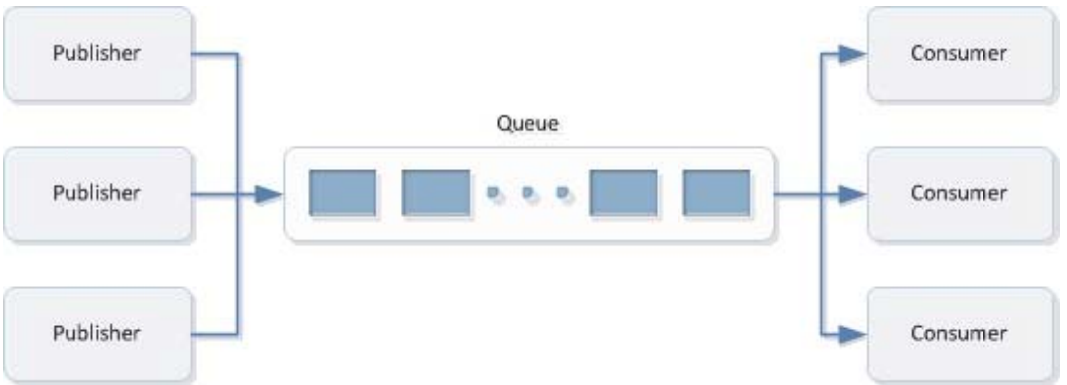
In This Section

See Also

"Now Available: The Service Bus September 2011 Release" article on the Windows Azure Blog
"Queues, Topics, and Subscriptions" article on the MSDN site.
"AppFabric Service Bus" topic on the MSDN site.
"Understanding Windows Azure AppFabric Queues (and Topics)" video on the AppFabric Team Blog.
"Building loosely-coupled apps with Windows Azure Service Bus Topics and Queues" video on the channel9 site.
"Service Bus Topics And Queues" video on the channel9 site.
"Securing Service Bus with ACS" video on the channel9 site.

Service Bus Queues

Queues provide messaging capabilities that enable a large and heterogeneous class of applications running on premises or in the cloud to exchange messages in a flexible, secure and reliable fashion across network and trust boundaries.



Queues are hosted in Windows Azure by a replicated, durable store infrastructure. The maximum size of a queue is 5GB. The maximum message size is instead 256KB, but the use of sessions allows you to create unlimited-size sequences of related messages. Queues are accessed through the following APIs:

1. The new messaging API available through the new Microsoft.ServiceBus.Messaging assembly.
2. Windows Communication Foundation (WCF) through the new NetMessagingBinding class.
3. Service Bus REST API.

Queue entities provide the following capabilities:

1. Session-based correlation, allowing you to build multiplexed request/reply paths easily.
2. Reliable delivery patterns using the PeekLock receive mode.
3. Transactions support to ensure that batches of messaging operations (both send and receive) are committed atomically.
4. Detection of inbound message duplicates, allowing clients to send the same message multiple times without adverse consequences.
5. Dead-letter facility for messages that cannot be processed or expire before being received.
6. Deferring messages for later processing. This functionality is particularly handy in the following cases:
   a. When messages are received out of the expected sequence and need to be saved while a process waits for a particular message to permit further progress.
   b. When messages must be processed based on a set of properties that define their priorities during a traffic peak.

In the .NET API the message entity is modeled by the BrokeredMessage class, which exposes properties (such as MessageId, SessionID, and CorrelationId) that you can use to exploit capabilities such as automatic duplicate detection or session-enabled communications. You can use a QueueDescription object to specify the metadata that models the behavior of the queue being created:

1. The DefaultMessageTimeToLive property specifies the default message time-to-live value.
2. The DuplicateDetectionHistoryTimeWindow property defines the duration of the duplicate detection history.
3. The EnableDeadLetteringOnMessageExpiration property allows you to enable or disable the dead-letter facility on message expiration.
4. The LockDuration property defines the duration of the lock used by a consumer when using the PeekLock receive mode. In the ReceiveAndDelete receive mode, a message is deleted from the queue as soon as a consumer reads it. Conversely, in the PeekLock receive mode, a message is hidden from other receivers until the timeout defined by the LockDuration property expires. By that time the receiver should have deleted the message invoking the Complete method on the BrokeredMessage object.
5. The MaxSizeInMegabytes property defines the maximum queue size in megabytes.
6. The RequiresDuplicateDetection property enables\disables duplicate message detection. The RequiresSession property enables\disables sessions.
7. The MessageCount return the number of messages in the queue. This is property is extremely useful because, based on its value, an intelligent system can decide to increase or decrease the number of competing consumers that concurrently receive and processes messages from the queue.

Warning

Since metadata cannot be changed once a messaging entity is created, modifying the duplicate detection behavior requires deleting and recreating the queue. The same principle applies to any other metadata.

Using queues permits you to flatten highly-variable traffic into a predictable stream of work and distribute the load across a set of worker processes size of which can vary dynamically to

accommodate the incoming message volume. In a competing consumers scenario, when a publisher writes a message to a queue, multiple consumers compete with each other to receive the message, but only one of them will receive and process the message in question.

In service-oriented or service bus architecture composed of multiple, heterogeneous systems, interactions between autonomous systems are asynchronous and loosely-coupled. In this context, the use of messaging entities like queues and topics (see the next section) allows increasing the agility, scalability and flexibility of the overall architecture and helps decreasing the loose coupling of individual systems.

See Also

"Understanding Windows Azure AppFabric Queues (and Topics)" video on the AppFabric Team Blog.

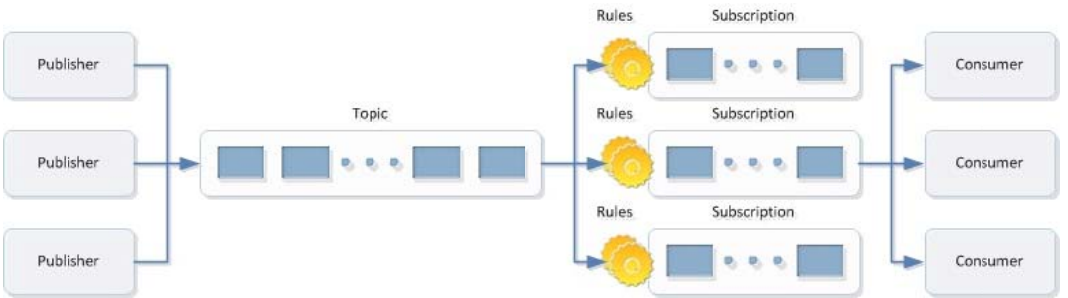
"An Introduction to Service Bus Queues" article on the AppFabric Team Blog.

"Windows Azure AppFabric Service Bus Queues and Topics" on Neil Mackenzie's Blog.

"Windows Azure AppFabric Learning Series" available on CodePlex.

Service Bus Topics

Topics extend the messaging features provided by queues with the addition of publish-subscribe capabilities.



A topic entity consists of a sequential message store like a queue, but it supports up to 2000 (this number is subject to vary in the future) concurrent and durable subscriptions, which relay message copies to a poll of worker processes. As depicted in the following figure, each subscription can define one or multiple rule entities.

Each rule specifies a filter expression that is used to filter messages that pass through the subscription and a filter action that can modify message properties. In particular, the SqlFilter class allows a SQL92-like condition to be defined on message properties, as in the following examples:

1. `OrderTotal > 5000 OR ClientPriority > 2`
2. `ShipDestinationCountry = 'USA' AND ShipDestinationState = 'WA'`

Conversely, the SqlRuleAction class can be used to modify, add or remove properties to a BrokeredMessage object using syntax similar to that used by the **SET** clause of an **UPDATE** SQL-command, as in the following examples.

1. `SET AuditRequired = 1`
2. `SET Priority = 'High', Severity = 1`

Warning

Each matching rule generates a separate copy of the published message, so any subscription can potentially generate more copies of the same message, one for each matching rule.

Topics, like queues, support a competing consumers scenario: in this context, a subscription can have a single consumer that receives all messages or a set of competing consumers that fetch messages on a first-come-first-served basis. Topics are a useful messaging solution for broadcasting messages to many consumers.

See Also

Understanding Windows Azure AppFabric Queues (and Topics) video on the AppFabric Team Blog.

An Introduction to Service Bus Topics article on the AppFabric Team Blog.

Windows Azure AppFabric Service Bus Queues and Topics on Neil Mackenzie's Blog.

Windows Azure AppFabric Learning Series available on CodePlex

BrokeredMessage

The BrokeredMessage class models the messages exchanged by applications that communicate through queues and topics. The class provides four different public constructors:

1. The BrokeredMessage() constructor initializes a new instance of the BrokeredMessage class with an empty payload. A BrokeredMessage object exposes a collection of key-value pairs in the Properties, which allows you to define a set of custom properties. It is common for a message to have an empty body because user-defined properties can carry the message payload.
2. The BrokeredMessage(Object) constructor initializes a new instance of the BrokeredMessage class from a given object by using a DataContractSerializer object with a binary XmlDictionaryWriter object.
3. BrokeredMessage(Stream, Boolean) constructor initializes a new instance of the BrokeredMessage class using the supplied stream as its body.
4. Finally, the BrokeredMessage(Object, XmlObjectSerializer) constructor initializes an instance of the BrokeredMessage class from a given object using the provided XmlObjectSerializer object.

The class exposes an interesting set of methods that allow execution of a wide range of actions at the message level:

1. In the PeekLock receive mode, the Abandon method can release the lock on a peek-locked message. The Complete method commits the receive operation of a message and indicates that the message should be marked as processed and then either deleted or archived. The Defer method indicates that the receiver wants to defer processing for a message. As mentioned before, deferring messages is a convenient way to handle situations where messages are received out of the expected sequence and need to be safely parked while the applications waits for a particular message before proceeding with message processing.
2. The DeadLetter and DeadLetter(String, String) methods allows an application to explicitly move a message to the dead-letter queue of a queue or a subscription. Take into account that when you create a queue entity using the managing API or the Windows Azure Management Portal, you can configure it to automatically move expired messages to the

dead-letter queue. In the same way, you can configure a subscription to move expired messages and messages to its dead-letter queue if they fail filter evaluation.

The BrokeredMessage class exposes a wide range of properties:

Property	Description
ContentType	Specifies the type of the content.
MessageId	Indicates the identifier of the message.
CorrelationId	Implement a request-reply message exchange pattern in which the client application uses the MessageId property of an outgoing request message and the CorrelationId property of an incoming response message to correlate the two messages. We will see an implementation of this technique later in this article.
SessionId	Sets or gets the session identifier for a message. A competing consumer scenario where multiple worker processes receive messages from the same session-enabled queue or subscription guarantees that messages sharing the same SessionId are received by the same consumer. In this context, when a client application sends a flow of request messages to a server application using a session-enabled queue or topic and waits for the correlated response messages on a separate session-enabled queue or a subscription, the client application can assign the receive session's ID to the ReplyToSessionId property of outgoing messages. This property value gives the server application the value to assign to the SessionId property of response messages.
ReplyTo	Gets or sets the address of the queue to reply to. In an asynchronous request-reply scenario a client application can send a request message to a server application through a Service Bus queue or topic and wait for a response message. In this scenario, by convention the client application can use the ReplyTo property of the request message to indicate to the server application the address of the queue or topic to send the response to. We will see an

	application of this technique when I introduce the second test case where an orchestration reads the ReplyTo address from a context property of the request message and assign its value to the Address property of a dynamic send port to return the response message to the expected queue or topic.
Label	Gets or sets an application specific label.
SequenceNumber	Returns the unique number that the Service Bus assigns to a message. Use this property to retrieve a deferred message from a queue or a subscription.
TimeToLive	Allows you to set or get the message's time-to-live value. The Service Bus does not enforce a maximum lifetime for messages waiting to be processed in a queue or a subscription. Nevertheless, you can define a default time to live for messages when you create a queue, topic, or subscription, or you can explicitly define an expiration timeout at the message level using the TimeToLive property.
DeliveryCount	Returns the number of message deliveries.
Properties	A collection that allows you to define application specific message properties. This collection is probably the most important feature of a BrokeredMessage entity because user-defined properties can be used for the following: 1. Carry the payload of a message. In this context, the body of the message could be empty. 2. Define application specific properties that can be used by a worker process to decide how to process the current message. 3. Specify filter and action expressions that can be used to define routing and data enrichment rules at a subscription level

If you are familiar with context properties for a BizTalk message, it is helpful to think of the user-defined properties contained in the BrokeredMessageProperties collection as the context properties of a BizTalk message. In fact, these properties can carry a piece of information or even the entire message payload. If you are using topics and subscriptions, these properties can route

the message to the proper destination. Hence, in a scenario where a third party system exchanges messages with a BizTalk application through the Service Bus, it's very important to translate the application specific properties carried by a BrokeredMessage object to and from the context properties of a BizTalk message. In the article and in the companion code I'll show you how to achieve this result.

Warning

As indicated in Windows Azure AppFabric Service Bus Quotas, the maximum size for each property is 32K. Cumulative size of all properties cannot exceed 64K. This applies to the entire header of the BrokeredMessage, which has both user properties as well as system properties (such as SequenceNumber, Label, MessageId, and so on). The space occupied by properties counts towards the overall size of the message and its maximum size of 256K. If an application exceeds any of the limits mentioned above, a SerializationException exception is generated, so you should expect to handle this error condition.

NetMessagingBinding

Service Bus Brokered Messaging supports the WCF programming model and provides a new binding called NetMessagingBinding that can be used by WCF-enabled applications to send and receive messages through queues, topics and subscriptions. NetMessagingBinding is the new name for the binding for queues and topics that provides full integration with WCF. From a functional perspective, NetMessagingBinding is similar to NetMsmqBinding, which supports queuing by using Message Queuing (MSMQ) as a transport and enables support for loosely-coupled applications. On the service-side, NetMessagingBinding provides an automatic message-pump that pulls messages off a queue or subscription and is integrated with WCF's **ReceiveContext** mechanism.

The new binding supports the standard interfaces IInputChannel, IOutputChannel, and IInputSessionChannel. When an application uses WCF and NetMessagingBinding to send a message to a queue or a topic, the message is wrapped in a SOAP envelope and encoded. To set BrokeredMessage-specific properties, you need to create a BrokeredMessageProperty object, set the properties on it, and add it to the Properties collection of the WCF message, as shown in the following example. When using the NetMessagingBinding to write a message to queue or a topic, the **ServiceBusOutputChannel** internal class looks for the BrokeredMessageProperty property in the Properties collection of the WCF message and copies all its properties to the BrokeredMessage object it creates. Then it copies the payload from the WCF message to the BrokeredMessage object and finally, publishes the resulting message to the target queue or topic.

```
static void Main(string[] args)
{
```

```
    try
    {
        // Create the
        var channelFactory = new ChannelFactory<IOrderService>("orderEndpoint");
        var clientChannel = channelFactory.CreateChannel();

        // Create a order object
        var order = new Order()
                        {
                            ItemId = "001",
                            Quantity = 10
                        };

        // Use the OperationContextScope to create a block within which to access the
current OperationScope
        using (var scope = new OperationContextScope((IContextChannel)clientChannel))
        {
            // Create a new BrokeredMessageProperty object
            var property = new BrokeredMessageProperty();

            // Use the BrokeredMessageProperty object to set the BrokeredMessage
properties
            property.Label = "OrderItem";
            property.MessageId = Guid.NewGuid().ToString();
            property.ReplyTo = "sb://acme.servicebus.windows.net/invoicequeue";

            // Use the BrokeredMessageProperty object to define application-specific
properties
            property.Properties.Add("ShipCountry", "Italy");
            property.Properties.Add("ShipCity", "Milan");

            // Add BrokeredMessageProperty to the OutgoingMessageProperties bag provided
            // by the current Operation Context

OperationContext.Current.OutgoingMessageProperties.Add(BrokeredMessageProperty.Name,
property);
            clientChannel.SendOrder(order);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
```

```
    }
}
```

Likewise, when you use a NetMessagingBinding-based service endpoint to receive messages from a queue or a topic, an application can retrieve the BrokeredMessageProperty object from the Properties collection of the incoming WCF Message, as shown in the following example. In particular, when the service endpoint receives messages, the **ServiceBusInputChannel** and **ServiceBusInputSessionChannel** internal classes create a new WCF Message and copy the payload from the body on the inbound to the body of the newly created WCF message. (**ServiceBusInputSessionChannel** is used to receive messages from sessionful queues and subscriptions). Then they copy the properties from the inbound BrokeredMessage to a new instance of the BrokeredMessageProperty class and, finally, add the new BrokeredMessageProperty object to the Properties collection of the incoming WCF Message.

```
[ServiceBehavior]
public class OrderService : IOrderService
{
    [OperationBehavior]
    public void ReceiveOrder(Order order)
    {
        // Get the BrokeredMessageProperty from the current OperationContext
        var incomingProperties = OperationContext.Current.IncomingMessageProperties;
        var property = incomingProperties[BrokeredMessageProperty.Name] as
BrokeredMessageProperty;

        ...
    }
}
```

Because the Service Bus does not support IInputSessionChannel, all applications sending messages to session-enabled queues must use a service contract where the value of the SessionMode property is different from that of the SessionMode.Required. However, because the Service Bus WCF runtime supports IInputSessionChannel and other interfaces order to receive messages from a sessionful queue or subscription that uses WCF and NetMessagingBinding, applications must implement a session-aware service contract. The following code snippet provides an example of a WCF service that receives messages from a sessionful queue/subscription.

```
// ServiceBus does not support IOutputSessionChannel.
// All senders sending messages to sessionful queue must use a contract which does not
enforce SessionMode.Required.
// Sessionful messages are sent by setting the SessionId property of the
BrokeredMessageProperty object.
[ServiceContract]
public interface IOrderService
{
```

```
    [OperationContract(IsOneWay = true)]
    [ReceiveContextEnabled(ManualControl = true)]
    void ReceiveOrder(Order order);
}

// ServiceBus supports both IInputChannel and IInputSessionChannel.
// A sessionful service listening to a sessionful queue must have SessionMode.Required in
its contract.
[ServiceContract(SessionMode = SessionMode.Required)]
public interface IOrderServiceSessionful : IOrderService
{
}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession, ConcurrencyMode =
ConcurrencyMode.Single)]
public class OrderService : IOrderServiceSessionful
{
    [OperationBehavior]
    public void ReceiveOrder(Order order)
    {
        // Get the BrokeredMessageProperty from the current OperationContext
        var incomingProperties = OperationContext.Current.IncomingMessageProperties;
        var property = incomingProperties[BrokeredMessageProperty.Name] as
BrokeredMessageProperty;

        ...

        //Complete the Message
        ReceiveContext receiveContext;
        if (ReceiveContext.TryGet(incomingProperties, out receiveContext))
        {
            receiveContext.Complete(TimeSpan.FromSeconds(10.0d));
            ...
        }
        else
        {
            throw new InvalidOperationException("...");
        }
    }
}
```

Note that the ManualControl property of the ReceiveContextEnabled operation attribute is set to **true**. This setting makes the service explicitly invoke the ReceiveContext.Complete method to

commit the receive operation. In fact, when the ManualControl property is set to **true**, the message received from the channel is delivered to the service operation with a lock for the message. The service implementation must either call Complete(TimeSpan) or Abandon(TimeSpan) to signal that the message has been received completely. Failure to call one of these methods results in the lock being held on the message until the lock timeout interval elapses. Once the lock is released (either by a call to Abandon(TimeSpan) or by a lock timeout) the message is dispatched again from the channel to the service. Calling Complete(TimeSpan) marks the message as successfully received.

Note also that the `OrderService` class sets the ServiceBehavior.InstanceContextMode property to InstanceContextMode.PerSession and the ConcurrencyMode property to ConcurrencyMode.Single. This way the ServiceHost creates a new service instance every time a new session is available in the specified queue or subscription, and the ServiceHost uses a single thread to receive messages sequentially from the queue or subscription. The SessionIdleTimeout property of the NetMessagingBinding controls the lifetime of the service. For more information on how to use WCF exchange messages through a sessionful queue, I strongly suggest you to look at the **WcfServiceSessionSample** sample included in the Windows Azure AppFabric SDK V1.5.

BizTalk WCF Adapters

Each WCF receive location in BizTalk Server is an instance of a WCF service class called BizTalkServiceInstance, which is hosted by a separate instance of a **ServiceHost**-derived class:

1. The **BtsServiceHost** for receive locations (RLs) running in an in-process host.
2. The **WebServiceHost** for RLs running in an isolated host.

When you enable a WCF receive location, the adapter initializes and opens the **ServiceHost**, which dynamically builds the WCF runtime components within the BizTalk service process (BtsNtSvc.exe for in in-process host, w3wp.exe for an isolated host). These components include the channel stack, the dispatcher, and the singleton service instance. WCF receive locations and send ports are message-type agnostic or, if you prefer, untyped. This design comes in handy when you need to configure a single receive port to accept numerous message types or versions that you can normalize (using BizTalk maps or a custom pipeline component) into a common message type before the messages are posted to the **BizTalkMsgBoxDb**. However, this design also implies that the WCF adapters must build on generic service contracts in order to remain message-type agnostic within the WCF implementation.

WCF receive locations must receive the incoming message bytes, perform any necessary SOAP and WS-* processing, and publish the message (or some part of it) to the message box. The WCF Adapters create a separate **ServiceHost** and singleton service object of this class for each receive location that handles client requests for the lifetime of the BizTalk Host instance that runs

WCF receive locations. The service object uses multiple threads to process messages concurrently unless the WCF-NetMsmq receive locations are used with the **Ordered** processing property enabled.. As shown in the picture below, the class is decorated with the ServiceBehavior attribute:

```
//BizTalkServiceInstance Class

[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single, ConcurrencyMode =
ConcurrencyMode.Multiple)]
internal sealed class BizTalkServiceInstance : ITwoWayAsync,
                                               ITwoWayAsyncVoid,
                                               IOneWayAsync,
                                               IOneWayAsyncTxn,
                                               ITwoWayAsyncVoidTxn
{
    ...
}
```

In particular, you can note that:

1. `InstanceContextMode = InstanceContextMode.Single`
2. `ConcurrencyMode = ConcurrencyMode.Multiple`

Hence, all incoming messages to a WCF receive location are received and processed by a single well-known instance of the BizTalkServiceInstance class. This use of a single service instance avoids service activation and deactivation costs and improves performance and scalability.

The BizTalkServiceInstance class implements multiple untyped, generic service contracts.

- IOneWayAsync
- IOneWayAsyncTxn
- ITwoWayAsync
- ITwoWayAsyncVoid
- ITwoWayAsyncVoidTxn

Each contract was designed for a different scope (as suggested by their names):

- One-way vs. two-way Message Exchange Pattern.
- Transactional vs. nontransactional communication.

```
[ServiceContract(Namespace = "http://www.microsoft.com/biztalk/2006/r2/wcf-adapter")]
public interface IOneWayAsync
{
    // Methods
    [OperationContract(AsyncPattern = true, IsOneWay = true, Action = "*")]
    IAsyncResult BeginOneWayMethod(Message message, AsyncCallback callback, object
state);
    [OperationContract(IsOneWay = true, Action = "BizTalkSubmit")]
    void BizTalkSubmit(Message message);
    void EndOneWayMethod(IAsyncResult result);
```

```
}

[ServiceContract(Namespace = "http://www.microsoft.com/biztalk/2006/r2/wcf-adapter")]
public interface IOneWayAsyncTxn
{
    // Methods
    [OperationContract(AsyncPattern = true, IsOneWay = true, Action = "*")]
    IAsyncResult BeginOneWayMethod(Message message, AsyncCallback callback, object
state);
    [OperationContract(IsOneWay = true, Action = "BizTalkSubmit")]
    void BizTalkSubmit(Message message);
    void EndOneWayMethod(IAsyncResult result);
}

[ServiceContract(Namespace = "http://www.microsoft.com/biztalk/2006/r2/wcf-adapter")]
public interface ITwoWayAsync
{
    // Methods
    [OperationContract(AsyncPattern = true, IsOneWay = false, Action = "*", ReplyAction =
"*")]
    IAsyncResult BeginTwoWayMethod(Message message, AsyncCallback callback, object
state);
    [OperationContract(IsOneWay = false, Action = "BizTalkSubmit")]
    Message BizTalkSubmit(Message message);
    Message EndTwoWayMethod(IAsyncResult result);
}

[ServiceContract(Namespace = "http://www.microsoft.com/biztalk/2006/r2/wcf-adapter")]
public interface ITwoWayAsyncVoid
{
    // Methods
    [OperationContract(AsyncPattern = true, IsOneWay = false, Action = "*", ReplyAction =
"*")]
    IAsyncResult BeginTwoWayMethod(Message message, AsyncCallback callback, object
state);
    [OperationContract(IsOneWay = false, Action = "BizTalkSubmit")]
    void BizTalkSubmit(Message message);
    void EndTwoWayMethod(IAsyncResult result);
}

[ServiceContract(Namespace = "http://www.microsoft.com/biztalk/2006/r2/wcf-adapter")]
public interface ITwoWayAsyncVoidTxn
```

```
{
    // Methods
    [TransactionFlow(TransactionFlowOption.Mandatory)]
    [OperationContract(AsyncPattern = true, IsOneWay = false, Action = "*", ReplyAction =
"*")]
    IAsyncResult BeginTwoWayMethod(Message message, AsyncCallback callback, object
state);
    [TransactionFlow(TransactionFlowOption.Mandatory), OperationContract(IsOneWay =
false, Action = "BizTalkSubmit")]
    void BizTalkSubmit(Message message);
    void EndTwoWayMethod(IAsyncResult result);
}
```

All the methods exposed by these service contracts are generic, asynchronous, and untyped. In fact, as shown in the previous example, each method is decorated by the **OperationContract** attribute and, in particular:

- `AsyncPattern = True` indicates that an operation is implemented asynchronously using a **Begin***methodName* and **End***methodName* method pair. In a service contract `Action = "*"` means that the method accepts a message with any action.
- `ReplyAction = "*"` means that the method can return a message with any action.
- Every method accepts as a parameter or returns a generic WCF message

As a consequence, each WCF receive location can accept multiple message types, and versions can be normalized into a canonical format using a different map before being published to the **MessageBox**. Also WCF send ports are message-type agnostic.

Scenarios

As I mentioned in the introduction, the main objective of this article is to demonstrate how to integrate a BizTalk Server application with the new messaging entities supplied by the Service Bus. To this purpose I designed and implemented a solution made up of several projects in which a Windows Forms client application sends a request message to a BizTalk application and waits for a response. The solution implements the following communication patterns:

- The client application sends a request message directly to BizTalk Server using the BasicHttpBinding and a synchronous request-response message exchange pattern. In this context, the BizTalk application uses a **WCF-BasicHttp** request-response receive location to receive requests and send responses. I will not cover this pattern in the article as it's out of scope and is mainly used to verify that the application works as expected
- The client application sends a request message to BizTalk Server through the Service Bus using the NetTcpRelayBinding and a synchronous request-response message exchange pattern. In this case, to receive requests and send responses, the BizTalk application uses a **WCF-Custom** request-response receive location that is configured to use the NetTcpRelayBinding and the TransportClientEndpointBehavior. TransportClientEndpointBehavior specifies the Service Bus credentials that the receive location uses to authenticate with the Access Control Service. Even if this pattern uses the Service Bus, I will not cover it in the article as it does use Relay Messaging and not the

Brokered Messaging features of the Service Bus. You can consider it as a sort of bonus track!

- The client application exchanges request and response messages with the BizTalk application using Service Bus queues, topics, and subscriptions. In the remainder of the article we'll analyze this case.

Warning

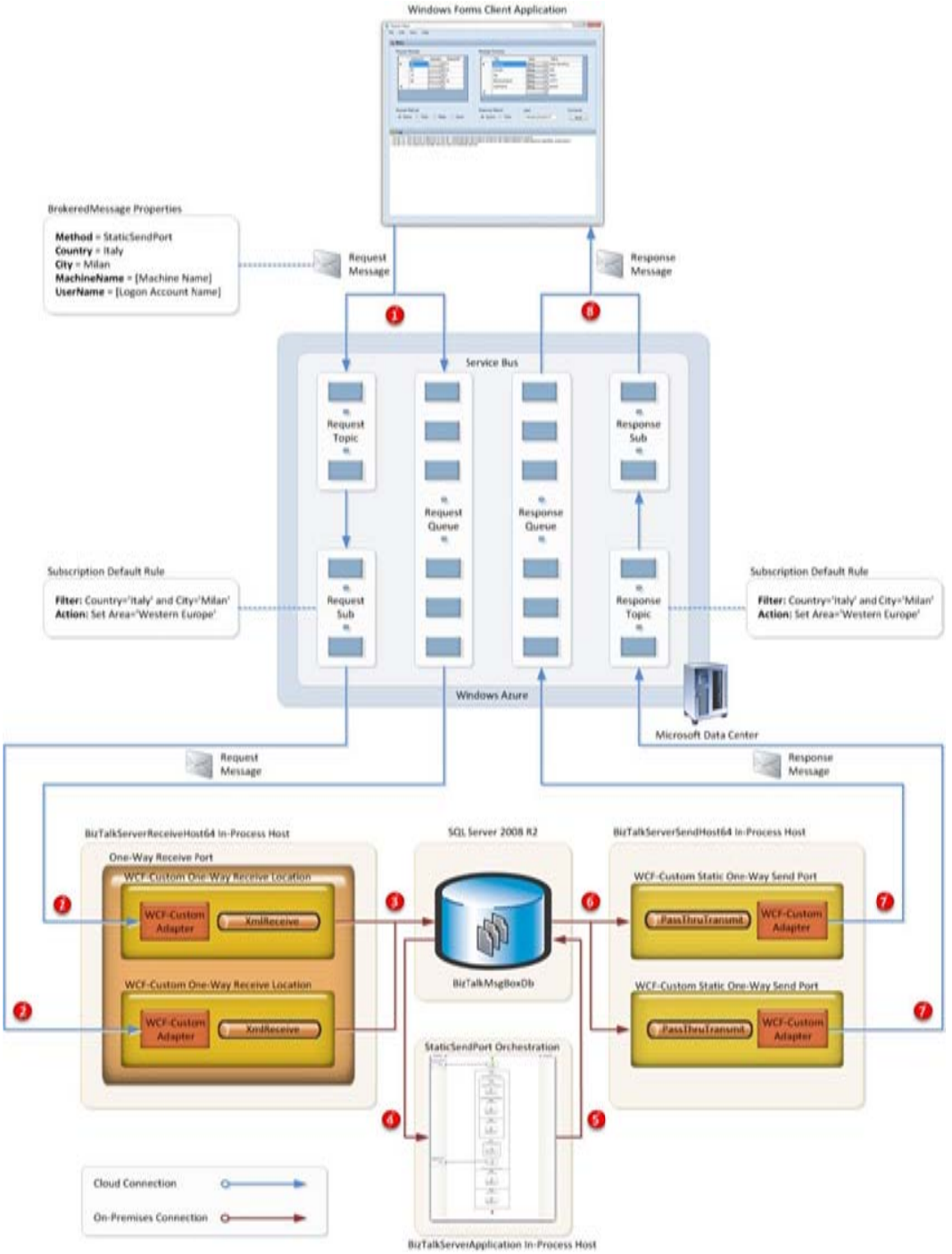
The Windows Azure AppFabric SDK V1.5 requires the .NET Framework version 4. As a consequence, in order to exploit the new messaging features provided by the Service Bus in a BizTalk application, you need to use BizTalk Server 2010, which is the only version to support the .NET Framework version 4.

StaticSendPortOrchestration

In this example, the **StaticSendPortOrchestration** can receive request messages from a queue A or a subscription X and return the corresponding response message to a queue B or topic Y. The latter are defined in two separate **WCF-Custom** static one-way send ports. Besides, the **StaticSendPortOrchestration** assigns the **MessageId** of the request message to the **CorrelationId** of the response to allow the client application to correlate the response to the initial request.

1. The client application sends the request message to a queue A and waits for the response message on a separate queue B. In a competing consumer scenario, the queue B is shared by multiple client applications to receive response messages. As a consequence, the response message could be read by a client other than that which sent the initial request.
2. The client application sends the request message to a queue A and waits for the response message on a subscription Z. In this case, the **StaticSendPortOrchestration** writes the response message to statically defined topic Y. In a competing scenario, different instances of the client application should use separate subscriptions defined by complementary rules to make sure that the response message is received by the same instance that initiated the request. Conversely, to implement a decoupled message fan-out scenario where the BizTalk application sends a notification message to multiple subscribers, you can create a separate subscription for each of the consumers and properly define rules to receive only those messages that satisfy a certain condition.
3. The client application sends the request message to a topic A and waits for the response message on a queue B. The same considerations made at point 1 apply to this case.
4. The client application sends the request message to a topic X and waits for the response message on a subscription Z. Even in this case, the **StaticSendPortOrchestration** writes the response message to statically defined topic Y. The same considerations made at point 2 apply to this case.

The following figure shows the architecture of the **StaticSendPortOrchestration** example.



1. The Windows Forms application uses a WCF proxy to send a request message to the requestqueue or to the **requesttopic**.
2. The BizTalk application uses a one-way receive port to receive request messages from the Service Bus. In particular, a WCF-Custom receive location called **ServiceBusSample.WCF-Custom.NetMessagingBinding.Queue.ReceiveLocation** is used to read request messages from the **requestqueue**, whereas another WCF-Custom receive location called

ServiceBusSample.WCF-Custom.NetMessagingBinding.Subscription.ReceiveLocation is used to receive request messages from the **italyMilan** subscription. Both receive locations are configured to use the following components:

- The NetMessagingBinding is used to receive messages from a queue or a subscription.
- A custom WCF Message Inspector called **ServiceBusMessageInspector**. At runtime, this component reads the BrokeredMessageProperty from the Properties collection of the inbound WCF Message and transforms its public properties (such as MessageId and ReplyTo) and application specific properties (the key/value pairs contained in the Properties collection of the BrokeredMessageProperty) into context properties of the BizTalk message.
- The TransportClientEndpointBehavior specifies the Service Bus credentials (in this sample I use **SharedSecret** credentials) used to authenticate with the Access Control Service.
- The **ListenUriEndpointBehavior** is used by the WCF-Custom receive location that retrieves messages from the **ItalyMilan** subscription defined on the **requesttopic**. This custom component is used to set the value of the ListenUri property of the service endpoint. See later in this article for more information about this component.
- If an application retrieves messages from a sessionful queue or subscription, the **SessionChannelEndpointBehavior** must be added to the configuration of the WCF-Custom receive location. This custom component is used to make sure that at runtime the WCF-Custom adapter creates an IInputSessionChannel in place of an IInputChannel. This is a mandatory requirement to receive messages from a sessionful messaging entity.

3. The **Message Agent** submits the request message to the MessageBox (BizTalkMsgBoxDb).
4. The inbound request starts a new instance of the **StaticSendPortOrchestration**. The latter uses a Direct Port to receive request messages that satisfy the following filter expression: `Microsoft.WindowsAzure.CAT.Schemas.Method == StaticSendPort`. The orchestration invokes a method exposed by the **RequestManager** helper component to calculate the response message. Then it copies the context properties from the request message to the response message and then assigns the value of the **MessageId** context property of the request message to the **CorrelationId** property of the response message. Finally, it checks the string value contained in the **ReplyTo** context property: if it contains the word "queue," the orchestration returns the response message through a logical port bound to a **WCF-Custom** send port configured to send messages to the **responsequeue**, otherwise it publishes the response to the MessageBox via a logical send port bound to a **WCF-Custom** send port configured to send messages to the **responsetopic**.
5. The **Message Agent** submits the request message to the MessageBox (BizTalkMsgBoxDb).
6. The response message is consumed by one of the following WCF-Custom send ports:
   - **ServiceBusSample.WCF-Custom.NetMessagingBinding.Queue.SendPort**: this send port writes message to the **responsequeue**.
   - **ServiceBusSample.WCF-Custom.NetMessagingBinding.Topic.SendPort**: this send port writes message to the **responsetopic**
7. The selected send port writes the response message to the **responsequeue** or **responsetopic**. Both send ports are configured to use the following components:
   - The NetMessagingBinding is used to send messages to the Service Bus.

- The **ServiceBusMessageInspector** transforms the message context properties into **BrokeredMessage** properties
- The TransportClientEndpointBehavior specifies the Service Bus credentials (in this sample I use **SharedSecret** credentials) used by to authenticate with the Access Control Service.

8. The client application uses a WCF service with 2 endpoints to retrieve the reply message from the **responsequeue** or from the **responsetopic**. In an environment with multiple client applications, each of them should use a separate queue or subscription to receive response messages from BizTalk.

DynamicSendPortOrchestration

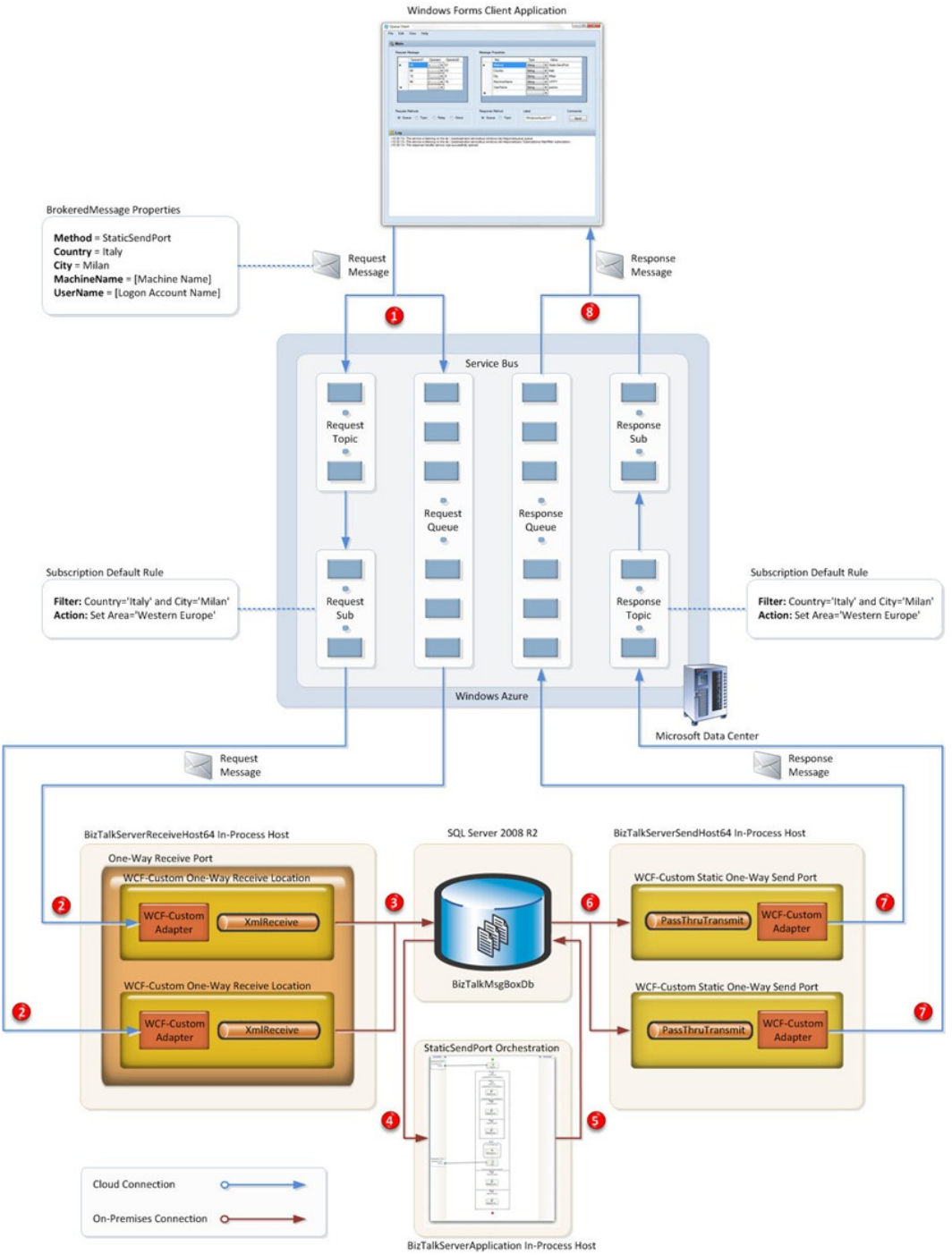
In this example, the **DynamicSendPortOrchestration** can receive request messages from a queue A or a subscription X and return the corresponding response message to a queue or a topic whose address has been provided by the client in the **ReplyTo** property of the **BrokeredMessage**. Different clients can use distinct queues or subscriptions defined on different topics to receive response messages. In my demo, the only requirement is that they all belong to the same Service Bus namespace. By the way, this constraint can be easily removed by using a repository (such as SQL Server) to store credentials for different namespaces. This test case demonstrates how to implement a dynamic request-reply message exchange pattern in which the sender communicates to the receiver the address of the messaging entity (queue or topic) where to send the response and the receiver (the orchestration) assigns the **MessageId** of the request message to the **CorrelationId** of the response to allow the client application to correlate the response to the initial request.

1. The client application specifies the address of the response queue B in the **ReplyTo** property of the request message sends the latter to a queue A and waits for the response message on a queue B. The **DynamicSendPortOrchestration** retrieves the URL of the response queue from the **ReplyTo** context property of the request message and assign it to the address of the dynamic send port used to send the response. In a competing consumer scenario, different clients can use distinct queues to receive response messages. This way, the response message will always be received by the client that sent the initial request.
2. The client application specifies the address of the response topic Y in the **ReplyTo** property of the request message sends the latter to a queue A and waits for the response message on a subscription Z. The **DynamicSendPortOrchestration** retrieves the URL of the response topic from the **ReplyTo** context property of the request message and assign it to the address of the dynamic send port used to send the response. In a competing scenario, different instances of the client application should use separate subscriptions defined by complementary rules to make sure that the response message is received by the same instance that initiated the request. However, the subscription they use to receive responses from the BizTalk application can be associated to different topics. Conversely, to implement a

decoupled message fan-out scenario where the BizTalk application sends a notification message to multiple subscribers, you can create a separate subscription for each of the consumers and properly define rules to receive only those messages that satisfy a certain condition.

3. The client application specifies the address of the response queue B in the **ReplyTo** property of the request message sends the latter to a topic X and waits for the response message on a queue B. The same considerations made at point 1 apply to this case.
4. The client application specifies the address of the response topic Y in the **ReplyTo** property of the request message sends the latter to a topic X and waits for the response message on a subscription Z. The same considerations made at point 2 apply to this case.

The following figure shows the architecture of the **DynamicSendPortOrchestration** example.



1. The Windows Forms application uses a WCF proxy to send a request message to the **requestqueue** or to the **requesttopic**.
2. The BizTalk application uses a one-way receive port to receive request messages from the Service Bus. In particular, a **WCF-Custom** receive location called **ServiceBusSample.WCF-Custom.NetMessagingBinding.Queue.ReceiveLocation** is used to read request messages from the **requestqueue**, whereas another **WCF-Custom** receive location called **ServiceBusSample.WCF-**

Custom.NetMessagingBinding.Subscription.ReceiveLocationis used to receive request messages from the **italyMilan** subscription. Both receive locations are configured to use the following components:

- The NetMessagingBinding is used to receive messages from a queue or a subscription
- A custom WCF message inspector called **ServiceBusMessageInspector**. At runtime, this component reads the BrokeredMessageProperty from the Properties collection of the inbound WCF Message and transforms its public properties (such as **MessageId** and **ReplyTo**) and application specific properties (the key/value pairs contained in the Properties collection of the BrokeredMessageProperty) into context properties of the BizTalk message.
- The TransportClientEndpointBehavior specifies the Service Bus credentials (in this sample I use **SharedSecret** credentials) used to authenticate with the Access Control Service.
- The **ListenUriEndpointBehavior** is used by the **WCF-Custom** receive location that retrieves messages from the **italyMilan** subscription defined on the **requesttopic**. This custom component is used to set the value of the ListenUri property of the service endpoint. See later in this article for more information about this component.
- When an application retrieves messages from a sessionful queue or subscription, the **SessionChannelEndpointBehavior** must be added to the configuration of the **WCF-Custom** receive location. This custom component is used to make sure that at runtime the **WCF-Custom** adapter creates an IInputSessionChannel in place of an IInputChannel. This is a mandatory requirement to receive messages from a sessionful messaging entity.

3. The **Message Agent** submits the request message to the MessageBox (**BizTalkMsgBoxDb**).
4. The inbound request starts a new instance of the **DynamicSendPortOrchestration**. The latter uses a Direct Port to receive request messages that satisfy the following filter expression: `Microsoft.WindowsAzure.CAT.Schemas.Method == DynamicSendPort`. The orchestration invokes a method exposed by the **RequestManager** helper component to calculate the response message. Then it copies the context properties from the request message to the response message and then assign the value of the **MessageId** context property of the request message to the **CorrelationId** property of the response message. Then it reads from the **ReplyTo** context property the address of the messaging entity (queue or topic) to which it should send the response. Finally, the orchestration configures the context properties of the response message and the properties of the dynamic send port to use the following components:
   - The NetMessagingBinding is used to send messages to the Service Bus.
   - The **ServiceBusMessageInspector** transforms the message context properties into **BrokeredMessage** properties
   - The TransportClientEndpointBehavior specifies the Service Bus credentials (in this sample I use **SharedSecret** credentials) used to authenticate with the Access Control Service.
5. The **Message Agent** submits the response message to the MessageBox (BizTalkMsgBoxDb).
6. The response message is consumed by the dynamic one-way send port named **ServiceBusSample.Dynamic.SendPort**.

7. The send port writes the response message to the messaging entity (in my sample **responsequeue** or **responsetopic**) whose address has been indicated by the client application in the **ReplyTo** property of the BrokeredMessageProperty.
8. The client application uses a WCF service with 2 endpoints to retrieve the reply message from the **responsequeue** or from the **responsetopic**. In an environment with multiple client applications, each of them should use a separate queue or subscription to receive response messages from BizTalk.

Solution

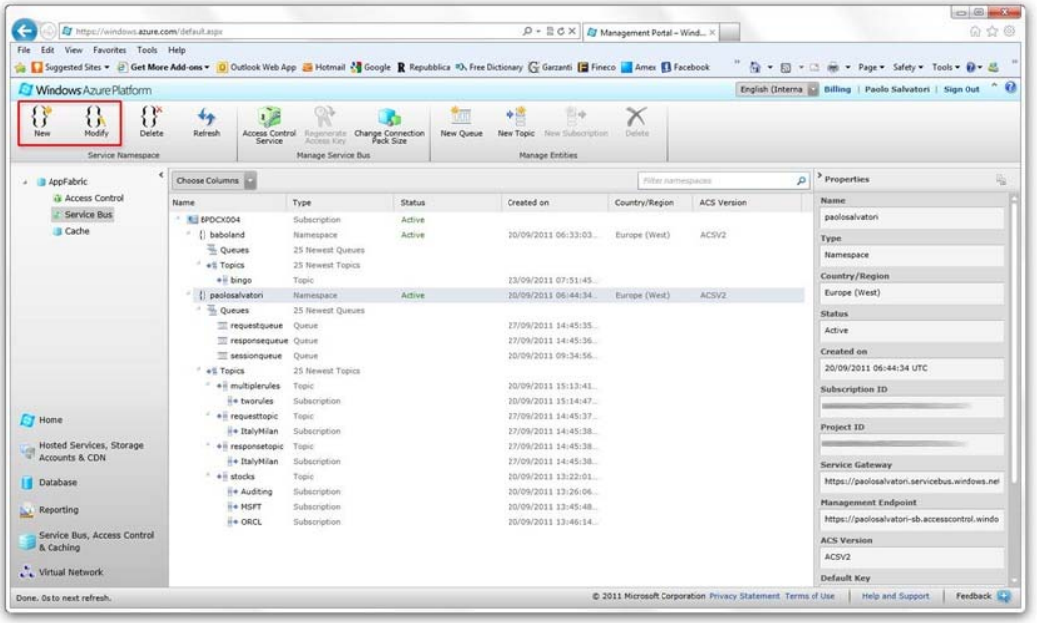
Now that we described the scenarios we can analyze in detail the main components of the solution.

In This Section

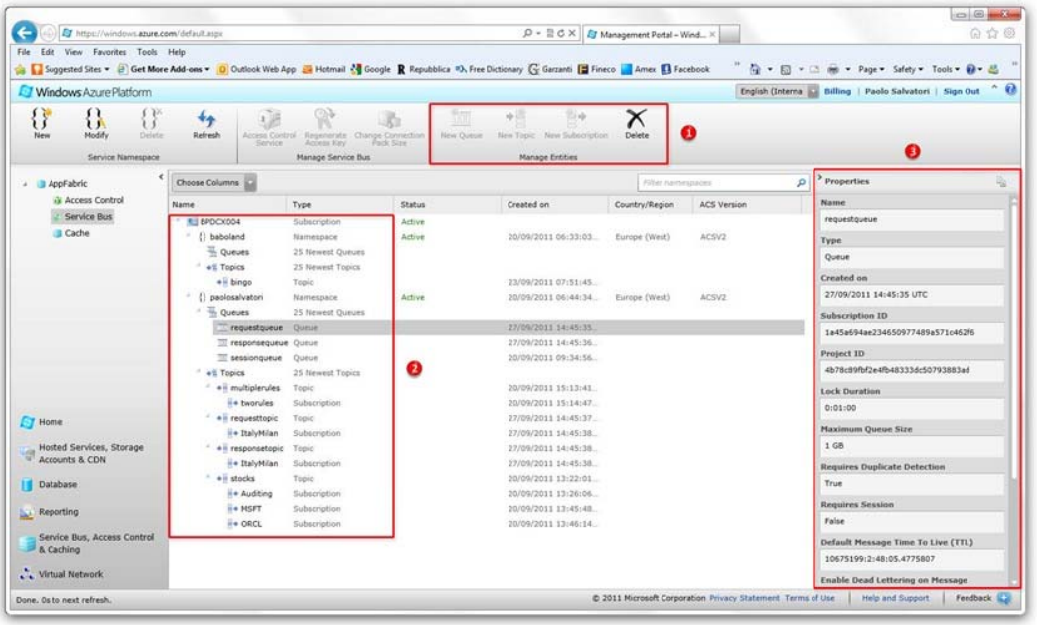
Queues, Topics and Subscriptions

The first operation to perform to properly configure the environment is creating the messaging entities used by the demo. The first operation to execute is to provision a new Service Bus namespace or modify an existing namespace to include the Service Bus. You can accomplish

this task from the **Windows Azure Management Portal** by clicking the **New** button or the **Modify** button, respectively, highlighted in the figure below.



After you provision a new Service Bus namespace or modify an existing namespace to include the Service Bus, the next step is to create the queue, topic and subscription entities required by the demo. As I mentioned in the introduction, you have many options to accomplish this task. The easiest way is by using the **Windows Azure Management Portal** using the buttons in the **Manage Entities** command bar highlighted in red at point 1 in the following figure.



You can use the navigation tree-view shown at point 2 to select an existing entity and display its properties in the vertical bar highlighted at point 3. To remove the selected entity, you press the **Delete** button in the **Manage Entities** command bar.

Note

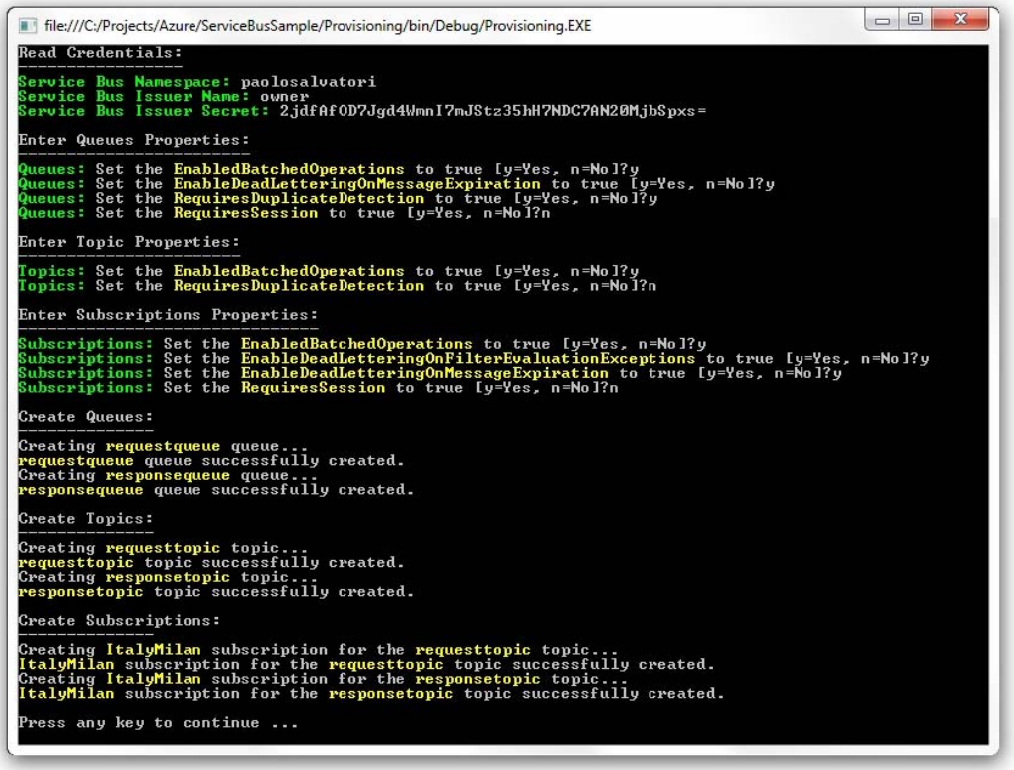
The current user interface supplied by the **Windows Azure Management Portal** allows a user to create queues, topics, and subscriptions and define their properties, but not to create or display rules for an existing subscription. At the moment, you can accomplish this task by only using the .NET messaging API. In particular, to add a new rule to an existing subscription you can use the AddRule(String, Filter) or the AddRule(RuleDescription) methods exposed by the SubscriptionClient class, whereas to enumerate the rules of an existing subscription, you can use the GetRules method of the NamespaceManager class. The latter can be used for managing entities, such as queues, topics, subscriptions, and rules, in a Service Bus namespace.In June, I created a tool called Service Bus Explorer that allows a user to create, delete and test queues, topics, subscriptions, and rules. My tool was able to manage entities in the AppFabric Labs Beta environment. However, the new version of the Service Bus API introduced some breaking changes, as you can read here, so I'm working at a new version of the Service Bus Explorer tool that will introduce a significant amount of interesting and hopefully useful features. So stay tuned and come back frequently on this site as I plan to publish it soon.

For your convenience, I created a console application called **Provisioning** that uses the functionality provided by the NamespaceManager class to create the queues, topics, and subscriptions required by the solution. When it starts, the console applications prompt for service namespace credentials. These are used to authenticate with the Access Control service, and acquire an access token that proves to the Service Bus infrastructure that the application is authorized to provision new messaging entities. Then the application prompts for the value to assign to the properties of the entities to create such as EnabledBatchedOperations and EnableDeadLetteringOnMessageExpiration for queues. The Provisioning application creates the following entities in the specified Service Bus namespace:

- A queue called **requestqueue** used by the client application to send request messages to BizTalk Server.
- A queue called **responsequeue** used by BizTalk Server to send response messages to the client application.
- A topic called **requesttopic** used by the client application to send request messages to BizTalk Server.
- A topic called **responsetopic** used by BizTalk Server to send response messages to the client application.
- A subscription called **ItalyMilan** for the **requesttopic**. The latter is used by BizTalk Server to receive request messages from the **requesttopic**. The subscription in question has a single rule defined as follows:
  a. **Filter**: `Country='Italy' and City='Milan'`
  b. **Action**: `Set Area='Western Europe'`

- A subscription called **ItalyMilan** for the **responsetopic**. The latter is used by client application to receive response messages from the **responsetopic**. The subscription in question has a single rule defined as follows:
    a. **Filter**: `Country='Italy' and City='Milan'`
    b. **Action**: `Set Area='Western Europe'`

The following figure displays the output of the **Provisioning** console application.



The following example contains the code of the **Provisioning** application:

```
#region Using Directives
using System;
using Microsoft.ServiceBus;
using Microsoft.ServiceBus.Messaging;
#endregion

namespace Microsoft.WindowsAzure.CAT.Samples.ServiceBus.Provisioning
{
    static class Program
    {
        #region Private Constants
        //***************************
```

```
// Constants
//***************************

private const string RequestQueue = "requestqueue";
private const string ResponseQueue = "responsequeue";
private const string RequestTopic = "requesttopic";
private const string ResponseTopic = "responsetopic";
private const string RequestSubscription = "ItalyMilan";
private const string ResponseSubscription = "ItalyMilan";
#endregion
static void Main()
{
    var defaultColor = Console.ForegroundColor;

    try
    {
        // Set Window Size
        Console.WindowWidth = 100;
        Console.BufferWidth = 100;
        Console.WindowHeight = 48;
        Console.BufferHeight = 48;

        // Print Header
        Console.WriteLine("Read Credentials:");
        Console.WriteLine("-----------------");

        // Read Service Bus Namespace
        Console.ForegroundColor = ConsoleColor.Green;
        Console.Write("Service Bus Namespace: ");
        Console.ForegroundColor = defaultColor;
        var serviceNamespace = Console.ReadLine();

        // Read Service Bus Issuer Name
        Console.ForegroundColor = ConsoleColor.Green;
        Console.Write("Service Bus Issuer Name: ");
        Console.ForegroundColor = defaultColor;
        var issuerName = Console.ReadLine();

        // Read Service Bus Issuer Secret
        Console.ForegroundColor = ConsoleColor.Green;
        Console.Write("Service Bus Issuer Secret: ");
        Console.ForegroundColor = defaultColor;
```

```
var issuerSecret = Console.ReadLine();

// Print Header
Console.WriteLine();
Console.WriteLine("Enter Queues Properties:");
Console.WriteLine("------------------------");

// Read Queue EnabledBatchedOperations
Console.ForegroundColor = ConsoleColor.Green;
Console.Write("Queues: ");
Console.ForegroundColor = defaultColor;
Console.Write("Set the ");
Console.ForegroundColor = ConsoleColor.Yellow;
Console.Write("EnabledBatchedOperations ");
Console.ForegroundColor = defaultColor;
Console.Write("to true [y=Yes, n=No]?");
var key = Console.ReadKey().KeyChar;
var queueEnabledBatchedOperations = key == 'y' || key == 'Y';
Console.WriteLine();

// Read Queue EnableDeadLetteringOnMessageExpiration
Console.ForegroundColor = ConsoleColor.Green;
Console.Write("Queues: ");
Console.ForegroundColor = defaultColor;
Console.Write("Set the ");
Console.ForegroundColor = ConsoleColor.Yellow;
Console.Write("EnableDeadLetteringOnMessageExpiration ");
Console.ForegroundColor = defaultColor;
Console.Write("to true [y=Yes, n=No]?");
key = Console.ReadKey().KeyChar;
var queueEnableDeadLetteringOnMessageExpiration = key == 'y' || key ==
'Y';
Console.WriteLine();

// Read Queue RequiresDuplicateDetection
Console.ForegroundColor = ConsoleColor.Green;
Console.Write("Queues: ");
Console.ForegroundColor = defaultColor;
Console.Write("Set the ");
Console.ForegroundColor = ConsoleColor.Yellow;
Console.Write("RequiresDuplicateDetection ");
Console.ForegroundColor = defaultColor;
```

```
Console.Write("to true [y=Yes, n=No]?");
key = Console.ReadKey().KeyChar;
var queueRequiresDuplicateDetection = key == 'y' || key == 'Y';
Console.WriteLine();

// Read Queue RequiresSession
Console.ForegroundColor = ConsoleColor.Green;
Console.Write("Queues: ");
Console.ForegroundColor = defaultColor;
Console.Write("Set the ");
Console.ForegroundColor = ConsoleColor.Yellow;
Console.Write("RequiresSession ");
Console.ForegroundColor = defaultColor;
Console.Write("to true [y=Yes, n=No]?");
key = Console.ReadKey().KeyChar;
var queueRequiresSession = key == 'y' || key == 'Y';
Console.WriteLine();

// Print Header
Console.WriteLine();
Console.WriteLine("Enter Topic Properties:");
Console.WriteLine("------------------------");

// Read Topic EnabledBatchedOperations
Console.ForegroundColor = ConsoleColor.Green;
Console.Write("Topics: ");
Console.ForegroundColor = defaultColor;
Console.Write("Set the ");
Console.ForegroundColor = ConsoleColor.Yellow;
Console.Write("EnabledBatchedOperations ");
Console.ForegroundColor = defaultColor;
Console.Write("to true [y=Yes, n=No]?");
key = Console.ReadKey().KeyChar;
var topicEnabledBatchedOperations = key == 'y' || key == 'Y';
Console.WriteLine();

// Read Topic RequiresDuplicateDetection
Console.ForegroundColor = ConsoleColor.Green;
Console.Write("Topics: ");
Console.ForegroundColor = defaultColor;
Console.Write("Set the ");
Console.ForegroundColor = ConsoleColor.Yellow;
```

```
                Console.Write("RequiresDuplicateDetection ");
                Console.ForegroundColor = defaultColor;
                Console.Write("to true [y=Yes, n=No]?");
                key = Console.ReadKey().KeyChar;
                var topicRequiresDuplicateDetection = key == 'y' || key == 'Y';
                Console.WriteLine();

                // Print Header
                Console.WriteLine();
                Console.WriteLine("Enter Subscriptions Properties: ");
                Console.WriteLine("-------------------------------");

                // Read Subscription EnabledBatchedOperations
                Console.ForegroundColor = ConsoleColor.Green;
                Console.Write("Subscriptions: ");
                Console.ForegroundColor = defaultColor;
                Console.Write("Set the ");
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write("EnabledBatchedOperations ");
                Console.ForegroundColor = defaultColor;
                Console.Write("to true [y=Yes, n=No]?");
                key = Console.ReadKey().KeyChar;
                var subscriptionnabledBatchedOperations = key == 'y' || key == 'Y';
                Console.WriteLine();

                // Read Subscription EnableDeadLetteringOnFilterEvaluationExceptions
                Console.ForegroundColor = ConsoleColor.Green;
                Console.Write("Subscriptions: ");
                Console.ForegroundColor = defaultColor;
                Console.Write("Set the ");
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write("EnableDeadLetteringOnFilterEvaluationExceptions ");
                Console.ForegroundColor = defaultColor;
                Console.Write("to true [y=Yes, n=No]?");
                key = Console.ReadKey().KeyChar;
                var subscriptionEnableDeadLetteringOnFilterEvaluationExceptions = key ==
'y' || key == 'Y';
                Console.WriteLine();

                // Read Subscription EnableDeadLetteringOnMessageExpiration
                Console.ForegroundColor = ConsoleColor.Green;
                Console.Write("Subscriptions: ");
```

```
                Console.ForegroundColor = defaultColor;
                Console.Write("Set the ");
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write("EnableDeadLetteringOnMessageExpiration ");
                Console.ForegroundColor = defaultColor;
                Console.Write("to true [y=Yes, n=No]?");
                key = Console.ReadKey().KeyChar;
                var subscriptionEnableDeadLetteringOnMessageExpiration = key == 'y' ||
key == 'Y';
                Console.WriteLine();

                // Read Subscription RequiresSession
                Console.ForegroundColor = ConsoleColor.Green;
                Console.Write("Subscriptions: ");
                Console.ForegroundColor = defaultColor;
                Console.Write("Set the ");
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write("RequiresSession ");
                Console.ForegroundColor = defaultColor;
                Console.Write("to true [y=Yes, n=No]?");
                key = Console.ReadKey().KeyChar;
                var subscriptionRequiresSession = key == 'y' || key == 'Y';
                Console.WriteLine();

                // Get ServiceBusNamespaceClient for management operations
                var managementUri =
                    ServiceBusEnvironment.CreateServiceUri("https", serviceNamespace,
string.Empty);
                var tokenProvider =
                    TokenProvider.CreateSharedSecretTokenProvider(issuerName,
issuerSecret);
                var namespaceManager = new NamespaceManager(managementUri,
tokenProvider);

                // Print Header
                Console.WriteLine();
                Console.WriteLine("Create Queues:");
                Console.WriteLine("--------------");

                // Create RequestQueue
                Console.Write("Creating ");
                Console.ForegroundColor = ConsoleColor.Yellow;
```

```
                Console.Write(RequestQueue);
                Console.ForegroundColor = defaultColor;
                Console.WriteLine(" queue...");
                if (namespaceManager.QueueExists(RequestQueue))
                {
                    namespaceManager.DeleteQueue(RequestQueue);
                }
                namespaceManager.CreateQueue(new QueueDescription(RequestQueue)
                                {
                                    EnableBatchedOperations =
queueEnabledBatchedOperations,
                                    EnableDeadLetteringOnMessageExpiration =
                                        queueEnableDeadLetteringOnMessageExpiration,
                                    RequiresDuplicateDetection =
queueRequiresDuplicateDetection,
                                    RequiresSession = queueRequiresSession
                                });
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write(RequestQueue);
                Console.ForegroundColor = defaultColor;
                Console.WriteLine(" queue successfully created.");

                // Create ResponseQueue
                Console.Write("Creating ");
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write(ResponseQueue);
                Console.ForegroundColor = defaultColor;
                Console.WriteLine(" queue...");
                if (namespaceManager.QueueExists(ResponseQueue))
                {
                    namespaceManager.DeleteQueue(ResponseQueue);
                }
                namespaceManager.CreateQueue(new QueueDescription(ResponseQueue)
                            {
                                EnableBatchedOperations = queueEnabledBatchedOperations,
                                EnableDeadLetteringOnMessageExpiration =
                                    queueEnableDeadLetteringOnMessageExpiration,
                                RequiresDuplicateDetection =
queueRequiresDuplicateDetection,
                                RequiresSession = queueRequiresSession
                            });
                Console.ForegroundColor = ConsoleColor.Yellow;
```

```
                Console.Write(ResponseQueue);
                Console.ForegroundColor = defaultColor;
                Console.WriteLine(" queue successfully created.");

                // Print Header
                Console.WriteLine();
                Console.WriteLine("Create Topics:");
                Console.WriteLine("--------------");

                // Create RequestTopic
                Console.Write("Creating ");
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write(RequestTopic);
                Console.ForegroundColor = defaultColor;
                Console.WriteLine(" topic...");
                if (namespaceManager.TopicExists(RequestTopic))
                {
                    namespaceManager.DeleteTopic(RequestTopic);
                }
                namespaceManager.CreateTopic(new TopicDescription(RequestTopic)
                                        {
                                            EnableBatchedOperations =
topicEnabledBatchedOperations,
                                            RequiresDuplicateDetection =
topicRequiresDuplicateDetection
                                        });
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write(RequestTopic);
                Console.ForegroundColor = defaultColor;
                Console.WriteLine(" topic successfully created.");

                // Create ResponseTopic
                Console.Write("Creating ");
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write(ResponseTopic);
                Console.ForegroundColor = defaultColor;
                Console.WriteLine(" topic...");
                if (namespaceManager.TopicExists(ResponseTopic))
                {
                    namespaceManager.DeleteTopic(ResponseTopic);
                }
                namespaceManager.CreateTopic(new TopicDescription(ResponseTopic)
```

```
                                    {
                                        EnableBatchedOperations =
topicEnabledBatchedOperations,
                                        RequiresDuplicateDetection =
topicRequiresDuplicateDetection
                                    });
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write(ResponseTopic);
                Console.ForegroundColor = defaultColor;
                Console.WriteLine(" topic successfully created.");

                // Print Header
                Console.WriteLine();
                Console.WriteLine("Create Subscriptions:");
                Console.WriteLine("--------------");

                // Create Request Subscription
                Console.Write("Creating ");
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write(RequestSubscription);
                Console.ForegroundColor = defaultColor;
                Console.Write(" subscription for the ");
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write(RequestTopic);
                Console.ForegroundColor = defaultColor;
                Console.WriteLine(" topic...");
                var ruleDescription = new RuleDescription(new SqlFilter("Country='Italy'
and City='Milan'"))
                                        {
                                            Name = "$Default",
                                            Action = new SqlRuleAction("Set
Area='Western Europe'")
                                        };
                var subscriptionDescription = new SubscriptionDescription(RequestTopic,
RequestSubscription)
                {
                    EnableBatchedOperations = subscriptionnabledBatchedOperations,
                    EnableDeadLetteringOnFilterEvaluationExceptions =
                            subscriptionEnableDeadLetteringOnFilterEvaluationExceptions,
                    EnableDeadLetteringOnMessageExpiration =
                            subscriptionEnableDeadLetteringOnMessageExpiration,
                    RequiresSession = subscriptionRequiresSession
```

```
                };
                namespaceManager.CreateSubscription(subscriptionDescription,
ruleDescription);
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write(RequestSubscription);
                Console.ForegroundColor = defaultColor;
                Console.Write(" subscription for the ");
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write(RequestTopic);
                Console.ForegroundColor = defaultColor;
                Console.WriteLine(" topic successfully created.");

                // Create Response Subscription
                Console.Write("Creating ");
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write(ResponseSubscription);
                Console.ForegroundColor = defaultColor;
                Console.Write(" subscription for the ");
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write(ResponseTopic);
                Console.ForegroundColor = defaultColor;
                Console.WriteLine(" topic...");
                ruleDescription = new RuleDescription(new SqlFilter("Country='Italy' and
City='Milan'"))
                                        {
                                            Action = new SqlRuleAction("Set Area='Western
Europe'")
                                        };
                subscriptionDescription = new SubscriptionDescription(ResponseTopic,
ResponseSubscription)
                {
                    EnableBatchedOperations = subscriptionnabledBatchedOperations,
                    EnableDeadLetteringOnFilterEvaluationExceptions =
                            subscriptionEnableDeadLetteringOnFilterEvaluationExceptions,
                    EnableDeadLetteringOnMessageExpiration =
                            subscriptionEnableDeadLetteringOnMessageExpiration,
                    RequiresSession = subscriptionRequiresSession
                };
                namespaceManager.CreateSubscription(subscriptionDescription,
ruleDescription);
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write(ResponseSubscription);
```

```
                Console.ForegroundColor = defaultColor;
                Console.Write(" subscription for the ");
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write(ResponseTopic);
                Console.ForegroundColor = defaultColor;
                Console.WriteLine(" topic successfully created.");
                Console.WriteLine();

                // Close the application
                Console.WriteLine("Press any key to continue ...");
                Console.ReadLine();
            }
            catch (Exception ex)
            {
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write("Exception: ");
                Console.ForegroundColor = defaultColor;
                Console.Write(ex.Message);
            }
        }
    }
}
```

Note

I did not test all the possible combinations of properties for queues and topics so the demo may not work as expected with all the configurations.

Data and Message Contracts

I started defining the data and message contracts for the request and response messages. These two types of contracts have different roles in WCF:

1. **Data contracts** provide a mechanism to map .NET CLR types that are defined in code and XML Schemas (XSD) defined by the W3C organization (www.w3c.org). Data contracts are published in the service's metadata, allowing clients to convert the neutral, technology-agnostic representation of the data types to their native representations.
2. **Message contracts** describe the structure of SOAP messages sent to and from a service and enable you to inspect and control most of the details in the SOAP header and body. Whereas data contracts enable interoperability through the XML Schema Definition (XSD) standard, message contracts enable you to interoperate with any system that communicates through SOAP. Using message contracts gives you complete control over the SOAP message sent to and from a service by providing access to the SOAP headers and bodies directly. This allows the use of simple or complex types to define the exact content of the SOAP parts.

In my solution, I created two separate projects called **DataContracts** and **MessageContracts** respectively. For your convenience, I included below the code of the classes used to define the Data and Message Contract of the request and response messages.

CalculatorRequest Class

```
[Serializable]
[XmlType(TypeName = "CalculatorRequest",
         Namespace = "http://windowsazure.cat.microsoft.com/samples/servicebus")]
[XmlRoot(ElementName = "CalculatorRequest",
         Namespace = http://windowsazure.cat.microsoft.com/samples/servicebus,
         IsNullable = false)]
[DataContract(Name = "CalculatorRequest",
              Namespace = "http://windowsazure.cat.microsoft.com/samples/servicebus")]
public class CalculatorRequest
{
    #region Private Fields
    private OperationList operationList;
    #endregion

    #region Public Constructors
    public CalculatorRequest()
    {
        operationList = new OperationList();
    }

    public CalculatorRequest(OperationList operationList)
    {
        this.operationList = operationList;
    }
    #endregion

    #region Public Properties
    [XmlArrayItem("Operation", Type=typeof(Operation), IsNullable = false)]
    [DataMember(Order = 1)]
    public OperationList Operations
    {
        get
        {
            return operationList;
        }
        set
        {
            operationList = value;
```

```
        }
    }
    #endregion
}

[CollectionDataContract(Name = "OperationList",
                        Namespace =
http://windowsazure.cat.microsoft.com/samples/servicebus,
                        ItemName = "Operation")]
public class OperationList : List<Operation>
{
}

[Serializable]
[XmlType(TypeName = "Operation",
         AnonymousType = true,
         Namespace = "http://windowsazure.cat.microsoft.com/samples/servicebus")]
[DataContract(Name = "Operation",
              Namespace = "http://windowsazure.cat.microsoft.com/samples/servicebus")]
public class Operation
{
    #region Private Fields
    private string op;
    private double operand1;
    private double operand2;
    #endregion

    #region Public Constructors
    public Operation()
    {
    }

    public Operation(string op,
                     double operand1,
                     double operand2)
    {
        this.op = op;
        this.operand1 = operand1;
        this.operand2 = operand2;
    }
    #endregion
```

```
#region Public Properties
[XmlElement]
[DataMember(Order = 1)]
public string Operator
{
    get
    {
        return op;
    }
    set
    {
        op = value;
    }
}

[XmlElement]
[DataMember(Order = 2)]
public double Operand1
{
    get
    {
        return operand1;
    }
    set
    {
        operand1 = value;
    }
}

[XmlElement]
[DataMember(Order = 3)]
public double Operand2
{
    get
    {
        return operand2;
    }
    set
    {
        operand2 = value;
    }
}
```

```
    #endregion
}
```

CalculatorResponse Class

```
[Serializable]
[XmlType(TypeName = "CalculatorResponse",
         Namespace = "http://windowsazure.cat.microsoft.com/samples/servicebus")]
[XmlRoot(ElementName = "CalculatorResponse",
         Namespace = http://windowsazure.cat.microsoft.com/samples/servicebus,
         IsNullable = false)]
[DataContract(Name = "CalculatorResponse",
         Namespace = "http://windowsazure.cat.microsoft.com/samples/servicebus")]
public class CalculatorResponse
{
    #region Private Fields
    private string status;
    private ResultList resultList;
    #endregion

    #region Public Constructors
    public CalculatorResponse()
    {
        status = default(string);
        resultList = new ResultList();
    }

    public CalculatorResponse(string status)
    {
        this.status = status;
        resultList = new ResultList();
    }

    public CalculatorResponse(string status, ResultList resultList)
    {
        this.status = status;
        this.resultList = resultList;
    }
    #endregion

    #region Public Properties
```

```
    [XmlElement]
    [DataMember(Order = 1)]
    public string Status
    {
        get
        {
            return status;
        }
        set
        {
            status = value;
        }
    }

    [XmlArrayItem("Result", Type=typeof(Result), IsNullable=false)]
    [DataMember(Order = 2)]
    public ResultList Results
    {
        get
        {
            return resultList;
        }
        set
        {
            resultList = value;
        }
    }
    #endregion
}

[CollectionDataContract(Name = "ResultList",
                        Namespace =
http://windowsazure.cat.microsoft.com/samples/servicebus,
                        ItemName = "Result")]
public class ResultList : List<Result>
{
}

[Serializable]
[XmlType(TypeName = "Result",
         AnonymousType = true,
         Namespace = "http://windowsazure.cat.microsoft.com/samples/servicebus")]
```

```
[DataContract(Name = "Result",
              Namespace = "http://windowsazure.cat.microsoft.com/samples/servicebus")]
public class Result
{
    #region Private Fields
    private double value;
    private string error;
    #endregion

    #region Public Constructors
    public Result()
    {
        value = default(double);
        error = default(string);
    }

    public Result(double value, string error)
    {
        this.value = value;
        this.error = error;
    }
    #endregion

    #region Public Properties
    [XmlElement]
    [DataMember(Order = 1)]
    public double Value
    {
        get
        {
            return value;
        }
        set
        {
            this.value = value;
        }
    }

    [XmlElement]
    [DataMember(Order = 2)]
    public string Error
    {
```

```
        get
        {
            return error;
        }
        set
        {
            error = value;
        }
    }
    #endregion
}
```

CalculatorRequestMessage Class

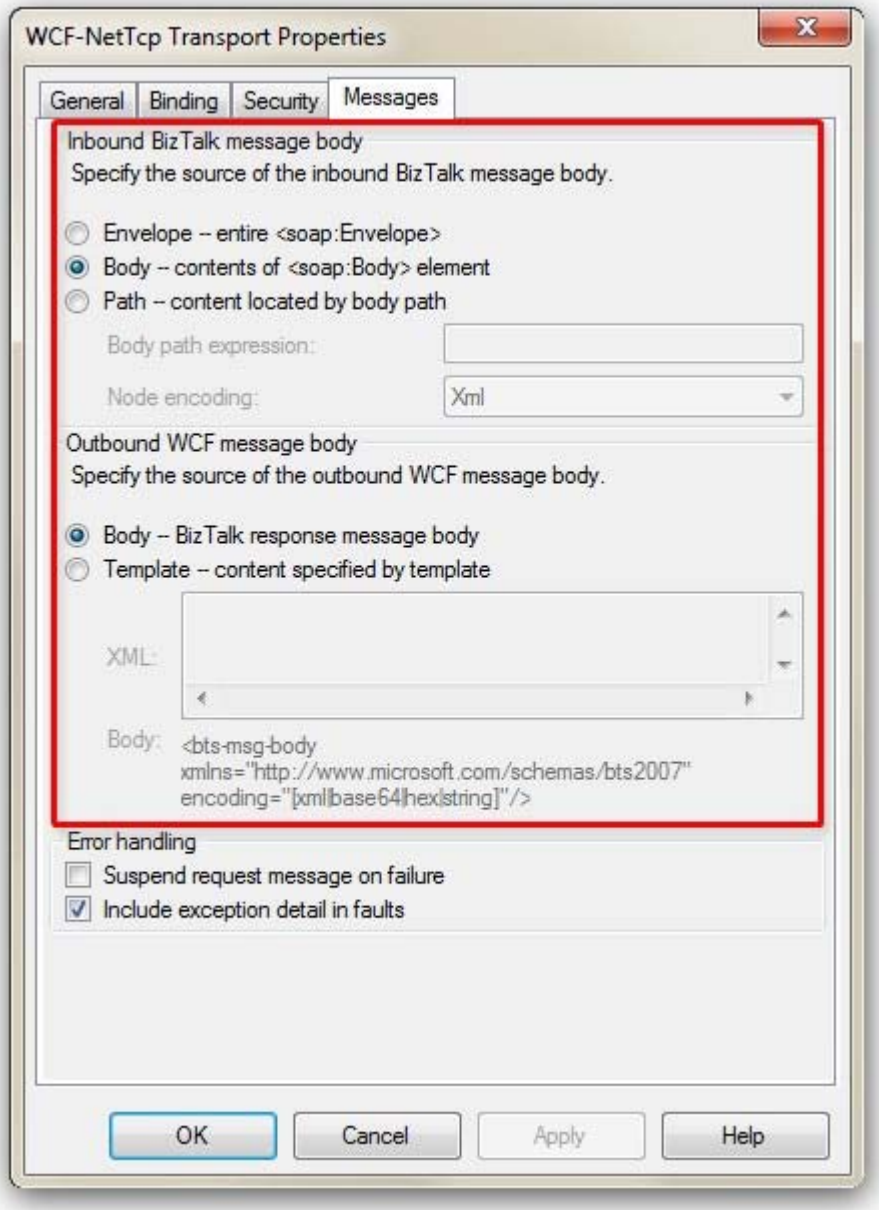
```
[MessageContract(IsWrapped=false)]
public class CalculatorRequestMessage
{
    #region Private Fields
    private CalculatorRequest calculatorRequest;
    #endregion

    #region Public Constructors
    public CalculatorRequestMessage()
    {
        this.calculatorRequest = null;
    }

    public CalculatorRequestMessage(CalculatorRequest calculatorRequest)
    {
        this.calculatorRequest = calculatorRequest;
    }
    #endregion

    #region Public Properties
    [MessageBodyMember(Namespace =
"http://windowsazure.cat.microsoft.com/samples/servicebus")]
    public CalculatorRequest CalculatorRequest
    {
        get
        {
            return this.calculatorRequest;
        }
        set
        {
```

```
            this.calculatorRequest = value;
        }
    }
    #endregion
}
```

CalculatorResponseMessage Class

```
[MessageContract(IsWrapped = false)]
public class CalculatorResponseMessage
{
    #region Private Fields
    private CalculatorResponse calculatorResponse;
    #endregion

    #region Public Constructors
    public CalculatorResponseMessage()
    {
        this.calculatorResponse = null;
    }

    public CalculatorResponseMessage(CalculatorResponse calculatorResponse)
    {
        this.calculatorResponse = calculatorResponse;
    }
    #endregion

    #region Public Properties
    [MessageBodyMember(Namespace =
"http://windowsazure.cat.microsoft.com/samples/servicebus")]
    public CalculatorResponse CalculatorResponse
    {
        get
        {
            return this.calculatorResponse;
        }
        set
        {
            this.calculatorResponse = value;
        }
    }
    #endregion
}
```

Indeed, I could have used just data contracts to model messages as message contracts add a degree of complexity. However, by assigning **false** to the IsWrapped property exposed by the MessageContractAttribute, you specify that the message body won't be contained in a wrapper element. Typically, the wrapper element of a request message is the name of the operation invoked and it's defined in the WSDL. Setting the value of the IsWrapped property to **false**, you can simply select the **Body** option in both the **Inbound BizTalk message body** and **Outbound WCF message body** sections on the **Messages** tab when configuring a WCF receive location. Otherwise you should define a **Path** in the **Inbound BizTalk message body** section to extract the payload from the inbound message, and specify a template in the **Outbound WCF message body** section to include the outgoing response message within a wrapper element.



Note

The namespace of the message and data contract classes match those defined by the XML schemas that model the request and response messages in the BizTalk Server application.

See Also

"Using Data Contracts" topic on MSDN.

"Using Message Contracts" topic on MSDN

Service Contracts

The next step was to define the ServiceContracts used by the client application to exchange messages with the Service Bus. To this purpose, I created a new project in my solution called **ServiceContracts**, and I defined two service contract interfaces used by the client application respectively to send and receive messages from the Service Bus messaging entities. Indeed, I created two different versions of the service contract used to receive response messages:

- The **ICalculatorResponse** interface is meant to be used to receive response messages from a non-sessionful queue or subscription
- The **ICalculatorResponseSessionful** interface inherits from the **ICalculatorResponse** service contract and is marked with the [ServiceContract(SessionMode = SessionMode.Required)] attribute. This service contract is meant to be used to receive response messages from a non-sessionful queue or subscription.

Note that the methods defined by all the contracts must be one-way.

ICalculatorRequest, ICalculatorResponse, ICalculatorResponseSessionful Interfaces

```
#region Using Directives
using System.ServiceModel;
using Microsoft.WindowsAzure.CAT.Samples.ServiceBus.MessageContracts;
#endregion

namespace Microsoft.WindowsAzure.CAT.Samples.ServiceBus.ServiceContracts
{
    [ServiceContract(Namespace =
"http://windowsazure.cat.microsoft.com/samples/servicebus",
                     SessionMode = SessionMode.Allowed)]
    public interface ICalculatorRequest
```

```
    {
        [OperationContract(Action = "SendRequest", IsOneWay = true)]
        [ReceiveContextEnabled(ManualControl = true)]
        void SendRequest(CalculatorRequestMessage calculatorRequestMessage);
    }

    [ServiceContract(Namespace =
"http://windowsazure.cat.microsoft.com/samples/servicebus",
                     SessionMode = SessionMode.Allowed)]
    public interface ICalculatorResponse
    {
        [OperationContract(Action = "ReceiveResponse", IsOneWay = true)]
        [ReceiveContextEnabled(ManualControl = true)]
        void ReceiveResponse(CalculatorResponseMessage calculatorResponseMessage);
    }

    [ServiceContract(Namespace =
"http://windowsazure.cat.microsoft.com/samples/servicebus",
                     SessionMode = SessionMode.Required)]
    public interface ICalculatorResponseSessionful : ICalculatorResponse
    {
    }
}
```

We are now ready to look at the code of the client application.

Client Application

Since the Windows Forms application exchanges messages with the **ServiceBusSample** BizTalk application asynchronously through Service Bus messaging entities, it acts as a client and a service application at the same time. Windows Forms uses WCF and the NetMessagingBinding to perform the following actions:

1. Send request messages to the **requestqueue**.
2. Send request messages to the **requesttopic**.
3. Receive response messages from the **responsequeue**.
4. Receive response messages from the **ItalyMilan** subscription of the **responsetopic**.

Let's start reviewing the configuration file of the client application that plays a central role in the definition of the WCF client and service endpoints used to communicate with the Service Bus.

App.Config

```
<?xml version="1.0"?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="System.ServiceModel.MessageLogging" switchValue="Warning,
ActivityTracing">
        <listeners>
          <add type="System.Diagnostics.DefaultTraceListener" name="Default">
            <filter type="" />
          </add>
          <add name="ServiceModelMessageLoggingListener">
            <filter type="" />
          </add>
        </listeners>
      </source>
    </sources>
    <sharedListeners>
      <add initializeData="C:\ServiceBusQueueClient.svclog"
           type="System.Diagnostics.XmlWriterTraceListener, System,
Version=2.0.0.0, Culture=neutral,
                 PublicKeyToken=b77a5c561934e089"
           name="ServiceModelMessageLoggingListener"
           traceOutputOptions="Timestamp">
        <filter type="" />
      </add>
    </sharedListeners>
    <trace autoflush="true" indentsize="4">
      <listeners>
        <clear/>
        <add name="LogTraceListener"
type="Microsoft.WindowsAzure.CAT.Samples.ServiceBus.Client.LogTraceListener, Client"
             initializeData="" />
      </listeners>
    </trace>
  </system.diagnostics>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/>
  </startup>
```

```
  36:    <system.serviceModel>
  37:      <diagnostics>
  38:        <messageLogging logEntireMessage="true"
  39:                        logMalformedMessages="false"
  40:                        logMessagesAtServiceLevel="true"
  41:                        logMessagesAtTransportLevel="false" />
  42:      </diagnostics>
  43:      <behaviors>
  44:        <endpointBehaviors>
  45:          <behavior name="securityBehavior">
  46:            <transportClientEndpointBehavior>
  47:              <tokenProvider>
  48:                <sharedSecret issuerName="owner"
  49:                              issuerSecret="[ISSUER SECRET RETRIEVED FROM THE AZURE
MGMT PORTAL]" />
  50:              </tokenProvider>
  51:            </transportClientEndpointBehavior>
  52:          </behavior>
  53:        </endpointBehaviors>
  54:      </behaviors>
  55:      <bindings>
  56:        <basicHttpBinding>
  57:          <binding name="basicHttpBinding"
  58:                   closeTimeout="00:10:00"
  59:                   openTimeout="00:10:00"
  60:                   receiveTimeout="00:10:00"
  61:                   sendTimeout="00:10:00">
  62:            <security mode="None">
  63:              <transport clientCredentialType="None" proxyCredentialType="None"
  64:                realm="" />
  65:              <message clientCredentialType="UserName" algorithmSuite="Default" />
  66:            </security>
  67:          </binding>
  68:        </basicHttpBinding>
  69:        <basicHttpRelayBinding>
  70:          <binding name="basicHttpRelayBinding"
  71:                   closeTimeout="00:10:00"
  72:                   openTimeout="00:10:00"
  73:                   receiveTimeout="00:10:00"
  74:                   sendTimeout="00:10:00">
  75:            <security mode="Transport"
relayClientAuthenticationType="RelayAccessToken" />
```

```
  76:          </binding>
  77:        </basicHttpRelayBinding>
  78:        <netTcpRelayBinding>
  79:          <binding name="netTcpRelayBinding"
  80:                   closeTimeout="00:10:00"
  81:                   openTimeout="00:10:00"
  82:                   receiveTimeout="00:10:00"
  83:                   sendTimeout="00:10:00">
  84:            <security mode="Transport"
relayClientAuthenticationType="RelayAccessToken" />
  85:          </binding>
  86:        </netTcpRelayBinding>
  87:        <netMessagingBinding>
  88:          <binding name="netMessagingBinding"
  89:                   sendTimeout="00:03:00"
  90:                   receiveTimeout="00:03:00"
  91:                   openTimeout="00:03:00"
  92:                   closeTimeout="00:03:00"
  93:                   sessionIdleTimeout="00:01:00"
  94:                   prefetchCount="-1">
  95:            <transportSettings batchFlushInterval="00:00:01" />
  96:          </binding>
  97:        </netMessagingBinding>
  98:      </bindings>
  99:      <client>
 100:        <!-- Invoke BizTalk via Service Bus Queue -->
 101:        <endpoint address="sb://paolosalvatori.servicebus.windows.net/requestqueue"
 102:                  behaviorConfiguration="securityBehavior"
 103:                  binding="netMessagingBinding"
 104:                  bindingConfiguration="netMessagingBinding"
 105:
contract="Microsoft.WindowsAzure.CAT.Samples.ServiceBus.ServiceContracts.ICalculatorReque
st"
 106:                  name="requestQueueClientEndpoint" />
 107:        <!-- Invoke BizTalk via Service Bus Topic -->
 108:        <endpoint address="sb://paolosalvatori.servicebus.windows.net/requesttopic"
 109:                  behaviorConfiguration="securityBehavior"
 110:                  binding="netMessagingBinding"
 111:                  bindingConfiguration="netMessagingBinding"
 112:
contract="Microsoft.WindowsAzure.CAT.Samples.ServiceBus.ServiceContracts.ICalculatorReque
st"
```

```
 113:                  name="BizTalkServiceBusTopic" />
 114:        <!-- Invoke BizTalk via Service Bus Relay Service -->
 115:        <endpoint
address="sb://paolosalvatori.servicebus.windows.net/nettcp/calculatorservice"
 116:                  behaviorConfiguration="securityBehavior"
 117:                  binding="netTcpRelayBinding"
 118:                  bindingConfiguration="netTcpRelayBinding"
 119:
contract="Microsoft.WindowsAzure.CAT.Samples.ServiceBus.ServiceContracts.ICalculatorServi
ce"
 120:                  name="netTcpRelayBindingClientEndpoint" />
 121:        <!-- Invoke BizTalk directly via WCF Receive Location -->
 122:        <endpoint
address="http://localhost/newcalculatorservice/calculatorservice.svc"
 123:                  binding="basicHttpBinding"
 124:                  bindingConfiguration="basicHttpBinding"
 125:
contract="Microsoft.WindowsAzure.CAT.Samples.ServiceBus.ServiceContracts.ICalculatorServi
ce"
 126:                  name="basicHttpBindingClientEndpoint" />
 127:      </client>
 128:      <services>
 129:        <service
name="Microsoft.WindowsAzure.CAT.Samples.ServiceBus.Service.ResponseHandlerService">
 130:          <endpoint
address="sb://paolosalvatori.servicebus.windows.net/responsequeue"
 131:                    behaviorConfiguration="securityBehavior"
 132:                    binding="netMessagingBinding"
 133:                    bindingConfiguration="netMessagingBinding"
 134:                    name="responseQueueServiceEndpoint"
 135:
contract="Microsoft.WindowsAzure.CAT.Samples.ServiceBus.ServiceContracts.ICalculatorRespo
nse" />
 136:          <endpoint
address="sb://paolosalvatori.servicebus.windows.net/responsetopic"
 137:
listenUri="sb://paolosalvatori.servicebus.windows.net/responsetopic/Subscriptions/ItalyMi
lan"
 138:                    behaviorConfiguration="securityBehavior"
 139:                    binding="netMessagingBinding"
 140:                    bindingConfiguration="netMessagingBinding"
 141:                    name="responseSubscriptionServiceEndpoint"
```

```
 142:
contract="Microsoft.WindowsAzure.CAT.Samples.ServiceBus.ServiceContracts.ICalculatorRespo
nse" />
 143:        </service>
 144:      </services>
 145:    </system.serviceModel>
 146:  </configuration>
```

Please find below a brief description of the main elements and sections of the configuration file:

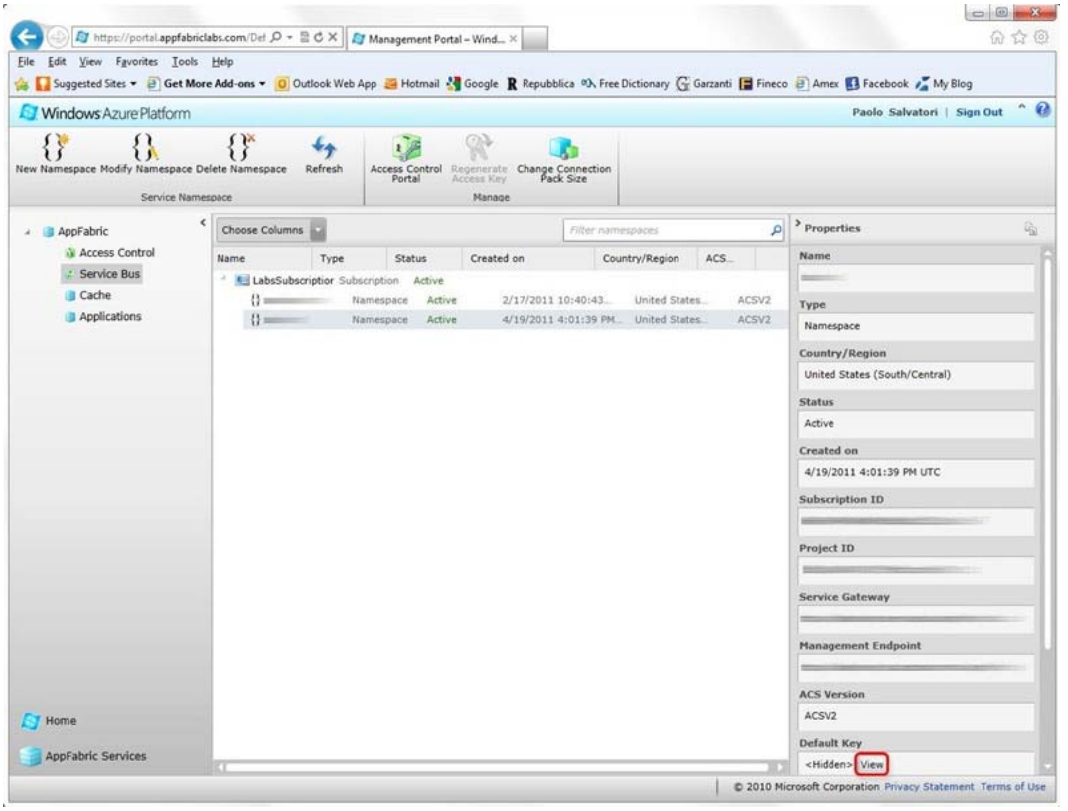
- Lines **[3-32]** define a custom trace listener called **LogTraceListener** used by the ResponseHandlerService to write response message to the log control of the Windows Forms application.
- Lines **[33-35]** the startup section specifies which versions of the common language runtime the application supports.
- Lines **[45-52]** contain the definition of the **securityBehavior** used by client and service endpoint to authenticate with the Access Control Service. In particular, the TransportClientEndpointBehavior is used to define shared secret credentials. For more information on how to retrieve credentials from the **Windows Azure Management Portal**, see the box below.
- Lines **[87-97]** contain the configuration of the NetMessagingBinding used client and service endpoints to exchange messages with the Service Bus.
- Lines **[101-106]** contain the definition of the **requestQueueClientEndpoint** used by the application to send request messages to the **requestqueue**. The **address** of the client endpoint is given by the concatenation of the URL of the service namespace and the name of the queue.
- Lines **[108-113]** contain the definition of the **requestTopicClientEndpoint** used by the application to send request messages to the **requesttopic**. The **address** of the client endpoint is given by the concatenation of the URL of the service namespace and the name of the topic
- Lines **[130-135]** contain the definition of the **responseQueueServiceEndpoint** used by the application to receive response messages from the **responsequeue**. The **address** of the service endpoint is given by the concatenation of the URL of the service namespace and the name of the queue.
- Lines **[108-113]** contain the definition of the **responseSubscriptionServiceEndpoint** used by the application to send receive response messages from the **ItalyMilan** subscription for the **responsetopic**. When you define a WCF service endpoint that uses the NetMessagingBinding to receive messages from a subscription, you have to proceed as follows (for more information on this, see the box below):
    - As value of the **address** attribute, specify the URL of the topic which the subscription belongs to. The URL of the topic is given by the concatenation of the URL of the service namespace and the name of the topic.
    - As value of the **listenUri** attribute, specify the URL of the subscription. The URL of the subscription is defined by the concatenation of the topic URL, the string **/Subscriptions/**, and the name of the subscription.
    - Assign the value **Explicit** to the **listenUriMode** attribute. The **default value** for the **listenUriMode** is **Explicit**, so this setting is optional.

Note

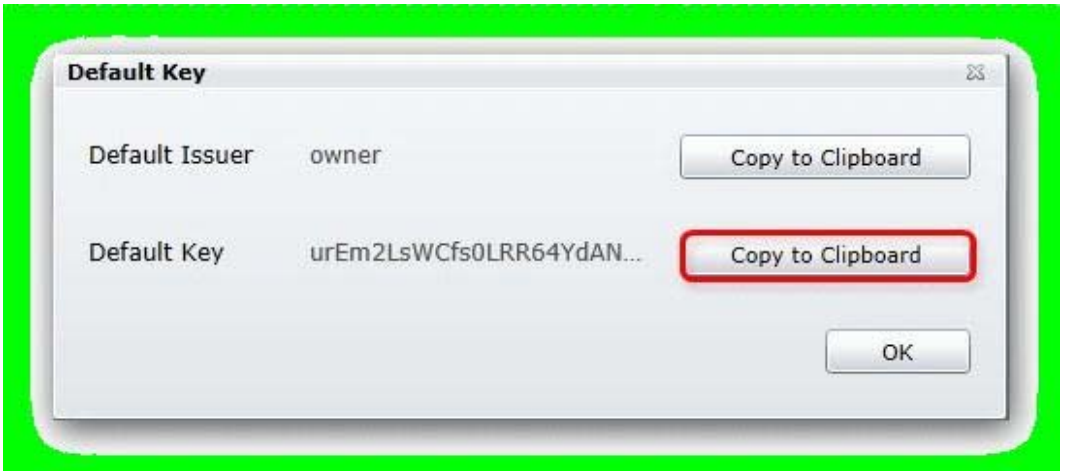
When you configure a WCF service endpoint to consume messages from a sessionful queue or subscription, the service contract needs to support sessions. Therefore, in our sample, when you configure the **responseQueueServiceEndpoint** or **responseSubscriptionServiceEndpoint** endpoints to receive, respectively, from a sessionful queue and subscription, you have to replace the **ICalculatorResponse** service contract with the sessionful **ICalculatorResponseSessionful** contract interface. For more information, see the Service Contracts section later in the article.

Note

The Service Bus supports three different types of credential schemes: **SAML**, **Shared Secret**, and **Simple Web Token**, but this version of the **Service Bus Explorer** supports only **Shared Secret** credentials. However, you can easily extend my code to support other credential schemes. You can retrieve the issuer-secret key from the **Windows Azure Management Portal** by clicking the **View** button highlighted in red in the picture below after selecting a certain namespace in the **Service Bus** section.



This opens up the modal dialog shown in the picture below where you can retrieve the key by clicking the **Copy to Clipboard** button highlighted in red.



Note

By convention, the name of the **Default Issuer** is always **owner**.

Note

When you define a WCF service endpoint that uses the NetMessagingBinding to receive messages from a subscription, if you make the mistake to assign the URL of the subscription to the address attribute of the service endpoint (as reported in configuration below), at runtime an FaultException like the following will occur:

```
The message with To 'sb://paolosalvatori.servicebus.windows.net/responsetopic' cannot be
processed at
the receiver, due to an AddressFilter mismatch at the EndpointDispatcher.
Check that the sender and receiver's EndpointAddresses agree."}
```

Wrong Configuration

```
<?xml version="1.0"?>
<configuration>
  ...
<system.serviceModel>
    ...
<services>
<service
name="Microsoft.WindowsAzure.CAT.Samples.ServiceBus.Service.ResponseHandlerService">
<endpoint
address="sb://paolosalvatori.servicebus.windows.net/responsetopic/Subscriptions/ItalyMila
n"
                  behaviorConfiguration="securityBehavior"
                  binding="netMessagingBinding"
                  bindingConfiguration="netMessagingBinding"
```

```
                    name="responseSubscriptionServiceEndpoint"

contract="Microsoft.WindowsAzure.CAT.Samples.ServiceBus.ServiceContracts.ICalculatorRespo
nse" />
</service>
</services>
</system.serviceModel>
</configuration>
```

The error is due to the fact the WS-Addressing To header of the message contains the address of the topic and not the address of the subscription:

```
<:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
xmlns:a="http://www.w3.org/2005/08/addressing">
<s:Header>
<a:Action s:mustUnderstand="1">ReceiveResponse</a:Action>
<a:MessageID>urn:uuid:64cb0b06-1622-4920-a035-c27b610cfcaf</a:MessageID>
<a:To
s:mustUnderstand="1">sb://paolosalvatori.servicebus.windows.net/responsetopic</a:To>
</s:Header>
<s:Body>... stream ...</s:Body>
</s:Envelope>
```

To correctly configure the service endpoint to receive messages from a subscription, you have to proceed as follows:

- As value of the **address** attribute, you have specify the URL of the topic which the subscription belongs to. The URL of the topic is given by the concatenation of the URL of the service namespace and the name of the topic.
- As value of the **listenUri** attribute, you have to specify the URL of the subscription. The URL of the subscription is defined by the concatenation of the topic URL, the string **/Subscriptions/** and the name of the subscription.
- Assign the value **Explicit** to the **listenUriMode** attribute. The default value for the **listenUriMode** is **Explicit**, so this setting is optional.

See the following page on MSDN for a description of the **address**, **listenUri** and **listenUriMode** attributes.

Correct Configuration

```
<?xml version="1.0"?>
<configuration>
  ...
<system.serviceModel>
    ...
<services>
```

```
<service
name="Microsoft.WindowsAzure.CAT.Samples.ServiceBus.Service.ResponseHandlerService">
<endpoint address="sb://paolosalvatori.servicebus.windows.net/responsetopic"

listenUri="sb://paolosalvatori.servicebus.windows.net/responsetopic/Subscriptions/ItalyMi
lan"
                  behaviorConfiguration="securityBehavior"
                  binding="netMessagingBinding"
                  bindingConfiguration="netMessagingBinding"
                  name="subscriptionEndpoint"

contract="Microsoft.WindowsAzure.CAT.Samples.ServiceBus.ServiceContracts.ICalculatorRespo
nse" />
</service>
</services>
</system.serviceModel>
</configuration>
```

To accomplish the same task via API, you have to properly set the value of the Address, ListenUri and ListenUriMode properties of your ServiceEndpoint instance as indicated in this note.

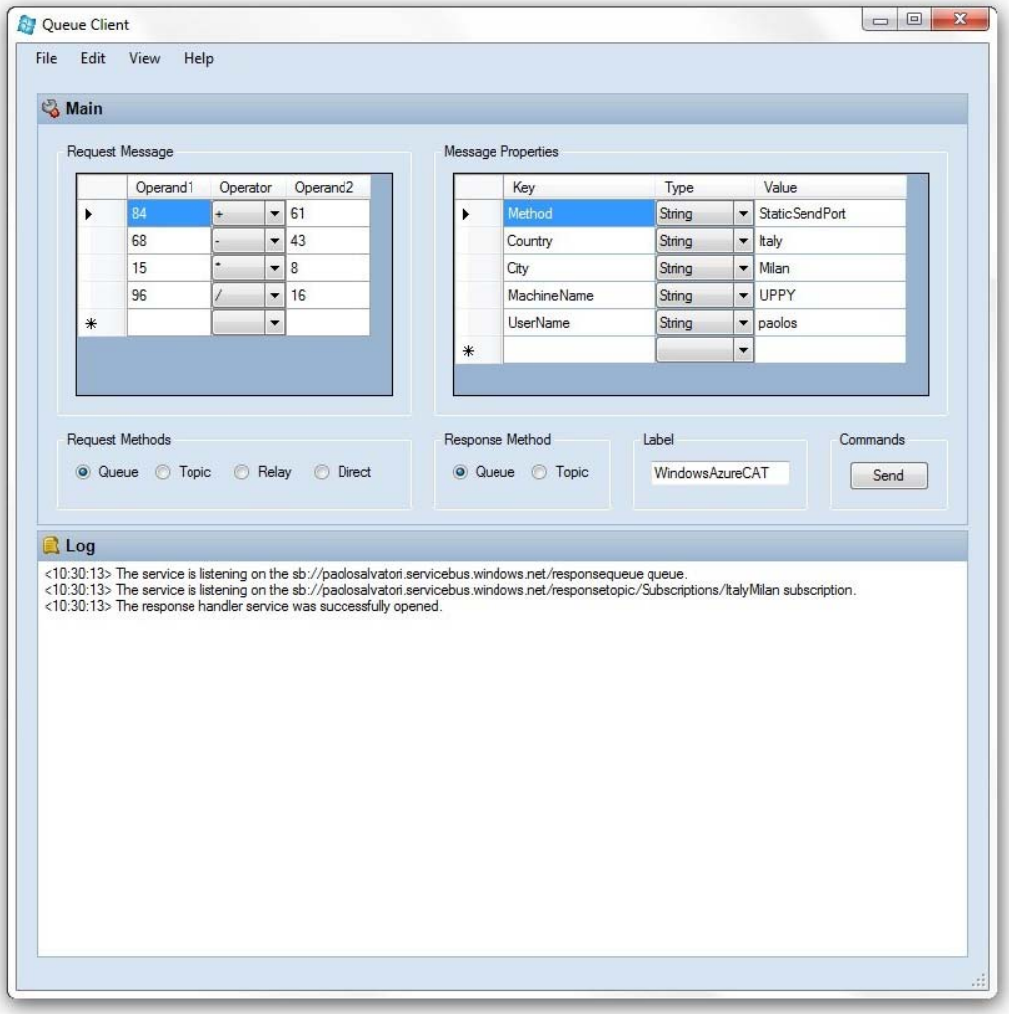
The following example shows the code used by the client application to start the **ResponseHandlerService** used to read response messages from the **responsequeue** and the **ItalyMilan** subscription of the **responsetopic**. We'll examine the code of the service in the next section.

StartServiceHost Method

```
private void StartServiceHost()
{
    try
    {
        // Creating the service host object as defined in config
        var serviceHost = new ServiceHost(typeof(ResponseHandlerService));

        // Add ErrorServiceBehavior for handling errors encounter by servicehost during
execution.
        serviceHost.Description.Behaviors.Add(new ErrorServiceBehavior());



        foreach (var serviceEndpoint in serviceHost.Description.Endpoints)
        {
            if (serviceEndpoint.Name == "responseQueueServiceEndpoint")
            {
                responseQueueUri = serviceEndpoint.Address.Uri.AbsoluteUri;
```

```
                WriteToLog(string.Format(ServiceHostListeningOnQueue,
                                         serviceEndpoint.Address.Uri.AbsoluteUri));
            }
            if (serviceEndpoint.Name == "responseSubscriptionServiceEndpoint")
            {
                responseTopicUri = serviceEndpoint.Address.Uri.AbsoluteUri;
                WriteToLog(string.Format(ServiceHostListeningOnSubscription,
                                             serviceEndpoint.ListenUri.AbsoluteUri));
            }
        }

        // Start the service host
        serviceHost.Open();
        WriteToLog(ServiceHostSuccessfullyOpened);
    }
    catch (Exception ex)
    {
        mainForm.HandleException(ex);
    }
}
```

The following picture shows the user interface of the client application.



The radio buttons contained in the **Request Method** group allow you to choose whether to send the request message to the **requestqueue** or the **requesttopic**, whereas the radio buttons contained in the **Response Method** group allows you to select whether to receive the response from the **responsequeue** or from the **ItalyMilan** subscription of the **responsetopic**. To communicate the selection to the underlying BizTalk application, the application uses a BrokeredMessageProperty object to assign the value of the **responseQueueUri** or **responseTopicUri** private fields to the ReplyTo property. The following table contains the code of the method used by the client application. For your convenience, comments have been added to the code to facilitate its understanding.

SendRequestMessageUsingWCF Method

```
private void SendRequestMessageUsingWCF(string endpointConfigurationName)
{
    try
    {
        if (string.IsNullOrEmpty(endpointConfigurationName))
```

```
        {
            WriteToLog(EndpointConfigurationNameCannotBeNull);
            return;
        }

        // Set the wait cursor
        Cursor = Cursors.WaitCursor;

        // Make sure that the request message contains at least an operation
        if (operationList == null ||
            operationList.Count == 0)
        {
            WriteToLog(OperationListCannotBeNull);
            return;
        }

        // Create warning collection
        var warningCollection = new List<string>();

        // Create request message
        var calculatorRequest = new CalculatorRequest(operationList);
        var calculatorRequestMessage = new CalculatorRequestMessage(calculatorRequest);

        // Create the channel factory for the currennt client endpoint
        // and cache it in the channelFactoryDictionary
        if (!channelFactoryDictionary.ContainsKey(endpointConfigurationName))
        {
            channelFactoryDictionary[endpointConfigurationName] =
                new ChannelFactory<ICalculatorRequest>(endpointConfigurationName);
        }

        // Create the channel for the currennt client endpoint
        // and cache it in the channelDictionary
        if (!channelDictionary.ContainsKey(endpointConfigurationName))
        {
            channelDictionary[endpointConfigurationName] =
                channelFactoryDictionary[endpointConfigurationName].CreateChannel();
        }

        // Use the OperationContextScope to create a block within which to access the
current OperationScope
```

```
        using (new
OperationContextScope((IContextChannel)channelDictionary[endpointConfigurationName]))
        {
            // Create a new BrokeredMessageProperty object
            var brokeredMessageProperty = new BrokeredMessageProperty();

            // Read the user defined properties and add them to the
            // Properties collection of the BrokeredMessageProperty object
            foreach (var e in propertiesBindingSource.Cast<PropertyInfo>())
            {
                try
                {
                    e.Key = e.Key.Trim();
                    if (e.Type != StringType && e.Value == null)
                    {
                        warningCollection.Add(string.Format(CultureInfo.CurrentUICulture,
                                                            PropertyValueCannotBeNull,
e.Key));
                    }
                    else
                    {
                        if (brokeredMessageProperty.Properties.ContainsKey(e.Key))
                        {
                            brokeredMessageProperty.Properties[e.Key] =
                                ConversionHelper.MapStringTypeToCLRType(e.Type, e.Value);
                        }
                        else
                        {
                            brokeredMessageProperty.Properties.Add(e.Key,
                                ConversionHelper.MapStringTypeToCLRType(e.Type,
e.Value));
                        }
                    }
                }
                catch (Exception ex)
                {
                    warningCollection.Add(string.Format(CultureInfo.CurrentUICulture,
                        PropertyConversionError, e.Key, ex.Message));
                }
            }

            // if the warning collection contains at least one or more items,
```

```
            // write them to the log and return immediately
            StringBuilder builder;
            if (warningCollection.Count > 0)
            {
                builder = new StringBuilder(WarningHeader);
                var warnings = warningCollection.ToArray<string>();
                for (var i = 0; i < warningCollection.Count; i++)
                {
                    builder.AppendFormat(WarningFormat, warnings[i]);
                }
                mainForm.WriteToLog(builder.ToString());
                return;
            }

            // Set the BrokeredMessageProperty properties
            brokeredMessageProperty.Label = txtLabel.Text;
            brokeredMessageProperty.MessageId = Guid.NewGuid().ToString();
            brokeredMessageProperty.SessionId = sessionId;
            brokeredMessageProperty.ReplyToSessionId = sessionId;
            brokeredMessageProperty.ReplyTo = responseQueueRadioButton.Checked
                                              ? responseQueueUri
                                              : responseTopicUri;

OperationContext.Current.OutgoingMessageProperties.Add(BrokeredMessageProperty.Name,

brokeredMessageProperty);

            // Send the request message to the requestqueue or requesttopic
            var stopwatch = new Stopwatch();
            try
            {
                stopwatch.Start();

channelDictionary[endpointConfigurationName].SendRequest(calculatorRequestMessage);
            }
            catch (CommunicationException ex)
            {
                if (channelFactoryDictionary[endpointConfigurationName] != null)
                {
                    channelFactoryDictionary[endpointConfigurationName].Abort();
                    channelFactoryDictionary.Remove(endpointConfigurationName);
                    channelDictionary.Remove(endpointConfigurationName);
```

```
                }
                HandleException(ex);
            }
            catch (Exception ex)
            {
                if (channelFactoryDictionary[endpointConfigurationName] != null)
                {
                    channelFactoryDictionary[endpointConfigurationName].Abort();
                    channelFactoryDictionary.Remove(endpointConfigurationName);
                    channelDictionary.Remove(endpointConfigurationName);
                }
                HandleException(ex);
            }
            finally
            {
                stopwatch.Stop();
            }
            // Log the request message and its properties
            builder = new StringBuilder();
            builder.AppendLine(string.Format(CultureInfo.CurrentCulture,
                    MessageSuccessfullySent,

channelFactoryDictionary[endpointConfigurationName].Endpoint.Address.Uri.AbsoluteUri,
                    brokeredMessageProperty.MessageId,
                    brokeredMessageProperty.SessionId,
                    brokeredMessageProperty.Label,
                    stopwatch.ElapsedMilliseconds));
            builder.AppendLine(PayloadFormat);
            for (var i = 0; i < calculatorRequest.Operations.Count; i++)
            {
                builder.AppendLine(string.Format(RequestFormat,
                                                 i + 1,

calculatorRequest.Operations[i].Operand1,

calculatorRequest.Operations[i].Operator,

calculatorRequest.Operations[i].Operand2));
            }
            builder.AppendLine(SentMessagePropertiesHeader);
            foreach (var p in brokeredMessageProperty.Properties)
            {
```

```
                builder.AppendLine(string.Format(MessagePropertyFormat,
                                                 p.Key,
                                                 p.Value));
            }
            var traceMessage = builder.ToString();
            WriteToLog(traceMessage.Substring(0, traceMessage.Length - 1));
        }
    }
    catch (Exception ex)
    {
        // Handle the exception
        HandleException(ex);
    }
    finally
    {
        // Restoire the defaulf cursor
        Cursor = Cursors.Default;
    }
}
```

Response Handler Service

The following example contains the code of the WCF service used by the client application to retrieve and log response messages from the **responsequeue** and **ItalyMilan** subscription of the **responsetopic**. To accomplish this result, the service exposes two different endpoints each of which uses the NetMessagingBinding and receives messages from one of the two queues. Indeed, each subscription can be seen as a virtual queue getting copies of messages published to the topic they belong to. The example below shows the code of the **ResponseHandlerService** class. As you can notice, the service retrieves the BrokeredMessageProperty from the Properties collection of the incoming WCF message and uses this object to access to the properties of the response message. Since in the **ICalculatorResponse** service contract the **ReceiveResponse** method is decorated with the [ReceiveContextEnabled(ManualControl = true)] attribute, the service method must explicitly signal the receive. This signaling requires the service to explicitly invoke the ReceiveContext.Complete method to commit the receive operation. In fact, as we said at the beginning of the article, when the ManualControl property is set to true, the message received from the channel is delivered to the service operation with a lock for the message. The service implementation must call either Complete(TimeSpan) or Abandon(TimeSpan) to signal the receive completion of the message. Failure to call either of these methods causes the lock to

be held on the message until the lock timeout interval elapses. Once the lock is released (either through a call to Abandon(TimeSpan)) or lock timeout) the message is redispatched from the channel to the service. Calling Complete(TimeSpan) marks the message as successfully received.

ResponseHandlerService Class

```
[ServiceBehavior(Namespace = "http://windowsazure.cat.microsoft.com/samples/servicebus")]
public class ResponseHandlerService : ICalculatorResponseSessionful
{
    #region Private Constants
    //***************************
    // Formats
    //***************************
    private const string MessageSuccessfullyReceived = "Response Message Received:\n -
EndpointUrl:[{0}]" +
                                                        "\n - CorrelationId=[{1}]\n -
SessionId=[{2}]\n - Label=[{3}]";
    private const string ReceivedMessagePropertiesHeader = "Properties:";
    private const string PayloadFormat = "Payload:";
    private const string StatusFormat = " - Status=[{0}]";
    private const string ResultFormat = " - Result[{0}]: Value=[{1}] Error=[{2}]";
    private const string MessagePropertyFormat = " - Key=[{0}] Value=[{1}]";

    //***************************
    // Constants
    //***************************
    private const string Empty = "EMPTY";
    #endregion

    #region Public Operations
    [OperationBehavior]
    public void ReceiveResponse(CalculatorResponseMessage calculatorResponseMessage)
    {
        try
        {
            if (calculatorResponseMessage != null &&
                calculatorResponseMessage.CalculatorResponse != null)
            {
                // Initialize calculatorResponse var
                var calculatorResponse = calculatorResponseMessage.CalculatorResponse;

                // Get the message properties
```

```
                var incomingProperties =
OperationContext.Current.IncomingMessageProperties;
                var brokeredMessageProperty =
incomingProperties[BrokeredMessageProperty.Name] as
                                            BrokeredMessageProperty;

                // Trace the response message
                var builder = new StringBuilder();
                if (brokeredMessageProperty != null)
                    builder.AppendLine(string.Format(MessageSuccessfullyReceived,

OperationContext.Current.Channel.LocalAddress.Uri.AbsoluteUri,

brokeredMessageProperty.CorrelationId ?? Empty,
                                                     brokeredMessageProperty.SessionId ??
Empty,
                                                     brokeredMessageProperty.Label ??
Empty));
                builder.AppendLine(PayloadFormat);
                builder.AppendLine(string.Format(StatusFormat,
                                                 calculatorResponse.Status));
                if (calculatorResponse.Results != null &&
                    calculatorResponse.Results.Count > 0)
                {
                    for (int i = 0; i < calculatorResponse.Results.Count; i++)
                    {
                        builder.AppendLine(string.Format(ResultFormat,
                                                         i + 1,

calculatorResponse.Results[i].Value,

calculatorResponse.Results[i].Error));
                    }
                }
                builder.AppendLine(ReceivedMessagePropertiesHeader);
                if (brokeredMessageProperty != null)
                {
                    foreach (var property in brokeredMessageProperty.Properties)
                    {
                        builder.AppendLine(string.Format(MessagePropertyFormat,
                                                         property.Key,
                                                         property.Value));
```

```
                }
            }
            var traceMessage = builder.ToString();
            Trace.WriteLine(traceMessage.Substring(0, traceMessage.Length - 1));

            //Complete the Message
            ReceiveContext receiveContext;
            if (ReceiveContext.TryGet(incomingProperties, out receiveContext))
            {
                receiveContext.Complete(TimeSpan.FromSeconds(10.0d));
            }
            else
            {
                throw new InvalidOperationException("An exception occurred.");
            }
        }
    }
    catch (Exception ex)
    {
        Trace.WriteLine(ex.Message);
    }
}
#endregion
}
```

ListenUri Endpoint Behavior

As you will see ahead in the article, it's easy to define a WCF-Custom receive location to read messages from a Service Bus queue or from a subscription. Nevertheless, in the previous section we have seen that when defining a WCF service endpoint to consume messages from a subscription, you have to you have to specify the URL of the topic as the address of the service endpoint and the URL of the subscription as its **listenUri**. Now, when configuring a WCF receive location using the WCF-Custom adapter, there is a textbox for specifying the service endpoint address on the **General** tab of the configuration dialog, however there's no field to specify a value of the **listenUri** and **ListenUriMode** properties.

For this reason, I decided to create a custom endpoint behavior that at runtime can set these values for a WCF-Custom receive location. The first attempt was to use the AddBindingParameters and ApplyDispatchBehavior methods exposed by the custom endpoint behavior to set the **listenUri** and **ListenUriMode** of the service endpoint passed as a parameter to both methods. However, this technique didn't work as expected.

Finally I solved the problem in the following way: I created a binding extension element class called **ListenUriBehaviorExtensionElement** to register the custom endpoint behavior in the

machine.config file. This component exposes a property called **listenUri** that allows a user to specify the URL of a subscription when configuring a WCF-Custom receive location.

At runtime, the **ListenUriBehaviorExtensionElement** component creates an instance of the **ListenUriEndpointBehavior** class. The AddBindingParameters method of the custom endpoint behavior replaces the original binding with a CustomBinding that contains the same binding elements and injects an instance of the **ListenUriBindingElement** at the top of the binding. This way, at runtime, the custom binding will be the first to execute. Finally, the BuildChannelListener method of the **ListenUriBindingElement** assigns the URL specified in the configuration of the receive location to the ListenUriBaseAddress property of the BindingContext and sets the value of its ListenUriMode property to Explicit. For you convenience, I included the code for the three classes below. Later in the article I'll show you how to use this component when defining a WCF-Custom receive location that receives messages from a Service Bus subscription.

ListenUriBehaviorExtensionElement Class

```
public class ListenUriBehaviorExtensionElement : IEndpointBehavior
{
#region Private Constants
    //***************************
    // Constants
    //***************************

    private const string ListenUriName = "listenUri";
    private const string IsTrackingEnabledName = "isTrackingEnabled";
    private const string ListenUriDescription = "Gets or sets the URI at which the
service endpoint listens.";
    private const string IsTrackingEnabledDescription = "Gets or sets a value indicating
whether tracking is enabled.";
    #endregion

    #region BehaviorExtensionElement Members
    //***************************
    // Protected Methods
    //***************************

    /// <summary>
    /// Creates a behavior extension based on the current configuration settings.
    /// </summary>
    /// <returns>The behavior extension.</returns>
    protected override object CreateBehavior()
    {
        return new ListenUriEndpointBehavior(ListenUri, IsTrackingEnabled);
    }
```

```
    /// <summary>
    /// Gets the type of behavior.
    /// </summary>
    public override Type BehaviorType
    {
        get
        {
            return typeof(ListenUriEndpointBehavior);
        }
    }
    #endregion

    #region Public Properties
    /// <summary>
    /// Gets or sets the URI at which the service endpoint listens.
    /// </summary>
    [ConfigurationProperty(ListenUriName, IsRequired = true)]
    [SettingsDescription(ListenUriDescription)]
    public string ListenUri
    {
        get
        {
            return (string)base[ListenUriName];
        }
        set
        {
            base[ListenUriName] = value;
        }
    }

    /// <summary>
    /// Gets or sets a value indicating whether the message inspector is enabled.
    /// </summary>
    [ConfigurationProperty(IsTrackingEnabledName, DefaultValue = true, IsRequired =
false)]
    [SettingsDescription(IsTrackingEnabledDescription)]
    public bool IsTrackingEnabled
    {
        get
        {
            return (bool)base[IsTrackingEnabledName];
        }
```

```
        set
        {
            base[IsTrackingEnabledName] = value;
        }
    }
    #endregion
}
```

ListenUriEndpointBehavior Class

```
public class ListenUriEndpointBehavior : IEndpointBehavior
{
    #region Private Constants
    //***************************
    // Constants
    //***************************

    private const string ListerUriMessageFormat = "[ListenUriEndpointBehavior] ListenUri
= [{0}].";
    #endregion

    #region Public Constructors
    private readonly string listenUri;
    private readonly bool isTrackingEnabled;
    #endregion

    #region Public Constructors
    /// <summary>
    /// Initializes a new instance of the ListenUriEndpointBehavior class.
    /// </summary>
    /// <param name="listenUri">The URI at which the service endpoint listens</param>
    /// <param name="isTrackingEnabled">A boolean value indicating whether tracking is
enabled</param>
    public ListenUriEndpointBehavior(string listenUri, bool isTrackingEnabled)
    {
        this.listenUri = listenUri;
        this.isTrackingEnabled = isTrackingEnabled;
    }
    #endregion

    #region IEndpointBehavior Members
    /// <summary>
    /// Implement to pass data at runtime to bindings to support custom behavior.
    /// </summary>
```

```
    /// <param name="endpoint">The endpoint to modify.</param>
    /// <param name="bindingParameters">The objects that binding elements require to
support the behavior.</param>
    void IEndpointBehavior.AddBindingParameters(ServiceEndpoint endpoint,
BindingParameterCollection bindingParameters)
    {
        if (endpoint != null &&
            !String.IsNullOrEmpty(listenUri))
        {
            // Read the binding elements from the original binding
            var bindingElementCollection = endpoint.Binding.CreateBindingElements();
            // Create an array of binding elements
            var bindingElementArray = new BindingElement[bindingElementCollection.Count +
1];
            // Add an instance of the ListenUriBindingElement as first binding element of
the array
            bindingElementArray[0] = new ListenUriBindingElement(listenUri);
            // Copy the binding elements of the original binding to the array
            bindingElementCollection.CopyTo(bindingElementArray, 1);
            // Create a custom binding with the same binding elements as the original
            // binding with the addition of the custom binding as first item
            var customBinding = new CustomBinding(bindingElementArray)
                                    {
                                        CloseTimeout = endpoint.Binding.CloseTimeout,
                                        OpenTimeout = endpoint.Binding.OpenTimeout,
                                        ReceiveTimeout = endpoint.Binding.ReceiveTimeout,
                                        SendTimeout = endpoint.Binding.SendTimeout,
                                        Name = endpoint.Binding.Name,
                                        Namespace = endpoint.Binding.Namespace
                                    };
            //Replace the original binding with the newly created binding
            endpoint.Binding = customBinding;
            Trace.WriteLineIf(isTrackingEnabled,
                              string.Format(ListerUriMessageFormat,
                                            listenUri));
        }
    }

    /// <summary>
    /// Implements a modification or extension of the client across an endpoint.
    /// </summary>
    /// <param name="endpoint">The endpoint that is to be customized.</param>
```

```
    /// <param name="clientRuntime">The client runtime to be customized.</param>
    void IEndpointBehavior.ApplyClientBehavior(ServiceEndpoint endpoint, ClientRuntime
clientRuntime)
    {
    }

    /// <summary>
    /// Implements a modification or extension of the service across an endpoint.
    /// </summary>
    /// <param name="endpoint">The endpoint that exposes the contract.</param>
    /// <param name="endpointDispatcher">The endpoint dispatcher to be modified or
extended.</param>
    void IEndpointBehavior.ApplyDispatchBehavior(ServiceEndpoint endpoint,
EndpointDispatcher endpointDispatcher)
    {
    }

    /// <summary>
    /// Implement to confirm that the endpoint meets some intended criteria.
    /// </summary>
    /// <param name="endpoint">The endpoint to validate.</param>
    void IEndpointBehavior.Validate(ServiceEndpoint endpoint)
    {
    }
    #endregion
}
```

ListenUriBindingElement Class

```
public class ListenUriBindingElement : BindingElement
{
    #region Private Fields
    private readonly string listenUri;
    #endregion

    #region Public Constructor
    /// <summary>
    /// Initializes a new instance of the ListenUriBindingElement class.
    /// </summary>
    /// <param name="listenUri">A BindingElement object that is a deep clone of the
original.</param>
    public ListenUriBindingElement(string listenUri)
    {
        this.listenUri = listenUri;
```

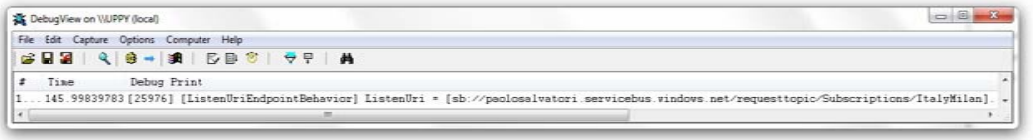
```
    }
    #endregion

    #region BindingElement Members
    /// <summary>
    /// returns a copy of the binding element object.
    /// </summary>
    /// <returns></returns>
    public override BindingElement Clone()
    {
        return new ListenUriBindingElement(listenUri);
    }

    /// <summary>
    /// Returns a typed object requested, if present, from the appropriate layer in the
binding stack.
    /// </summary>
    /// <typeparam name="T">The typed object for which the method is
querying.</typeparam>
    /// <param name="context">The BindingContext for the binding element.</param>
    /// <returns>The typed object T requested if it is present or nullif it is not
present.</returns>
    public override T GetProperty<T>(BindingContext context)
    {
        return context.GetInnerProperty<T>();
    }

    /// <summary>
    /// Returns a value that indicates whether the binding element can build a
    /// channel factory for a specific type of channel.
    /// </summary>
    /// <typeparam name="TChannel">The type of channel the channel factory
produces.</typeparam>
    /// <param name="context">The BindingContext that provides context for the binding
element.</param>
    /// <returns>true if the IChannelFactory<TChannel/>of type TChannel can be built by
    ///          the binding element; otherwise, false.</returns>
    public override bool CanBuildChannelFactory<TChannel>(BindingContext context)
    {
        return false;
    }
```

```
    /// <summary>
    /// Initializes a channel factory for producing channels of a specified type from the
binding context.
    /// </summary>
    /// <typeparam name="TChannel">The type of channel the factory builds.</typeparam>
    /// <param name="context">The BindingContext that provides context for the binding
element.</param>
    /// <returns>The IChannelFactory<TChannel/>of type TChannel initialized from the
context. </returns>
    public override IChannelFactory<TChannel>
BuildChannelFactory<TChannel>(BindingContext context)
    {
        throw new NotSupportedException();
    }

    /// <summary>
    /// Returns a value that indicates whether the binding element can build a channel
    /// listener for a specific type of channel.
    /// </summary>
    /// <typeparam name="TChannel">The type of channel listener to build.</typeparam>
    /// <param name="context">The BindingContext for the binding element.</param>
    /// <returns>true if a channel listener of the specified type can be built;
    ///          otherwise, false. The default is false. </returns>
    public override bool CanBuildChannelListener<TChannel>(BindingContext context)
    {
        return context.CanBuildInnerChannelListener<TChannel>();
    }

    /// <summary>
    /// Initializes a channel listener for producing channels of a specified type from
the binding context.
    /// </summary>
    /// <typeparam name="TChannel">The type of channel that the listener is built to
accept.</typeparam>
    /// <param name="context">The BindingContext for the binding element.</param>
    /// <returns>The IChannelListener<TChannel/>of type IChannel initialized from the
context.</returns>
    public override IChannelListener<TChannel>
BuildChannelListener<TChannel>(BindingContext context)
    {
        if (!string.IsNullOrEmpty(listenUri))
        {
```

```
            context.ListenUriBaseAddress = new Uri(listenUri);
            context.ListenUriMode = ListenUriMode.Explicit;
        }
        return context.BuildInnerChannelListener<TChannel>();
    }
    #endregion
}
```

Component Tracking

You can enable the component tracking and use DebugView, as shown in the picture below, to monitor its runtime behavior.



SessionChannel Endpoint Behavior

In the first part of the article we have seen that to receive messages from a sessionful queue or subscription, a WCF service needs to implement a session-aware contract interface like **IOrderServiceSessionful** in the following code snippet:

```
// ServiceBus does not support IOutputSessionChannel.
// All senders sending messages to sessionful queue must use a contract which does not
enforce SessionMode.Required.
// Sessionful messages are sent by setting the SessionId property of the
BrokeredMessageProperty object.
[ServiceContract]
public interface IOrderService
{
    [OperationContract(IsOneWay = true)]
    [ReceiveContextEnabled(ManualControl = true)]
    void ReceiveOrder(Order order);
}

// ServiceBus supports both IInputChannel and IInputSessionChannel.
```

```
// A sessionful service listening to a sessionful queue must have SessionMode.Required in
its contract.
[ServiceContract(SessionMode = SessionMode.Required)]
public interface IOrderServiceSessionful : IOrderService
{
}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession, ConcurrencyMode =
ConcurrencyMode.Single)]
public class OrderService : IOrderServiceSessionful
{
    [OperationBehavior]
    public void ReceiveOrder(Order order)
    {
        ...
    }
}
```

In the section on BizTalk Server WCF adapters we have noticed that WCF receive locations are implemented by a singleton instance of a WCF service class called BizTalkServiceInstance which implements multiple untyped, generic service contracts.

- IOneWayAsync
- IOneWayAsyncTxn
- ITwoWayAsync
- ITwoWayAsyncVoid
- ITwoWayAsyncVoidTxn

All of these interfaces are marked with the following attribute:

```
[ServiceContract(Namespace = "http://www.microsoft.com/biztalk/2006/r2/wcf-adapter",
                 SessionMode = SessionMode.Allowed)]
```

In particular, the SessionMode property of the ServiceContract attribute is set to SessionMode.Allowed. This setting specifies that the contract supports sessions if the incoming binding supports them. In other words, all the service contracts implemented by the BizTalkServiceInstance support, but not require, a session.

When the following conditions are satisfied, the WCF runtime always selects the IInputChannel over the IInputSessionChannel when it selects the channel to use:

- The service contract exposed by a service endpoint allows, but not requires sessions (SessionMode = SessionMode.Allowed).
- The binding used by the service endpoint supports both sessionless and session-based communication (like the NetMessagingBinding).

This rule implies that when you define a WCF-Custom receive location to consume messages from a sessionful queue or subscription, when you enable it, you will see an error like the following in the **Application Log**:

```
The adapter "WCF-Custom" raised an error message. Details
"System.InvalidOperationException: It is not
possible for an entity that requires sessions to create a non-sessionful message
receiver..
TrackingId:9163a41f-d792-406d-acbe-eb93ab2defb8_1_1,TimeStamp:10/3/2011 12:39:02 PM --->
System.ServiceModel.FaultException`1[System.ServiceModel.ExceptionDetail]: It is not
possible for an entity
that requires sessions to create a non-sessionful message receiver..
TrackingId:9163a41f-d792-406d-acbe-eb93ab2defb8_1_1,TimeStamp:10/3/2011 12:39:02 PM
```

To avoid this problem, when we control the definition of the service contract, we can simply mark its ServiceContract attribute as requiring sessions. Unfortunately, we cannot change the definition of the service contracts implemented by the BizTalkServiceInstance class. So, how can we force a WCF receive location to use an IInputSessionChannel instead of an IInputChannel when consuming messages from a sessionful queue or subscription? Even in this case I solved the problem by using a WCF extension.

I created a binding extension element class called **SessionChannelBehaviorExtensionElement** to register the custom endpoint behavior in the **machine.config**. At runtime, this component creates an instance of the **SessionChannelEndpointBehavior** class. The AddBindingParameters method of the custom endpoint behavior replaces the original binding (in our case the NetMessagingBinding) with a CustomBinding that contains the same binding elements and injects an instance of the **SessionChannelBindingElement** at the top of the new binding. This way, at runtime, the custom binding will be the first to execute. Finally, I customized the BindingElement.CanBuildChannelListener<TChannel> method to return false when the type of channel is IInputChannel. In a nutshell, this component forces the WCF runtime to skip the IInputChannel. Hence, you can use it in a WCF-Custom receive location to force the WCF adapter to use the **IInputSessionChannel**. For your convenience, I included the code of the three classes below. Later in the article I'll show you how to use this component when defining a WCF-Custom receive location that consumes messages from a sessionful queue or subscription.

SessionChannelBehaviorExtensionElement Class

```
public class SessionChannelBehaviorExtensionElement : BehaviorExtensionElement
{
    #region Private Constants
    //***************************
    // Constants
    //***************************

    private const string IsTrackingEnabledName = "isTrackingEnabled";
    private const string IsTrackingEnabledDescription =
                                            "Gets or sets a value indicating whether
tracking is enabled.";
    #endregion
```

```
    #region BehaviorExtensionElement Members
    //****************************
    // Protected Methods
    //****************************

    /// <summary>
    /// Creates a behavior extension based on the current configuration settings.
    /// </summary>
    /// <returns>The behavior extension.</returns>
    protected override object CreateBehavior()
    {
        return new SessionChannelEndpointBehavior(IsTrackingEnabled);
    }

    /// <summary>
    /// Gets the type of behavior.
    /// </summary>
    public override Type BehaviorType
    {
        get
        {
            return typeof(SessionChannelEndpointBehavior);
        }
    }
    #endregion

    #region Public Properties
    /// <summary>
    /// Gets or sets a value indicating whether the message inspector is enabled.
    /// </summary>
    [ConfigurationProperty(IsTrackingEnabledName, DefaultValue = true, IsRequired =
false)]
    [SettingsDescription(IsTrackingEnabledDescription)]
    public bool IsTrackingEnabled
    {
        get
        {
            return (bool)base[IsTrackingEnabledName];
        }
        set
        {
            base[IsTrackingEnabledName] = value;
```

```
        }
    }
    #endregion
}
```

SessionChannelBindingElement Class

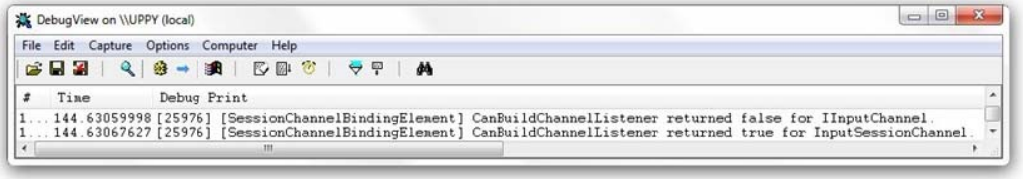
```
public class SessionChannelBindingElement : BindingElement
{
    #region Private Constants
    //***************************
    // Constants
    //***************************

    private const string CanBuildIInputSessionChannel =
                        "[SessionChannelBindingElement] CanBuildChannelListener returned
true for InputSessionChannel.";
    private const string CannotBuildIInputChannel =
                        "[SessionChannelBindingElement] CanBuildChannelListener returned
false for IInputChannel.";
    #endregion

    #region Private Fields
    private readonly bool isTrackingEnabled;
    #endregion

    #region Public Constructor
    /// <summary>
    /// Initializes a new instance of the SessionChannelBindingElement class.
    /// </summary>
    /// <param name="isTrackingEnabled">A boolean value indicating whether tracking is
enabled</param>
    public SessionChannelBindingElement(bool isTrackingEnabled)
    {
        this.isTrackingEnabled = isTrackingEnabled;
    }
    #endregion

    #region BindingElement Members
    /// <summary>
    /// returns a copy of the binding element object.
    /// </summary>
    /// <returns></returns>
    public override BindingElement Clone()
```

```
    {
        return new SessionChannelBindingElement(isTrackingEnabled);
    }

    /// <summary>
    /// Returns a typed object requested, if present, from the appropriate layer in the
binding stack.
    /// </summary>
    /// <typeparam name="T">The typed object for which the method is
querying.</typeparam>
    /// <param name="context">The BindingContext for the binding element.</param>
    /// <returns>The typed object T requested if it is present or nullif it is not
present.</returns>
    public override T GetProperty<T>(BindingContext context)
    {
        return context.GetInnerProperty<T>();
    }

    /// <summary>
    /// Returns a value that indicates whether the binding element can build
    /// a channel factory for a specific type of channel.
    /// </summary>
    /// <typeparam name="TChannel">The type of channel the channel factory
produces.</typeparam>
    /// <param name="context">The BindingContext that provides context for the binding
element.</param>
    /// <returns>true if the IChannelFactory<TChannel/>of type TChannel can be built by
    ///          the binding element; otherwise, false.</returns>
    public override bool CanBuildChannelFactory<TChannel>(BindingContext context)
    {
        return false;
    }

    /// <summary>
    /// Initializes a channel factory for producing channels of a specified type from the
binding context.
    /// </summary>
    /// <typeparam name="TChannel">The type of channel the factory builds.</typeparam>
    /// <param name="context">The BindingContext that provides context for the binding
element.</param>
    /// <returns>The IChannelFactory<TChannel/>of type TChannel initialized from the
context. </returns>
```

```
    public override IChannelFactory<TChannel>
BuildChannelFactory<TChannel>(BindingContext context)
    {
        throw new NotSupportedException();
    }

    /// <summary>
    /// Returns a value that indicates whether the binding element can build a
    /// channel listener for a specific type of channel.
    /// </summary>
    /// <typeparam name="TChannel">The type of channel listener to build.</typeparam>
    /// <param name="context">The BindingContext for the binding element.</param>
    /// <returns>true if a channel listener of the specified type can be built;
    ///          otherwise, false. The default is false. </returns>
    public override bool CanBuildChannelListener<TChannel>(BindingContext context)
    {
        var ok = typeof(TChannel) != typeof(IInputChannel) &&
context.CanBuildInnerChannelListener<TChannel>();
        Trace.WriteLineIf(isTrackingEnabled, ok ? CanBuildIInputSessionChannel :
CannotBuildIInputChannel);
        return ok;
    }

    /// <summary>
    /// Initializes a channel listener for producing channels of a specified type from
the binding context.
    /// </summary>
    /// <typeparam name="TChannel">The type of channel that the listener is built to
accept.</typeparam>
    /// <param name="context">The BindingContext for the binding element.</param>
    /// <returns>The IChannelListener<TChannel/>of type IChannel initialized from the
context.</returns>
    public override IChannelListener<TChannel>
BuildChannelListener<TChannel>(BindingContext context)
    {
        return context.BuildInnerChannelListener<TChannel>();
    }
    #endregion
}
```

Component Tracking

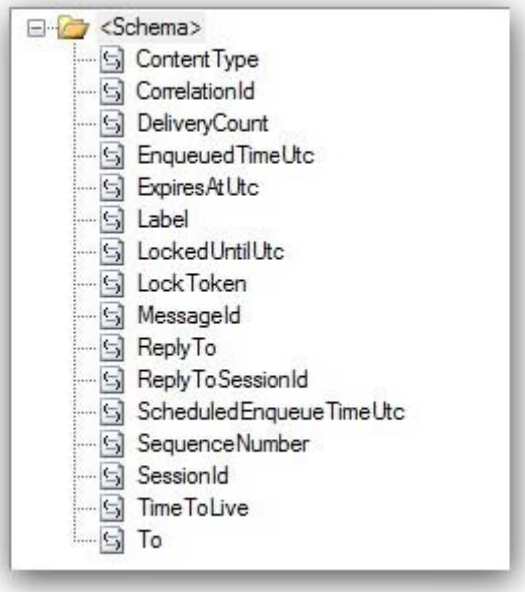
You can enable the component tracking and use DebugView, as shown in the picture below, to monitor its runtime behavior.



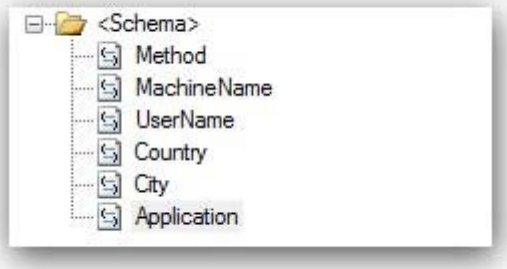
Property Schemas

To achieve a full integration between the Service Bus and BizTalk Server, it's necessary to translate the properties exposed by a BrokeredMessage into context properties of a BizTalk message and vice versa. The first step in this direction has been the definition of two property schemas:

- The first property schema is called **BrokeredMessagePropertySchema** and contains an element for each property of the BrokeredMessage class. This property schema is independent from a specific application and is fully reusable.



- The second property schema is called just **PropertySchema** and contains the specific context properties used by the **ServiceBusSample** application.



Note

All of the context properties defined in both property schemas inherit from the MessageContextPropertyBase base class. This inheritance allows you to define any of these properties independently from the schema of the received message when you define the filter expression of a direct bound orchestration.

Schemas

The request and response messages exchanged by the client and BizTalk applications have the following formats:

CalculatorRequest Message

```
<CalculatorRequest xmlns="http://windowsazure.cat.microsoft.com/samples/servicebus">
<Method>DynamicSendPortOrchestration</Method>
<Operations>
<Operation>
<Operator>+</Operator>
<Operand1>82</Operand1>
<Operand2>18</Operand2>
</Operation>
<Operation>
<Operator>-</Operator>
<Operand1>30</Operand1>
<Operand2>12</Operand2>
</Operation>
<Operation>
<Operator>*</Operator>
<Operand1>25</Operand1>
```

```
<Operand2>8</Operand2>
</Operation>
<Operation>
<Operator>\</Operator>
<Operand1>100</Operand1>
<Operand2>25</Operand2>
</Operation>
<Operation>
<Operator>+</Operator>
<Operand1>100</Operand1>
<Operand2>32</Operand2>
</Operation>
</Operations>
</CalculatorRequest>
```

CalculatorResponse Message

```
<CalculatorResponse xmlns="http://windowsazure.cat.microsoft.com/samples/servicebus">
<Status>Ok</Status>
<Results>
<Result>
<Value>100</Value>
<Error>None</Error>
</Result>
<Result>
<Value>18</Value>
<Error>None</Error>
</Result>
<Result>
<Value>200</Value>
<Error>None</Error>
</Result>
<Result>
<Value>4</Value>
<Error>None</Error>
</Result>
<Result>
<Value>132</Value>
<Error>None</Error>
</Result>
</Results>
</CalculatorResponse>
```

Both message types are defined by an XSD file contained in the `Schemas` project that you can find in the companion code for this article.

Service Bus Message Inspector

The final step to bridge the gap between the Service Bus and BizTalk Server has been to create a component to translate the properties of a BrokeredMessage into context properties of a BizTalk message and vice versa. Indeed, a WCF-Custom receive location can use the NetMessagingBinding to receive messages from a queue or subscription without the need to use any custom component to extend the normal behavior of the WCF adapter runtime. However, using this approach, a BizTalk application can read only the body of the message retrieved from the Service Bus, not the value of the properties specified by the sender. At the beginning of the article, I highlighted that it's quite common to have a message with an empty body and the entire payload specified in the user defined properties. For this reason, it's extremely important, not to say mandatory, to create a component that turns the properties of a BrokeredMessage into context properties of a BizTalk message and vice versa.

In the first part of the article we have seen that a WCF service which uses the NetMessagingBinding to read messages from a queue or a subscription, can retrieve a BrokeredMessageProperty object from the Properties collection of the incoming WCF message. This fact gives us the possibility to write a custom message inspector to intercept the incoming WCF message and read the BrokeredMessageProperty from its properties. For obvious reasons, this operation cannot be done using a custom pipeline component because a custom pipeline does not provide access to the WCF message. In the previous step we created the property schemas which define, respectively, the object specific and user defined properties for Service Bus messages. The last piece to complete the puzzle is to find a way to promote or write a property to the BizTalk message context. Fortunately, the WCF adapter runtime provides an easy way to accomplish this task.

As explained in the BizTalk Server documentation on MSDN, to promote a set of properties to the BizTalk message context you need to add their xml qualified name and value to a collection of key/value pairs. Next you must add a new property to the inbound WCF message whose key must be equal to:

- http://schemas.microsoft.com/BizTalk/2006/01/Adapters/WCF-properties/Promote

and specify the collection of key/value pairs as its value. The WCF adapter looks for this property in the WCF message and will promote all the properties contained in the collection.

Likewise, to write but not promote a set of properties to the BizTalk message context, you need to add their XML qualified name and value to another collection of key/value pairs. Next you must add another property to the inbound WCF message whose key must be equal to:

- http://schemas.microsoft.com/BizTalk/2006/01/Adapters/WCF-properties/WriteToContext

and specify the second collection of key/value pairs as its value. The WCF adapter looks for this property in the WCF message and writes all the properties contained in the collection to the BizTalk message context.

The **ServiceBusMessageInspector** component exploits this feature to promote or write the properties of the BrokeredMessageProperty to the context of the BizTalk message when a WCF-Custom receive location reads a message from a Service Bus queue or subscription. Conversely, the component uses another characteristic of the WCF adapter to retrieve the context property from an outbound WCF message and translate them into properties of a BrokeredMessageProperty. In fact, the send handler of the WCF adapter copies all the message context properties into the Properties collection when it creates a WCF message from a BizTalk message. Therefore, when using a WCF-Custom send port, this allows access to any message context properties within a custom message inspector or channel component at runtime.

- The **ServiceBusMessageInspectorBehaviorExtensionElement** can be used to register the message inspector in the machine.config file. It exposes the following properties that you can configure when creating a WCF-Custom receive location or send port:
- **IsComponentEnabled**: gets or sets a value indicating whether the message inspector is enabled.
- **IsTrackingEnabled**: Gets or sets a value indicating whether the message inspector is enabled.
- **PropertiesToPromote**: defines which user defined properties must be extracted from the Properties collection of the BrokeredMessageProperty and promoted to the BizTalk message context. This property is intended to be used only when the message inspector is used in a WCF-Custom receive location
- **PropertiesToWrite**: defines which user defined properties must be extracted from the Properties collection of the BrokeredMessageProperty and written but not promoted to the BizTalk message context. This property is intended to be used only when the message inspector is used in a WCF-Custom receive location.
- **PropertiesToSend**: defines which context properties must be extracted from the Properties collection of the WCF message and added to the Properties collection of the BrokeredMessageProperty. This property is intended to be used only when the message inspector is used in a WCF-Custom send port.

For you convenience, I included the code of the custom message inspector in the following example.

ServiceBusMessageInspector Class

```
/// <summary>
/// This class is used to promote\write properties to the message context.
/// </summary>
public class ServiceBusMessageInspector : IDispatchMessageInspector,
IClientMessageInspector
{
    #region Private Constants
    //***************************
    // Constants
    //***************************

    private const string Source = "ServiceBusMessageInspector";
```

```
    private const string BrokeredMessagePropertySchemaNamespace =
        "http://schemas.microsoft.com/servicebus/2011/brokered-message-property";
    private const string PropertiesToPromoteKey =
        "http://schemas.microsoft.com/BizTalk/2006/01/Adapters/WCF-properties/Promote";
    private const string PropertiesToWriteKey =
        "http://schemas.microsoft.com/BizTalk/2006/01/Adapters/WCF-
properties/WriteToContext";
    private const string ContentTypeProperty =
        "http://schemas.microsoft.com/servicebus/2011/brokered-message-
property#ContentType";
    private const string CorrelationIdProperty =
        "http://schemas.microsoft.com/servicebus/2011/brokered-message-
property#CorrelationId";
    private const string LabelProperty =
        "http://schemas.microsoft.com/servicebus/2011/brokered-message-property#Label";
    private const string ReplyToProperty =
        "http://schemas.microsoft.com/servicebus/2011/brokered-message-property#ReplyTo";
    private const string ReplyToSessionIdProperty =
        "http://schemas.microsoft.com/servicebus/2011/brokered-message-
property#ReplyToSessionId";
    private const string SessionIdProperty =
        "http://schemas.microsoft.com/servicebus/2011/brokered-message-
property#SessionId";
    private const string ToProperty =
        "http://schemas.microsoft.com/servicebus/2011/brokered-message-property#To";
    private const string ContentType = "ContentType";
    private const string CorrelationId = "CorrelationId";
    private const string DeliveryCount = "DeliveryCount";
    private const string EnqueuedTimeUtc = "EnqueuedTimeUtc";
    private const string ExpiresAtUtc = "ExpiresAtUtc";
    private const string Label = "Label";
    private const string MessageId = "MessageId";
    private const string ReplyTo = "ReplyTo";
    private const string ReplyToSessionId = "ReplyToSessionId";
    private const string ScheduledEnqueueTimeUtc = "ScheduledEnqueueTimeUtc";
    private const string SequenceNumber = "SequenceNumber";
    private const string SessionId = "SessionId";
    private const string TimeToLive = "TimeToLive";
    private const string To = "To";
    private const string PropertiesToPromote = "PropertiesToPromote";
    private const string PropertiesToWrite = "PropertiesToWrite";
    private const string PropertiesToSend = "PropertiesToSend";
```

```
    //****************************
    // Messages
    //****************************
    private const string PropertiesElementCannotBeNull =
        "[ServiceBusMessageInspector] The PropertiesElement object cannot be null.";
    private const string NameAndNamespaceNumberAreDifferent =
        "[ServiceBusMessageInspector] The number of items in the name and namespace
comma-separated " +
        "lists is different in the {0} property.";
    private const string PropertyAddedToList = "[ServiceBusMessageInspector] The
following property was added to " +
        "the [{0}] list:\n\r - Name=[{1}]\n\r - Namespace=[{2}]\n\r - Value=[{3}]";
    private const string ExceptionFormat = "[ServiceBusMessageInspector] The following
exception occurred=[{0}].";
    #endregion

    #region Private Fields
    //****************************
    // Private Fields
    //****************************

    private readonly bool isComponentEnabled;
    private readonly bool isTrackingEnabled;
    private readonly Dictionary<string, string> propertiesToPromoteDictionary = new
Dictionary<string, string>();
    private readonly Dictionary<string, string> propertiesToWriteDictionary = new
Dictionary<string, string>();
    private readonly Dictionary<string, string> propertiesToSendDictionary = new
Dictionary<string, string>();
    #endregion

    #region Public Constructors
    //****************************
    // Public Constructors
    //****************************

    /// <summary>
    /// Initializes a new instance of the ServiceBusMessageInspector class.
    /// </summary>
    public ServiceBusMessageInspector()
    {
```

```
        isComponentEnabled = true;
    }

    /// <summary>
    /// Initializes a new instance of the ServiceBusMessageInspector class.
    /// </summary>
    /// <param name="isComponentEnabled">This value indicates whether the message
inspector is enabled.</param>
    /// <param name="isTrackingEnabled">This value indicates whether tracking is
enabled.</param>
    /// <param name="propertiesToPromote">This object defines the properties to
promote.</param>
    /// <param name="propertiesToWrite">This object defines the properties to
write.</param>
    /// <param name="propertiesToSend">This object defines the properties to
send.</param>
    public ServiceBusMessageInspector(bool isComponentEnabled,
                                      bool isTrackingEnabled,
                                      PropertiesElement propertiesToPromote,
                                      PropertiesElement propertiesToWrite,
                                      PropertiesElement propertiesToSend)
    {
        this.isComponentEnabled = isComponentEnabled;
        this.isTrackingEnabled = isTrackingEnabled;

        // Propulate the dictionary that contains the properties to promote
        InitializeDictionary(PropertiesToPromote, propertiesToPromote,
propertiesToPromoteDictionary);

        // Propulate the dictionary that contains the properties to write
        InitializeDictionary(PropertiesToWrite, propertiesToWrite,
propertiesToWriteDictionary);

        // Propulate the dictionary that contains the properties to send out
        InitializeDictionary(PropertiesToSend, propertiesToSend,
propertiesToSendDictionary);
    }
    #endregion

    #region IDispatchMessageInspector Members
    //**********************************
    // IDispatchMessageInspector Members
```

```
    //**********************************

    /// <summary>
    /// Called after an inbound message has been received but
    /// before the message is dispatched to the intended operation.
    /// This method extracts the properties from the BrokeredMessageProperty
    /// object contained in the WCF message properties and write or promote
    /// them in the message context.
    /// </summary>
    /// <param name="request">The request message.</param>
    /// <param name="channel">The incoming channel.</param>
    /// <param name="instanceContext">The current service instance.</param>
    /// <returns>The object used to correlate state. This object is passed back
    ///          in the BeforeSendReply method.</returns>
    public object AfterReceiveRequest(ref Message request, IClientChannel channel,
InstanceContext instanceContext)
    {
        try
        {
            if (!isComponentEnabled)
            {
                return Guid.NewGuid();
            }

            var propertiesToPromoteList = new List<KeyValuePair<XmlQualifiedName,
object>>();
            var propertiesToWriteList = new List<KeyValuePair<XmlQualifiedName,
object>>();

            // Retreive the BrokeredMessageProperty from the current operation context
            var incomingProperties = OperationContext.Current.IncomingMessageProperties;
            var brokeredMessageProperty =
(BrokeredMessageProperty)incomingProperties[BrokeredMessageProperty.Name];

            if (brokeredMessageProperty != null)
            {
                // Include BrokeredMessageProperty explicit properties
                // in the list of properties to write to the BizTalk message context
                if (!string.IsNullOrEmpty(brokeredMessageProperty.ContentType))
                {
                    AddItemToList(ContentType,
                                  BrokeredMessagePropertySchemaNamespace,
```

```
                    brokeredMessageProperty.ContentType,
                    PropertiesToWrite,
                    propertiesToWriteList);
}
if (!string.IsNullOrEmpty(brokeredMessageProperty.CorrelationId))
{
    AddItemToList(CorrelationId,
                    BrokeredMessagePropertySchemaNamespace,
                    brokeredMessageProperty.CorrelationId,
                    PropertiesToWrite,
                    propertiesToWriteList);
}
AddItemToList(DeliveryCount,
                BrokeredMessagePropertySchemaNamespace,
                brokeredMessageProperty.DeliveryCount,
                PropertiesToWrite,
                propertiesToWriteList);
AddItemToList(EnqueuedTimeUtc,
                BrokeredMessagePropertySchemaNamespace,
                brokeredMessageProperty.EnqueuedTimeUtc,
                PropertiesToWrite,
                propertiesToWriteList);
AddItemToList(ExpiresAtUtc,
                BrokeredMessagePropertySchemaNamespace,
                brokeredMessageProperty.ExpiresAtUtc,
                PropertiesToWrite,
                propertiesToWriteList);
if (!string.IsNullOrEmpty(brokeredMessageProperty.Label))
{
    AddItemToList(Label,
                    BrokeredMessagePropertySchemaNamespace,
                    brokeredMessageProperty.Label,
                    PropertiesToWrite,
                    propertiesToWriteList);
}

if (!string.IsNullOrEmpty(brokeredMessageProperty.MessageId))
{
    AddItemToList(MessageId,
                    BrokeredMessagePropertySchemaNamespace,
                    brokeredMessageProperty.MessageId,
                    PropertiesToWrite,
```

```
                    propertiesToWriteList);
}
if (!string.IsNullOrEmpty(brokeredMessageProperty.ReplyTo))
{
    AddItemToList(ReplyTo,
                    BrokeredMessagePropertySchemaNamespace,
                    brokeredMessageProperty.ReplyTo,
                    PropertiesToWrite,
                    propertiesToWriteList);
}
if (!string.IsNullOrEmpty(brokeredMessageProperty.ReplyToSessionId))
{
    AddItemToList(ReplyToSessionId,
                    BrokeredMessagePropertySchemaNamespace,
                    brokeredMessageProperty.ReplyToSessionId,
                    PropertiesToWrite,
                    propertiesToWriteList);
}
AddItemToList(ScheduledEnqueueTimeUtc,
                BrokeredMessagePropertySchemaNamespace,
                brokeredMessageProperty.ScheduledEnqueueTimeUtc,
                PropertiesToWrite,
                propertiesToWriteList);
AddItemToList(SequenceNumber,
                BrokeredMessagePropertySchemaNamespace,
                brokeredMessageProperty.SequenceNumber,
                PropertiesToWrite,
                propertiesToWriteList);
if (!string.IsNullOrEmpty(brokeredMessageProperty.SessionId))
{
    AddItemToList(SessionId,
                    BrokeredMessagePropertySchemaNamespace,
                    brokeredMessageProperty.SessionId,
                    PropertiesToWrite,
                    propertiesToWriteList);
}
AddItemToList(TimeToLive,
                BrokeredMessagePropertySchemaNamespace,
                brokeredMessageProperty.TimeToLive.TotalSeconds,
                PropertiesToWrite,
                propertiesToWriteList);
if (!string.IsNullOrEmpty(brokeredMessageProperty.To))
```

```
    {
        AddItemToList(To,
                        BrokeredMessagePropertySchemaNamespace,
                        brokeredMessageProperty.To,
                        PropertiesToWrite,
                        propertiesToWriteList);
    }

    // Promote or write properties indicated in the
    // configuration of the message inspector.
    if (brokeredMessageProperty.Properties != null &&
        brokeredMessageProperty.Properties.Count > 0)
    {
        foreach (var property in brokeredMessageProperty.Properties)
        {
            if (propertiesToPromoteDictionary.ContainsKey(property.Key))
            {
                AddItemToList(property.Key,
                                propertiesToPromoteDictionary[property.Key],
                                property.Value,
                                PropertiesToPromote,
                                propertiesToPromoteList);
                continue;
            }
            if (propertiesToWriteDictionary.ContainsKey(property.Key))
            {
                AddItemToList(property.Key,
                                propertiesToWriteDictionary[property.Key],
                                property.Value,
                                PropertiesToWrite,
                                propertiesToWriteList);
            }
        }
    }
}

// Notify the WCF Adapter the properties to promote
if (propertiesToPromoteList.Count > 0)
{
    request.Properties.Add(PropertiesToPromoteKey, propertiesToPromoteList);
}
```

```
            // Notify the WCF Adapter the properties to write
            if (propertiesToWriteList.Count > 0)
            {
                request.Properties.Add(PropertiesToWriteKey, propertiesToWriteList);
            }
        }
        catch (Exception ex)
        {
            Trace.WriteLineIf(isTrackingEnabled, string.Format(ExceptionFormat,
ex.Message));
            EventLog.WriteEntry(Source, ex.Message, EventLogEntryType.Error);
            throw;
        }
        return Guid.NewGuid();
    }

    /// <summary>
    /// Called after the operation has returned but before the reply message is sent.
    /// </summary>
    /// <param name="reply">The reply message. This value is null if the operation is one
way.</param>
    /// <param name="correlationState">The correlation object returned from the
AfterReceiveRequest method.</param>
    public void BeforeSendReply(ref Message reply, object correlationState)
    {
        return;
    }
    #endregion

    #region IClientMessageInspector Members
    //**********************************
    // IClientMessageInspector Members
    //**********************************

    /// <summary>
    /// Enables inspection or modification of a message after a reply message is
    /// received but prior to passing it back to the client application.
    /// </summary>
    /// <param name="reply">The message to be transformed into types and handed
    ///                     back to the client application.</param>
    /// <param name="correlationState">Correlation state data.</param>
    public void AfterReceiveReply(ref Message reply, object correlationState)
```

```
    {
        return;
    }

    /// <summary>
    /// Enables inspection or modification of a message before a request message is sent
to a service.
    /// </summary>
    /// <param name="message">The message to be sent to the service.</param>
    /// <param name="channel">The client object channel.</param>
    /// <returns>The object that is returned as the correlationState
    ///          argument of the AfterReceiveReply method.</returns>
    public object BeforeSendRequest(ref Message message, IClientChannel channel)
    {
        try
        {
            if (!isComponentEnabled)
            {
                return message.Headers.Action;
            }

            // Create a new BrokeredMessageProperty object
            var brokeredMessageProperty = new BrokeredMessageProperty();

            // Adds properties to the BrokeredMessageProperty object
            foreach (var property in message.Properties)
            {
                if (string.Compare(property.Key, ContentTypeProperty, true) == 0)
                {
                    brokeredMessageProperty.ContentType =
message.Properties[ContentTypeProperty] as string;
                    Trace.WriteLineIf(isTrackingEnabled,
string.Format(PropertyAddedToList,
                                                                        PropertiesToSend,
                                                                        ContentType,

BrokeredMessagePropertySchemaNamespace,

brokeredMessageProperty.ContentType));
                    continue;
                }
                if (string.Compare(property.Key, CorrelationIdProperty, true) == 0)
```

```
                {
                    brokeredMessageProperty.CorrelationId =
message.Properties[CorrelationIdProperty] as string;
                    Trace.WriteLineIf(isTrackingEnabled,
string.Format(PropertyAddedToList,
                                                                        PropertiesToSend,
                                                                        CorrelationId,

BrokeredMessagePropertySchemaNamespace,

brokeredMessageProperty.CorrelationId));
                    continue;
                }
                if (string.Compare(property.Key, LabelProperty, true) == 0)
                {
                    brokeredMessageProperty.Label = message.Properties[LabelProperty] as
string;
                    Trace.WriteLineIf(isTrackingEnabled,
string.Format(PropertyAddedToList,
                                                                        PropertiesToSend,
                                                                        Label,

BrokeredMessagePropertySchemaNamespace,

brokeredMessageProperty.Label));
                    continue;
                }
                if (string.Compare(property.Key, ReplyToProperty, true) == 0)
                {
                    brokeredMessageProperty.ReplyTo = message.Properties[ReplyToProperty]
as string;
                    Trace.WriteLineIf(isTrackingEnabled,
string.Format(PropertyAddedToList,
                                                                        PropertiesToSend,
                                                                        ReplyTo,

BrokeredMessagePropertySchemaNamespace,

brokeredMessageProperty.ReplyTo));
                    continue;
                }
                if (string.Compare(property.Key, ReplyToSessionIdProperty, true) == 0)
```

```
                {
                    brokeredMessageProperty.ReplyToSessionId =
message.Properties[ReplyToSessionIdProperty] as string;
                    Trace.WriteLineIf(isTrackingEnabled,
string.Format(PropertyAddedToList,
                                                                        PropertiesToSend,
                                                                        ReplyToSessionId,

BrokeredMessagePropertySchemaNamespace,

brokeredMessageProperty.ReplyToSessionId));
                    continue;
                }
                if (string.Compare(property.Key, SessionIdProperty, true) == 0)
                {
                    brokeredMessageProperty.SessionId =
message.Properties[SessionIdProperty] as string;
                    Trace.WriteLineIf(isTrackingEnabled,
string.Format(PropertyAddedToList,
                                                                        PropertiesToSend,
                                                                        SessionId,

BrokeredMessagePropertySchemaNamespace,

brokeredMessageProperty.SessionId));
                    continue;
                }
                if (string.Compare(property.Key, ToProperty, true) == 0)
                {
                    brokeredMessageProperty.To = message.Properties[ToProperty] as
string;
                    Trace.WriteLineIf(isTrackingEnabled,
string.Format(PropertyAddedToList,
                                                                        PropertiesToSend,
                                                                        To,

BrokeredMessagePropertySchemaNamespace,

brokeredMessageProperty.To));
                    continue;
                }
```

```
                // Include properties indicated in the
                // configuration of the message inspector.
                string[] parts = property.Key.Split('#');
                if (parts.Length == 2)
                {
                    if (propertiesToSendDictionary.ContainsKey(parts[1]) &&
                        propertiesToSendDictionary[parts[1]] == parts[0])
                    {
                        brokeredMessageProperty.Properties[parts[1]] = property.Value;
                        Trace.WriteLineIf(isTrackingEnabled,
string.Format(PropertyAddedToList,

PropertiesToSend,
                                                                                parts[0],
                                                                                parts[1],

property.Value));
                    }
                }
            }

            // Add the BrokeredMessageProperty to the WCF message properties
            message.Properties[BrokeredMessageProperty.Name] = brokeredMessageProperty;
        }
        catch (Exception ex)
        {
            Trace.WriteLineIf(isTrackingEnabled, string.Format(ExceptionFormat,
ex.Message));
            EventLog.WriteEntry(Source, ex.Message, EventLogEntryType.Error);
            throw;
        }
        return Guid.NewGuid();
    }
    #endregion

    #region Private Methods
    //******************
    // Private Methods
    //******************

    /// <summary>
    /// Initialize the dictionary passed as second parameter
```

```
    /// with the information read from the first argument.
    /// In particular, the Name and Namespace properties of the
    /// PropertiesElement object contain respectively
    /// the comma-separated list of property names and namespaces
    /// of a set of properties. The name of each property will
    /// become the key in the dictionary, whereas the namespace
    /// will become the corresponding value.
    /// </summary>
    /// <param name="dictionaryName">The name of the dictionary.</param>
    /// <param name="properties">A PropertiesElement object.</param>
    /// <param name="dictionary">The dictionary to populate.</param>
    private void InitializeDictionary(string dictionaryName,
                                      PropertiesElement properties,
                                      Dictionary<string, string> dictionary)
    {
        if (properties == null)
        {
            Trace.WriteLineIf(isTrackingEnabled, PropertiesElementCannotBeNull);
            throw new ApplicationException(PropertiesElementCannotBeNull);
        }
        if (!string.IsNullOrEmpty(properties.Name) &&
            !string.IsNullOrEmpty(properties.Namespace))
        {
            var nameArray = properties.Name.Split(new[] { ',', ';', '|' });
            var namespaceArray = properties.Namespace.Split(new[] { ',', ';', '|' });
            if (namespaceArray.Length == nameArray.Length)
            {
                if (nameArray.Length > 0)
                {
                    for (int i = 0; i < nameArray.Length; i++)
                    {
                        dictionary[nameArray[i]] = namespaceArray[i];
                    }
                }
            }
            else
            {
                Trace.WriteLineIf(isTrackingEnabled,
string.Format(NameAndNamespaceNumberAreDifferent,
                                                                    dictionaryName));
                throw new
ApplicationException(string.Format(NameAndNamespaceNumberAreDifferent,
```

```
                                                                dictionaryName));
            }
        }
    }

    /// <summary>
    /// Adds a new property to the specified list.
    /// </summary>
    /// <param name="name">The name of a property.</param>
    /// <param name="ns">The namespace of a property.</param>
    /// <param name="value">The value of a property.</param>
    /// <param name="listName">The name of the list.</param>
    /// <param name="list">The list to add the property to.</param>
    private void AddItemToList(string name,
                               string ns,
                               object value,
                               string listName,
                               List<KeyValuePair<XmlQualifiedName, object>> list)
    {
        list.Add(new KeyValuePair<XmlQualifiedName, object>(new XmlQualifiedName(name,
ns), value));
        Trace.WriteLineIf(isTrackingEnabled, string.Format(PropertyAddedToList,
                                                            listName,
                                                            name,
                                                            ns,
                                                            value));
    }
    #endregion
}
```

Component Description

The constructor of the message inspector reads the configuration data and initializes three dictionaries that contain, respectively, the properties to promote, to write, and to send out.

- The AfterReceiveRequest method is executed when a WCF-Custom receive location, configured to use the message inspector, receives a message from the Service Bus. This method performs the following actions:
    a. Uses the current OperationContext to read the **BrokeredMessageProperty** from the **Properties** collection of the WCF message,
    b. Includes the BrokeredMessageProperty properties in the list of the properties to write to the BizTalk message context
    c. Reads the user-defined properties from the Properties collection of the BrokeredMessageProperty and promote or write only those that have been specified in

the configuration of the WCF receive location using the **PropertiesToPromote** and **PropertiesToWrite** properties of the endpoint behavior.

The BeforeSendRequest method is executed when a WCF-Custom send port, configured to use the message inspector, is about to send a message to the Service Bus. This method performs the following actions:

1. Creates a new BrokeredMessageProperty object.
2. Reads the context properties, defined in the **BrokeredMessagePropertySchema**, from the Properties collection of the WCF message and assigns their values to the corresponding properties of the BrokeredMessageProperty.
3. Adds the custom context properties, in our sample defined in the **PropertySchema**, to the Properties collection of the BrokeredMessageProperty.
4. Finally, adds the BrokeredMessageProperty to the WCF message properties.

Component Tracking

You can enable the component tracking and use DebugView, as shown in the picture below, to monitor its runtime behavior.

```
#   Time        Debug Print
12  46.12773132 [25820] [ServiceBusMessageInspector] The following property was added to the [PropertiesToWrite] list:
13  46.12773132 [25820]   - Name=[ContentType]
14  46.12773132 [25820]   - Namespace=[http://schemas.microsoft.com/servicebus/2011/brokered-message-property]
15  46.12773132 [25820]   - Value=[application/soap+msbin1]
16  46.12776184 [25820] [ServiceBusMessageInspector] The following property was added to the [PropertiesToWrite] list:
17  46.12776184 [25820]   - Name=[DeliveryCount]
18  46.12776184 [25820]   - Namespace=[http://schemas.microsoft.com/servicebus/2011/brokered-message-property]
19  46.12776184 [25820]   - Value=[1]
20  46.12781143 [25820] [ServiceBusMessageInspector] The following property was added to the [PropertiesToWrite] list:
21  46.12781143 [25820]   - Name=[EnqueuedTimeUtc]
22  46.12781143 [25820]   - Namespace=[http://schemas.microsoft.com/servicebus/2011/brokered-message-property]
23  46.12781143 [25820]   - Value=[9/29/2011 8:24:12 AM]
24  46.12786484 [25820] [ServiceBusMessageInspector] The following property was added to the [PropertiesToWrite] list:
25  46.12786484 [25820]   - Name=[ExpiresAtUtc]
26  46.12786484 [25820]   - Namespace=[http://schemas.microsoft.com/servicebus/2011/brokered-message-property]
27  46.12786484 [25820]   - Value=[12/31/9999 11:59:59 PM]
28  46.12791443 [25820] [ServiceBusMessageInspector] The following property was added to the [PropertiesToWrite] list:
29  46.12791443 [25820]   - Name=[Label]
30  46.12791443 [25820]   - Namespace=[http://schemas.microsoft.com/servicebus/2011/brokered-message-property]
31  46.12791443 [25820]   - Value=[WindowsAzureCAT]
32  46.12792587 [25820] [ServiceBusMessageInspector] The following property was added to the [PropertiesToWrite] list:
33  46.12792587 [25820]   - Name=[MessageId]
34  46.12792587 [25820]   - Namespace=[http://schemas.microsoft.com/servicebus/2011/brokered-message-property]
35  46.12792587 [25820]   - Value=[15fad380-e38b-4937-b18c-8b085bba618f]
36  46.12797165 [25820] [ServiceBusMessageInspector] The following property was added to the [PropertiesToWrite] list:
37  46.12797165 [25820]   - Name=[ReplyTo]
38  46.12797165 [25820]   - Namespace=[http://schemas.microsoft.com/servicebus/2011/brokered-message-property]
39  46.12797165 [25820]   - Value=[sb://paolosalvatori.servicebus.windows.net/responsetopic]
40  46.12799835 [25820] [ServiceBusMessageInspector] The following property was added to the [PropertiesToWrite] list:
41  46.12799835 [25820]   - Name=[ReplyToSessionId]
42  46.12799835 [25820]   - Namespace=[http://schemas.microsoft.com/servicebus/2011/brokered-message-property]
43  46.12799835 [25820]   - Value=[0a643bac-d49d-45ff-996e-e54da915a230]
44  46.12801743 [25820] [ServiceBusMessageInspector] The following property was added to the [PropertiesToWrite] list:
45  46.12801743 [25820]   - Name=[ScheduledEnqueueTimeUtc]
46  46.12801743 [25820]   - Namespace=[http://schemas.microsoft.com/servicebus/2011/brokered-message-property]
47  46.12801743 [25820]   - Value=[1/1/0001 12:00:00 AM]
48  46.12804794 [25820] [ServiceBusMessageInspector] The following property was added to the [PropertiesToWrite] list:
49  46.12804794 [25820]   - Name=[SequenceNumber]
50  46.12804794 [25820]   - Namespace=[http://schemas.microsoft.com/servicebus/2011/brokered-message-property]
51  46.12804794 [25820]   - Value=[8]
52  46.12807465 [25820] [ServiceBusMessageInspector] The following property was added to the [PropertiesToWrite] list:
53  46.12807465 [25820]   - Name=[SessionId]
54  46.12807465 [25820]   - Namespace=[http://schemas.microsoft.com/servicebus/2011/brokered-message-property]
55  46.12807465 [25820]   - Value=[0a643bac-d49d-45ff-996e-e54da915a230]
56  46.12820435 [25820] [ServiceBusMessageInspector] The following property was added to the [PropertiesToWrite] list:
57  46.12820435 [25820]   - Name=[TimeToLive]
58  46.12820435 [25820]   - Namespace=[http://schemas.microsoft.com/servicebus/2011/brokered-message-property]
59  46.12820435 [25820]   - Value=[922337203685.478]
60  46.12825394 [25820] [ServiceBusMessageInspector] The following property was added to the [PropertiesToPromote] list:
61  46.12825394 [25820]   - Name=[Method]
62  46.12825394 [25820]   - Namespace=[http://windowsazure.cat.microsoft.com/samples/servicebus/propertyschema]
63  46.12825394 [25820]   - Value=[DynamicSendPort]
64  46.12827301 [25820] [ServiceBusMessageInspector] The following property was added to the [PropertiesToWrite] list:
65  46.12827301 [25820]   - Name=[Country]
66  46.12827301 [25820]   - Namespace=[http://windowsazure.cat.microsoft.com/samples/servicebus/propertyschema]
67  46.12827301 [25820]   - Value=[Italy]
68  46.12832642 [25820] [ServiceBusMessageInspector] The following property was added to the [PropertiesToWrite] list:
69  46.12832642 [25820]   - Name=[City]
70  46.12832642 [25820]   - Namespace=[http://windowsazure.cat.microsoft.com/samples/servicebus/propertyschema]
71  46.12832642 [25820]   - Value=[Milan]
72  46.12835312 [25820] [ServiceBusMessageInspector] The following property was added to the [PropertiesToWrite] list:
73  46.12835312 [25820]   - Name=[MachineName]
74  46.12835312 [25820]   - Namespace=[http://windowsazure.cat.microsoft.com/samples/servicebus/propertyschema]
75  46.12835312 [25820]   - Value=[UPPY]
76  46.12837219 [25820] [ServiceBusMessageInspector] The following property was added to the [PropertiesToWrite] list:
77  46.12837219 [25820]   - Name=[UserName]
78  46.12837219 [25820]   - Namespace=[http://windowsazure.cat.microsoft.com/samples/servicebus/propertyschema]
79  46.12837219 [25820]   - Value=[paolos]
```

If you stop both orchestrations and use the client application to send a request message to BizTalk Server, the BizTalk Server is suspended. This suspension allows you to use the **BizTalk Server Administration Console** to examine the context properties of the inbound document and, in particular, to review the properties promoted or written to the message context, in order, by the following components:

- WCF-Custom adapter (see properties at points 2 and 5).
- **ServiceBusMessageInspector** (see properties at point 3)
- **Xml Disassembler** component within the **XMLReceive** pipeline (see properties at points 1 and 4).

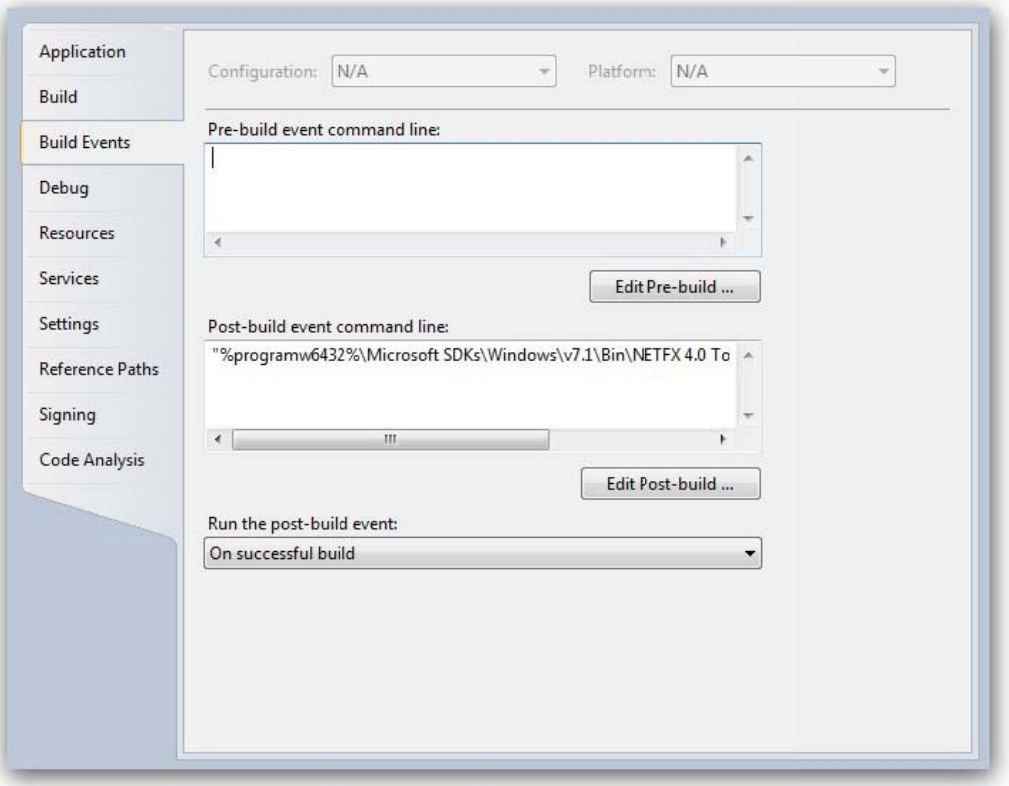
Name	Value	Type	Namespace
PartNames	System.String[]	Not Promoted	http://schemas.microsoft.com/BizTalk/2003/messageagent-properties
ActivityIdentity	[AA6B282B-1081-41F0-85C4-645A7CC7CF3D]	Not Promoted	http://schemas.microsoft.com/BizTalk/2003/messagetracking-properties
AdapterReceiveCompleteTime	10/3/2011 12:09:21 PM	Not Promoted	http://schemas.microsoft.com/BizTalk/2003/messagetracking-properties
PartyName		Not Promoted	http://schemas.microsoft.com/BizTalk/2003/messagetracking-properties
PortName	ServiceBusSample.OneWay.ReceivePort	Not Promoted	http://schemas.microsoft.com/BizTalk/2003/messagetracking-properties
MethodName	SendRequest	Promoted	http://schemas.microsoft.com/BizTalk/2003/soap-properties
InboundTransportLocation	sb://paolosalvatori.servicebus.windows.net/requestqueue	Promoted	http://schemas.microsoft.com/BizTalk/2003/system-properties
InterchangeID	{CD28183D-A2DD-4E15-A787-17F8ED94F0E6}	Not Promoted	http://schemas.microsoft.com/BizTalk/2003/system-properties
LastInterchangeMessage	True	Promoted	http://schemas.microsoft.com/BizTalk/2003/system-properties
ReceiveInstanceID	{32040754-AA10-4935-93F0-37D9706BE1CB}	Not Promoted	http://schemas.microsoft.com/BizTalk/2003/system-properties
ReceiveLocationName	ServiceBusSample.WCF-Custom.NetMessagingBinding.Queu...	Not Promoted	http://schemas.microsoft.com/BizTalk/2003/system-properties
ReceivePortID	{D83FECAE-0484-45D9-BFC1-54EA3C3EB533}	Promoted	http://schemas.microsoft.com/BizTalk/2003/system-properties
ReceivePortName	ServiceBusSample.OneWay.ReceivePort	Promoted	http://schemas.microsoft.com/BizTalk/2003/system-properties
AuthenticationRequiredOnReceivePort	False	Not Promoted	http://schemas.microsoft.com/BizTalk/2003/system-properties
InboundTransportType	WCF-Custom	Promoted	http://schemas.microsoft.com/BizTalk/2003/system-properties
InterchangeSequenceNumber	1	Not Promoted	http://schemas.microsoft.com/BizTalk/2003/system-properties
LRPMsgBodyTracking	0	Not Promoted	http://schemas.microsoft.com/BizTalk/2003/system-properties
MessageExchangePattern	1	Not Promoted	http://schemas.microsoft.com/BizTalk/2003/system-properties
ReceivePipelineID	{E7BC2D5-80AE-4CD5-B668-CCA77437D311}	Not Promoted	http://schemas.microsoft.com/BizTalk/2003/system-properties
ReceivePortAuth	0	Not Promoted	http://schemas.microsoft.com/BizTalk/2003/system-properties
SourcePartyID	e-1-5-7	Not Promoted	http://schemas.microsoft.com/BizTalk/2003/system-properties
MessageType	http://windowsazure.cat.microsoft.com/samples/servicebusR...	Promoted	http://schemas.microsoft.com/BizTalk/2003/system-properties
SchemaStrongName	Microsoft.WindowsAzure.CAT.Samples.BizTalkQueueIntegrati...	Not Promoted	http://schemas.microsoft.com/BizTalk/2003/system-properties
SourceCharset	utf-8	Not Promoted	http://schemas.microsoft.com/BizTalk/2003/xmlnorm-properties
Action	SendRequest	Promoted	http://schemas.microsoft.com/BizTalk/2006/01/Adapters/WCF-properties
InboundHeaders	<headers><a:Action s:mustUnderstand="1" xmlns:s="http://w...	Not Promoted	http://schemas.microsoft.com/BizTalk/2006/01/Adapters/WCF-properties
To	sb://paolosalvatori.servicebus.windows.net/requestqueue	Promoted	http://schemas.microsoft.com/BizTalk/2006/01/Adapters/WCF-properties
Label	WindowsAzureCAT	Not Promoted	http://schemas.microsoft.com/servicebus/2011/brokered-message-property
ReplyToSessionId	b63670eb-56e4-4a66-92d7-5e381d243611	Not Promoted	http://schemas.microsoft.com/servicebus/2011/brokered-message-property
ScheduledEnqueueTimeUtc	12/30/1899 12:00:00 AM	Not Promoted	http://schemas.microsoft.com/servicebus/2011/brokered-message-property
ContentType	application/soap+msbin1	Not Promoted	http://schemas.microsoft.com/servicebus/2011/brokered-message-property
DeliveryCount	1	Not Promoted	http://schemas.microsoft.com/servicebus/2011/brokered-message-property
EnqueuedTimeUtc	10/3/2011 12:09:20 PM	Not Promoted	http://schemas.microsoft.com/servicebus/2011/brokered-message-property
ExpiresAtUtc	12/31/9999 11:59:59 PM	Not Promoted	http://schemas.microsoft.com/servicebus/2011/brokered-message-property
ReplyTo	sb://paolosalvatori.servicebus.windows.net/responsequeue	Not Promoted	http://schemas.microsoft.com/servicebus/2011/brokered-message-property
SessionId	b63670eb-56e4-4a66-92d7-5e381d243611	Not Promoted	http://schemas.microsoft.com/servicebus/2011/brokered-message-property
MessageId	1ea34666-5420-44fa-8195-32e03321709e	Not Promoted	http://schemas.microsoft.com/servicebus/2011/brokered-message-property
SequenceNumber	17	Not Promoted	http://schemas.microsoft.com/servicebus/2011/brokered-message-property
TimeToLive	922337203685.478	Not Promoted	http://schemas.microsoft.com/servicebus/2011/brokered-message-property
Country	Italy	Not Promoted	http://windowsazure.cat.microsoft.com/samples/servicebus/propertyschema
MachineName	UPPY	Not Promoted	http://windowsazure.cat.microsoft.com/samples/servicebus/propertyschema
UserName	paolos	Not Promoted	http://windowsazure.cat.microsoft.com/samples/servicebus/propertyschema
City	Milan	Not Promoted	http://windowsazure.cat.microsoft.com/samples/servicebus/propertyschema
Method	StaticSendPort	Promoted	http://windowsazure.cat.microsoft.com/samples/servicebus/propertyschema
Action	<a:Action s:mustUnderstand="1" xmlns:s="http://www.w3.org...	Not Promoted	http://www.w3.org/2005/08/addressing
To	<a:To s:mustUnderstand="1" xmlns:s="http://www.w3.org/20...	Not Promoted	http://www.w3.org/2005/08/addressing

Installing Components in GAC

In order to use the **ListenUriEndpointBehavior** and **ServiceBusMessageInspector** classes in a BizTalk application, we need to perform the following operations:

1. Install the corresponding assemblies in the global assembly cache (GAC).
2. Register the components in the proper machine.config file on the BizTalk Server computer.

The first step is pretty straightforward and can be accomplished using the gacutil tool. You can automatically install your assembly to the GAC whenever you build the class library by including the execution of the gacutil tool in the post-build event command-line of the project, as shown in following picture:



An easy and handy way to verify that the two assemblies have been successfully installed in the GAC is to use the following commands:

```
gacutil /lr Microsoft.WindowsAzure.CAT.Samples.ServiceBus.ListenUriEndpointBehavior
gacutil /lr Microsoft.WindowsAzure.CAT.Samples.ServiceBus.MessageInspector
```

Registering Components in the machine.config file

To accomplish this task you can proceed as follow:

- Use a text editor such as Notepad to open the **machine.config** file, which can be found in the following folders:
  - %windir%\Microsoft.NET\Framework\v4.0.30319\Config
  - %windir%\Microsoft.NET\Framework64\v4.0.30319\Config
- Add the following items to the `<configuration><system.serviceModel>\<extensions><bindingElementExtensions>` section. These items contain the fully-qualified names (FQDN) of the behavior extension elements of the two components:

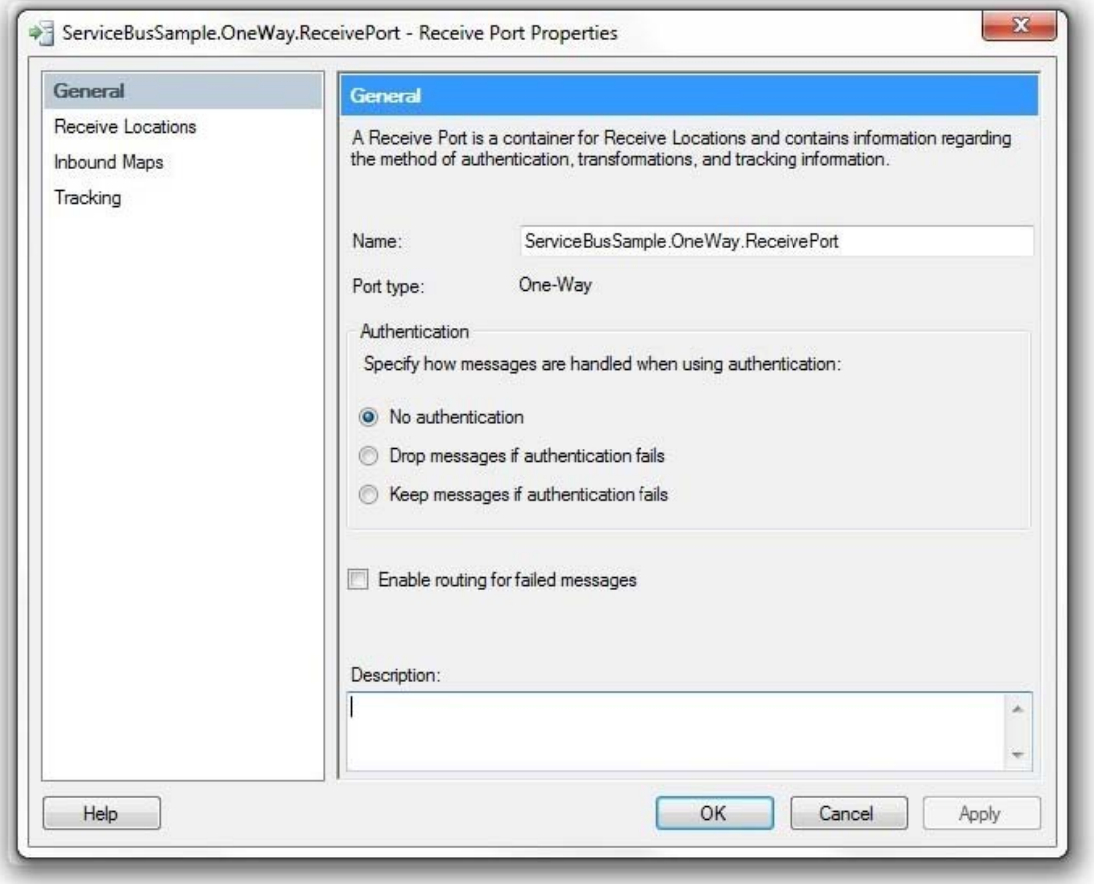
```
<add name="serviceBusMessageInspector"
type="Microsoft.WindowsAzure.CAT.Samples.ServiceBus.MessageInspector.
ServiceBusMessageInspectorBehaviorExtensionElement,
Microsoft.WindowsAzure.CAT.Samples.ServiceBus.MessageInspector,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=197ec3eb961f773c"/>
<add name="setListenUri"
type="Microsoft.WindowsAzure.CAT.Samples.ServiceBus.
ListenUriEndpointBehavior.ListenUriBehaviorExtensionElement,
Microsoft.WindowsAzure.CAT.Samples.ServiceBus.ListenUriEndpointBehavior,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=197ec3eb961f773c"/>
<add name="setSessionChannel"
type="Microsoft.WindowsAzure.CAT.Samples.ServiceBus.
SessionChannelEndpointBehavior.SessionChannelBehaviorExtensionElement,
Microsoft.WindowsAzure.CAT.Samples.ServiceBus.
SessionChannelEndpointBehavior,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=197ec3eb961f773c"/>
```

Receive Locations

We have now all the elements we need to create and configure the artifacts that compose our BizTalk application. In this section we will describe the steps necessary to create the WCF-Custom receive location used by the **ServiceBusSample** application to read messages from the **requestqueue**, then we'll see how to define a WCF-Custom receive location to retrieve messages from the **ItalyMilan** subscription defined on the **requesttopic**.

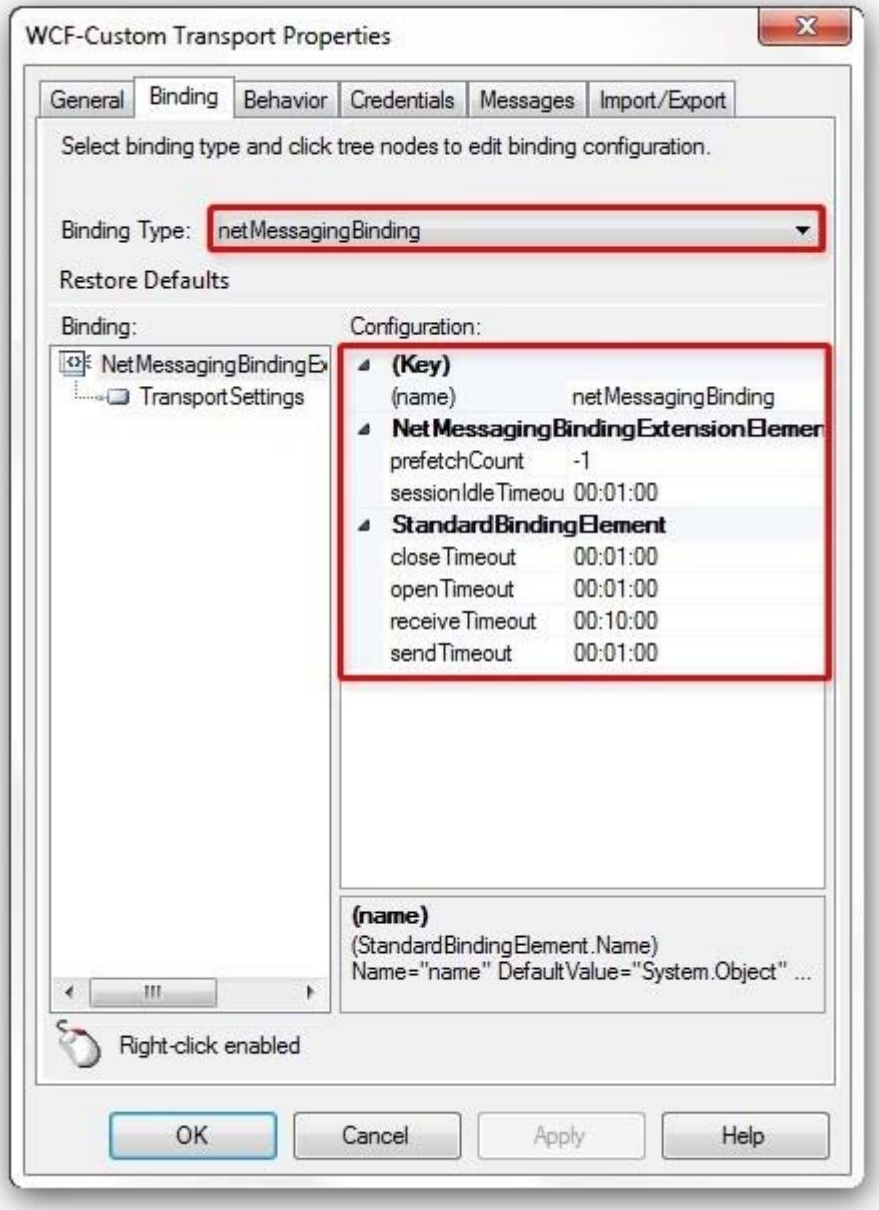
Let's start by looking at the steps necessary to create the WCF receive location used to read request messages from the **requestqueue**.

- Open the **BizTalk Server Administration Console** and expand the **Receive Ports** node of the **ServiceBusSample** application.
- Create a new one-way receive port as shown in the figure below:

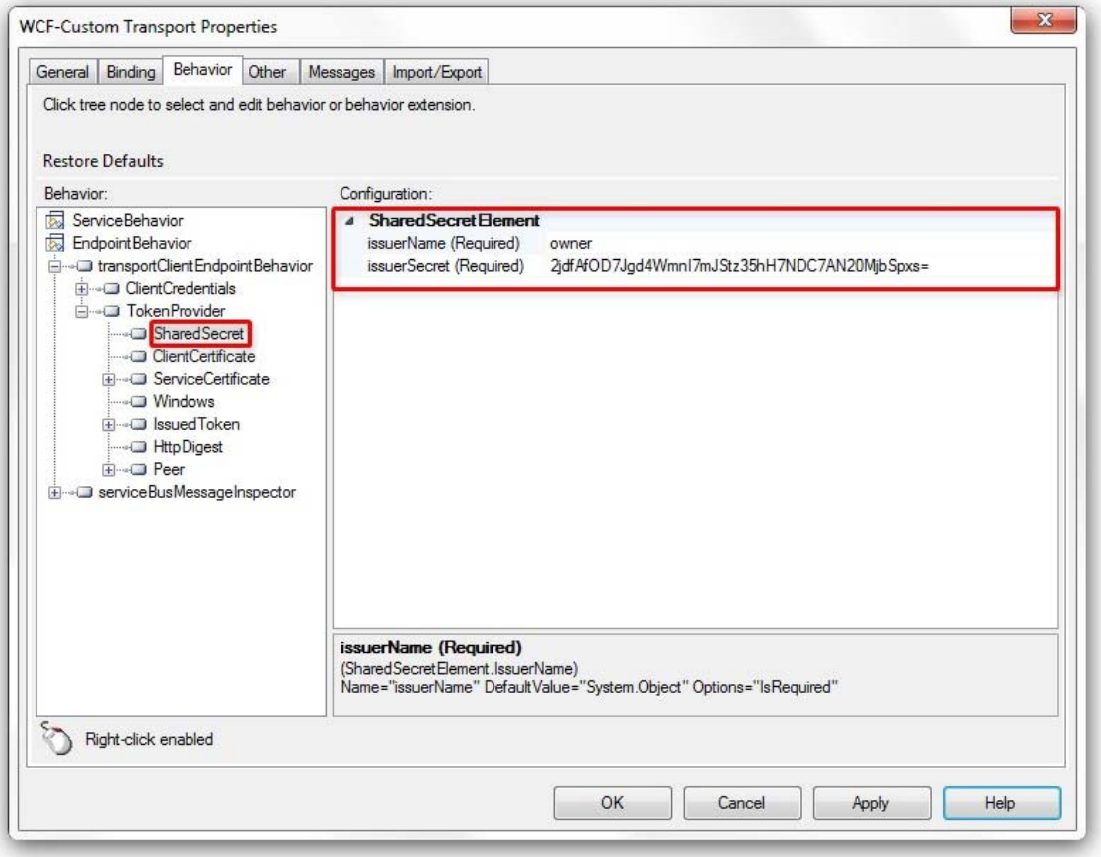


- Right-click the newly created receive port and select **New Receive Location** from the context menu.
- Specify a name for the receive location and choose a pipeline based on your requirements. Since in our sample we want to process incoming requests using an orchestration, we choose the **XMLReceive** pipeline to promote the context properties of the inbound document like the MessageType that is used in the subscription expression of the **StaticSendPortOrchestration** and **DynamicSendPortOrchestration**. Then choose the **WCF-Custom** adapter and click the **Configure** button.
- Specify the URL of the **requestqueue** in the **Address (URI)** field in the **General** tab.

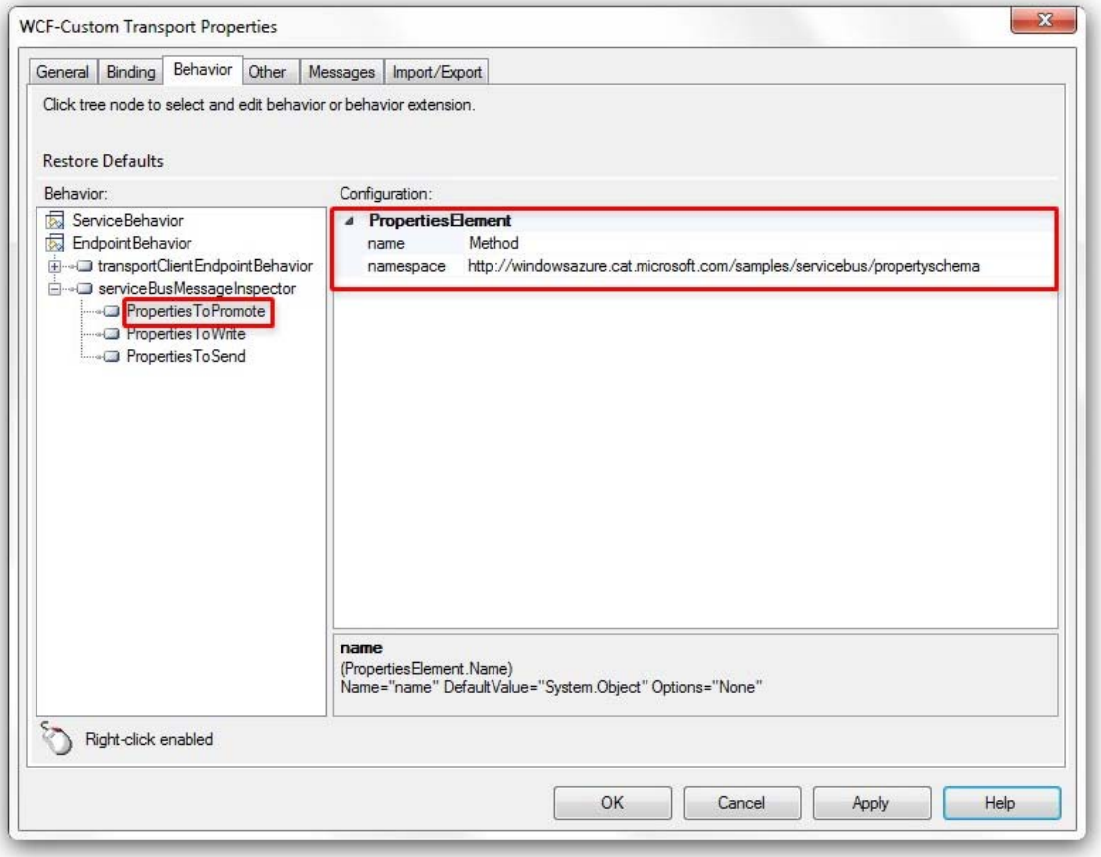
- Select the **Binding** tab, and choose the NetMessagingBinding from the **Binding Type** drop-down list. Then set the binding properties in the **Configuration** property grid.



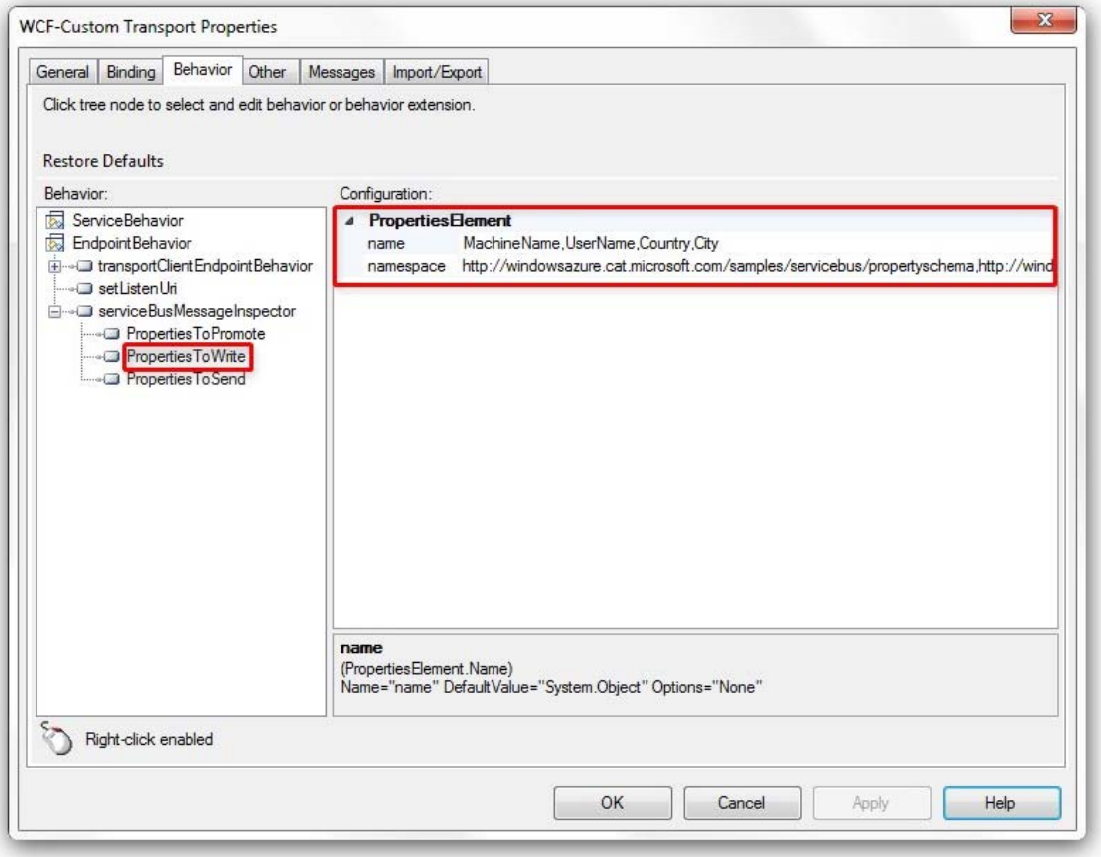
- Select the **Behavior** tab, then right click the **EndpointBehavior** node and select the **transportClientEndpointBehavior** component. Expand the corresponding node and enter the shared secret credentials for your service namespace that you previously retrieved from the **Windows Azure Management Portal**.



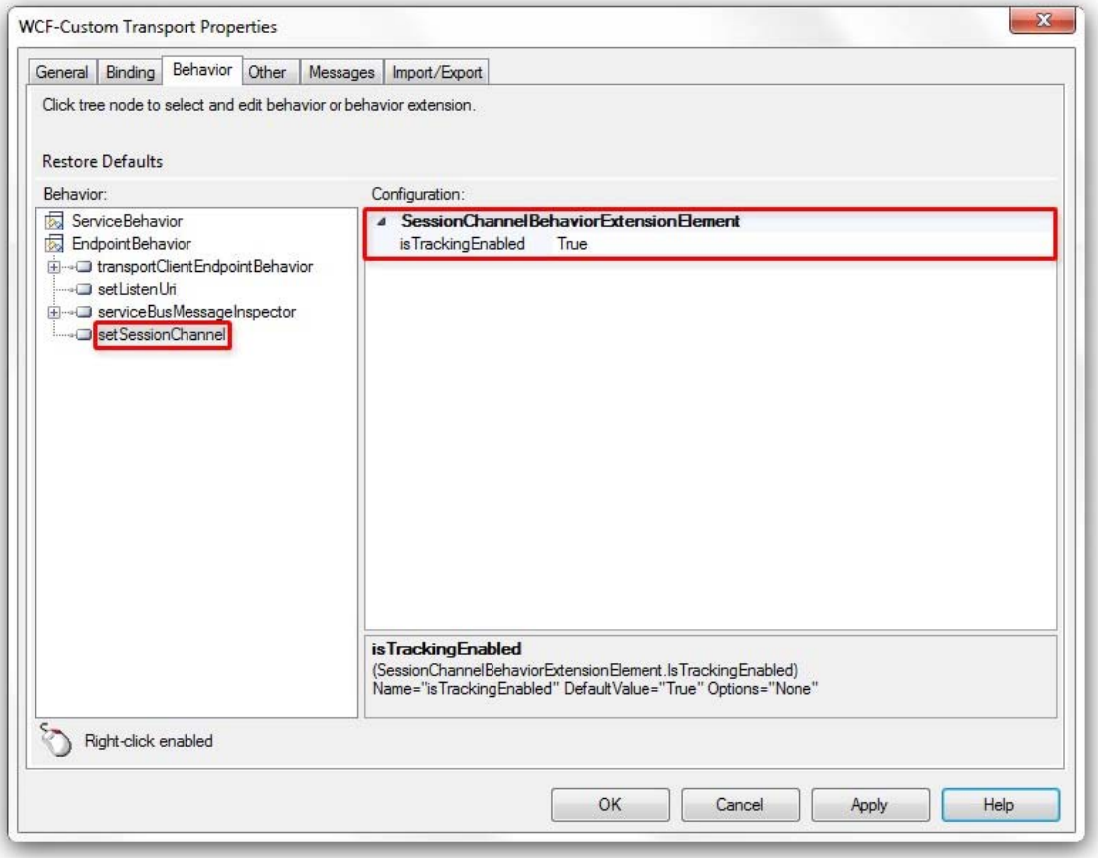
- Right click the **EndpointBehavior** node and select the **serviceBusMessageInspector** component. Then click the **PropertiesToPromote** node and specify the name and namespace lists of the user-defined properties that you want to promote. In our sample we need to promote the **Method** property, which is used in the subscription expression of **StaticSendPortOrchestration** and **DynamicSendPortOrchestration**.



- Click the **PropertiesToWrite** node, and specify the name and namespace lists of the user-defined properties that you want to promote to write to the message context. In our sample we define the following properties:
  - **MachineName**, **UserName**: these two properties are optional in our demo.
  - `Country`, `City`: since the client application uses a subscription whose filter expression is equal to **Country='Italy' and City='Milan'**, when the BizTalk application publishes the response to the **responsetopic**, the client must set the **Country** and **City** properties to **Italy** and **Milan** respectively,.in order for the client to receive the message For this reason, both **StaticSendPortOrchestration** and **DynamicSendPortOrchestration** copy the context properties, including **Country** and **City**, from the request to the response message.



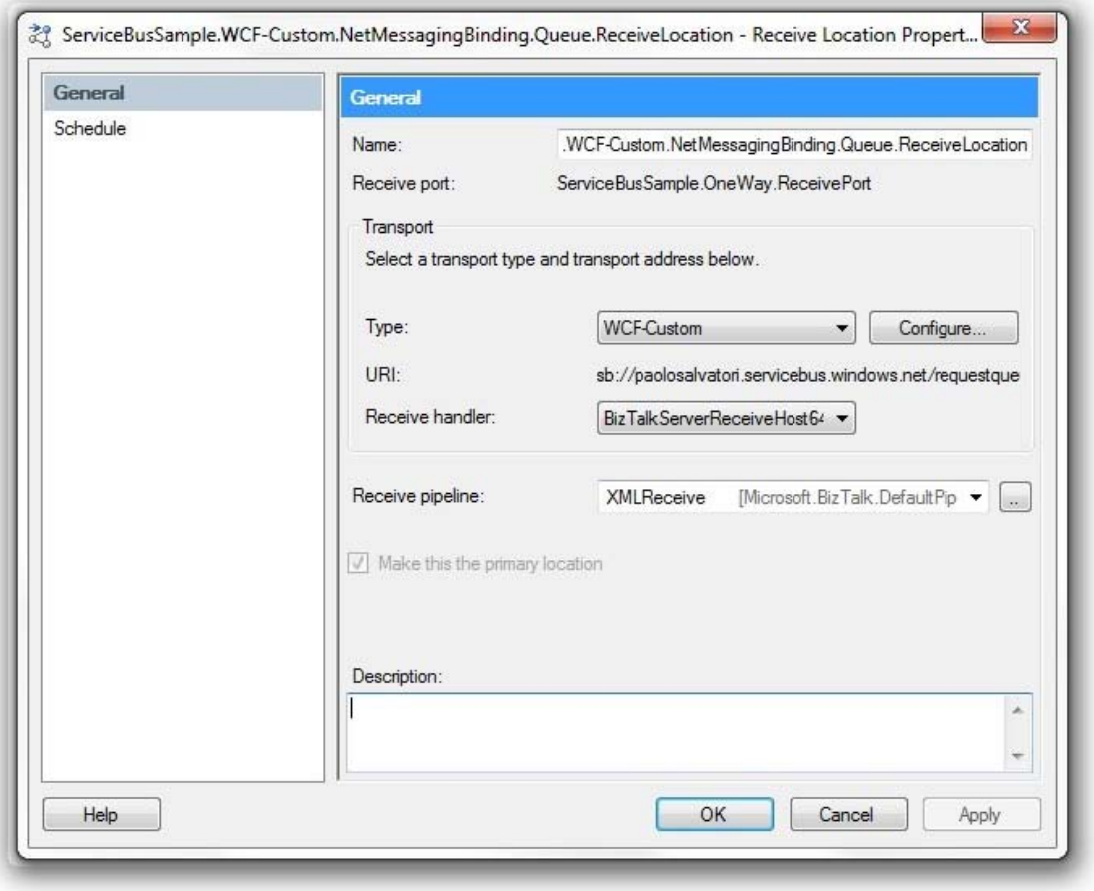
- Because the **PropertiesToSend** property applies only to send ports, it can be ignored when you are defining a WCF-Custom receive location.
- When the WCF-Custom receive location is configured to receive messages from a sessionful queue, you must perform an additional step: right click the **EndpointBehavior** node, and select the **setSessionChannel** component. At runtime, this component makes sure that the WCF-Custom adapter uses a session-aware channel (IInputSessionChannel) to receive messages from a sessionful queue or subscription. For more information, see the note at the end of this section



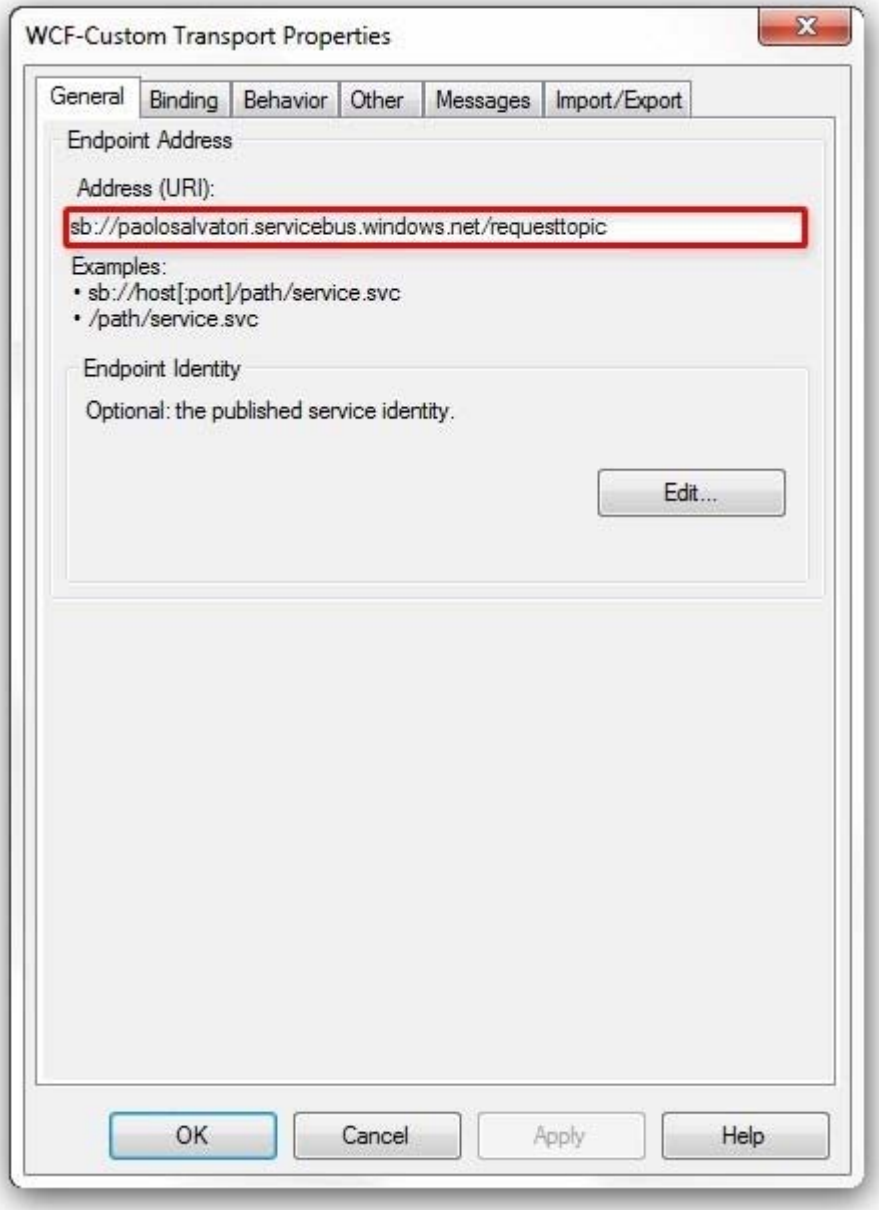
- Click the **OK** to close the WCF-Custom adapter configuration dialog.
- Click the **OK** button to finish creating the WCF-Custom receive location.

Let's now go through the steps to configure the WCF receive location used to read request messages from the **ItalyMilan** subscription of the **requesttopic**.

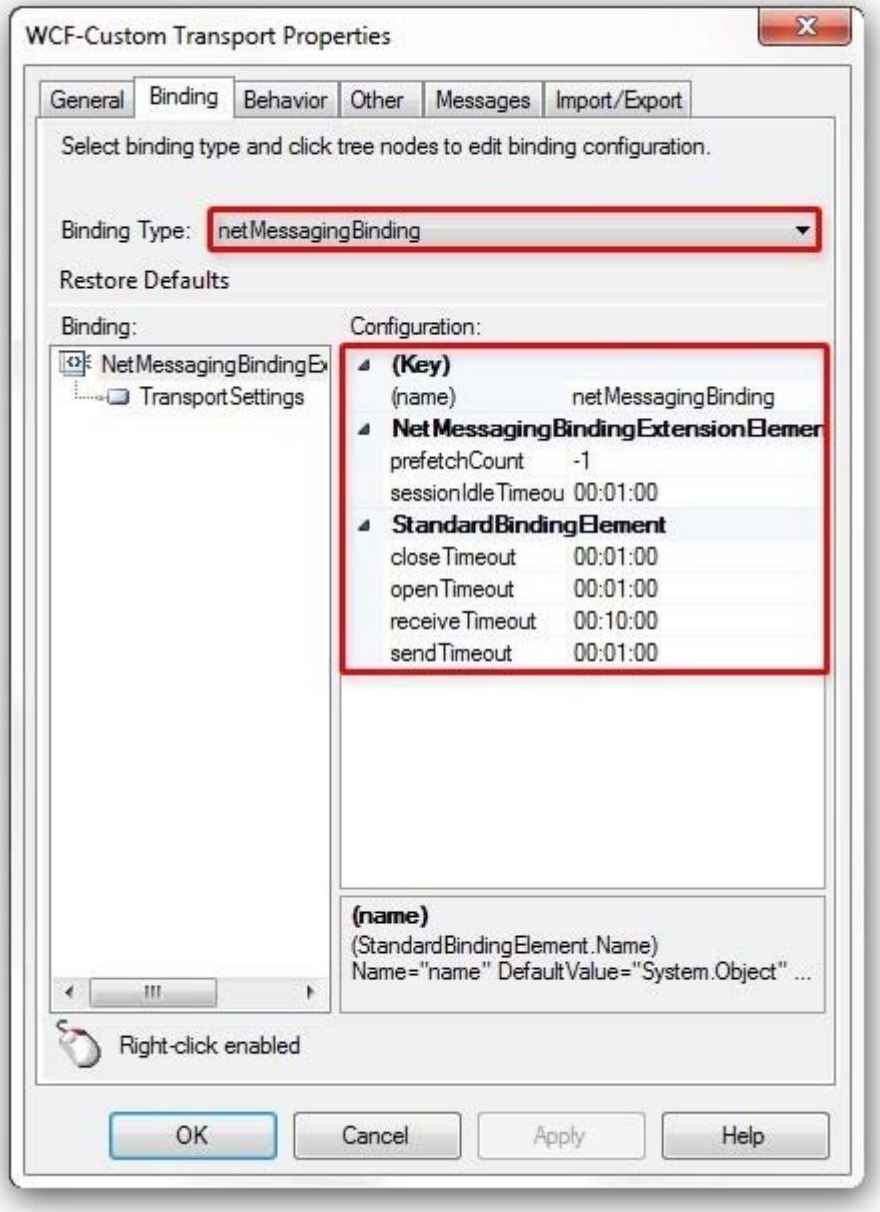
1. Open the **BizTalk Server Administration Console** and expand the **Receive Ports** node of the **ServiceBusSample** application.
2. Right-click the **ServiceBusSample.OneWay.ReceivePort**receive port and select **New Receive Location** from the context menu.
3. Specify a name for the receive location, and choose a pipeline based on your requirements. Since in our sample we want to process incoming requests using an orchestration, we choose the **XMLReceive** pipeline to promote the context properties of the inbound document like the MessageType that is used in the subscription expression of the **StaticSendPortOrchestration** and **DynamicSendPortOrchestration**. Then choose the **WCF-Custom** adapter and click the **Configure** button.



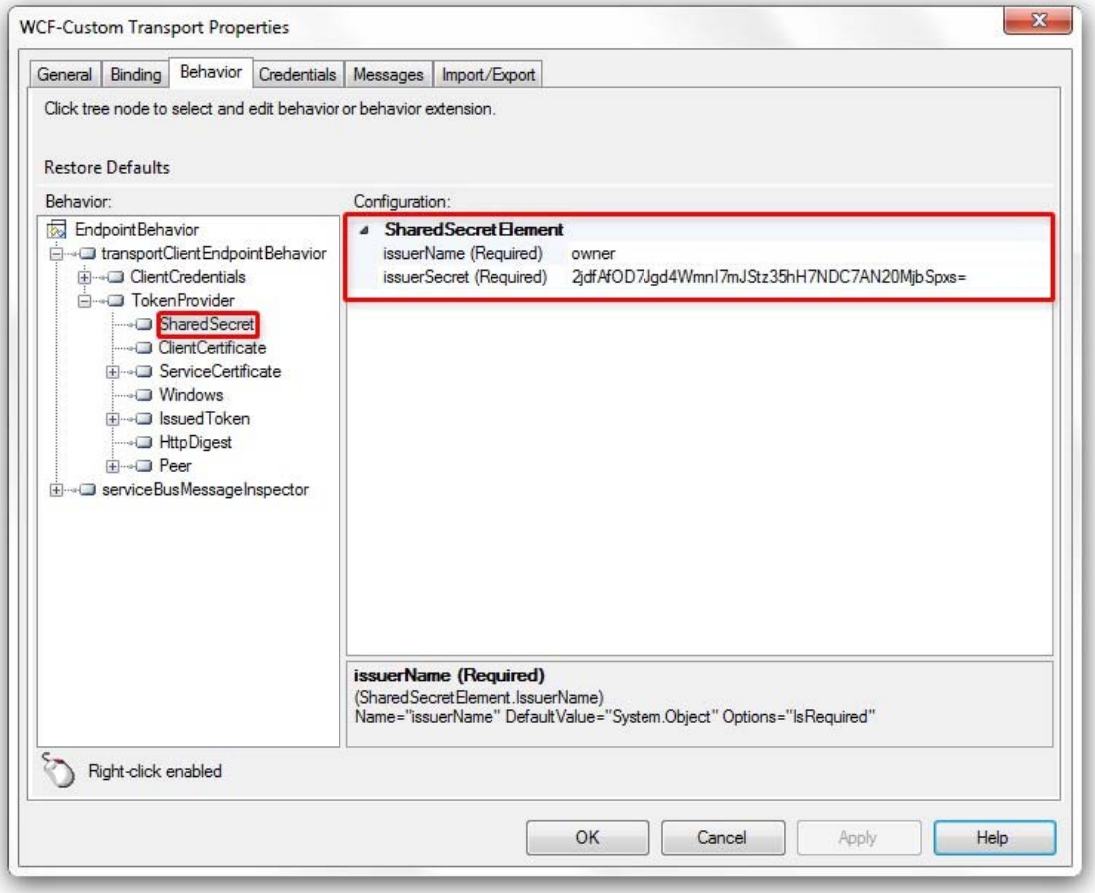
4. Specify the URL of the **requesttopic** in the **Address (URI)** field in the **General** tab.



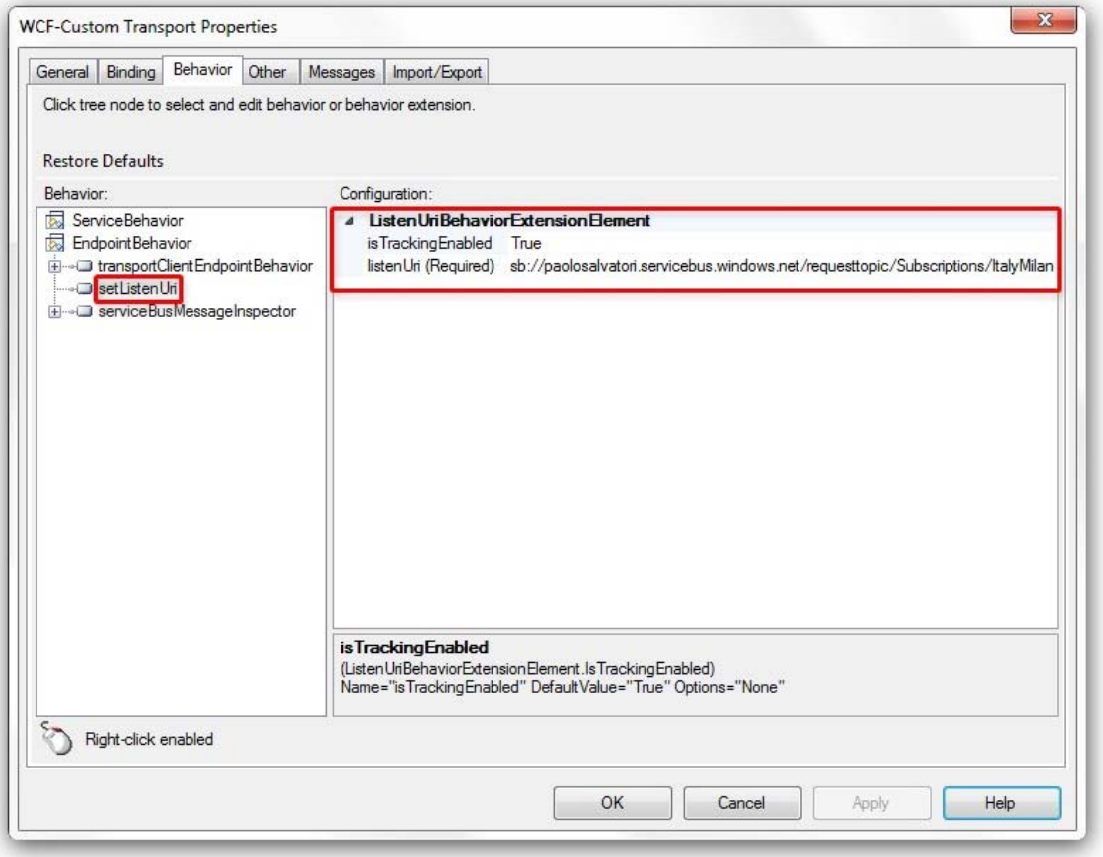
5. Select the **Binding** tab then choose NetMessagingBinding from the **Binding Type** drop-down list. Then set the binding properties in the **Configuration** property grid.



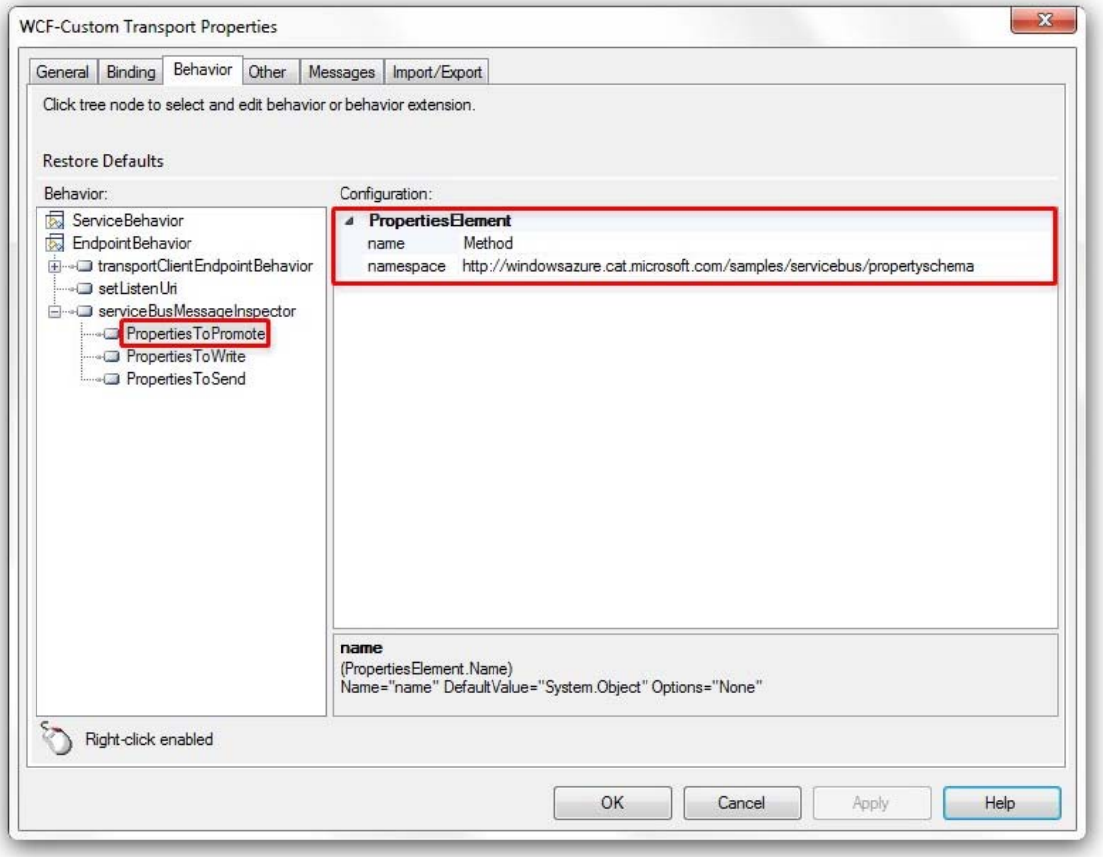
6. Select the **Behavior** tab then right click the **EndpointBehavior** node and select the **transportClientEndpointBehavior** component. Expand the corresponding node and enter the shared secret credentials for your service namespace that you previously retrieved from the **Windows Azure Management Portal**.



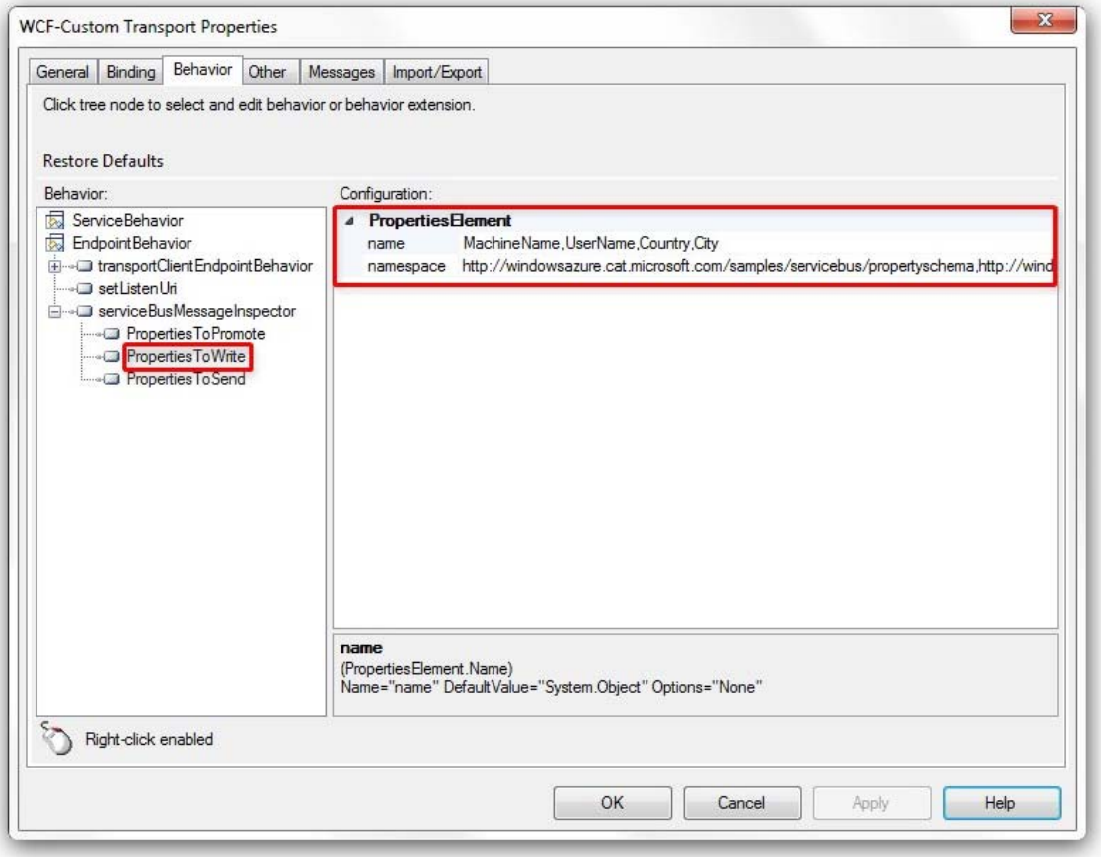
7. Right click the **EndpointBehavior** node and select the **setListenUri** component. Then click the **setListenUri** node in the **Behavior** panel and assign the URL of the **ItalyMilan** subscription to the **ListenUri** property in the **Configuration** property grid. For more information, see the note at the end of this section.



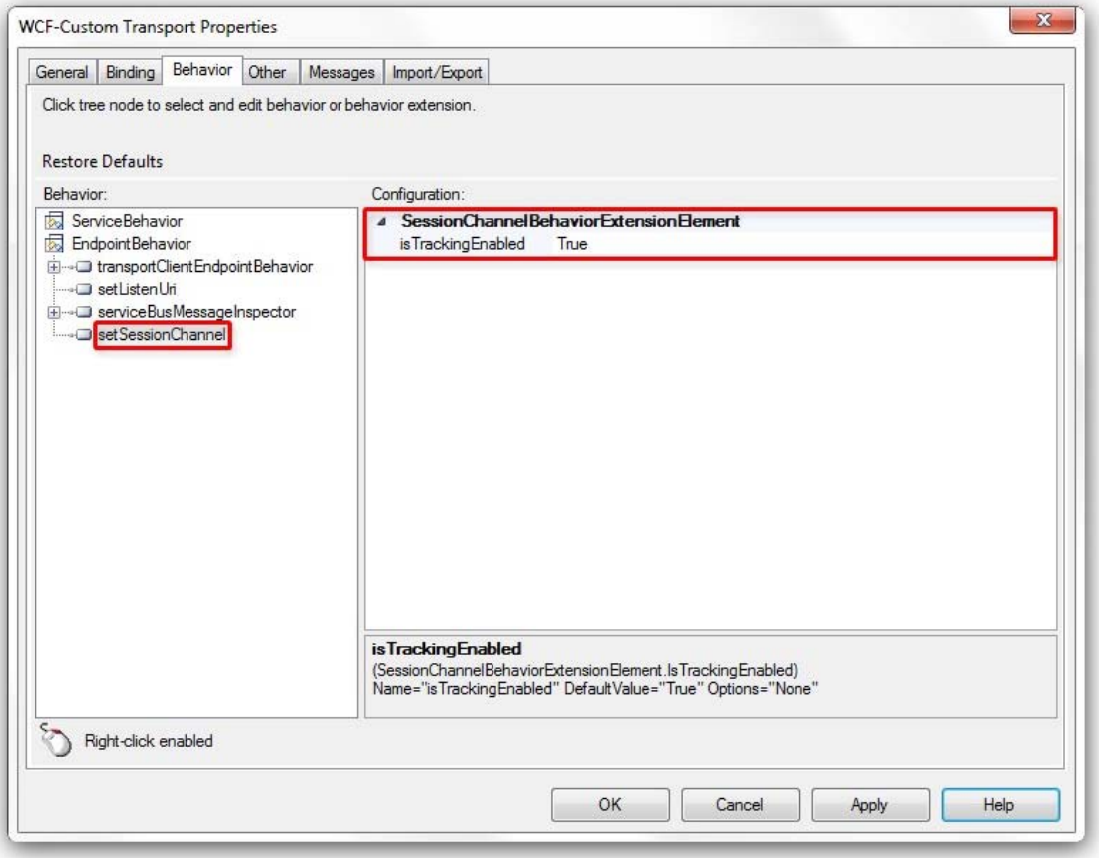
8. Right click the **EndpointBehavior** node and select the **serviceBusMessageInspector** component. Then click the **PropertiesToPromote** node and specify the name and namespace lists of the user-defined properties to promote. In our sample we need to promote the **Method** property, which is used in the subscription expression of the **StaticSendPortOrchestration** and **DynamicSendPortOrchestration**.



9. Click the **PropertiesToWrite** node and specify the name and namespace lists of the user-defined properties promote to write to the message context. In our sample we define the following properties:
   - **MachineName**, **UserName**: these two properties are optional in our demo.
   - **Country**, **City**: since the client application uses a subscription whose filter expression is equal to **Country='Italy' and City='Milan'**, when the BizTalk application publishes the response to the **responsetopic**, in order for the client to receive the message, the client must set the **Country** and **City** properties to **Italy** and **Milan**, respectively, in order for the client to receive the message. For this reason, both the **StaticSendPortOrchestration** and **DynamicSendPortOrchestration** copy the context properties, including **Country** and **City**, from the request to the response message.



10. Because **PropertiesToSend** applies only to send ports, it can be ignored when you define a WCF-Custom receive location.
11. When the WCF-Custom receive location is configured to receive messages from a sessionful subscription, you have to perform an additional step. Right click the **EndpointBehavior** node and select the **setSessionChannel** component. At runtime, this component makes sure that the WCF-Custom adapter uses a session-aware channel (IInputSessionChannel) to receive messages from a sessionful queue or subscription. For more information, see the note at the end of this section.



12. Click the **OK** button to close the WCF-Custom adapter configuration dialog.
13. Click the **OK** button to finish creating the WCF-Custom receive location.

Note

If you make the mistake of specifying the URL of a subscription instead of the URL of the topic to which the subscription belongs in the **Address (URI)** field of a WCF-Custom receive location, you will see an error like the following in the **Application Log**.

```
The adapter "WCF-Custom" raised an error message. Details
"System.ServiceModel.CommunicationException:
Internal Server Error: The server did not provide a meaningful reply; this might be
caused by a premature
session shutdown.TrackingId:1025c2be-5f49-4639-be01-4b2e5749c7ba, Timestamp:9/30/2011
8:43:47 AM
Server stack trace:
Exception rethrown at [0]:
   at Microsoft.ServiceBus.Common.AsyncResult.End[TAsyncResult](IAsyncResult result)
   at
Microsoft.ServiceBus.Messaging.Sbmp.DuplexRequestBindingElement.DuplexRequestSessionChann
el.
```

```
                DuplexCorrelationAsyncResult.End(IAsyncResult result)
   at Microsoft.ServiceBus.Messaging.Channels.
               ReconnectBindingElement.ReconnectChannelFactory`1.RequestSessionChannel.
                RequestAsyncResult.b__13(RequestAsyncResult thisPtr, IAsyncResult r)
   at Microsoft.ServiceBus.Messaging.IteratorAsyncResult`1.StepCallback(IAsyncResult
result)

Exception rethrown at [1]:
   at Microsoft.ServiceBus.Messaging.Channels.
                ServiceBusInputChannelBase`1.OnException(Exception exception)
   at Microsoft.ServiceBus.Messaging.Channels.
                ServiceBusInputChannelBase`1.OnEndTryReceive(IAsyncResult result,
                                                              BrokeredMessage&
brokeredMessage)
   at Microsoft.ServiceBus.Messaging.Channels.
                ServiceBusInputChannelBase`1.System.ServiceModel.Channels.
                IInputChannel.EndTryReceive(IAsyncResult result, Message& wcfMessage)
   at System.ServiceModel.Dispatcher.
                InputChannelBinder.EndTryReceive(IAsyncResult result, RequestContext&
requestContext)
   at System.ServiceModel.Dispatcher.
                ErrorHandlingReceiver.EndTryReceive(IAsyncResult result, RequestContext&
requestContext)".
```

Note

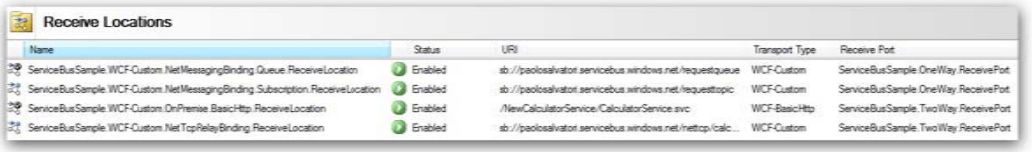
If you configure a WCF-Custom receive location to retrieve messages from a sessionful queue or subscription and you forget to add **setSessionChannel** to the list of endpoint behaviors used by the artifact, then, when you enable the receive location, you will see an error like the following in the **Application Log**:

```
The adapter "WCF-Custom" raised an error message. Details
"System.InvalidOperationException: It is not
possible for an entity that requires sessions to create a non-sessionful message
receiver..
TrackingId:9163a41f-d792-406d-acbe-eb93ab2defb8_1_1,TimeStamp:10/3/2011 12:39:02 PM --->
System.ServiceModel.FaultException`1[System.ServiceModel.ExceptionDetail]: It is not
possible for an entity
that requires sessions to create a non-sessionful message receiver..
TrackingId:9163a41f-d792-406d-acbe-eb93ab2defb8_1_1,TimeStamp:10/3/2011 12:39:02 PM
Server stack trace:
   at Microsoft.ServiceBus.Messaging.Sbmp.DuplexRequestBindingElement.
      DuplexRequestSessionChannel.ThrowIfFaultMessage(Message wcfMessage)
   at Microsoft.ServiceBus.Messaging.Sbmp.DuplexRequestBindingElement.
```

```
      DuplexRequestSessionChannel.HandleMessageReceived(IAsyncResult result)
Exception rethrown at [0]:
   at Microsoft.ServiceBus.Common.AsyncResult.End[TAsyncResult](IAsyncResult result)
   at Microsoft.ServiceBus.Messaging.Sbmp.DuplexRequestBindingElement.
      DuplexRequestSessionChannel.DuplexCorrelationAsyncResult.End(IAsyncResult result)
   at
Microsoft.ServiceBus.Messaging.Channels.ReconnectBindingElement.ReconnectChannelFactory`1
.
      RequestSessionChannel.RequestAsyncResult.<GetAsyncSteps>b__13(RequestAsyncResult
thisPtr, IAsyncResult r)
   at Microsoft.ServiceBus.Messaging.IteratorAsyncResult`1.StepCallback(IAsyncResult
result)
Exception rethrown at [1]:
   at Microsoft.ServiceBus.Common.AsyncResult.End[TAsyncResult](IAsyncResult result)
   at Microsoft.ServiceBus.Messaging.Channels.ReconnectBindingElement.
      ReconnectChannelFactory`1.RequestSessionChannel.EndRequest(IAsyncResult result)
   at Microsoft.ServiceBus.Messaging.Sbmp.SbmpMessageReceiver.
      EndReceiveCommand(IAsyncResult result, IEnumerable`1& messages)
   --- End of inner exception stack trace ---
Server stack trace:
   at Microsoft.ServiceBus.Messaging.Sbmp.SbmpMessageReceiver.
      EndReceiveCommand(IAsyncResult result, IEnumerable`1& messages)
   at Microsoft.ServiceBus.Messaging.IteratorAsyncResult`1.StepCallback(IAsyncResult
result)
Exception rethrown at [0]:
   at Microsoft.ServiceBus.Common.AsyncResult.End[TAsyncResult](IAsyncResult result)
   at Microsoft.ServiceBus.Messaging.Sbmp.SbmpMessageReceiver.
      OnEndTryReceive(IAsyncResult result, IEnumerable`1& messages)
   at Microsoft.ServiceBus.Messaging.MessageReceiver.EndTryReceive(IAsyncResult result,
IEnumerable`1& messages)
   at Microsoft.ServiceBus.Messaging.Channels.ServiceBusInputChannelBase`1.
      OnEndTryReceive(IAsyncResult result, BrokeredMessage& brokeredMessage)
   at
Microsoft.ServiceBus.Messaging.Channels.ServiceBusInputChannelBase`1.System.ServiceModel.
Channels.
      IInputChannel.EndTryReceive(IAsyncResult result, Message& wcfMessage)
   at System.ServiceModel.Dispatcher.InputChannelBinder.
      EndTryReceive(IAsyncResult result, RequestContext& requestContext)
   at System.ServiceModel.Dispatcher.ErrorHandlingReceiver.
      EndTryReceive(IAsyncResult result, RequestContext& requestContext)".
```

The following picture shows the two receive locations we just created along with two other request-response receive locations used that use the BasicHttpBinding and the

NetTcpRelayBinding. As we said in the Scenarios section, the description of these two receive locations is out of the scope of this article.

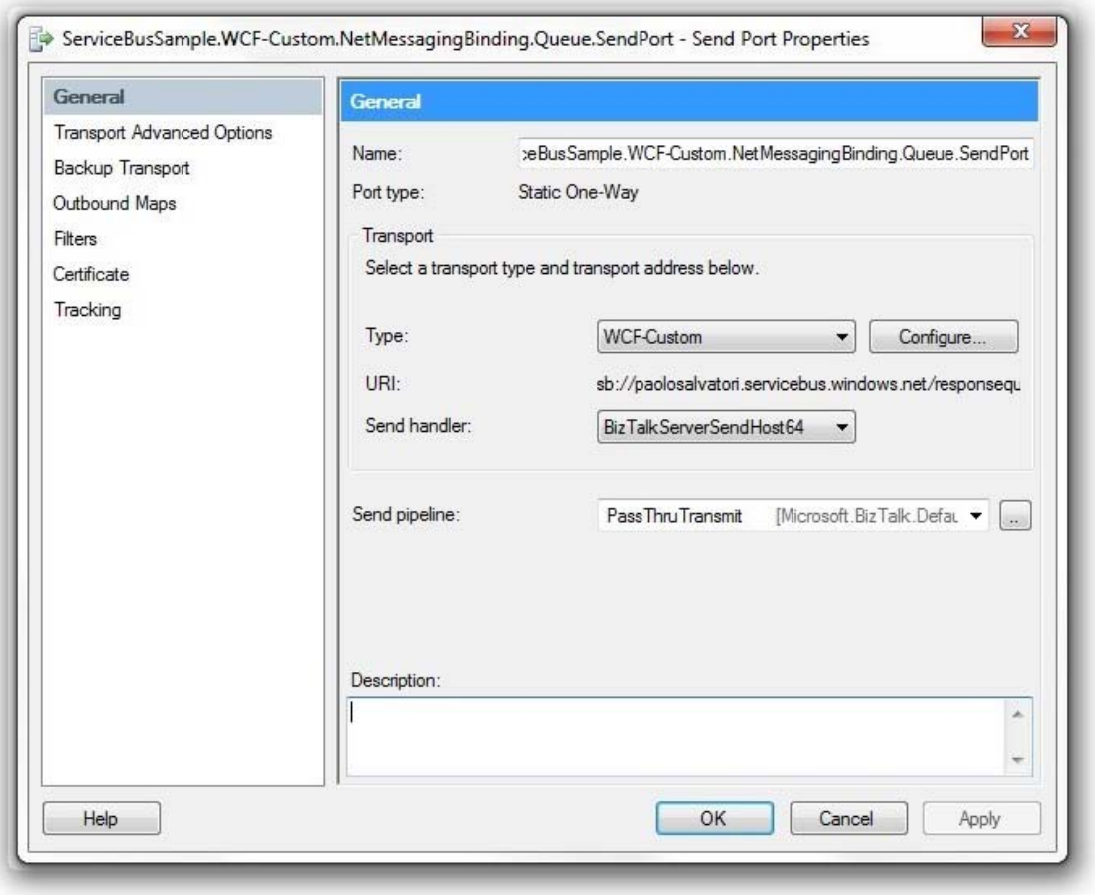


Send Ports

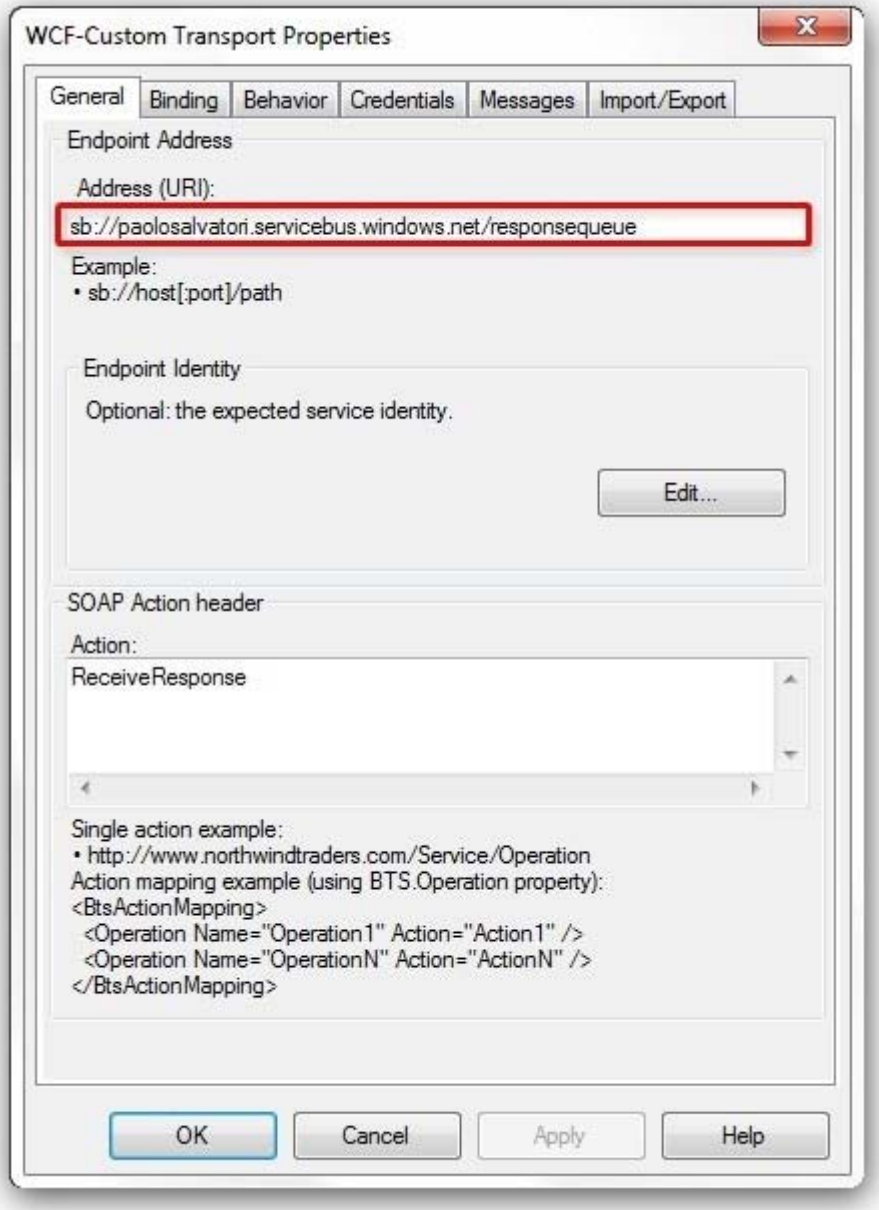
In this section we will describe the steps necessary to create the WCF-Custom send ports used by the **StaticSendPortOrchestration** and **DynamicSendPortOrchestration** to send response messages to the **requestqueue** and **requesttopic**. In particular, the **StaticSendPortOrchestration** uses two static one-way send ports, one for each messaging entity, while the **DynamicSendPortOrchestration** uses a dynamic send port to decide at runtime, based on the address specified by the client application and contained in the **ReplyTo** context property, where to send the response message.

Let's start by looking at the steps necessary to create the WCF send port used by the **StaticSendPortOrchestration** to send response messages to the **responsequeue**.

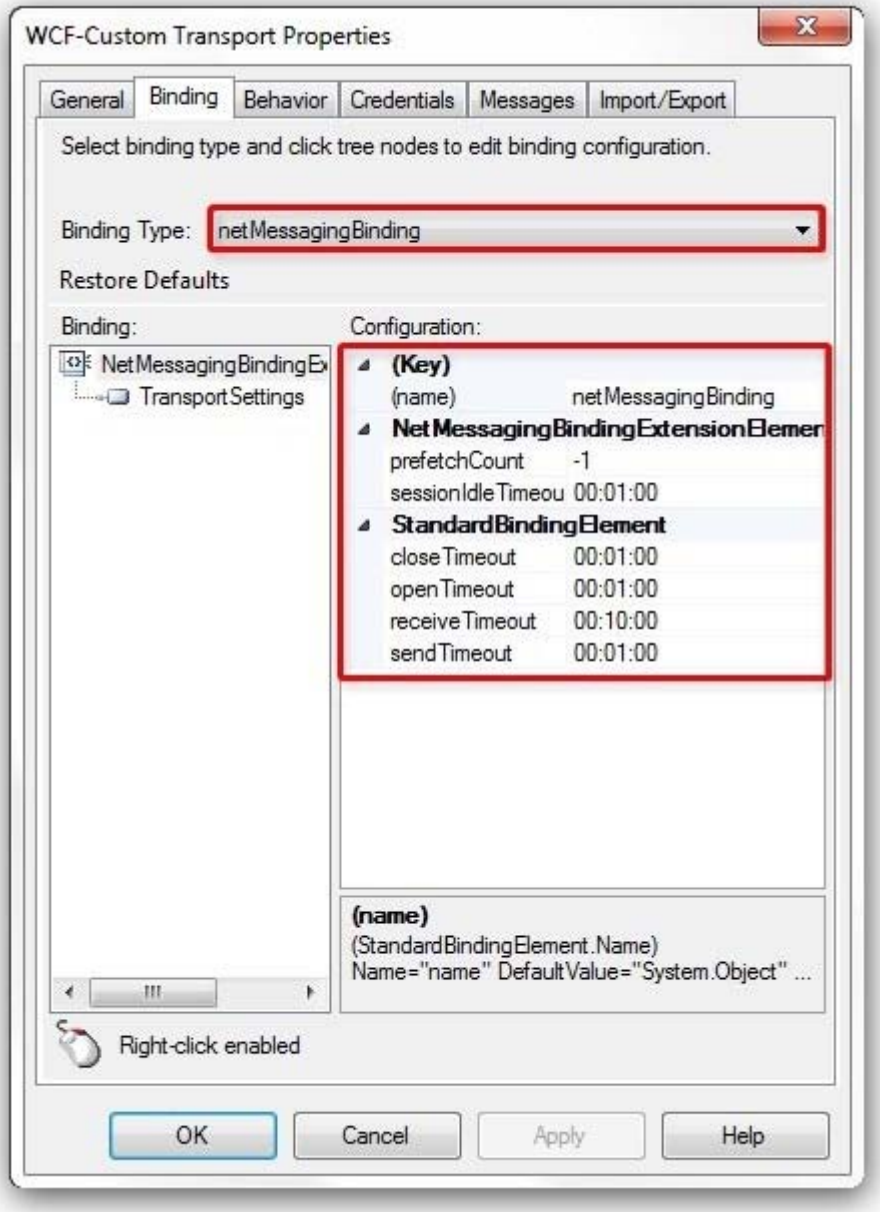
- Open the **BizTalk Server Administration Console** and expand the **Send Ports** node of the **ServiceBusSample** application
- Create a new **Static One-way Send Port** as shown in the figure below. Specify a name for the send port and choose a pipeline based on your requirements. In this demo, we will use the **PassThruTransmit** pipeline for performance reasons. Then choose the **WCF-Custom** adapter and click the **Configure** button.



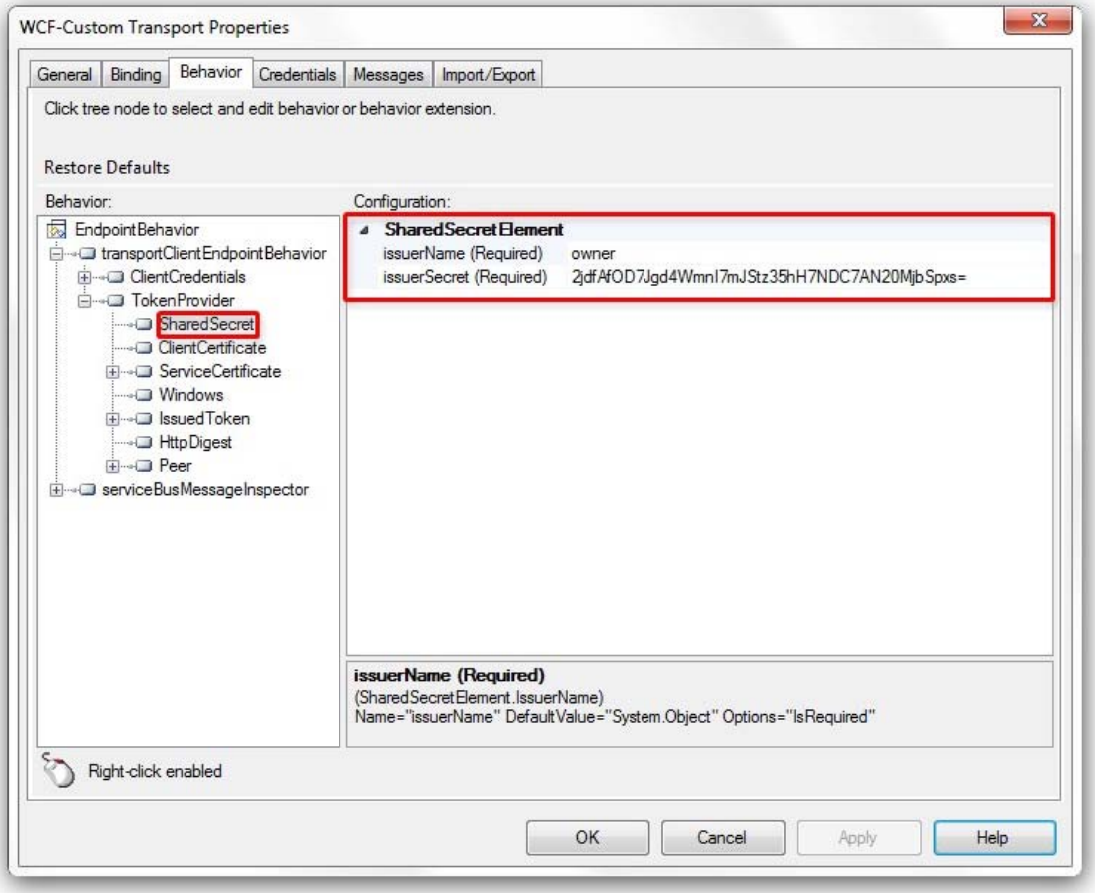
- Specify the URL of the **responsequeue** in the **Address (URI)** field in the **General** tab.



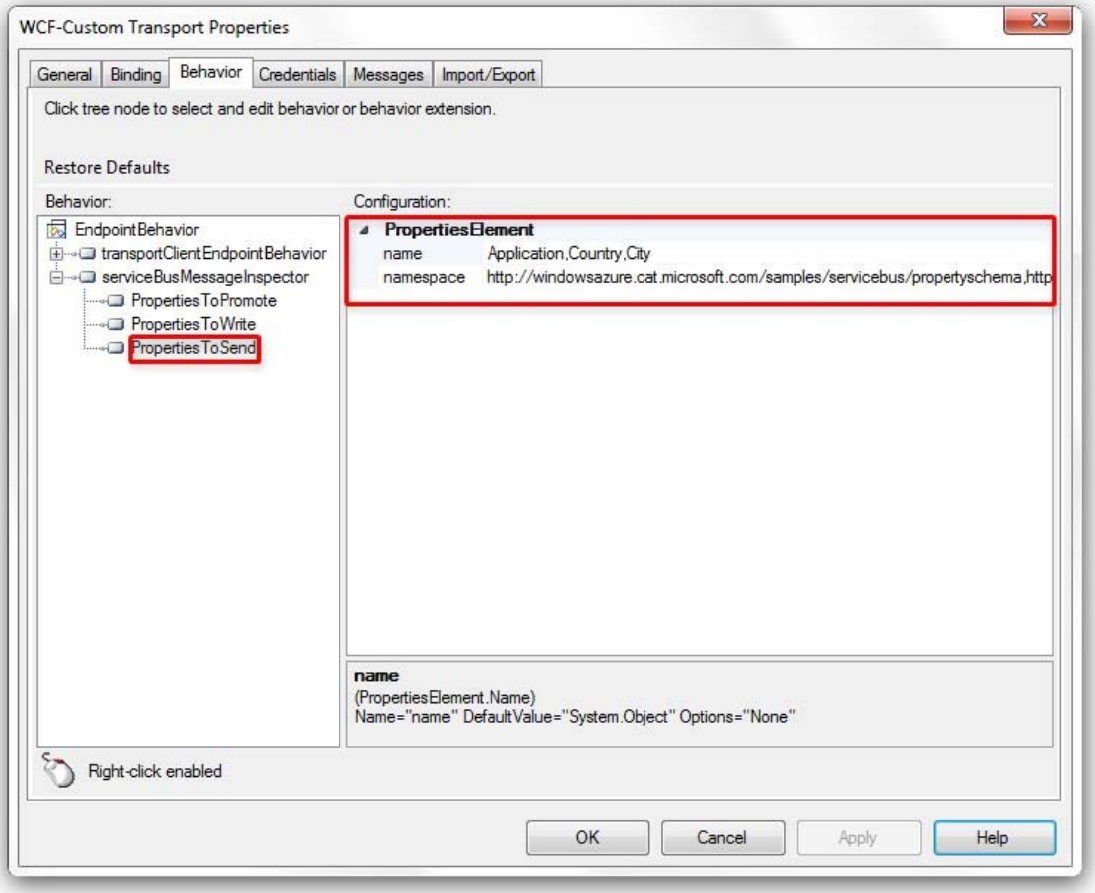
- Select the **Binding** tab, and then choose NetMessagingBinding from the **Binding Type** drop-down list. Then set the binding properties in the **Configuration** property grid.



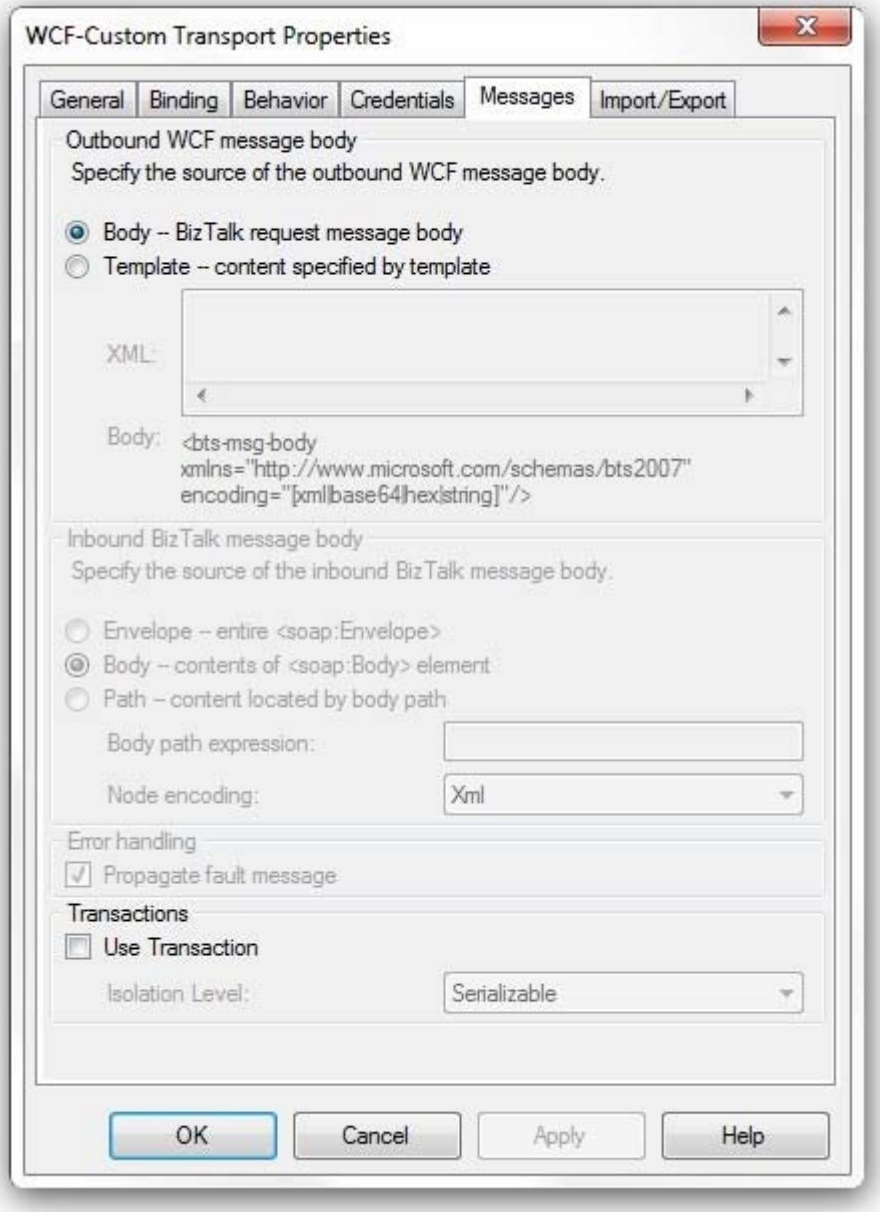
- Select the **Behavior** tab then right click the **EndpointBehavior** node and select the **transportClientEndpointBehavior** component. Expand the corresponding node and enter the shared secret credentials for your service namespace that you previously retrieved from the Windows Azure Platform Management Portal.



- Right click the **EndpointBehavior** node and select the **serviceBusMessageInspector** component. Then click the **PropertiesToSend** node and specify the name and namespace lists of the user-defined properties to include in the outgoing message. In our sample, the **ServiceBusSample** application includes the **Application**, **Country** and **City** in the outgoing Service Bus message. In particular, the use of the **Application** context property demonstrates how a BizTalk application can send context information out-of-band to another application using a Service Bus message. Please note that in this case the use of the **serviceBusMessageInspector** is not strictly required for routing purposes, as the client would be able to retrieve the message from the **responsequeue** even if the message did not contain the **Application**, **Country** and **City** properties.



- Select the **Messages** tab then uncheck **Use Transaction** in the **Transactions** section.

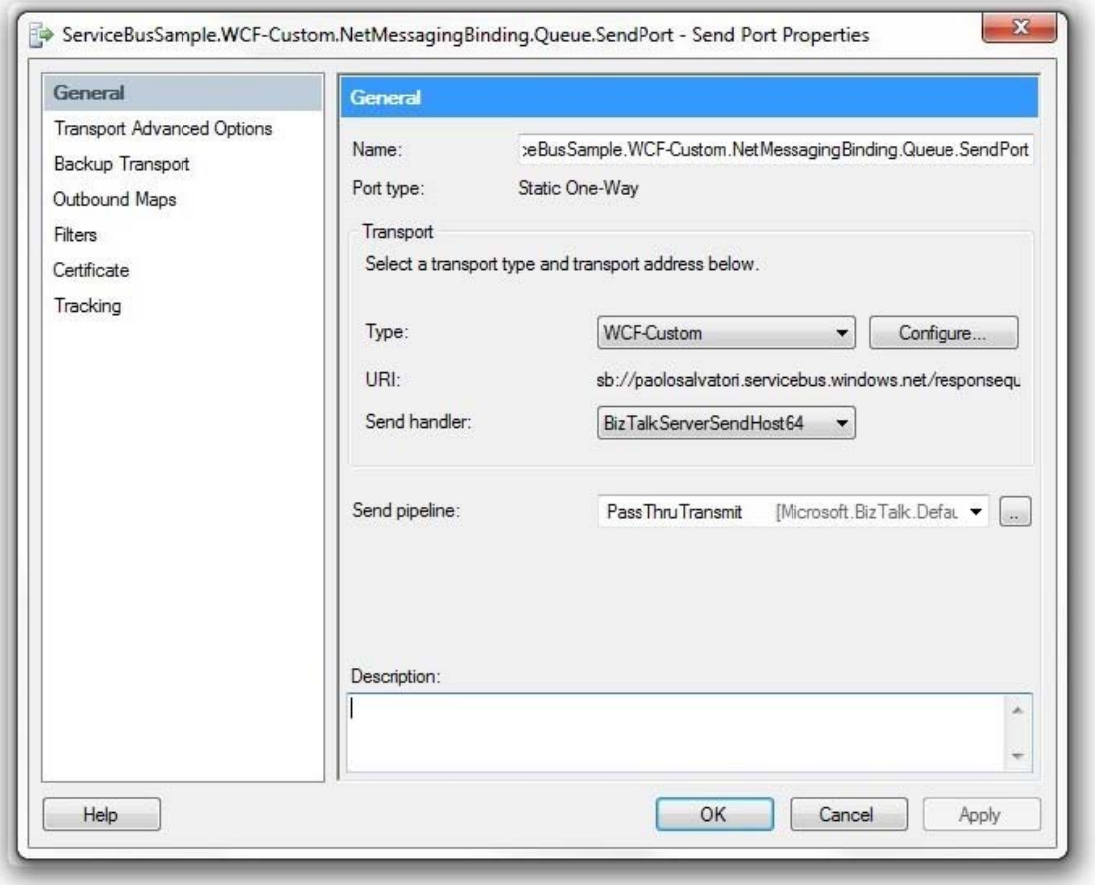


- Press the **OK** button to close the **WCF-Custom** adapter configuration dialog.
- Click the **OK** button to complete the creation of the **WCF-Custom** send port.

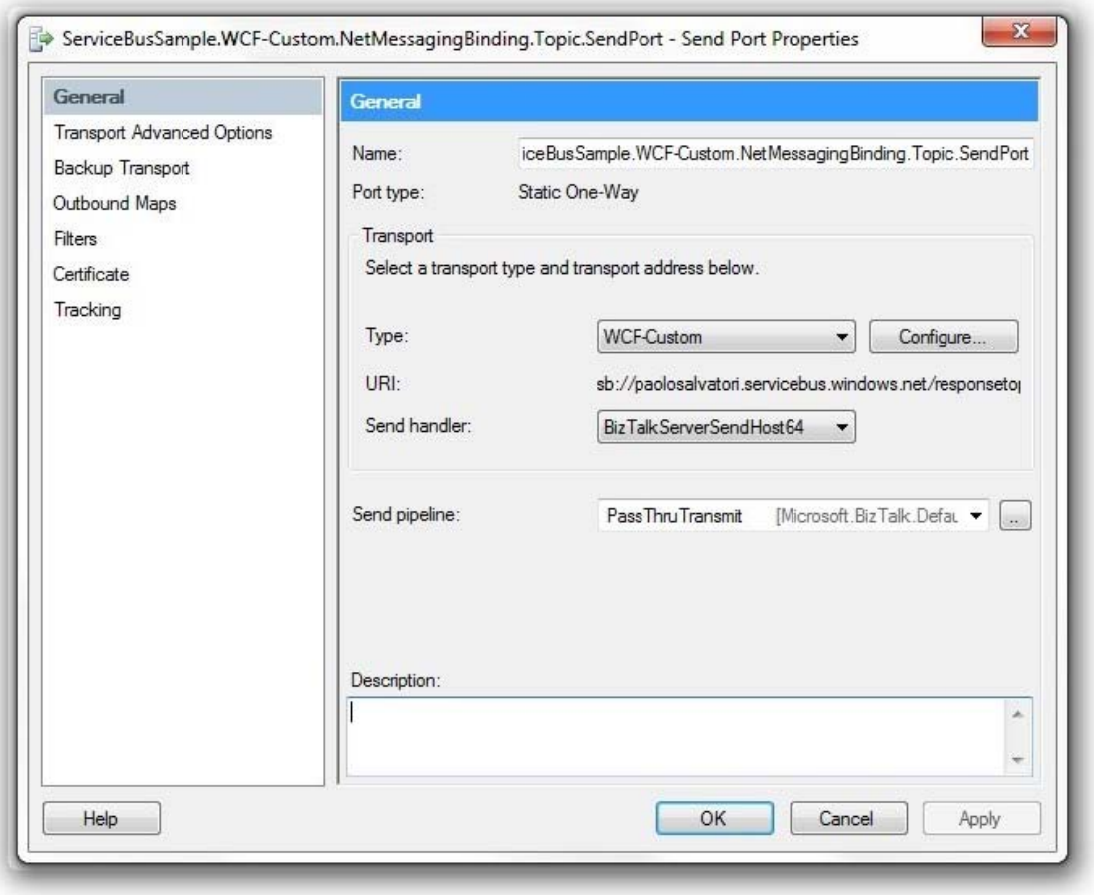
Now, let's go through the steps to create the WCF send port used by the **StaticSendPortOrchestration** to send response messages to the **responsetopic**.

- Open the **BizTalk Server Administration Console** and expand the **Send Ports** node of the **ServiceBusSample** application.
- Create a new static one-way send port as shown in the figure below. Specify a name for the send port and choose a pipeline based on your requirements. In this demo, we will use the

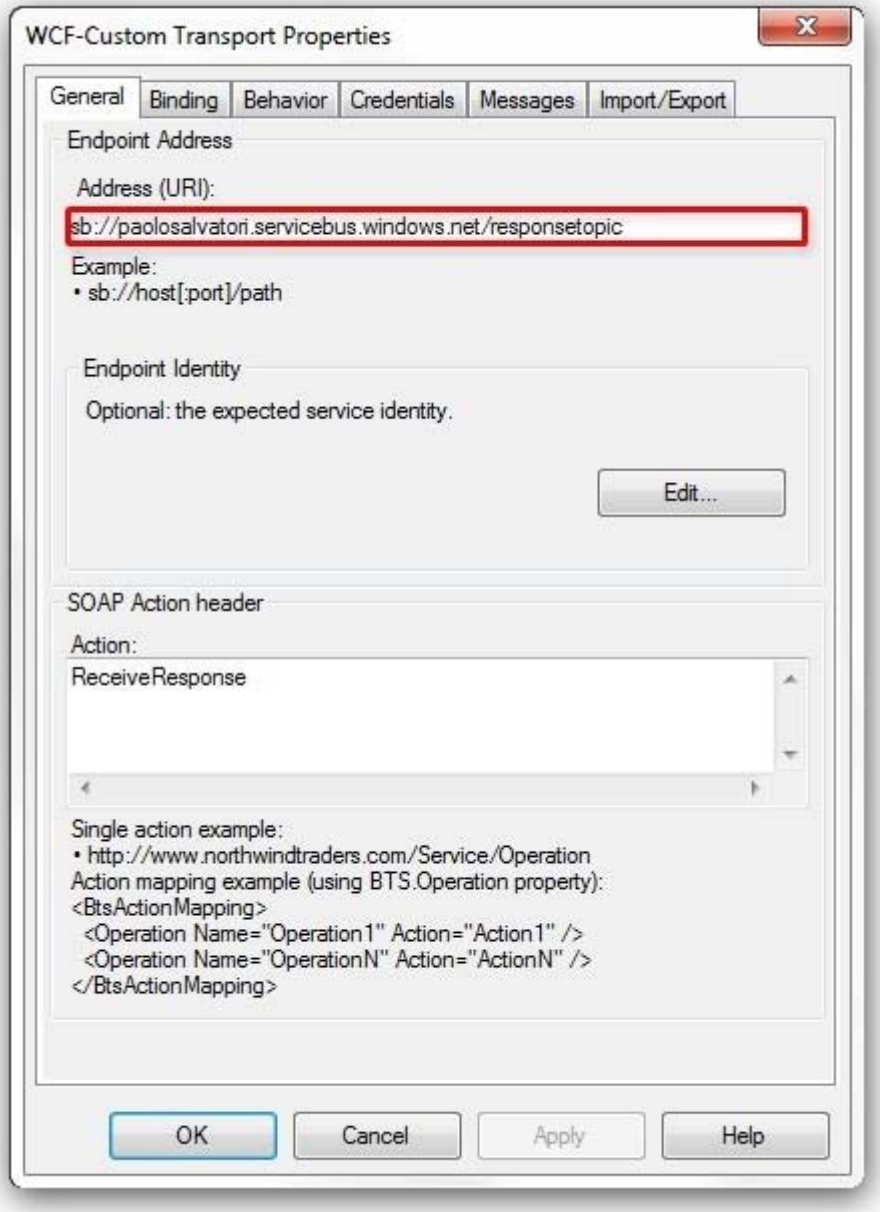
PassThruTransmit pipeline for performance reasons. Then choose the **WCF-Custom** adapter and click the **Configure** button.



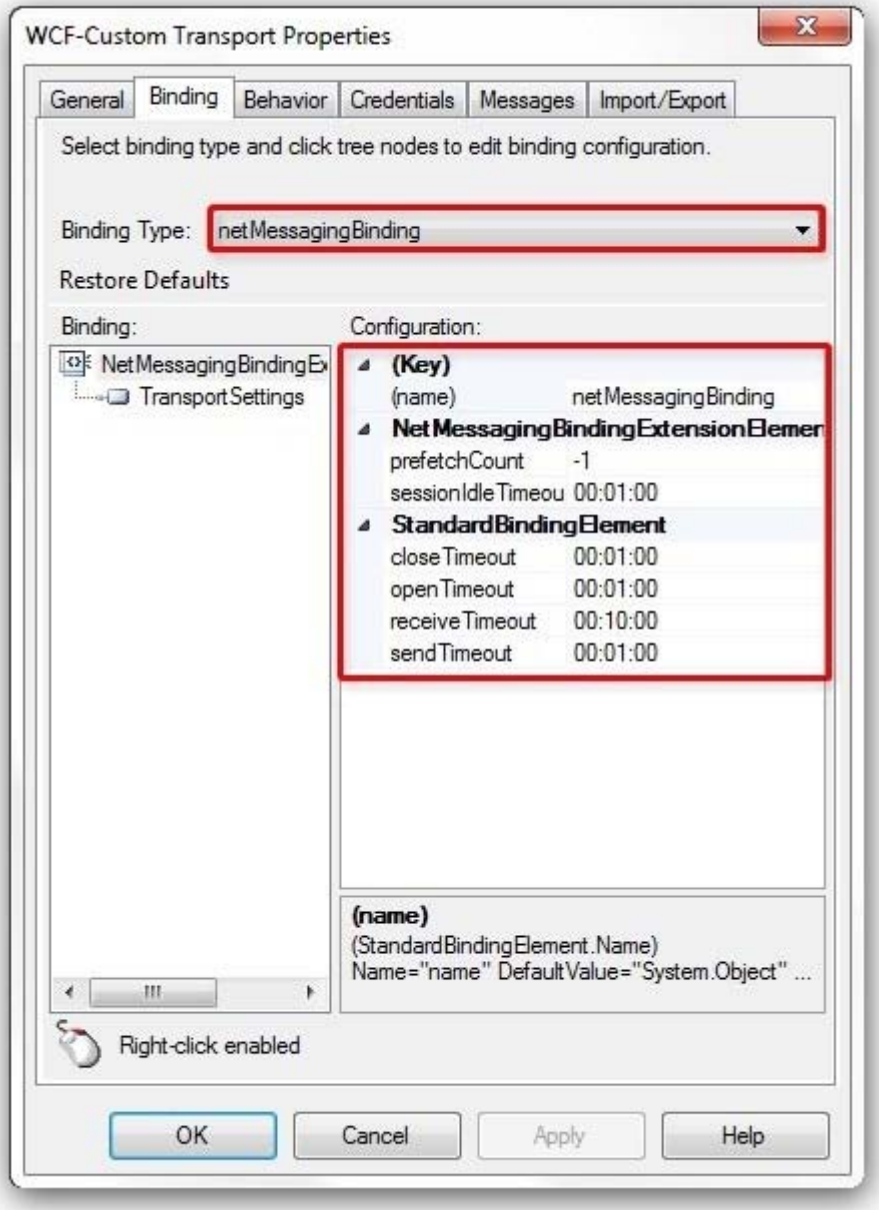
- Specify the URL of the **responsetopic** in the **Address (URI)** field in the **General** tab.



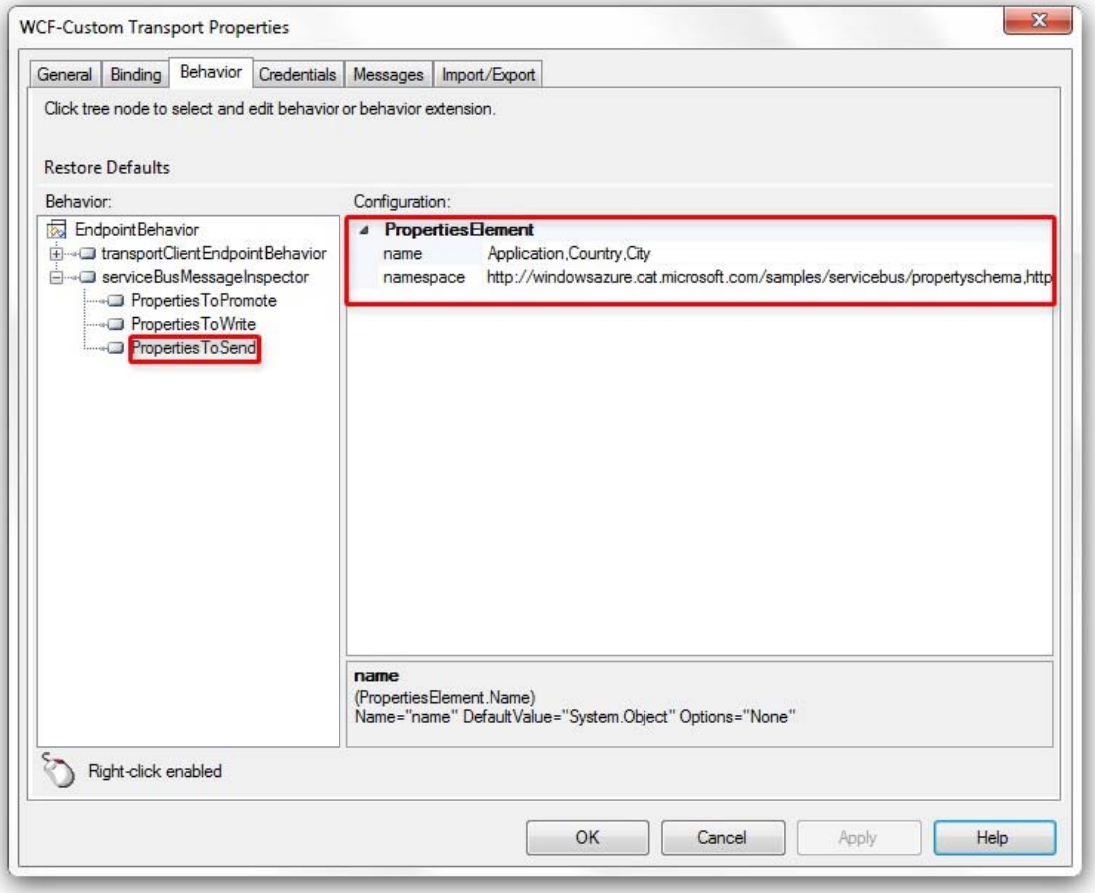
- Select the **Binding** tab then choose NetMessagingBinding from the **Binding Type** drop-down list. Then set the binding properties in the **Configuration** property grid.



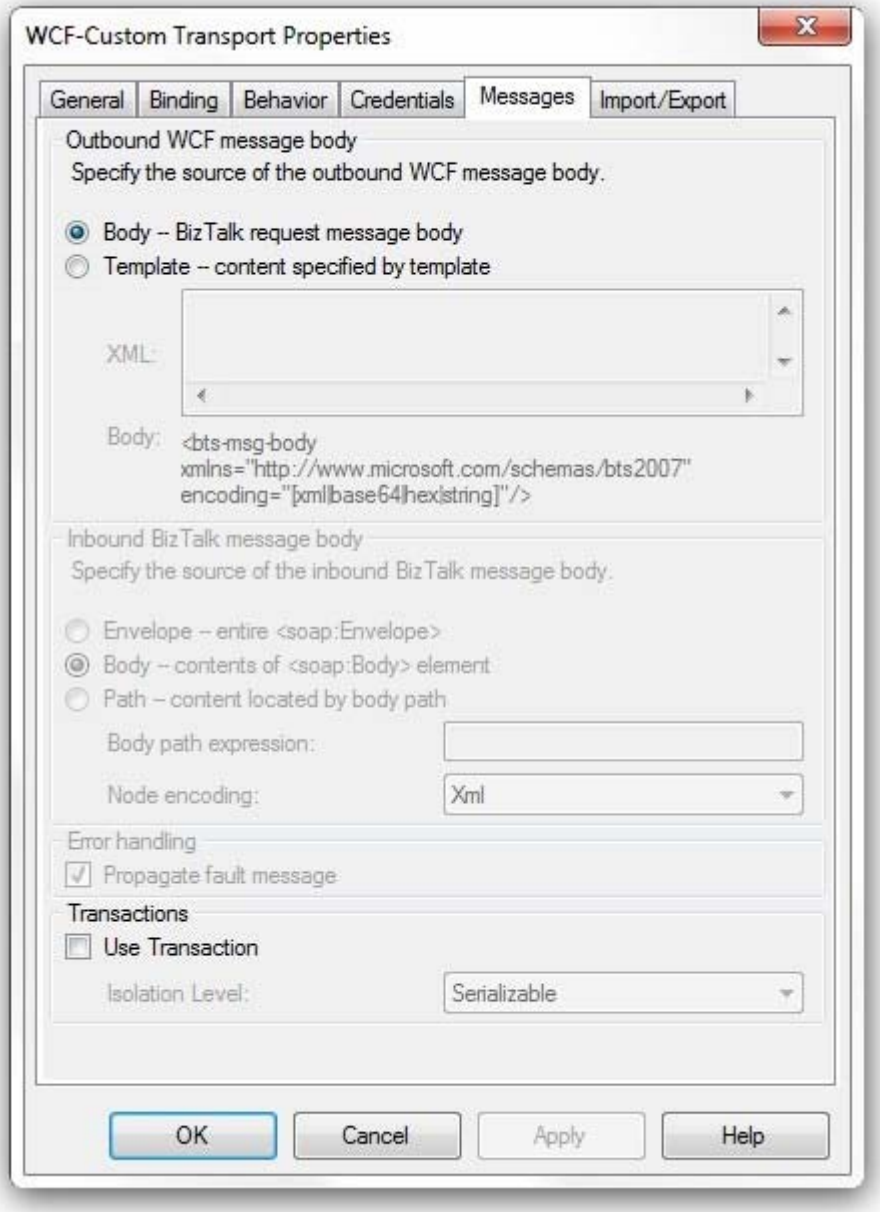
- Select the **Behavior** tab then right click the **EndpointBehavior** node and select the **transportClientEndpointBehavior** component. Expand the corresponding node and enter the shared secret credentials for your service namespace that you previously retrieved from the Windows Azure Platform Management Portal.



- Right click the **EndpointBehavior** node and select the **serviceBusMessageInspector** component. Then click the **PropertiesToSend** node and specify the name and namespace lists of the user-defined properties to include in the outgoing message. In our sample, the **ServiceBusSample** application includes the values of the **Application**, **Country**, and **City** properties in the outgoing Service Bus message. In particular, the use of the **Application** context property demonstrates how a BizTalk application can send context information out-of-band to another application through a Service Bus message. Please note that in this case the use of the **serviceBusMessageInspector** is not strictly required for routing purposes as the client would be able to retrieve the message from the **responsequeue** even if the message did not contain the **Application**, **Country**, and **City** properties.



- Select the **Messages** tab then uncheck **Use Transaction** in the **Transactions** section.

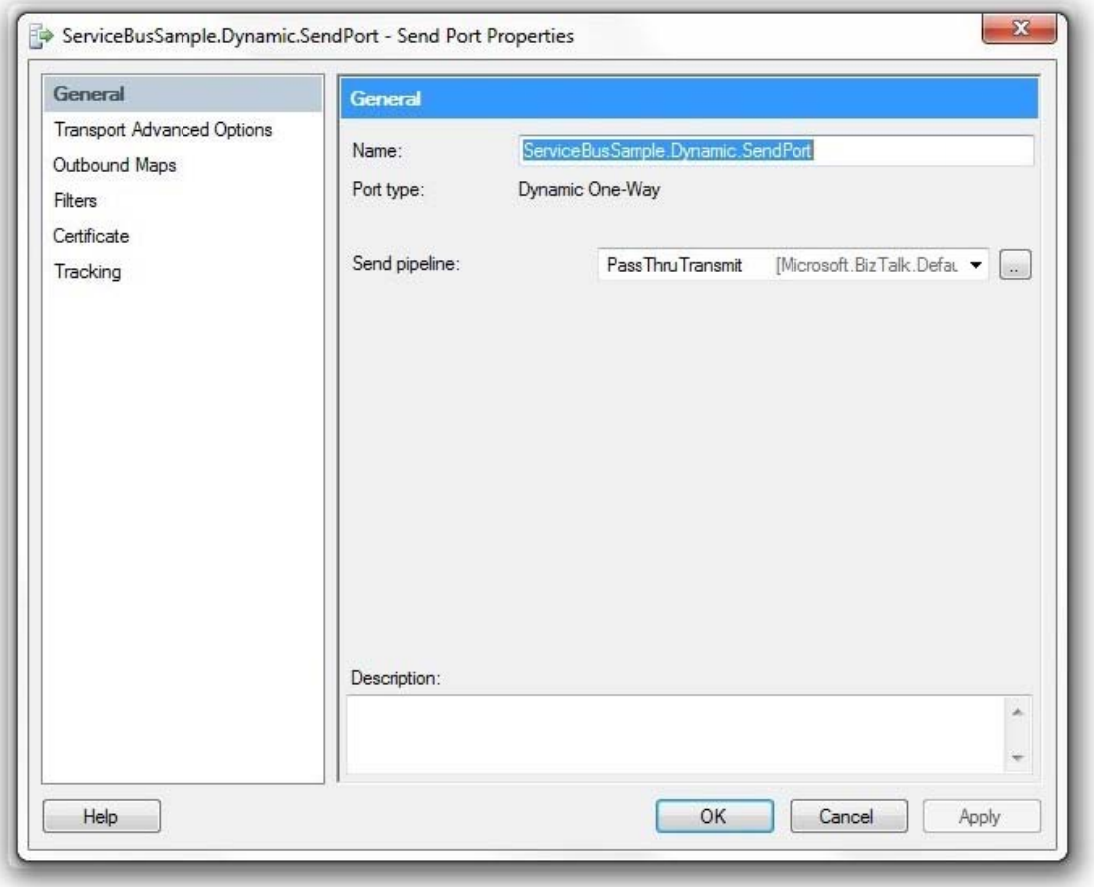


- Click the **OK** button to close the WCF-Custom adapter configuration dialog.
- Click the **OK** button to finish creating the WCF-Custom send port.

As the last step, let's create the dynamic send port used by the **DynamicSendPortOrchestration** to send messages to the **responsequeue** or **responsetopic**, based on the URL specified by the client application in the **ReplyTo** property of the Service Bus message.

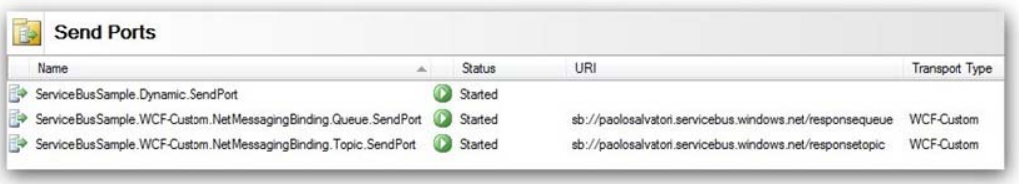
- Open the BizTalk Server Administration Console and expand the **Send Ports** node of the **ServiceBusSample** application.

- Create a new dynamic one-way send port as shown in the figure below. Specify a name for the send port and choose a pipeline based on your requirements. In this demo, we will use the **PassThruTransmit** pipeline for performance reasons.



- Click the **OK** button to complete the creation of the WCF-Custom send port.

The following picture shows the three send ports we created in the **BizTalk Server Administration Console**.



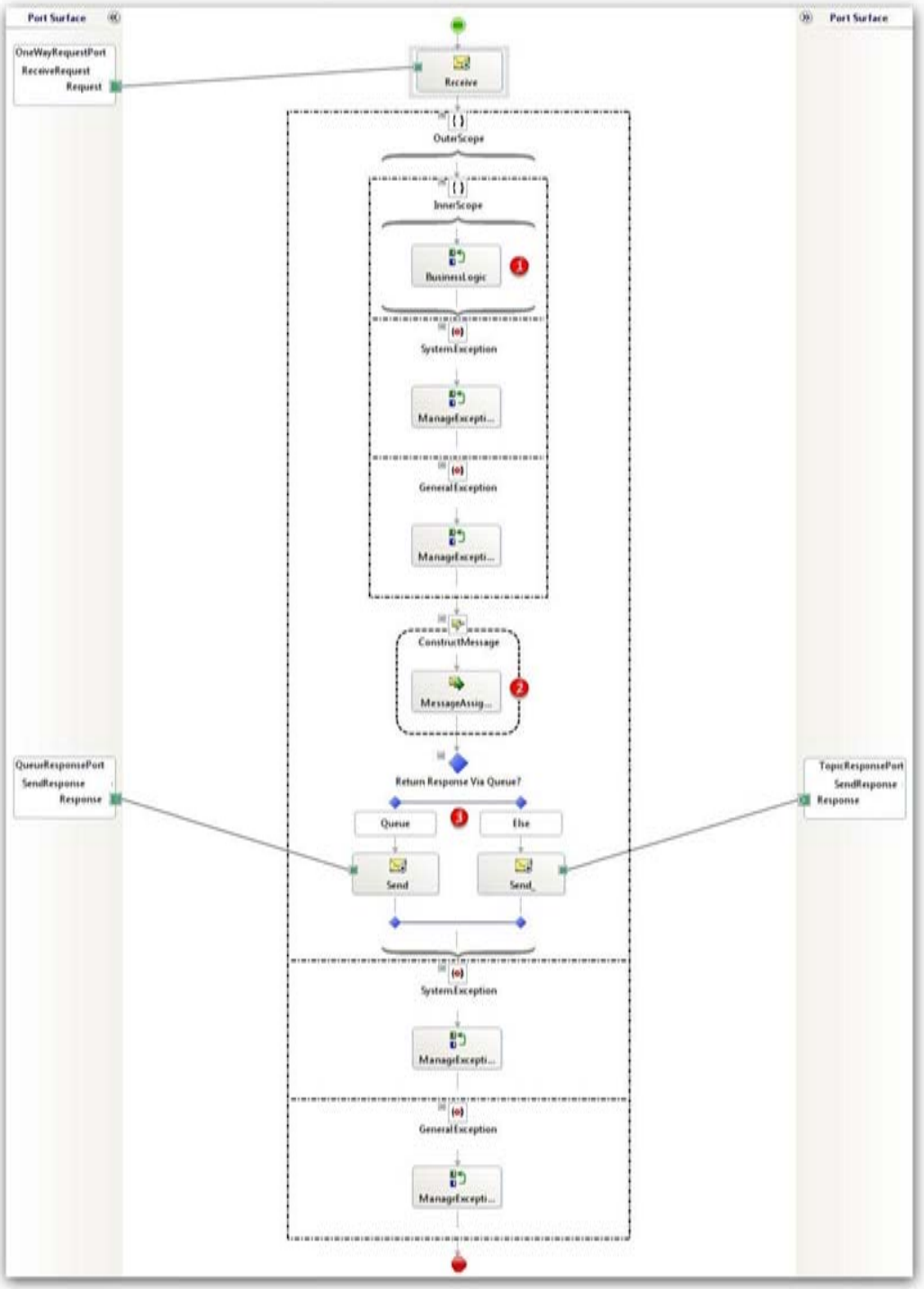
Orchestrations

In this section we will examine the structure of the **StaticSendPortOrchestration** and **DynamicSendPortOrchestration** and the code of the helper classes used by them.

This section covers the structure of the orchestrations and the code of the components they use. In particular, with the **DynamicSendPortOrchestration**, we'll examine how to implement an asynchronous, request-response pattern where the client application dynamically indicates to the BizTalk application the address of the queue or topic where to send the response message.

StaticSendPortOrchestration

The following figure shows the structure of the **StaticSendPortOrchestration**.



The **BusinessLogic** expression shape at point 1 contains the code in the box below. The orchestration invokes the **ProcessRequestReturnStream** method on a **RequestManager** object that returns the payload of the response message as a stream. I will omit for brevity the description of the **RequestManager** helper component as the analysis of its code is unnecessary for the understanding of the operating mode of the solution.

```
logHelper.WriteLine("[StaticSendPortOrchestration] Request message received from [{0}].",
                    requestMessage(WCF.To));
stream = requestManager.ProcessRequestReturnStream(requestMessage);
logHelper.WriteLine("[StaticSendPortOrchestration] Request message successfully
processed.");
```

The **MessageAssignment** shape at point 2 creates the response message.

```
responseMessage = null;
responseManager.SetResponse(responseMessage, stream);
responseMessage(*) = requestMessage(*);
guid = System.Guid.NewGuid();
responseMessage(Microsoft.WindowsAzure.CAT.Schemas.Application) = "ServiceBusSample";
responseMessage(Microsoft.WindowsAzure.CAT.Schemas.MessageId) = guid.ToString();
responseMessage(Microsoft.WindowsAzure.CAT.Schemas.CorrelationId) =
responseManager.GetMessageId(requestMessage);
responseMessage(Microsoft.WindowsAzure.CAT.Schemas.SessionId) =
responseManager.GetReplyToSessionId(requestMessage);
```

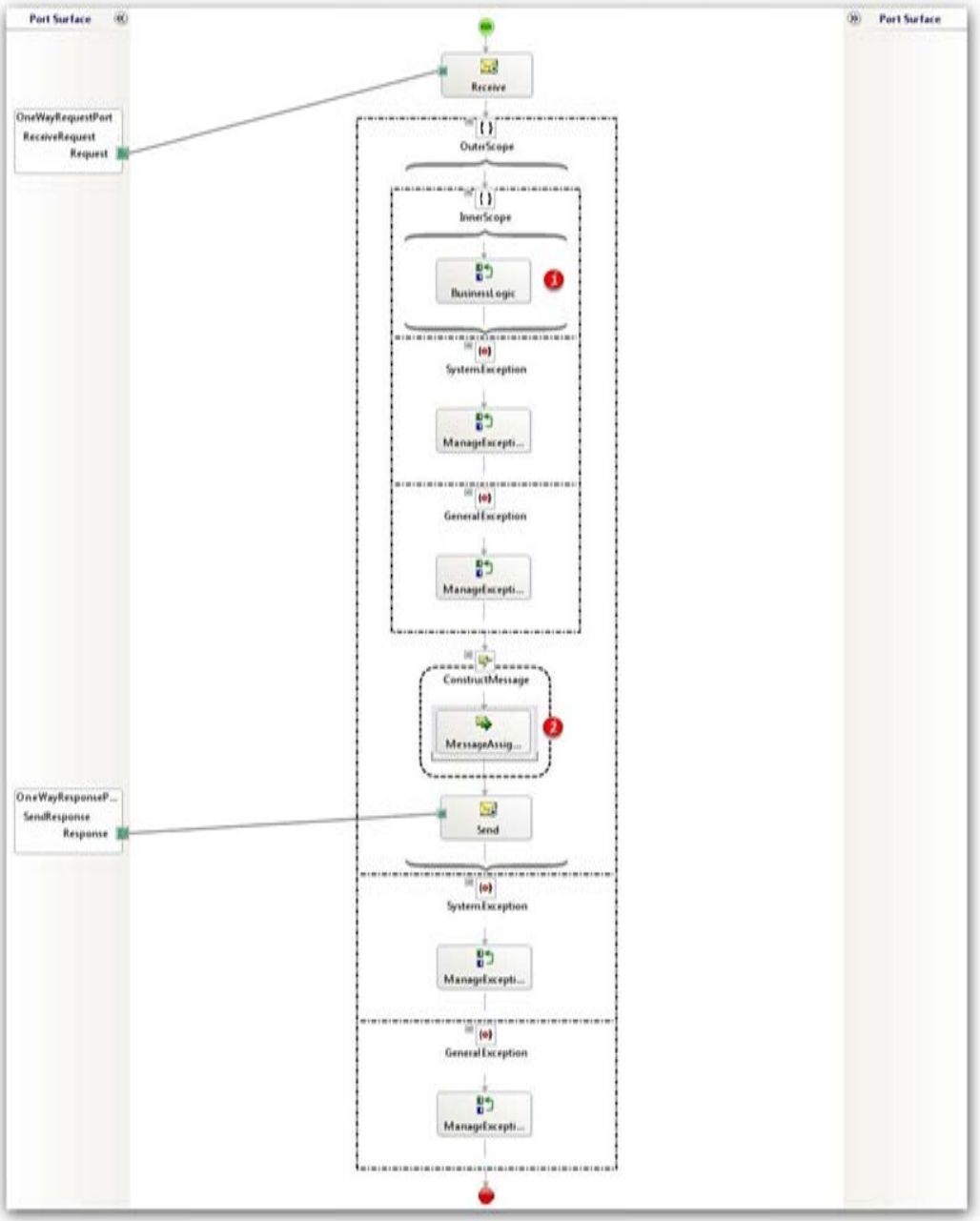
As you can notice, in this shape the orchestration performs the following actions:

- Invokes the **SetResponse** method on a **ResponseManager** object to assign the stream containing the payload to the response message.
- Copies the context properties from the request to the response message.
- Assign a value to the **Application** context property. In particular, the use of this context property demonstrates how a BizTalk application can send context information out-of-band to another application through a Service Bus message. In fact, as we have seen in the previous section, the static send ports are configured to translate this context property into a user-defined property of the outgoing BrokeredMessage.
- Creates a **MessageId** for the response message.
- Copies the **MessageId** from the request message to the **CorrelationId** of the response message. This copy allows the client to correlate the response message with the initial request.
- Copies the **ReplyToSessionId** from the request message to the **SessionId** of the response message. Please note that using sessions is necessary when multiple consumers use the same sessionful queue or subscription to receive response messages. In this case, a publisher application can use the ReplyToSessionId property of a BrokeredMessage to communicate to the consumer the value to assign to the SessionId of response messages.

The queue rule of the code snippet at point 3 invokes the **ReturnResponseViaQueue** method of the **RequestManager** object, which returns **true** if the value of the **ReplyTo** context property contains the word “queue” or false otherwise. In other words, the orchestration checks the value of the **ReplyTo** context property that contains the address specified by the client in the ReplyTo property of the BrokeredMessage and chooses whether to send the response message to the **responsequeue** or **responsetopic** based on the client demand.

DynamicSendPortOrchestration

The following figure shows the structure of the **DynamicSendPortOrchestration**.



The **BusinessLogic** expression shape at point 1 contains the code in the box below. The orchestration invokes the **ProcessRequestReturnStream** method on a **RequestManager** object that returns the payload of the response message as a stream. I will omit for brevity the description of the **RequestManager** helper component as the analysis of its code is unnecessary for the understanding of the operating mode of the solution.

```
logHelper.WriteLine("[DynamicSendPortOrchestration] Request message received from
[{0}].",
                    requestMessage(WCF.To));
stream = requestManager.ProcessRequestReturnStream(requestMessage);
logHelper.WriteLine("[DynamicSendPortOrchestration] Request message successfully
processed.");
```

The **MessageAssignment** shape at point 2 creates the response message.

```
responseMessage = null;
responseManager.SetResponse(responseMessage, stream);
responseMessage(*) = requestMessage(*);
guid = System.Guid.NewGuid();
responseMessage(Microsoft.WindowsAzure.CAT.Schemas.Application) = "ServiceBusSample";
responseMessage(Microsoft.WindowsAzure.CAT.Schemas.MessageId) = guid.ToString();
responseMessage(Microsoft.WindowsAzure.CAT.Schemas.CorrelationId) =
responseManager.GetMessageId(requestMessage);
responseMessage(Microsoft.WindowsAzure.CAT.Schemas.SessionId) =
responseManager.GetReplyToSessionId(requestMessage);
responseMessage(WCF.InboundHeaders) = "";
responseMessage(WCF.Action)= responseManager.GetAction(requestMessage);
responseMessage(WCF.BindingType)="netMessagingBinding";
responseMessage(WCF.EnableTransaction) = false;
responseMessage(WCF.BindingConfiguration) =
responseManager.GetBindingConfiguration(requestMessage);
responseMessage(WCF.EndpointBehaviorConfiguration) =
responseManager.GetEndpointBehaviorConfiguration(requestMessage);
OneWayResponsePort(Microsoft.XLANGs.BaseTypes.Address) =
responseManager.GetReplyTo(requestMessage);
OneWayResponsePort(Microsoft.XLANGs.BaseTypes.TransportType)="WCF-Custom";
```

As you can see in this shape the orchestration performs the following actions:

- Invokes the **SetResponse** method on a **ResponseManager** object to assign the stream containing the payload to the response message.
- Copies the context properties from the request to the response message.
- Assign a value to the **Application** context property. In particular, the use of this context property demonstrates how a BizTalk application can send context information out-of-band to another application through a Service Bus message. In fact, as we have seen in the previous section, the static send ports are configured to translate this context property into a user-defined property of the outgoing **BrokeredMessage**.
- Creates a **MessageId** for the response message.
- Copies the **MessageId** from the request message to the **CorrelationId** of the response message. This allows the client to correlate the response message with the initial request. Finally, the orchestration.

- Copies the **ReplyToSessionId** from the request message to the **SessionId** of the response message. Please note that sessions are necessary when multiple consumers use the same sessionful queue or subscription to receive response messages. In this case, a publisher application can use the ReplyToSessionId property of a BrokeredMessage to communicate to the consumer the value to assign to the SessionId of response messages.
- Invokes the **GetAction** method on the **ResponseManager** object to get the value for the WCF.Action.
- Specifies the NetMessagingBinding as WCF.BindingType for the dynamic send port.
- Assigns **false** to the WCF.EnableTransaction property.
- Invokes the **GetBindingConfiguration** method on the **ResponseManager** object to get the value for the WCF.BindingConfiguration.
- Invokes the **GetEndpointBehaviorConfiguration** method on the **ResponseManager** object to get the value for the WCF.EndpointBehaviorConfiguration.
- Invokes the **GetReplyTo** method on the **ResponseManager** object to get the address of the dynamic send port.
- Sets the **TransportType** of the dynamic send port to **WCF-Custom**.

For your convenience, I included the code of the **ResponseManager** class in the example below.

ResponseManager Class

```
public class ResponseManager
{
    #region Private Fields
    //***************************
    // Private Fields
    //***************************
    private readonly LogHelper logHelper = new LogHelper();
    #endregion

    #region Public Methods
    //***************************
    // Public Methods
    //***************************

    /// <summary>
    /// Sets the content of the XLANGMessage passed as first parameter.
    /// </summary>
    /// <param name="message">An XLANGMessage object.</param>
    /// <param name="stream">The stream containing the payload to assign to the
message.</param>
    public void SetResponse(XLANGMessage message, Stream stream)
    {
        try
```

```
        {
            if (stream != null &&
                message != null &&
                message.Count > 0)
            {
                if (stream.CanSeek)
                {
                    stream.Seek(0, SeekOrigin.Begin);
                }
                message[0].LoadFrom(stream);
            }
        }
        catch (Exception ex)
        {
            logHelper.WriteLine(string.Format("[ResponseManager] {0}", ex.Message));
        }
        finally
        {
            if (message != null)
            {
                message.Dispose();
            }
        }
    }
    /// <summary>
    /// Gets the action for the response request.
    /// </summary>
    /// <param name="request">The request request.</param>
    /// <returns>The action to assign to the Action header of the response
request.</returns>
    public string GetAction(XLANGMessage request)
    {
        const string action = "ReceiveResponse";

        try
        {
            // In a real application, the action should be retrieved from a
            // configuration repository based on the request request content and context
            // information. In addition, this data should be cached to improve
performance.
            return action;
        }
```

```
        catch (Exception ex)
        {
            logHelper.WriteLine(string.Format("[ResponseManager] {0}", ex.Message));
        }
        finally
        {
            logHelper.WriteLine(string.Format("[ResponseManager] Action=[{0}]", action ??
"NULL"));
            if (request != null)
            {
                request.Dispose();
            }
        }
        // Return default Action.
        return action;
    }

    /// <summary>
    /// Gets the binding configuration for the response request.
    /// </summary>
    /// <param name="request">The request request.</param>
    /// <returns>The binding configuration of the dynamic send port used to send
    /// out the response request.</returns>
    public string GetBindingConfiguration(XLANGMessage request)
    {
        const string bindingConfiguration = @"<binding name=""netMessagingBinding"" " +
                            @"sendTimeout=""00:03:00"" " +
                            @"receiveTimeout=""00:03:00"" " +
                            @"openTimeout=""00:03:00"" " +
                            @"closeTimeout=""00:03:00"" " +
                            @"sessionIdleTimeout=""00:01:00"" " +
                            @"prefetchCount=""-1""> " +
                            @"<transportSettings batchFlushInterval=""00:00:01"" />" +
                            @"</binding>";
        try
        {
            // In a real application, the binding configuration should be retrieved from
a
            // configuration repository based on the request request content and context
            // information. In addition, this data should be cached to improve
performance.
            return bindingConfiguration;
```

```
        }
        catch (Exception ex)
        {
            logHelper.WriteLine(string.Format("[ResponseManager] {0}", ex.Message));
        }
        finally
        {
            logHelper.WriteLine(string.Format("[ResponseManager]
BindingConfiguration=[{0}]",
                                                bindingConfiguration ?? "NULL"));
            if (request != null)
            {
                request.Dispose();
            }
        }
        // Return default binding configuration.
        return bindingConfiguration;
    }

    /// <summary>
    /// Gets endpoint behavior configuration of the dynamic
    /// send port used to send out the response request.
    /// </summary>
    /// <param name="request">The request request.</param>
    /// <returns>The endpoint behavior configuration of the dynamic
    /// send port used to send out the response request.</returns>
    public string GetEndpointBehaviorConfiguration(XLANGMessage request)
    {
        const string endpointBehaviorConfiguration = @"<behavior
name=""EndpointBehavior"">" +
        @"<transportClientEndpointBehavior>" +
        @"<tokenProvider>" +
        @"<sharedSecret issuerName=""owner""
issuerSecret=""2jdfAfOD7Jgd4WmnI7mJStz35hH7NDC7AN20MjbSpxs="" />" +
        @"</tokenProvider>" +
        @"</transportClientEndpointBehavior>" +
        @"<serviceBusMessageInspector>" +
        @"<propertiesToSend name=""Application,Country,City"" " +

@"namespace=""http://windowsazure.cat.microsoft.com/samples/servicebus/propertyschema," +
        @"http://windowsazure.cat.microsoft.com/samples/servicebus/propertyschema," +
        @"http://windowsazure.cat.microsoft.com/samples/servicebus/propertyschema"" />" +
```

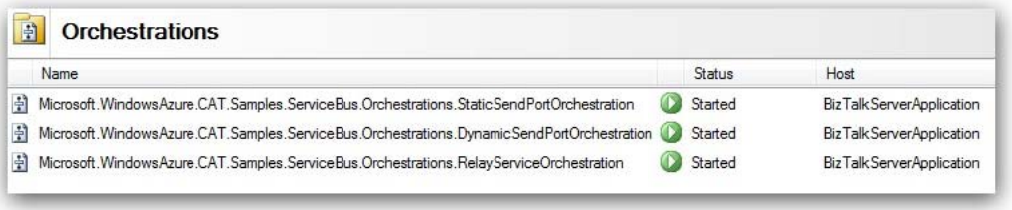
```
        @"</serviceBusMessageInspector>" +
        @"</behavior>";
        try
        {
            // In a real application, the endpoint behavior configuration should be
retrieved from a
            // configuration repository based on the request request content and context
            // information. In addition, this data should be cached to improve
performance.
            return endpointBehaviorConfiguration;
        }
        catch (Exception ex)
        {
            logHelper.WriteLine(string.Format("[ResponseManager] {0}", ex.Message));
        }
        finally
        {
            logHelper.WriteLine(string.Format("[ResponseManager]
EndpointBehaviorConfiguration=[{0}]",
                                                endpointBehaviorConfiguration ?? "NULL"));
            if (request != null)
            {
                request.Dispose();
            }
        }
        // Return default endpoint behavior configuration.
        return string.Empty;
    }

    /// <summary>
    /// Gets the value of the MessageId context property.
    /// </summary>
    /// <param name="request">The request request.</param>
    /// <returns>The value of the MessageId context property.</returns>
    public string GetMessageId(XLANGMessage request)
    {
        try
        {
            if (request != null)
            {
                var messageId = request.GetPropertyValue(typeof (MessageId)) as string;
                if (!string.IsNullOrEmpty(messageId))
```

```
                {
                    logHelper.WriteLine(string.Format("[ResponseManager]
MessageId=[{0}]", messageId));
                    return messageId;
                }
            }
        }
        catch (Exception ex)
        {
            logHelper.WriteLine(string.Format("[ResponseManager] Exception: {0}",
ex.Message));
        }
        finally
        {
            if (request != null)
            {
                request.Dispose();
            }
        }
        return string.Empty;
    }

    /// <summary>
    /// Gets the value of the ReplyTo context property.
    /// </summary>
    /// <param name="request">The request request.</param>
    /// <returns>The value of the ReplyTo context property.</returns>
    public string GetReplyTo(XLANGMessage request)
    {
        try
        {
            if (request != null)
            {
                var replyTo = request.GetPropertyValue(typeof(ReplyTo)) as string;
                if (!string.IsNullOrEmpty(replyTo))
                {
                    logHelper.WriteLine(string.Format("[ResponseManager] ReplyTo=[{0}]",
replyTo));
                    return replyTo;
                }
            }
        }
```

```
        catch (Exception ex)
        {
            logHelper.WriteLine(string.Format("[ResponseManager] Exception: {0}",
ex.Message));
        }
        finally
        {
            if (request != null)
            {
                request.Dispose();
            }
        }
        return string.Empty;
    }

    /// <summary>
    /// Gets the value of the ReplyToSessionId context property.
    /// </summary>
    /// <param name="request">The request request.</param>
    /// <returns>The value of the ReplyToSessionId context property.</returns>
    public string GetReplyToSessionId(XLANGMessage request)
    {
        try
        {
            if (request != null)
            {
                var replyToSessionId = request.GetPropertyValue(typeof(ReplyToSessionId))
as string;
                if (!string.IsNullOrEmpty(replyToSessionId))
                {
                    logHelper.WriteLine(string.Format("[ResponseManager]
ReplyToSessionId=[{0}]",
                                                      replyToSessionId));
                    return replyToSessionId;
                }
                return !string.IsNullOrEmpty(replyToSessionId) ? replyToSessionId :
string.Empty;
            }
        }
        catch (Exception ex)
        {
```

```
            logHelper.WriteLine(string.Format("[ResponseManager] Exception: {0}",
ex.Message));
        }
        finally
        {
            if (request != null)
            {
                request.Dispose();
            }
        }
        return string.Empty;
    }
    #endregion
}
```

The following picture shows the two orchestrations in the **BizTalk Server Administration Console** along with another orchestration called **RelayServiceOrchestration** used to serve requests coming from the request-response receive locations. The description of this orchestration is out of the scope of this article.

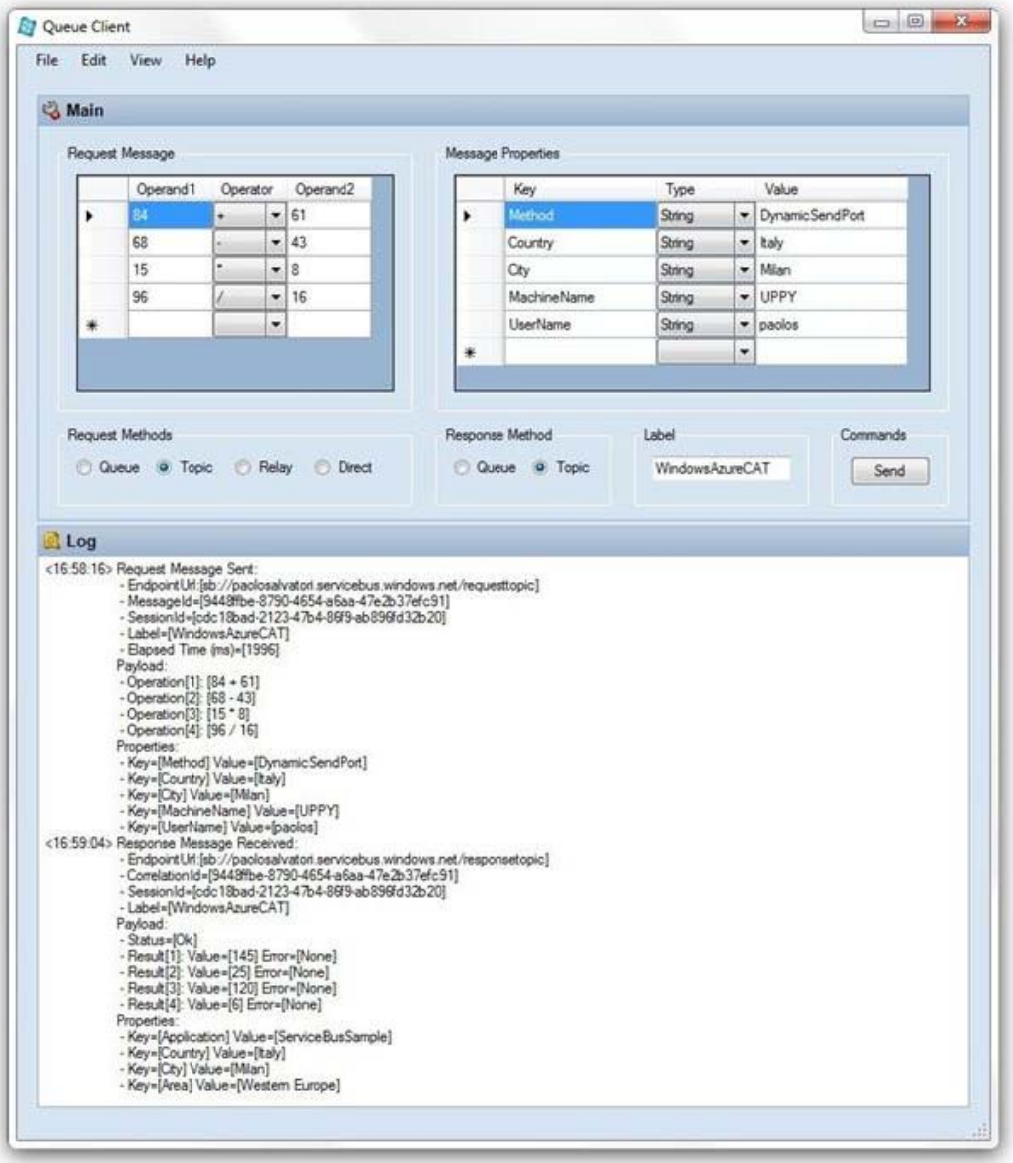


Testing the Solution

Assuming that you have already deployed and properly configured the solution, you can proceed as follows to test it.

- To send a request message to the **StaticSendPortOrchestration**, set the value of the **Method** property to **StaticSendPort**.
- To send a request message to the **DynamicSendPortOrchestration**, set the value of the **Method** property to **DynamicSendPort**.
- To send a request message to BizTalk Server via the **requestqueue**, select the **Queue** radio button in the **Request Methods** group
- To send a request message to BizTalk Server using the **requesttopic**, select the **Topic** radio button in the **Request Methods** group.
- To ask BizTalk Server to send the response message to the **responsequeue**, select the **Queue** radio button in the **Response Methods** group
- To ask BizTalk Server to send the response message to the **responsetopic**, select the **Topic** radio button in the **Response Methods** group.

The figure below shows the most interesting combination:



- The client sends the request message to **requesttopic**.
- The BizTalk application reads the request from the **ItalyMilan** subscription for the **requesttopic** using the WCF-Custom receive location.
- The **DynamicSendPortOrchestration** sends the response message to the **responsetopic**.
- The client application receives the response message through the **ItalyMilan** subscription defined on the **responsetopic**.

Looking at the log, you can note the following:

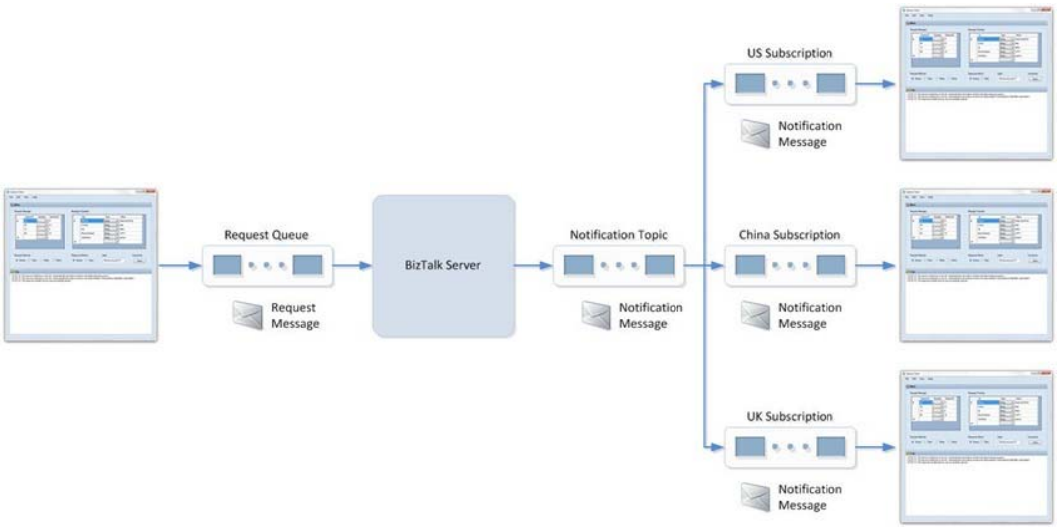
- The request message was sent to the **requesttopic**.
- The response message was received from the **responsetopic**.

- The **CorrelationId** of the response message is equal to the **MessageId** of the request message.
- The **Label** property of the request message has been translated by the message inspector into the **Label** context property in the WCF receive location. Then the orchestration copied the **Label** context property from the request message to the response message. Finally, the WCF send port copied the value of the **Label** context property to the **Label** property of the Service Bus message. The user defined properties **Country** and **City** followed the same data flow.
- The orchestration added the **Application** context property to the response message, and the WCF send port translated this property into a user-defined property of the Service Bus message.
- The **Area** property was added by the rule action defined on the **ItalyMilan** subscription.

Implementing a Message Fan-Out Scenario

You can use the **ServiceBusSample** solution to simulate a message fan-out scenario where BizTalk receives a request through a Service Bus queue or topic from a back-office application and sends a notification to multiple front-office applications located in different location via a Service Bus topic. In this context, each subscriber uses a different subscription to receive notifications sent by BizTalk Server.

The following figure shows the architecture a demo that simulates a message fan-out scenario. In this scenario one instance of the client application simulates a back-office application sending requests to BizTalk Server, while the other instances simulate front-office applications that receive notifications of a change of state (for example, if the price of a product has changed).



To set up this scenario, you need to proceed as follows:

- Create the individual subscriptions for the **requesttopic**s called, respectively, **US**, **China** and **UK**.
- Copy the client application's configuration file to three different folders called **US**, **China** and **UK**.

- Open the configuration files of the three instances and change the **listenUri** of the subscription service endpoint as follows:
  - **US**: `sb://paolosalvatori.servicebus.windows.net/responsetopic/Subscriptions/US`
  - **China**: `sb://paolosalvatori.servicebus.windows.net/responsetopic/Subscriptions/China`
  - **UK**: `sb://paolosalvatori.servicebus.windows.net/responsetopic/Subscriptions/UK`
- Use a forth instance of the client application to send a message to the BizTalk application, and make sure to select **Topic** in the **Response Methods**.

Conclusion

In this article we have seen how to bridge the gap between BizTalk Server and the Service Bus and how to achieve full interoperability between the two simply by using the native features provided out-of-the-box by these technologies and by adding a couple of extensions and a pinch of imagination. Is this the only way to integrate BizTalk Server with the Service Bus? The answer is clearly no. An alternative solution is to build a custom Service Bus adapter by using the WCF LOB Adapter SDK and the .NET Messaging API. This adapter should integrate the functionalities that I described in this article and implemented in my code.

Another alternative, and an easier way to enable a BizTalk application to exchange messages with Service Bus queues and topics, is to create a reusable helper class. This component should use the .NET Messaging API and the BizTalk Runtime libraries to accomplish the following tasks:

- Transform an XLANGMessage into a BrokeredMessage and vice versa. In particular, map BizTalk message context properties into BrokeredMessage user-defined properties and vice versa.
- Receive messages from a Service Bus queue or subscription.
- Send messages to a Service Bus queue or topic.

I look forward hearing your feedback. In the meantime, here you can download the companion code for this article.

Managing and Testing Topics, Queues and Relay Services with the Service Bus Explorer Tool

Authors: Paolo Salvatori

Introduced in the September 2011, queues and topics represent the foundation of a new cloud-based messaging and integration infrastructure that provides reliable message queuing and durable publish/subscribe messaging capabilities to both cloud and on-premises applications based on Microsoft and non-Microsoft technologies. .NET applications can use the new functionality offered by queues and topics by using the new messaging API (Microsoft.ServiceBus.Messaging) released with the Windows Azure AppFabric SDK V1.5 or via WCF by using the new NetMessagingBinding. Likewise, any Microsoft or non-Microsoft applications can use a Service Bus REST API to manage and access messaging entities over HTTPS.

Queues and topics were first introduced by the Community Technology Preview (CTP) of Windows Azure AppFabric that was released in May 2011. At that time, the Windows Azure Management Portal didn't provide any user interface to administer, create and delete messaging entities and the only way to accomplish this task was using the .NET or REST API. For this reason, In June 2011 I decided to build a tool called Service Bus Explorer that would allow developers and system administrators to connect to a Service Bus namespace and administer its messaging entities. During the last few months I continued to develop this tool and add new features with the intended goal to facilitate the development and administration of new Service Bus-enabled applications. In the meantime, the Windows Azure Management Portal introduced the ability for a user to create queues, topics, and subscriptions and define their properties, but not to define or display rules for an existing subscription. Besides, the Service Bus Explorer enables to accomplish functionalities, such as importing, exporting and testing entities, that are not currently provided by the Windows Azure Management Portal. For this reason, the Service Bus Explorer tool represents the perfect companion for the official Azure portal and it can also be used to explore the features (session-based correlation, configurable detection of duplicate messages, deferring messages, etc.) provided out-of-the-box by the Service Bus brokered messaging.

This post is the first of a two-part article where I will explain the functioning and implementation details of my tool whose source code is available on MSDN Code Gallery for use or modification by any developer. In particular, in the first part I will explain how to use my tool to manage and test Queues and Topics whereas in the second post I will go through the code and explain the approach I followed to realize the application.

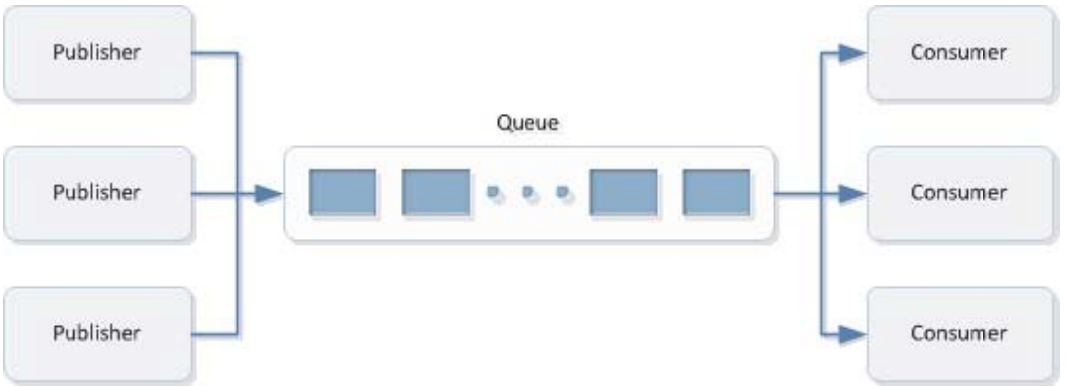
For more information on the AppFabric Service Bus, you can use the following resources:

- Service Bus topic on the MSDN Site.
- Now Available: The Service Bus September 2011 Release article on the Windows Azure Blog.
- Queues, Topics, and Subscriptions article on the MSDN site.
- AppFabric Service Bus topic on the MSDN site.
- Understanding Windows Azure AppFabric Queues (and Topics) video on the AppFabric Team Blog.

- Building loosely-coupled apps with Windows Azure Service Bus Topics and Queues video on the channel9 site.
- Service Bus Topics And Queues video on the channel9 site.
- Securing Service Bus with ACS video on the channel9 site.

Queues

Queues provide messaging capabilities that enable a large and heterogeneous class of applications running on premises or in the cloud to exchange messages in a flexible, secure and reliable fashion across network and trust boundaries.



Queues are hosted in Windows Azure by a replicated, durable store infrastructure. The maximum size of a queue during the CTP is 100MB, which is expected to rise to 1GB (or more) for production. The maximum message size is 256KB, but the use of sessions allows creating unlimited-size sequences of related messages. Queues functionality are accessible through the following APIs:

- The new messaging API available via the new Microsoft.ServiceBus.Messaging assembly
- WCF via the new ServiceBusMessagingBinding
- An extension of the Service Bus REST API.

Queue entities provide the following capabilities:

- Session-based correlation, meaning that you can build multiplexed request/reply paths in a easy way.
- Reliable delivery patterns such as Peek/Lock.
- Detection of inbound message duplicates, allowing clients to send the same message multiple times without adverse consequences.
- Dead-letter facility for messages that cannot be processed or expire before being received.
- Deferring messages for later processing. This functionality is particularly handy when messages are received out of the expected sequence and need to be safely put on the side while the process waits for a particular message to permit further progress or when messages need to be processed based on a set of properties that define their priority during a traffic peak.

In the .NET API the message entity is modeled by the BrokeredMessage class that exposes various properties such as MessageId, SessionID and CorrelationId that can be used to exploit capabilities such as automatic duplicate detection or session-enabled communications. A

QueueDescription object can be used to specify the metadata that models the behavior of the queue being created:

- The DefaultMessageTimeToLive property specifies the default message time to live value.
- The DuplicateDetectionHistoryTimeWindow property defines the duration of the duplicate detection history.
- The EnableDeadLetteringOnMessageExpiration property allows to enable\disable the dead-lettering on message expiration.
- The LockDuration property defines the duration of the lock used by a consumer when using the PeekLock receive mode. In the ReceiveAndDelete receive mode, a message is deleted from the queue as soon as it is read by a consumer. Conversely, in the PeekLock receive mode, a message is hidden from other receivers until the timeout defined by the LockDuration property expires. By that time the receiver should have deleted the message invoking the Complete method on the BrokeredMessage object.
- The MaxQueueSizeInBytes property defines the maximum queue size in bytes.
- The RequiresDuplicateDetection property enables\disables duplicate message detection. Since metadata cannot be changed once a messaging entity is created, modifying the duplicate detection behavior requires deleting and recreating the queue. The same principle applies to any other metadata.
- The RequiresSession property enables\disables sessions.

The use of Queues permits to flatten a highly-variable traffic into a predictable stream of work and distribute the load across a set of worker processes which size can vary dynamically to accommodate the incoming message volume. In a Competing Consumers scenario, when a publisher writes a message to a queue, multiple consumers compete with each other to receive the message, but only one of them will receive and process the message in question.

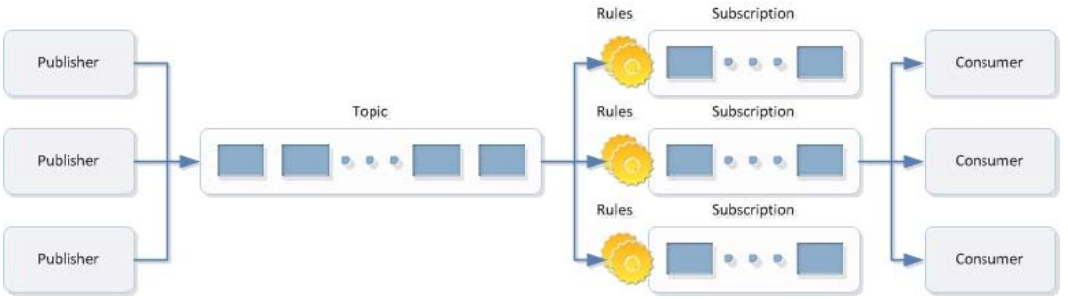
In a service-oriented or a service bus architecture composed of multiple, heterogeneous systems, interactions between autonomous systems are asynchronous and loosely-coupled. In this context, the use of messaging entities like Queues and Topics (see the next section) allows to increase the agility, scalability and flexibility of the overall architecture and helps decreasing the loose coupling of individual systems.

For more information on Queues, see the following articles:

- Introducing the Windows Azure AppFabric Service Bus May 2011 CTP article on the AppFabric Team Blog.
- Understanding Windows Azure AppFabric Queues (and Topics) video on the AppFabric Team Blog.
- An Introduction to Service Bus Queues article on the AppFabric Team Blog.
- Windows Azure AppFabric Service Bus Queues and Topics on Neil Mackenzie's Blog.
- Windows Azure AppFabric Learning Series available on CodePlex.

Topics

Topics extend the messaging features provided by Queues with the addition of Publish-Subscribe capabilities.



A Topic entity consists of a sequential message store like a Queue, but it supports up to 2000 (this number is subject to vary in the future) concurrent and durable subscriptions which relay message copies to a poll of worker processes. Each Subscription can define one or multiple Rule entities. Each Rule specifies a filter expression that is used to filter messages that pass through the subscription and a filter action that can modify message properties. In particular, the SqlFilterExpression class allows you to define a SQL92-like condition on message properties:

- OrderTotal > 5000 OR ClientPriority > 2
- ShipDestinationCountry = 'USA' AND ShipDestinationState = 'WA'

Conversely, the SqlFilterAction class can be used to modify, add or remove properties, as the filter expression is true and the message selected, using a syntax similar to that used by the SET clause of an UPDATE SQL-command.

- SET AuditRequired = 1
- SET Priority = 'High', Severity = 1

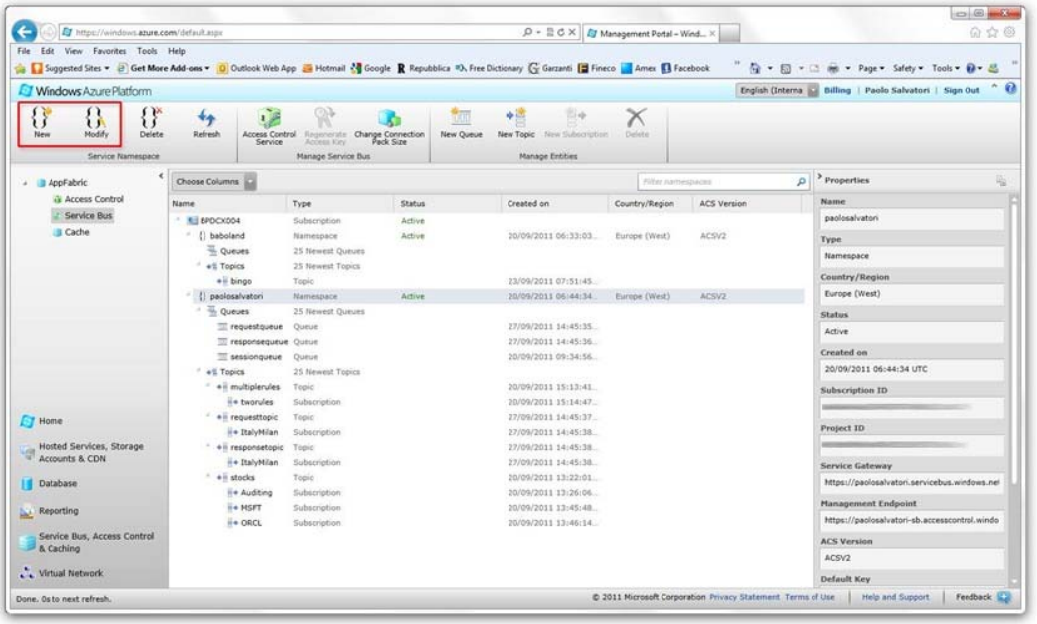
Each matching rule generates a separate copy of the published message, so any subscription can potentially generate more copies of the same message, one for each matching rule. As Queues, Topics support a Competing Consumers scenario: in this context, a subscription can have a single consumer that receives all messages or a set of competing consumers that fetch messages on a first-come-first-served basis. Topics are the ideal messaging solution to broadcast messages to many consumer applications or distribute the work load across multiple sets of competing worker processes.

The following section provides basic information on the Queue and Topic entities in Windows Azure AppFabric Service Bus. For more information, see the following articles:

- Understanding Windows Azure AppFabric Queues (and Topics) video on the AppFabric Team Blog.
- An Introduction to Service Bus Topics article on the AppFabric Team Blog.
- Windows Azure AppFabric Service Bus Queues and Topics on Neil Mackenzie's Blog.
- Windows Azure AppFabric Learning Series available on CodePlex.

Queues, Topics and Subscriptions

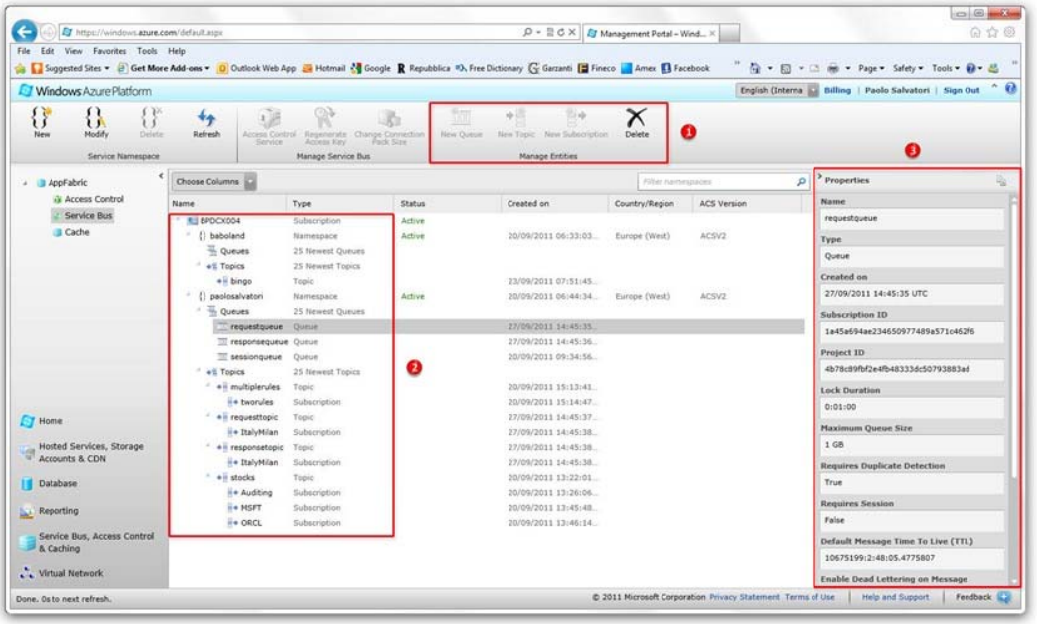
In order to use the Brokered and Relay messaging functionality provided by the Service Bus, the first operation to perform is to provision a new Service Bus namespace or modify an existing namespace to include the Service Bus. You can accomplish this task from the Windows Azure Management Portal respectively by clicking the **New** button or the **Modify** button highlighted in the figure below.



Once completed this step, you can start creating and using queues, topics and subscriptions. You have many options to provision and manage messaging entities:

- This first option is to exploit the CreateQueue, CreateTopic and CreateSubscription methods exposed by the NamespaceManager class that can be used to manage entities, such as queues, topics, subscriptions, and rules, in your service namespace. This is exactly the approach that I followed to build the provisioning functionality exposed by the Service Bus Explorer tool.
- The second approach is to rely on the Service Bus REST API to create and delete messaging entities. By using REST, you can write applications in any language that supports HTTP requests, without the need for a client SDK. In particular, this allows applications based on non-Microsoft technologies (e.g. Java, Ruby, etc.) not only to send and receive messages from the Service Bus, but also to create or delete queues, topics and subscription in a given namespace.

Finally, you can administer messaging entities from the user interface supplied by the Windows Azure Management Portal. In particular, the buttons in the **Manage Entities** command bar highlighted in red at point 1 in the figure below allow creating a new queue, topic and subscription entity. Then you can use the navigation tree-view shown at point 2 to select an existing entity and display its properties in the vertical bar highlighted at point 3. To remove the selected entity, you press the **Delete** button in the **Manage Entities** command bar.



Using Windows Azure Management Portal is a handy and convenient manner to handle the messaging entities in a given Service Bus namespace. However, at least at the moment, the set of operations that a developer or a system administrator can perform using its user interface is quite limited. For example, the Windows Azure Management Portalactually allows a user to create queues, topics, and subscriptions and define their properties, but not to create or display rules for an existing subscription. At the moment, you can accomplish this task only by using the .NET Messaging API. In particular, to add a new rule to an existing subscription you can use the AddRule(String, Filter) or the AddRule(RuleDescription) methods exposed by the SubscriptionClient class, while to enumerate the rules of an existing subscription, you can use the GetRules method of the NamespaceManager class. Besides, the Windows Azure Management Portal actually does not provide the ability to perform the following operations:

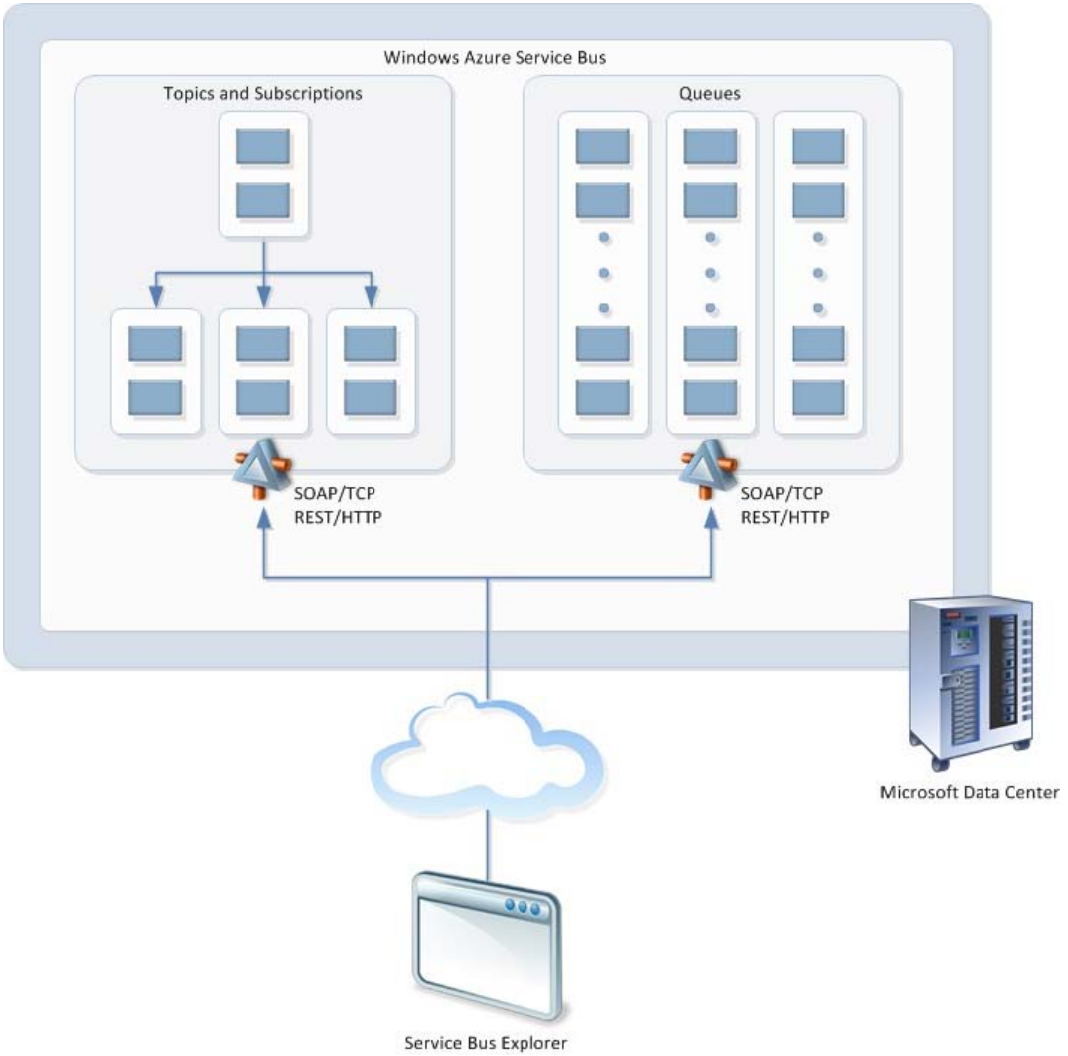
- Properly visualize entities in a hierarchical manner. Actually, the Windows Azure Management Portaldisplays queues, topics and subscriptions with a flat treeview. However, you can organize messaging entities in a hierarchical structure simply by specifying their name as an absolute path composed of multiple path segments, for example crm/prod/queues/orderqueue.
- Export the messaging entities contained in a given Service Bus namespace to an XML binding file (a-la BizTalk Server). Instead, the Service Bus Explorer tool provides the ability to select and export
    - Individual entities.
    - Entities by type (queues or topics).
    - Entities contained in a certain path (e.g. crm/prod/queues).
    - All the entities in a given namespace.
- Individual entities.
- Entities by type (queues or topics).
- Entities contained in a certain path (e.g. crm/prod/queues).

- All the entities in a given namespace.
- Import queues, topics and subscriptions from an XML file to an existing namespace. The Service Bus Explorer supports the ability to export entities to an XML file and reimport them on the same or another Service Bus namespace. This feature comes in a handy to perform a backup and restore of a namespace or simply to transfer queues and topics from a testing to a production namespace.

That's why in June, I created a tool called Service Bus Explorer that allows a user to create, delete and test queues, topics, subscriptions, and rules. My tool was able to manage entities in the AppFabric Labs Beta environment. However, the new version of the Service Bus API introduced some breaking changes, as you can read here, so I built a new version of the Service Bus Explorer tool that introduces a significant amount of new features.

Solution

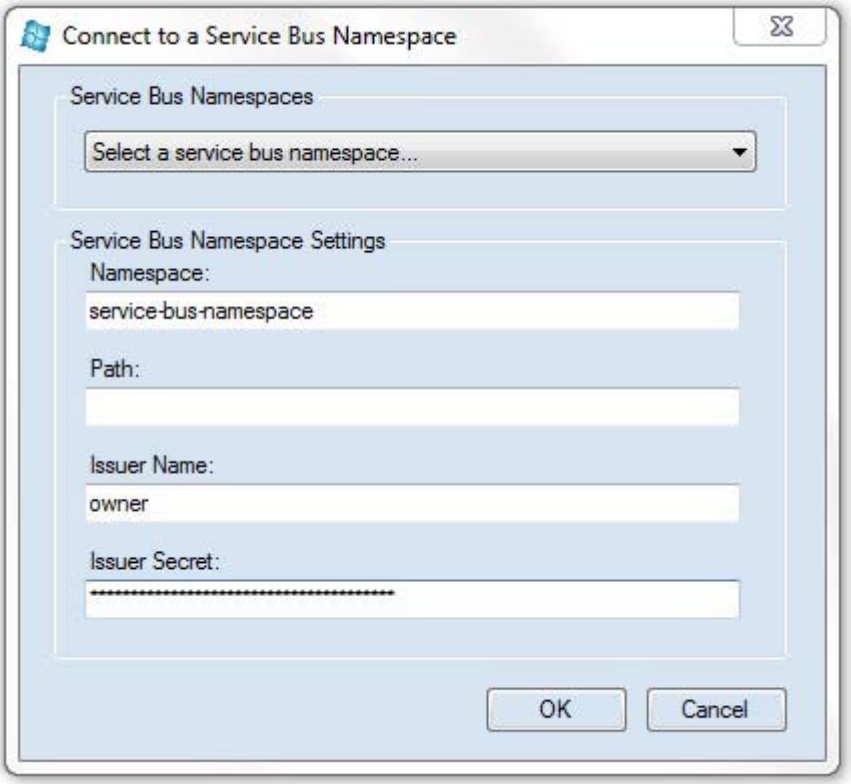
The following picture illustrates the high-level architecture of the Service Bus Explorer tool. The application has been written in C# using Visual Studio 2010 and requires the installation of the .NET Framework 4.0 and Windows Azure AppFabric SDK V1.5. The tool can be copied and used on any workstation that satisfies the prerequisites mentioned above to manage and test the Brokered and Relay messaging services defined in a given Service Bus namespace.



In the remainder of the article I'll explain how to configure and use the Service Bus Explorer tool.

Connecting to an existing Service Bus Namespace

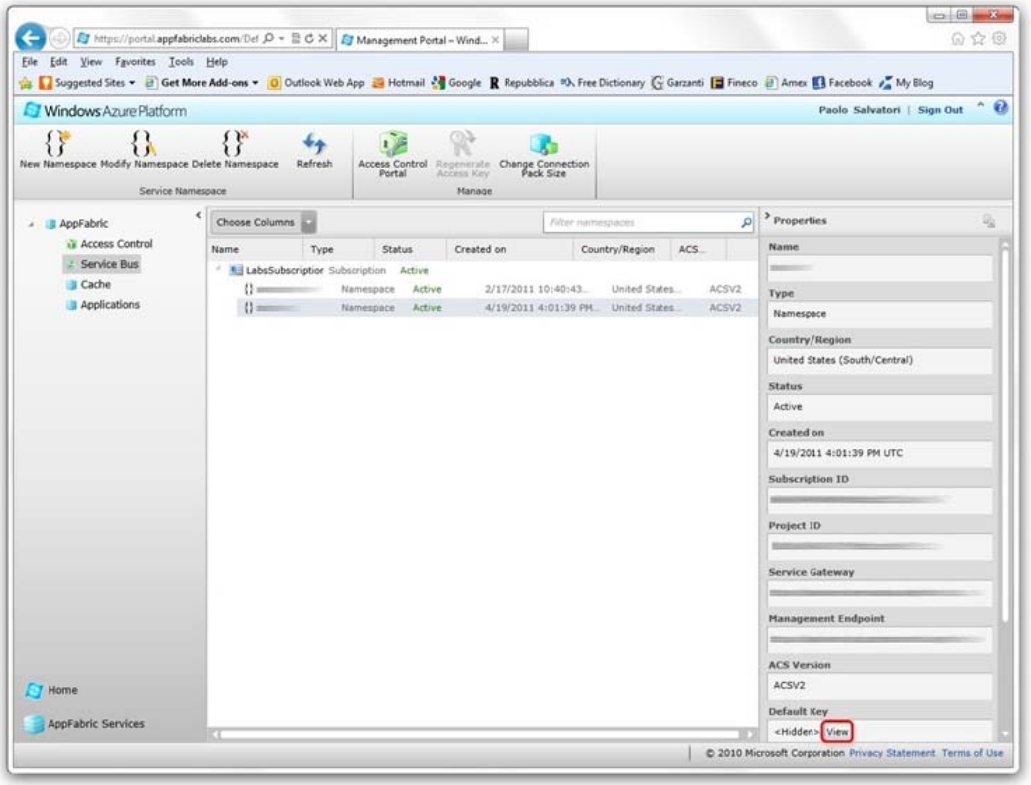
When you launch the application, you can connect to an existing Service Bus namespace to manage its entities by choosing the Connect command under the File menu. This operation opens up a modal dialog shown in the picture below that allows the user to enter the name of the Service Bus namespace that you want to connect to and the corresponding authentication credentials.



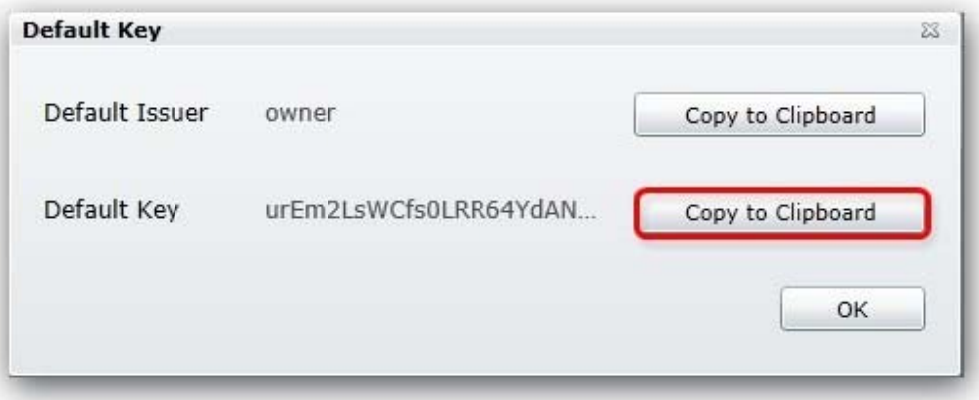
The authentication and authorization to use a service exposed by the Service Bus are controlled by the Access Control Service. In particular, the Service Bus currently supports three different types of credential schemes:

- SharedSecret, a slightly more complex but easy-to-use form of username/password authentication.
- Saml, which can be used to interact with SAML 2.0 authentication systems.
- SimpleWebToken, which uses the OAuth Web Resource Authorization Protocol (WRAP) and Simple Web Tokens (SWT).

The Service Bus Explorer tool currently supports only SharedSecret credentials. However, you can easily extend its code to support other credential schemes. You can retrieve the issuer-secret key for a given namespace from the Windows Azure Management Portal by clicking the View button highlighted in red in the picture below.



This opens up a modal dialog where you can retrieve the key by clicking the **Copy to Clipboard** button highlighted in red in the picture below.



Note

By convention, the name of the **Default Issuer** is always the **owner**.

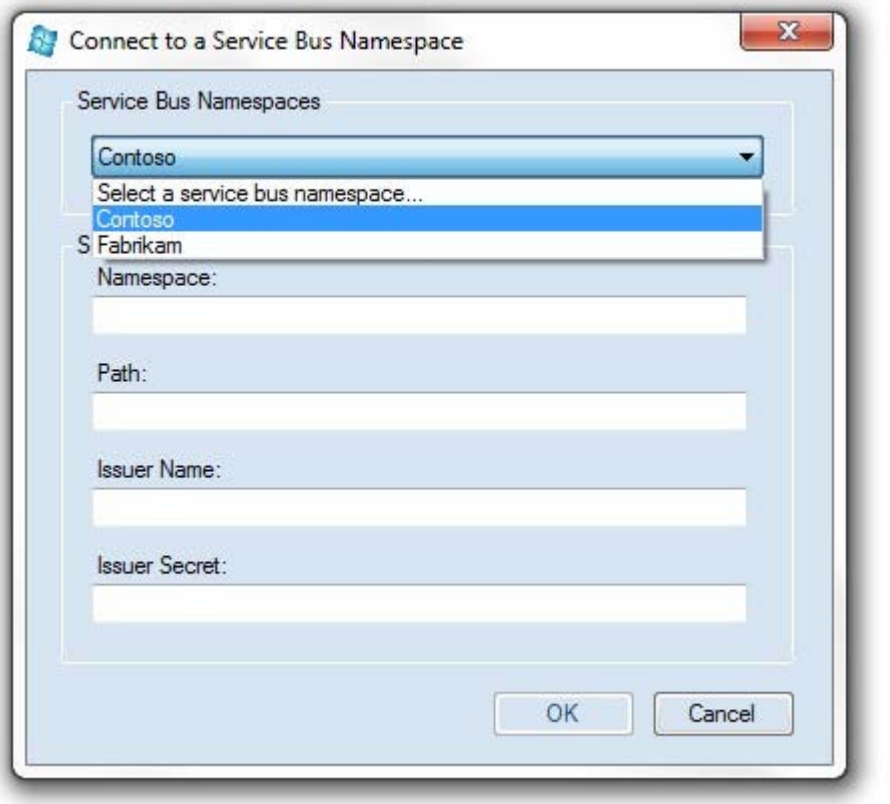
For your convenience, the tool gives the possibility to define, in the configuration file, a connection string for the most frequently accessed namespaces. As you can see in the xml snippet below, each connection string is composed of 4 elements:

- **namespace**: this property defines the Service Bus namespace.

- **servicePath**: this property is optional and specifies the service path within the namespace.
- **issuerName**: this field contains the **Issuer Name**. As mentioned above, by convention, the default value for this field is **owner**.
- **issuerSecret**: this element contains the **Issuer Secret** that you can copy from the **Windows Azure Platform Management Portal**.

```
<?xml version="1.0"?>
<configuration>
<configSections>
<section name="serviceBusNamespaces"
      type="System.Configuration.DictionarySectionHandler,
      System, Version=4.0.0.0, Culture=neutral,
      PublicKeyToken=b77a5c561934e089"/>
</configSections>
<serviceBusNamespaces>
<add key="Namespace1" value="namespace=namespace1;
      servicePath=;issuerName=owner;issuerSecret=..."/>
<add key="Namespace2" value="namespace=namespace2;servicePath=;
                     issuerName=owner;issuerSecret=..."/>
</serviceBusNamespaces>
<appSettings>
<add key="debug" value="true"/>
<add key="scheme" value="sb"/>
<add key="message" value="Hi mate, how are you?" />
<add key="file" value="C:\Sample.xml" />
<add key="label" value="Service Bus Explorer" />
<add key="retryCount" value="10" />
<add key="retryTimeout" value="100" />
<add key="messageDeferProvider"
value="Microsoft.AppFabric.CAT.WindowsAzure.Samples.ServiceBusExplorer.InMemoryMessageDef
erProvider,ServiceBusExplorer"/>
</appSettings>
<system.net>
<connectionManagement>
<add address="*" maxconnection="50"/>
</connectionManagement>
</system.net>
<startup>
<supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/>
</startup>
</configuration>
```

If you define one or more namespaces in the configuration file, this gives you the ability to select one of them from the corresponding drop-down list and the fields within the modal dialog will be automatically populated with the corresponding connection string data.



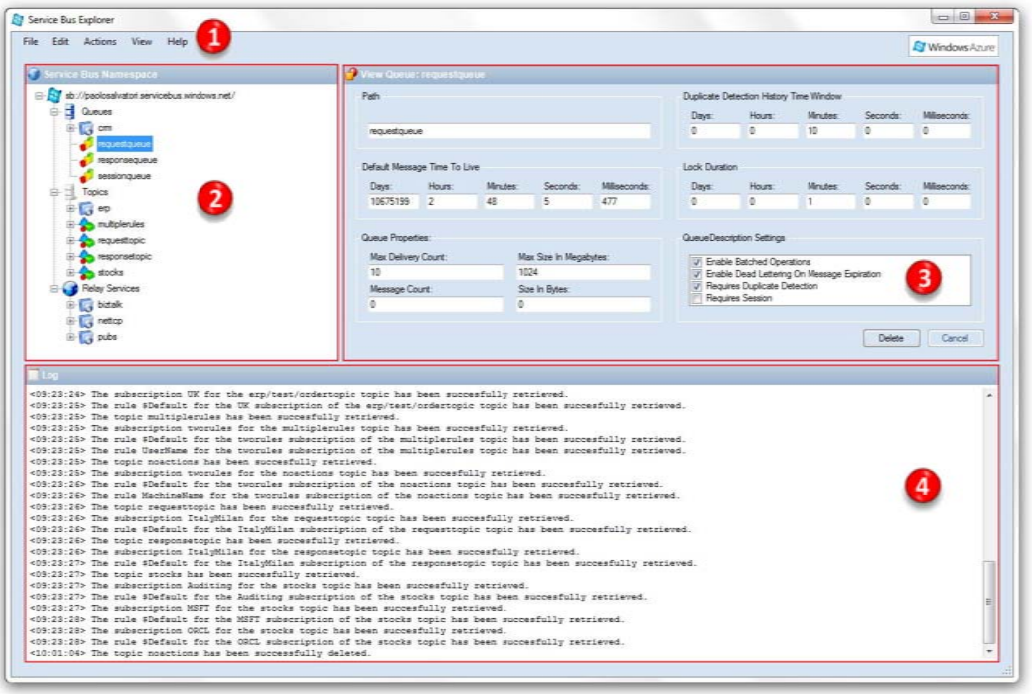
In the **appSettings** section of the configuration file you can define the following default settings:

- **debug**: some components use a custom trace listener to write message in the log view. By setting this parameter to false, you can disable this behavior.
- **scheme**: this parameter can be used to define the transport protocol that the tool uses to communicate with the Service Bus. At the moment, the only supported scheme is sb.
- **message**: you can use this parameter to specify the default text for the test message.
- **file**: this parameter can be used to define the location of the default message when testing queues, topics and relay services.
- **label**: this setting defines the default value for the Label property of a brokered message.
- **retryCount**: this parameter can be used the default retry count in case of exception.
- **retryTimeout**: this parameter allows to control the time in milliseconds between two consecutive attemps.
- **messageDeferProvider**: when testing deferred messages, the application uses a default provider to store in-process the sequence number of deferred messages. You can create an alternative provider and replace the default one. In this case, you have to use this parameter to declare its fully-qualified name.

Graphical User Interface

The GUI of the Service Bus Explorer is composed of following parts:

1. The **Menu** bar contains the main menu that allows the user to select commands and options. In particular, the commands displayed in the **Actions** menu vary based on the selected entity.
2. The **Service Bus Namespace** panel contains a **TreeView** control that allows to navigate, browse and select entity nodes. Each node is right-click enabled and offers the possibility to access a context-menu which commands depend on the selected entity.
3. The Main panel displays the information for the selected entity. Based on the command selected by the user, it provides the ability to create, delete or test the current entity. In case of relay services, the Service Bus Explorer tool only supports the possibility to test the service endpoints defined in the Service Bus registry by a given relay service using the ServiceRegistrySettings endpoint behavior.
4. The **Log** panel displays information and error messages. The log functionality comes in handy especially when you use the Service Bus Explorer to test existing queues, topics and relay services to trace send and receive operations performed by individual sender and receiver tasks. The Log Panel can be hidden by deselecting the **Log Window** option under the **View** menu.

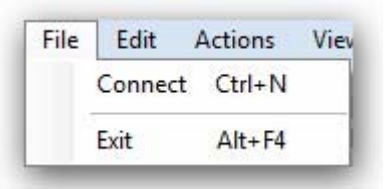


Once you connected to an existing Service Bus namespace, you can use the tool to perform the following operations.

Menu Bar

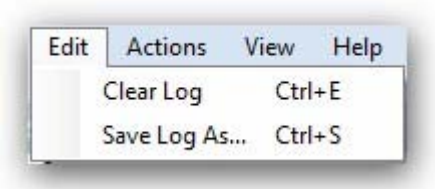
The **Menu** bar contains the following menus:

- **File**:



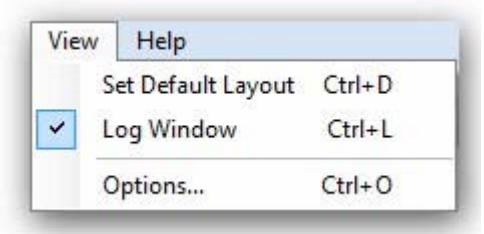
- **Connect**: opens a modal dialog that allows to connect to a given Service Bus namespace.
- **Exit**: closes the application.

- **Edit**:



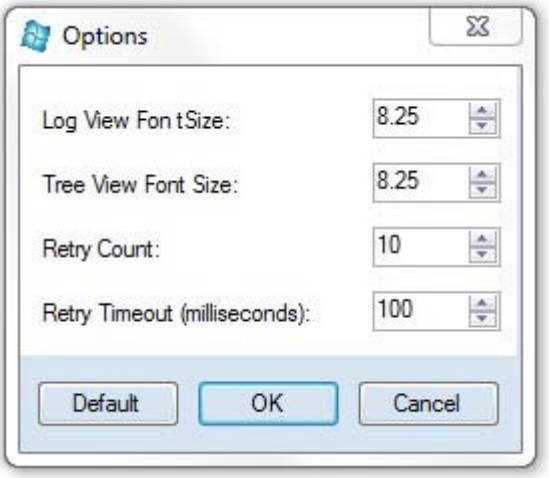
- **Clear Log**: clears the log view.
- **Save Log As…**: saves the content of the log to a text file.

- **Actions**: contains the same commands available in the context menu of the currently selected node in the treeview.
- **View**:



- **Set Default Layout**: reset the layout to the default configuration
- Log Window: enable or disable the display of the log window.

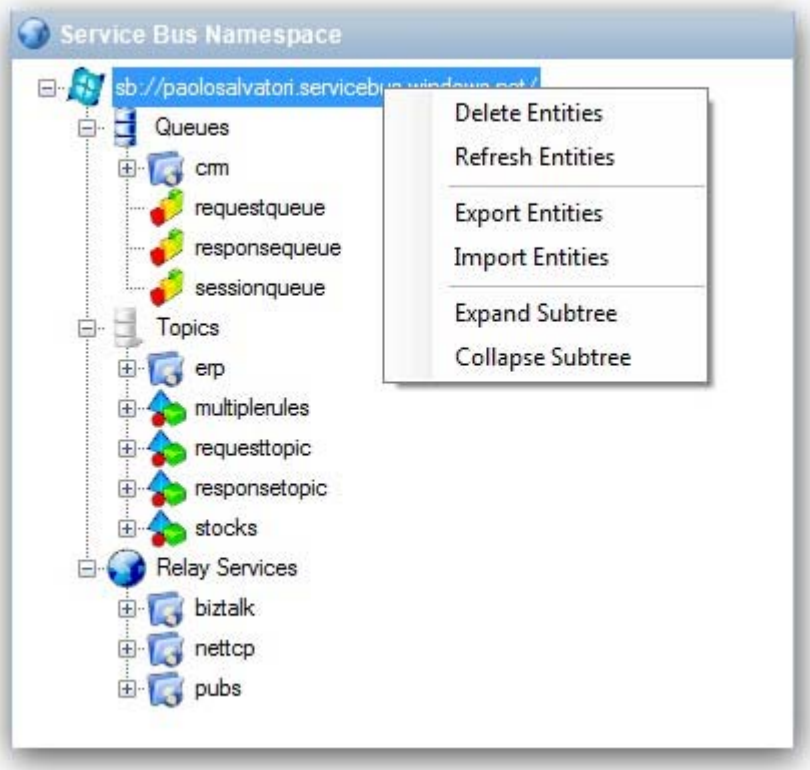
- **Options**: shows a modal dialog that allows to define the layout parameters:



- **Help**: contains the About Service Bus Explorer command.

Namespace Commands

By right clicking the namespace root node in the **Service Bus Namespace** panel, you can open the following context menu. The commands provided by the context menu are also available under the **Actions** menu in the **Menu** bar.



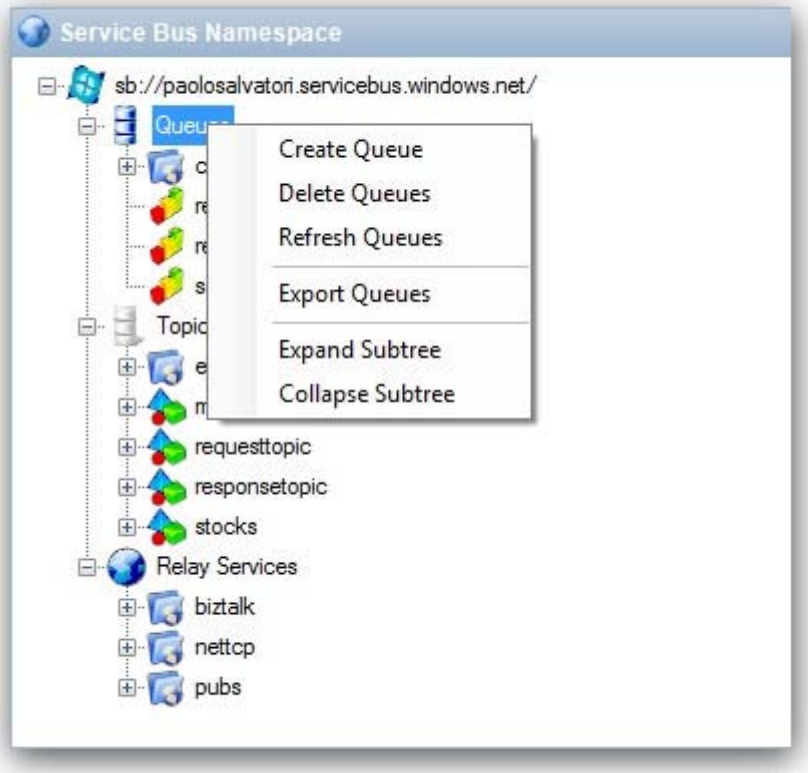
The context menu provides access to the following commands:

- **Delete Entities**: as shown in the picture below, prompts the user to confirm whether to delete all the entities in the current **Service Bus namespace**.

- **Refresh Entities**: refreshes all the entities (queues, topics, subscriptions, rules, relay services) in the current Service Bus namespace.
- **Export Entities**: exports all the entities in the current Service Bus namespace to an XML file.
- **Import Entities**: import entities from an XML file. The file in question can contain the definition of one or multiple queues and topics.
- **Expand Subtree**: expands the subtree below the namespace root node.
- **Collapse Subtree**: collapse the subtree below the namespace root node.

Queues Commands

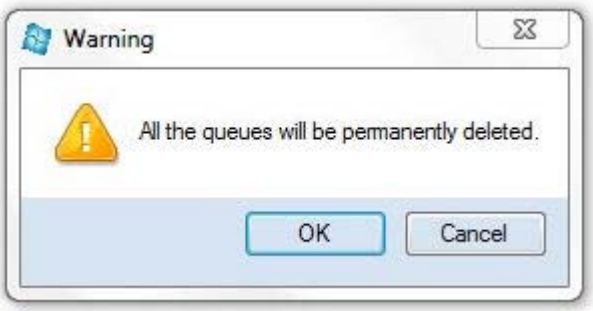
By right clicking the **Queues** node in the Service Bus Namespace panel, you can open the following context menu. The commands provided by the context menu are also available under the **Actions** menu in the **Menu** bar.



The context menu provides access to the following commands:

Create Queue: displays a custom control in the **Main Panel** that allows to specify the properties of a new queue. For more information on how to create a new queue, see later in this article.

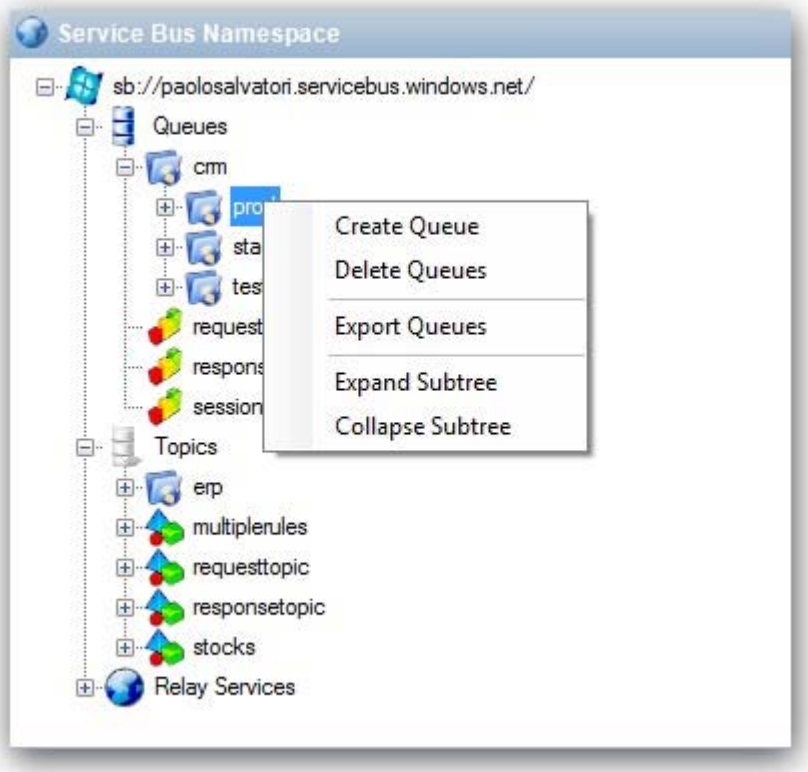
Delete Queues: prompts the user to confirm whether to delete all the queues in the current Service Bus namespace.



- **Refresh Queues**: refreshes all the queues in the current Service Bus namespace.
- **Export Queues**: exports all the queues in the current Service Bus namespace to an XML file.
- **Expand Subtree**: expands the subtree below the **Queues** node.
- **Collapse Subtree**: collapse the subtree below the **Queues** node.

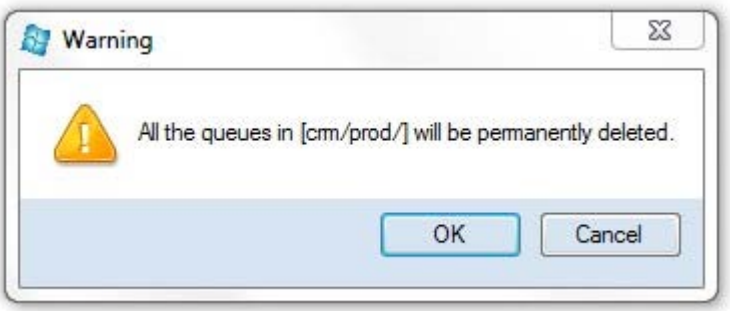
Queue Path Segment Commands

By right clicking a **Path Segment node** in the **Queues** subtree, as shown in the picture below, you can open a context menu. The commands provided by the context menu are also available under the **Actions** menu in the **Menu** bar.



The context menu provides access to the following commands:

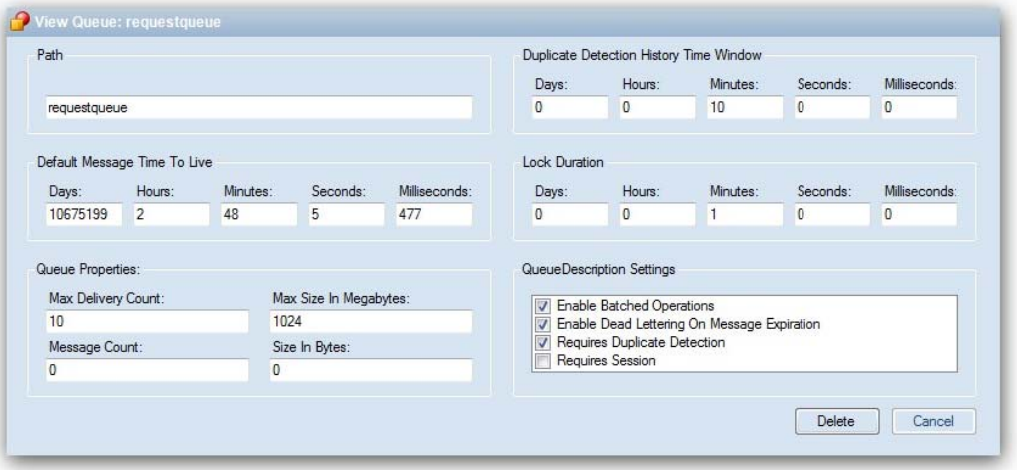
- **Create Queue**: displays a custom control in the Main Panel that allows to specify the properties of a new queue. In particular, it initializes the value of the queue Name field with the absolute path of the selected node (e.g. crm/prod in the sample above). For more information on how to create a new queue, see later in this article.
- **Delete Queues**: as shown in the picture below, prompts the user to confirm whether to delete all the queues in the current path.



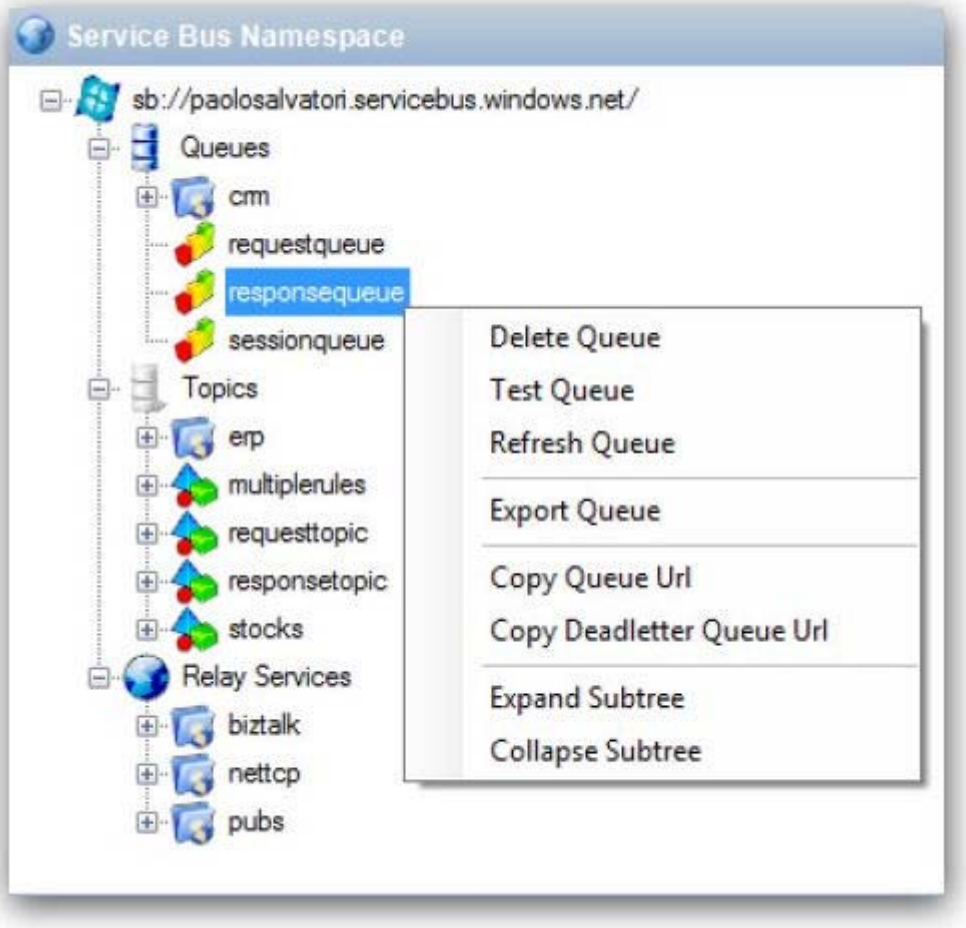
- **Export Queues**: exports all the queues in the current path to an XML file.
- **Expand Subtree**: expands the subtree below the selected path segment node.
- **Collapse Subtree**: collapse the subtree below the selected path segment node.

Queue Commands

If you click an existing **Queue** node in the **Service Bus Namespace** panel, its information will be displayed in the **Main** panel, as shown in the picture below.

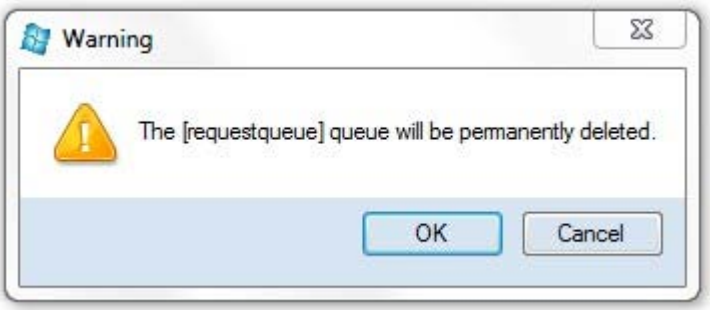


By right clicking a **Queue** node in the Queues subtree, you can open the following context menu. The commands provided by the context menu are also available under the Actions menu in the Menu bar.



The context menu provides access to the following commands:

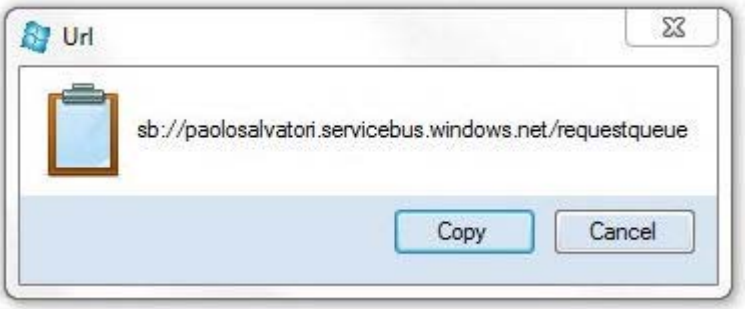
- **Delete Queue**: as shown in the picture below , prompts the user to confirm whether to delete the selected queue in the current Service Bus namespace.



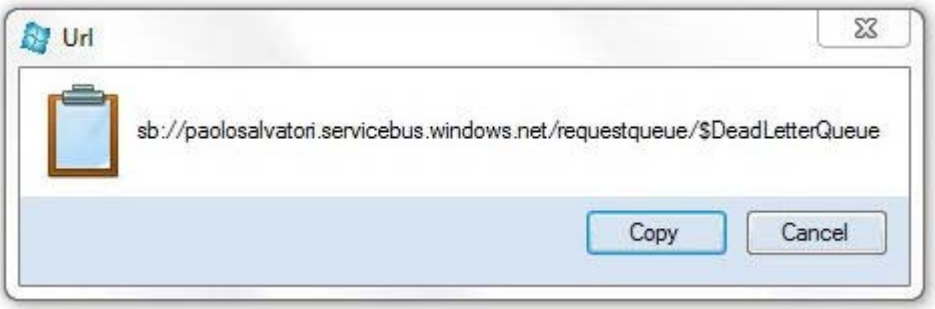
- **Test Queue**: displays a custom control in the Main Panel that allows to test the selected queue by sending and receiving messages. The tool allows the user to select a wide range of testing options. In addition, the Service Bus Explorer enables to monitor the progress and performance of an ongoing test run by using the information traced in the log view. Finally, the tool allows to display real-time performance data, such as latency and throughput for

sender and receiver tasks, simply by enabling statistics and chart features. For more information on how to test a queue, see later in this article

- **Refresh Queue**: refreshes the current information for the selected queue. This feature can be used to refresh the value of the MessageCount and SizeInBytes properties for the current queue.
- **Export Queue**: exports the current queue definition to an XML file.
- **Copy Queue Url**: as shown in the picture below, this command copies the url of the current queue to the clipboard.



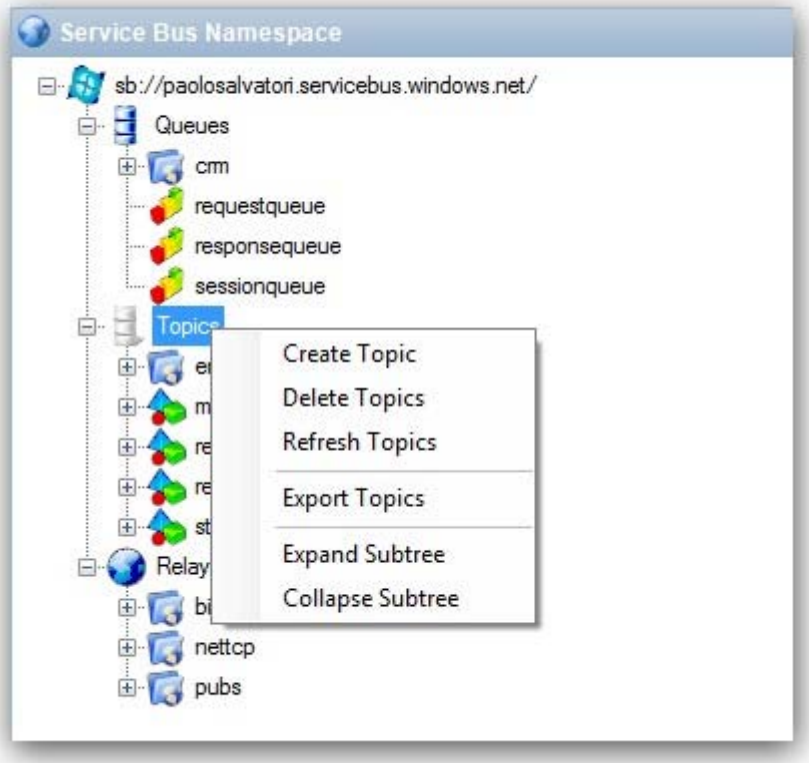
- **Copy Deadletter Queue Url**: as shown in the picture below, this command copies the url of the **deadletter** queue for the current queue to the clipboard.



- **Expand Subtree**: expands the subtree below the **Queues** node.
- **Collapse Subtree**: collapse the subtree below the **Queues** node.

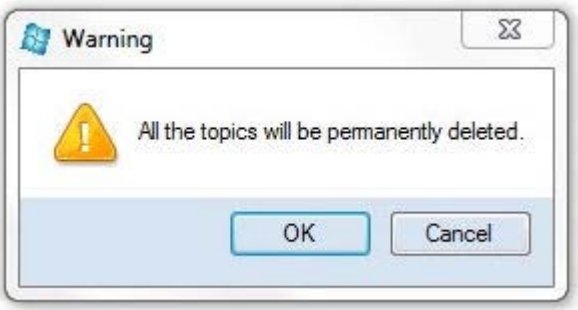
Topics Commands

By right clicking the **Topics** node in the **Service Bus Namespace** panel, you can open the following context menu. The commands provided by the context menu are also available under the Actions menu in the Menu bar.



The context menu provides access to the following commands:

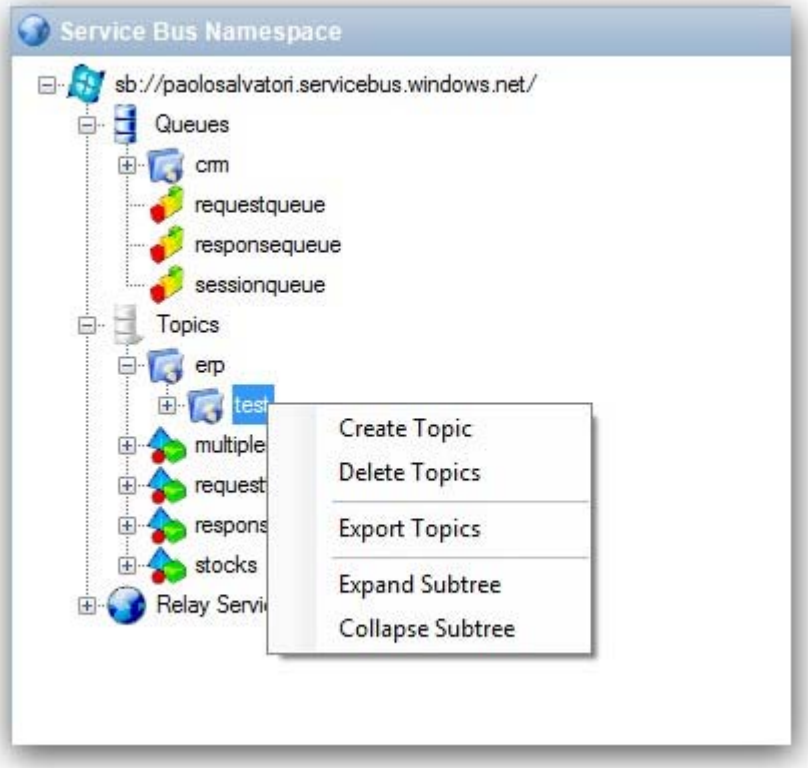
- **Create Topic**: displays a custom control in the Main Panel that allows to specify the properties of a new topic. For more information on how to create a new topic, see later in this article.
- **Delete Topics**: as shown in the picture below, prompts the user to confirm whether to delete all the topics in the current Service Bus namespace.



- **Refresh Topics**: refreshes all the topics in the current Service Bus namespace.
- **Export Topics**: exports all the topics in the current Service Bus namespace to an XML file.
- **Expand Subtree**: expands the subtree below the **Topics** node.
- **Collapse Subtree**: collapse the subtree below the **Topics** node.

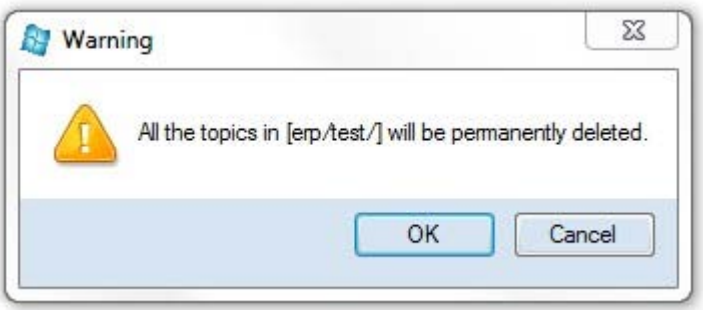
Topic Path Segment Commands

By right clicking a **Path Segment** node in the **Topics** subtree, as shown in the picture below, you can open a context menu. The commands provided by the context menu are also available under the **Actions** menu in the main Menu bar.



The context menu provides access to the following commands:

- **Create Topic**: displays a custom control in the Main Panel that allows to specify the properties of a new topic. In particular, it initializes the value of the topic Name field with the absolute path of the selected node (e.g. erp/test in the sample above). For more information on how to create a new topic, see later in this article.
- **Delete Topics**: as shown in the picture below, prompts the user to confirm whether to delete all the topics in the current path.



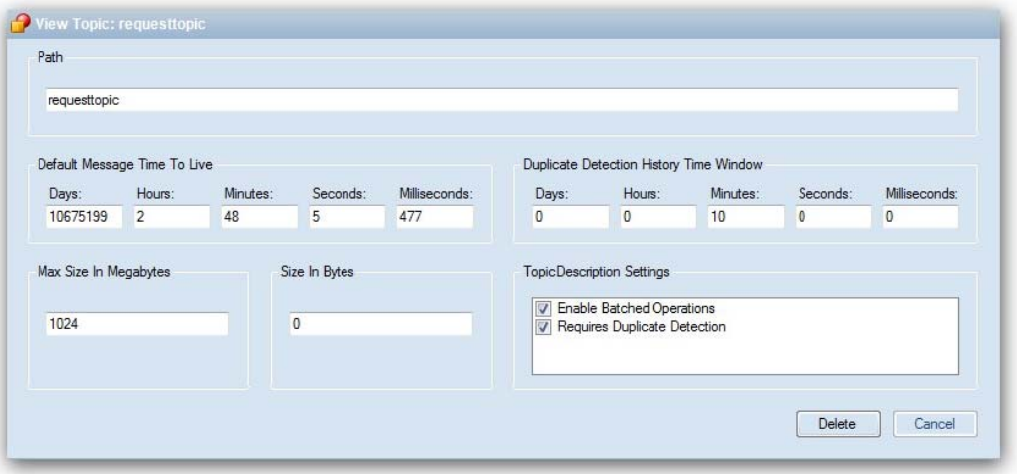
Export Topics: exports all the topics in the current path to an XML file.

Expand Subtree: expands the subtree below the selected path segment node.

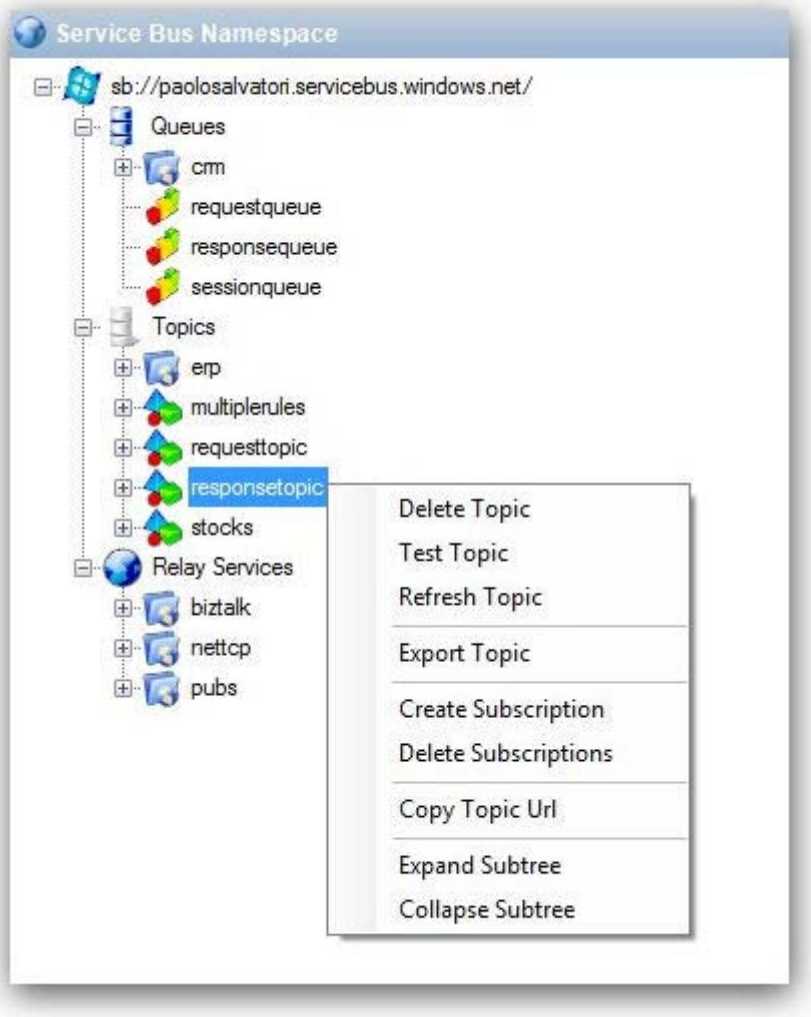
Collapse Subtree: collapse the subtree below the selected path segment node.

Topic Commands

If you click an existing Topic node in the Service Bus Namespace panel, its information will be displayed in the Main panel, as shown in the picture below.

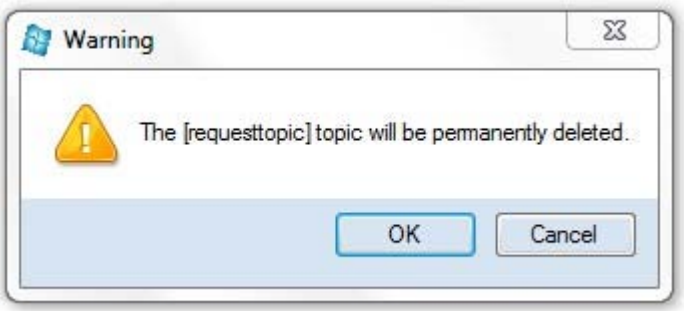


By right clicking a Topic node in the Topics subtree, you can open the following context menu. The commands provided by the context menu are also available under the Actions menu in the Menu bar.



The context menu provides access to the following commands:

- **Delete Topic**: as shown in the picture below, prompts the user to confirm whether to delete the selected topic in the current Service Bus namespace.

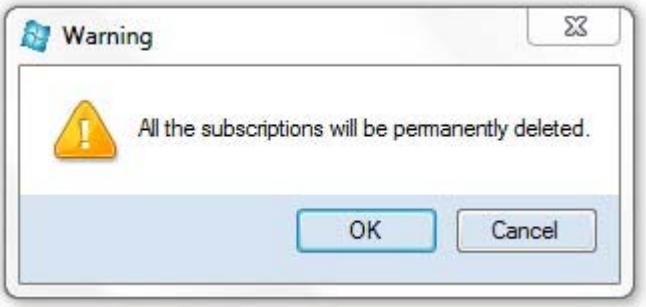


- Test Topic: displays a custom control in the Main Panel that allows to test the selected topic by sending and receiving messages. The tool allows the user to select a wide range of testing options. In addition, the Service Bus Explorer enables to monitor the progress and

performance of an ongoing test run by using the information traced in the log view. Finally, the tool allows to display real-time performance data, such as latency and throughput for sender and receiver tasks, simply by enabling statistics and chart features. For more information on how to test a queue, see later in this article.

- Refresh Topic: refreshes the current information for the selected topic. This feature can be used to refresh the value of the SizeInBytes property for the current topic.
- **Export Topic**: exports the current topic definition to an XML file along with its subscriptions.
- **Create Subscription**: displays a custom control in the Main Panel that allows to specify the properties of a new subscription for the selected topic. For more information on how to create a new subscription, see later in this article.

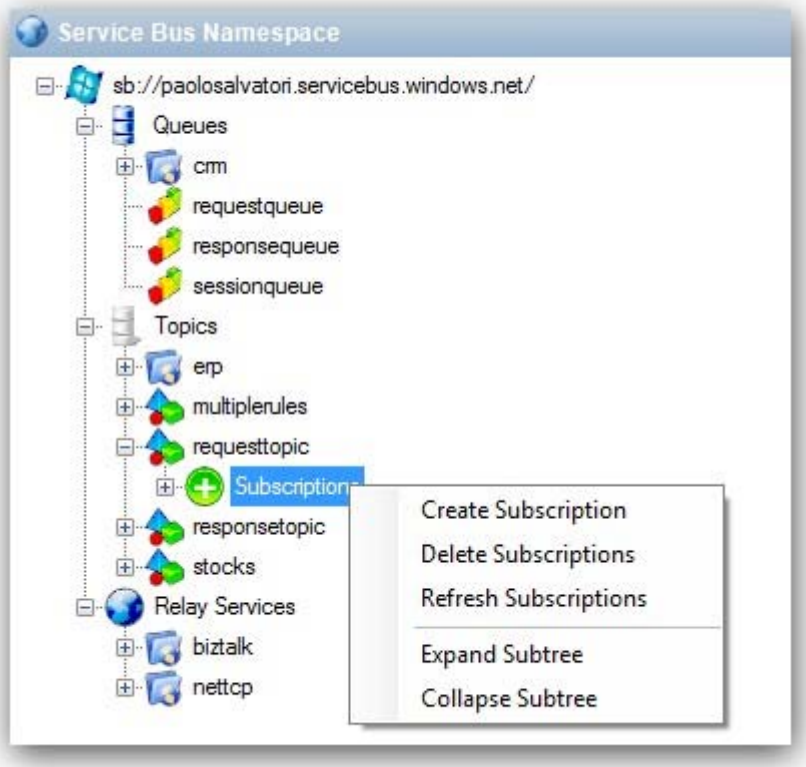
Delete Subscriptions: as shown in the picture below, prompts the user to confirm whether to delete all the subscriptions for the selected topic.



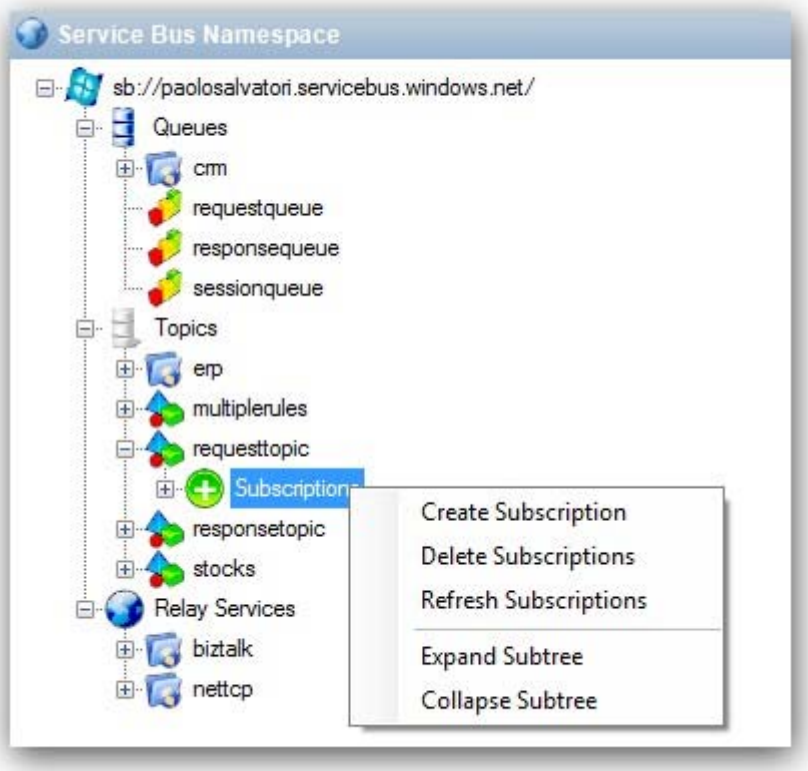
- **Copy Topic Url**: as shown in the picture below, this command copies the url of the current topic to the clipboard.
- **Expand Subtree**: expands the subtree below the Topics node.
- **Collapse Subtree**: collapse the subtree below the Topics node.

Subscriptions Commands

By right clicking a **Subscriptions** node as shown in the picture below, you can open the following context menu. The commands provided by the context menu are also available under the **Actions** menu in the **Menu** bar.



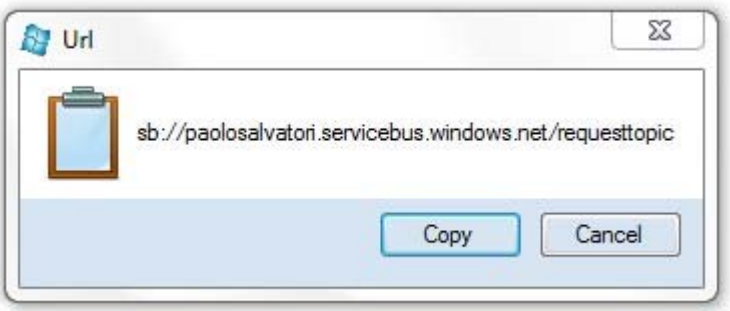
- The context menu provides access to the following commands:
- **Create Subscription**: displays a custom control in the Main Panel that allows to specify the properties of a new subscription for the current topic. For more information on how to create a new subscription, see later in this article.
- **Delete Subscriptions**: as shown in the picture below, prompts the user to confirm whether to delete all the subscriptions for the current topic.



- **Refresh Subscriptions**: refreshes the current information for the subscriptions of the current topic.
- **Expand Subtree**: expands the subtree below the Subscriptions node.
- **Collapse Subtree**: collapse the subtree below the Subscriptions node.

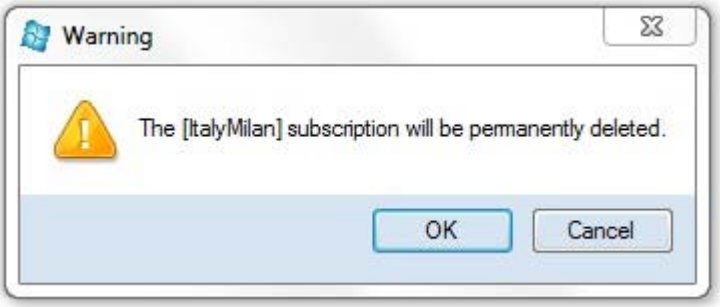
Subscription Commands

By right clicking a Subscription node as shown in the picture below, you can open the following context menu. The commands provided by the context menu are also available under the Actions menu in the Menu bar.

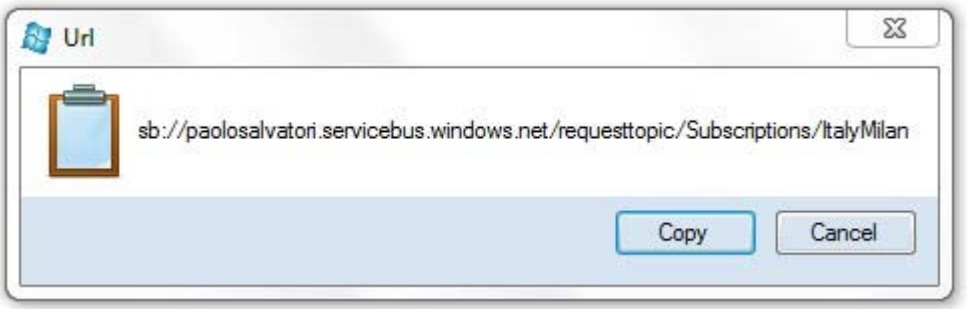


The context menu provides access to the following commands:

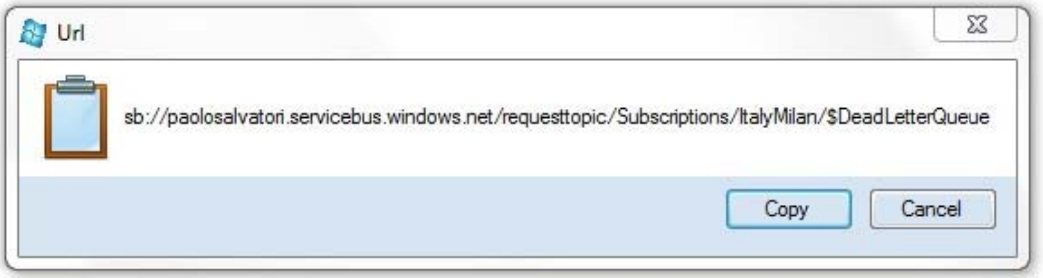
- **Delete Subscription**: as shown in the picture below, prompts the user to confirm whether to delete the selected subscription.



- **Test Subscription**: displays a custom control in the Main Panel that allows to test the selected subscription by receiving messages from it. The tool allows the user to select a wide range of testing options. In addition, the Service Bus Explorer enables to monitor the progress and performance of an ongoing test run by using the information traced in the log view. Finally, the tool allows to display real-time performance data, such as latency and throughput for receiver tasks, simply by enabling statistics and chart features. For more information on how to test a queue, see later in this article.
- **Refresh Subscription**: refreshes the current information for the selected subscription. This feature can be used to refresh the value of the MessageCount property for the current subscription.
- **Add Rule**: displays a custom control in the Main Panel that allows to specify the properties of a new rule for the currently selected subscription. For more information on how to create a new topic, see later in this article.
- **Copy Queue Url**: as shown in the picture below, this command copies the url of the current queue to the clipboard.



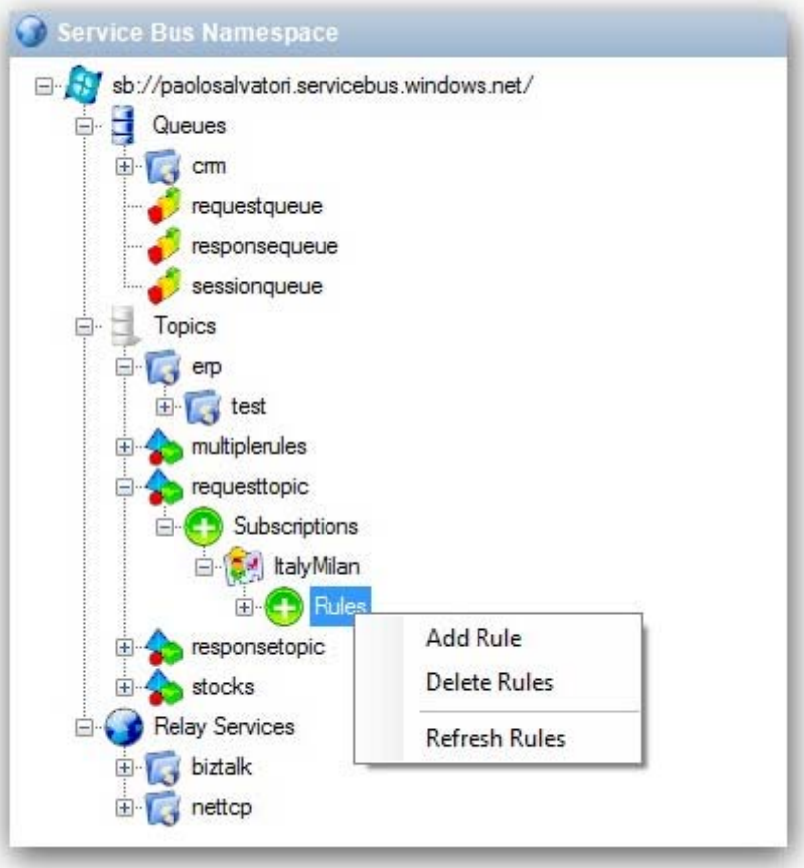
Copy Deadletter Queue Url: as shown in the picture below, this command copies the url of the deadletter queue for the current queue to the clipboard.



- **Expand Subtree**: expands the subtree below the Subscription node.
- **Collapse Subtree**: collapse the subtree below the Subscription node

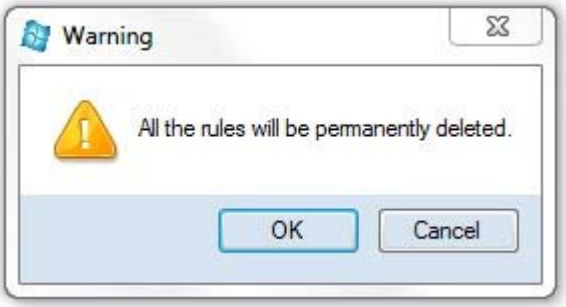
Rules Commands

By right clicking a Rules node as shown in the picture below, you can open the following context menu. The commands provided by the context menu are also available under the **Actions** menu in the **Menu** bar.



The context menu provides access to the following commands:

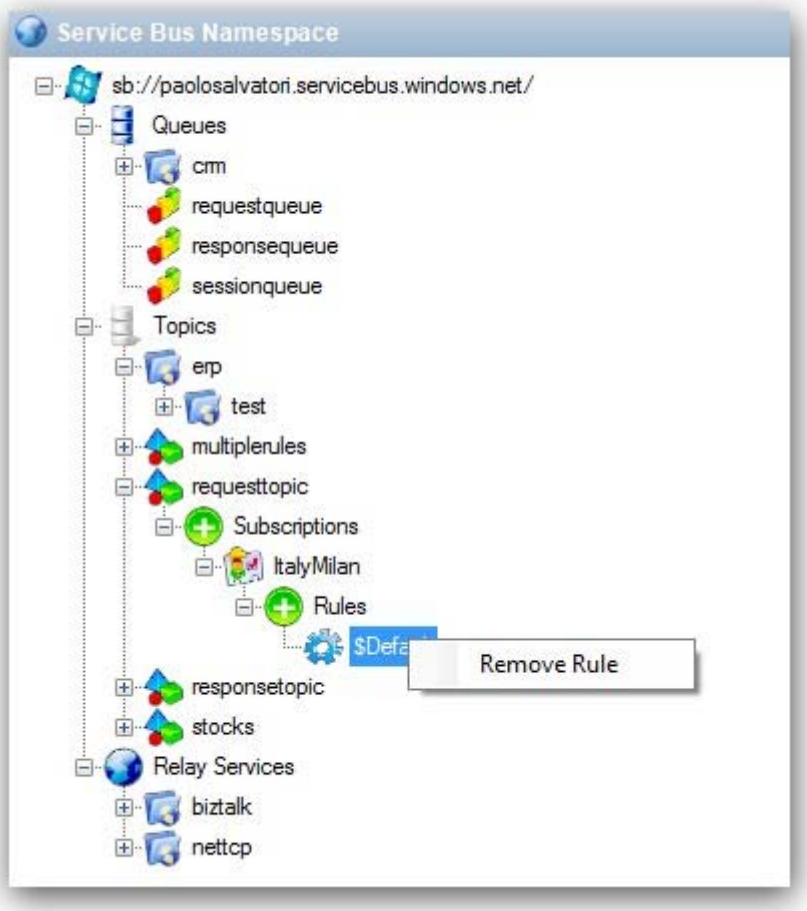
- **Add Rule**: displays a custom control in the Main Panel that allows to specify the properties of a new rule for the currently selected subscription. For more information on how to create a new topic, see later in this article.
- **Delete Rules**: as shown in the picture below, prompts the user to confirm whether to delete all the rules for the current subscription.



- **Refresh Rules**: refreshes the current information for the rules of the current topic.

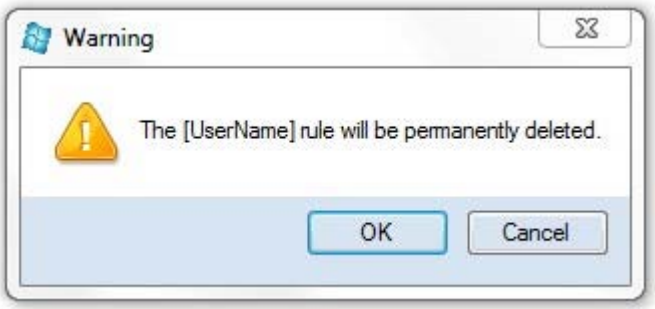
Rule Commands

By right clicking a **Rule** node as shown in the picture below, you can open the following context menu. The commands provided by the context menu are also available under the **Actions** menu in the **Menu** bar.



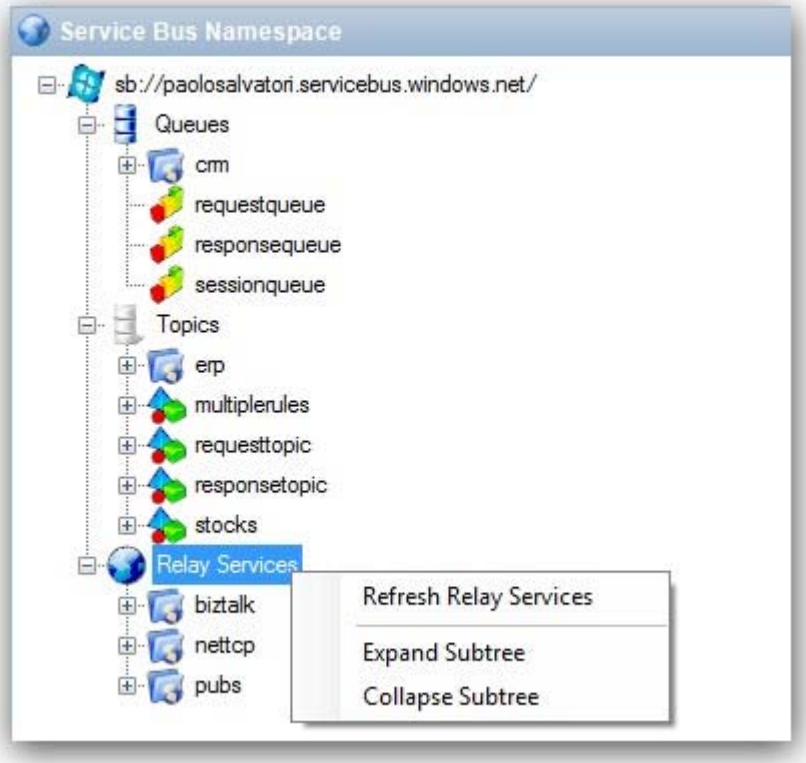
The context menu provides access to the following commands:

Remove Rule: as shown in the picture below, prompts the user to confirm whether to remove the selected rule.



Relay Services Commands

By right clicking the Relay Services node in the Service Bus Namespace panel, you can open the following context menu. The commands provided by the context menu are also available under the Actions menu in the Menu bar.



The context menu provides access to the following commands:

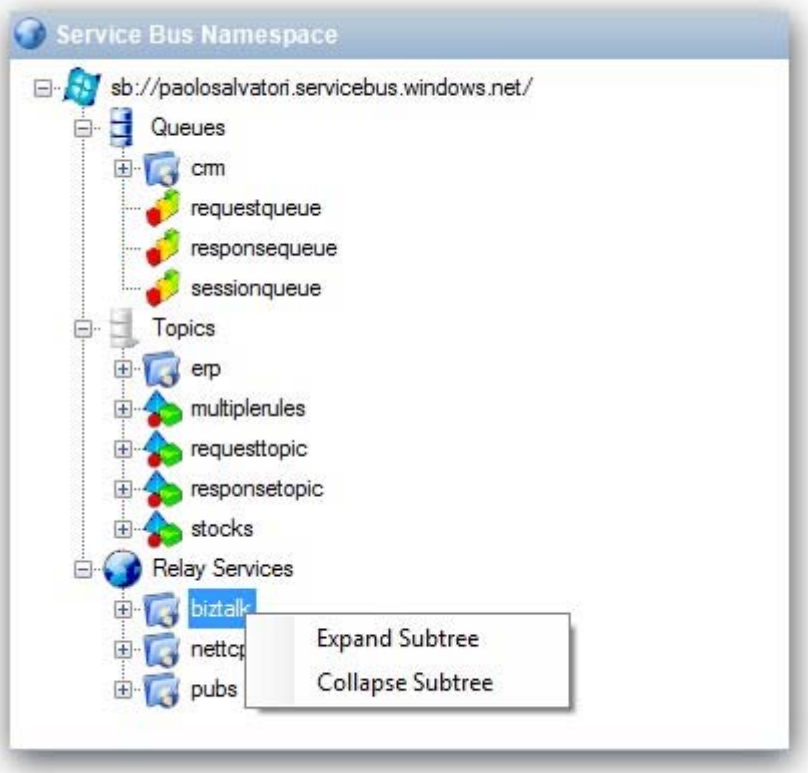
Refresh Relay Services: refreshes all the relay services in the current Service Bus namespace. In order to appear in the treeview, a relay service needs to use the ServiceRegistrySettings endpoint behavior to define its public endpoints in the Service Bus registry.

Expand Subtree: expands the subtree below the Relay Services node.

Collapse Subtree: collapse the subtree below the Relay Services node.

Relay Service Path Segment Commands

By right clicking a **Path Segment** node in the Relay Service subtree, as shown in the picture below, you can open the following context menu. The commands provided by the context menu are also available under the **Actions** menu in the **Menu** bar.



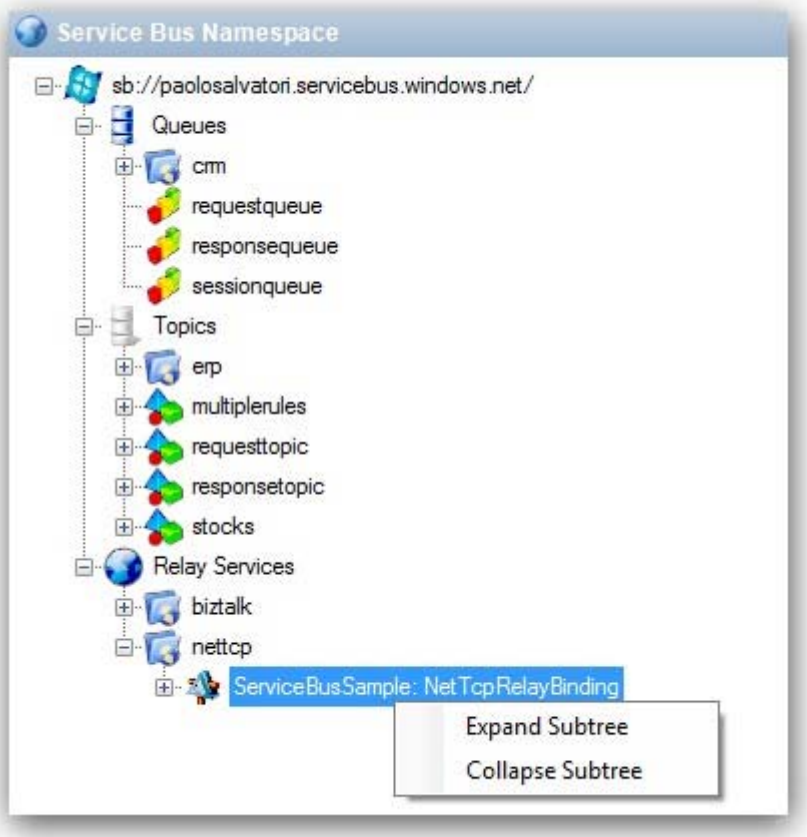
The context menu provides access to the following commands:

Expand Subtree: expands the subtree below the selected path segment node.

Collapse Subtree: collapse the subtree below the selected path segment node.

Relay Service Commands

By right clicking a Relay Service node as shown in the picture below, you can open the following context menu. The commands provided by the context menu are also available under the Actions menu in the Menu bar.

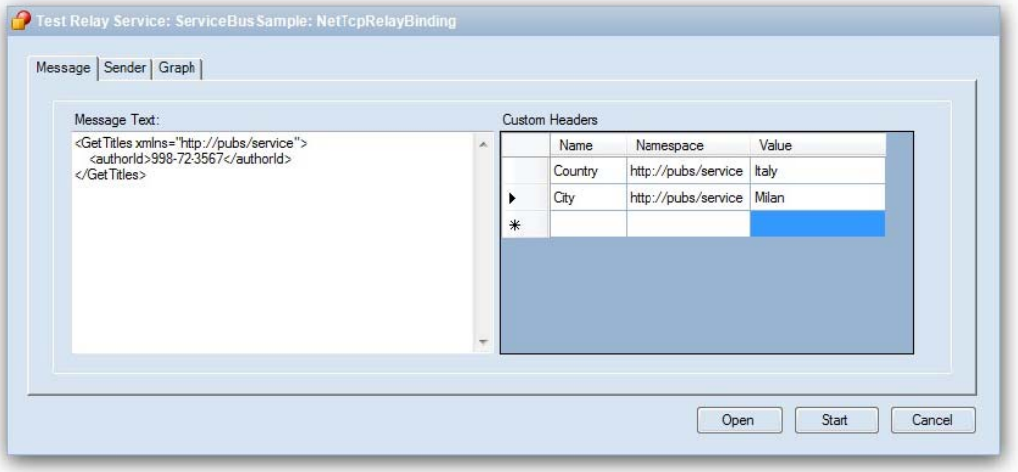


The context menu provides access to the following commands:

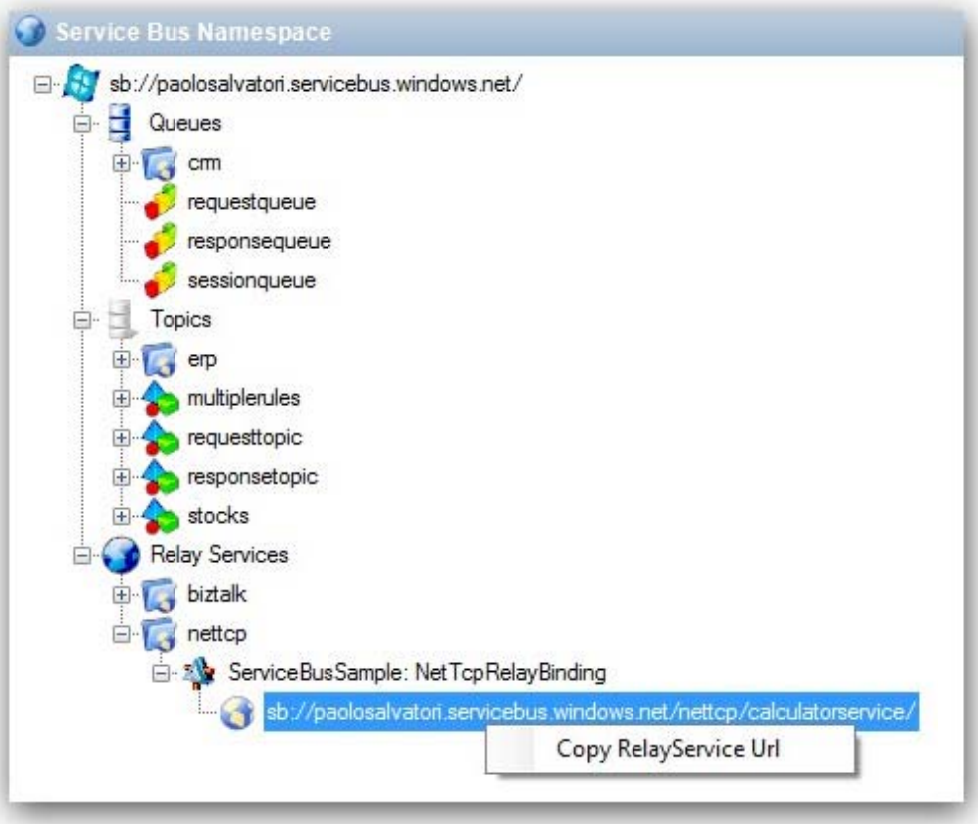
- **Expand Subtree**: expands the subtree below the selected path segment node.
- **Collapse Subtree**: collapse the subtree below the selected path segment node.

Relay Service Endpoint Commands

If you click an Endpoint node of a relay service in the Service Bus Namespace panel, the following custom control appears in the Main panel. This control allows to configure a client endpoint and send messages to the selected endpoint. For more information on how to test a relay service, see later in this article

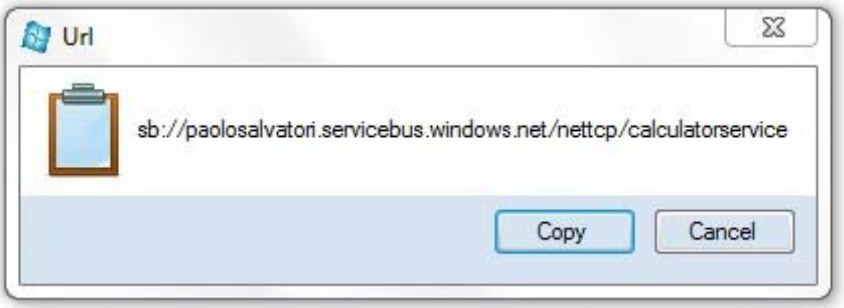


By right clicking an Endpoint node of a relay service, you can open the following context menu. The commands provided by the context menu are also available under the Actions menu in the Menu bar.



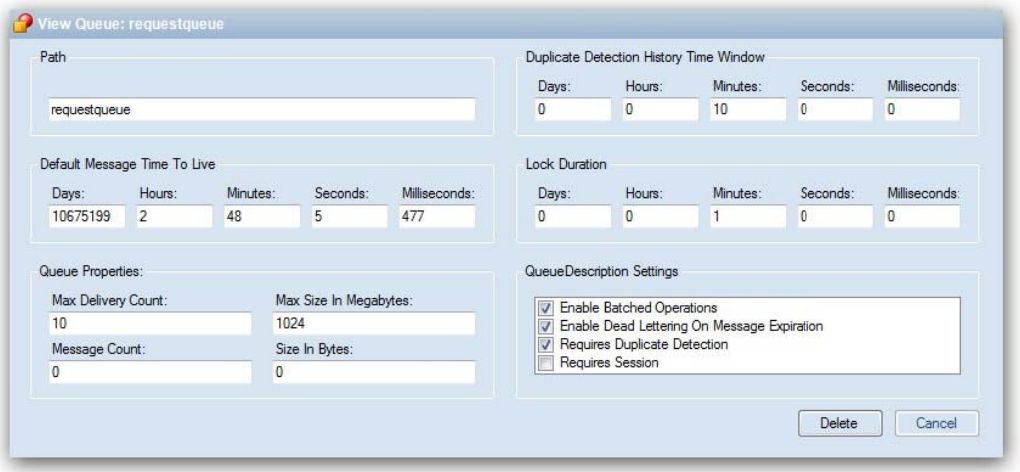
The context menu provides access to the following commands:

- **Copy Relay Service Url**: as shown in the picture below, this command copies the url of the current service endpoint to the clipboard.



View a Queue

To view the properties of an existing queue in the Main Panel, you can just select the corresponding node in the treeview.



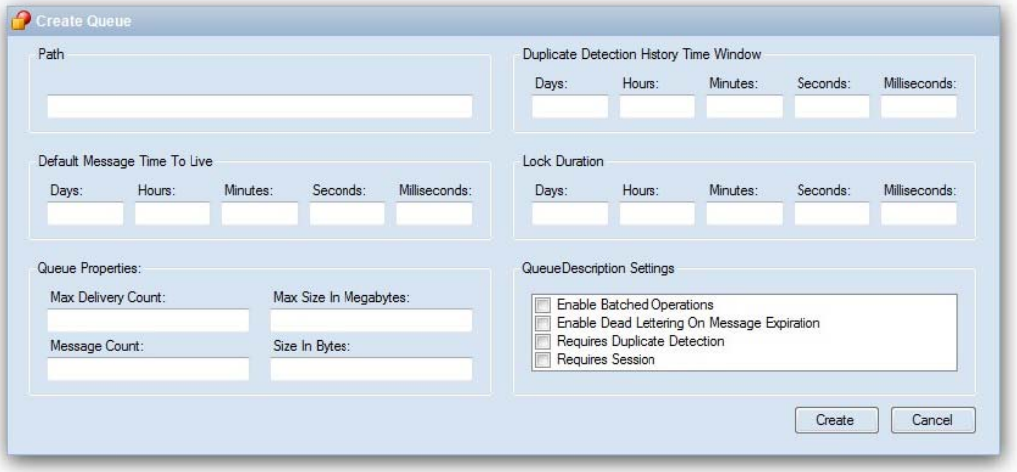
The current version of the Service Bus does not allow to change the properties of an existing queue once created. The same principle applies to the other entities (topics, subscriptions, rules).

Create a Queue

To create a new queue in the current namespace, you can select the **Queues** node or a path segment node under the **Queues** subtree and then perform one of the following actions:

1. Right-click the node and choose **Create Queue** from the context menu.
2. Choose **Create Queue** command under the Actions menu.

This operation displays a custom control (see the picture below) in the Main Panel that allows the user to specify the following properties for the new queue:



- **Path**: this field defines the path for the queue within the namespace. The value can be an absolute path like erp/prod/queues/orderqueue.
- **Duplicate Detection History Time Window**: this section gives the possibility to enter a TimeSpan value for the DuplicateDetectionHistoryTimeWindow property that defines the duration of the duplicate detection history.
- Default Message Time To Live: this section gives the possibility to enter a TimeSpan value for the DefaultMessageTimeToLive property that specifies the default message time to live.
- **Lock Duration**: this section gives the possibility to enter a TimeSpan value the LockDuration property that defines the duration of the lock used by a consumer when using the PeekLock receive mode. In the ReceiveAndDelete receive mode, a message is deleted from the queue as soon as it is read by a consumer. Conversely, in the PeekLock receive mode, a message is hidden from other receivers until the timeout defined by the LockDuration property expires. By that time the receiver should have deleted the message invoking the Complete method on the BrokeredMessage object or called the Abandon method to release the lock.
- **Max Delivery Count**: the MaxDeliveryCount property defines the maximum delivery count. A message is automatically deadlettered after this number of deliveries.
- **Max Size In Megabytes**: the MaxSizeInMegabytes property defines the maximum size in megabytes, which is the size of memory allocated for the queue.
- **Enable Batched Operations**: the EnableBatchedOperations property accepts a Boolean value that indicates whether server-side batched operations are enabled.
- **Enable Dead Lettering on Message Expiration**: this checkbox specifies a Boolean value for the `EnableDeadLetteringOnMessageExpiration` property that enables or disables the dead-lettering functionality on message expiration.
- **Requires Duplicate Detection**: this checkbox specifies a Boolean value for the RequiresDuplicateDetection property that enables or disables the duplicate message detection feature.
- **Requires Session**: this checkbox specifies a Boolean value for the RequiresSession property that enables or disables sessions support on the queue being created.

The MessageCount and SizeInBytes are read-only properties and cannot be set when creating a queue. In addition, since metadata cannot be changed once a messaging entity is created,

modifying the duplicate detection behavior requires deleting and recreating a queue. The same principle applies to the other entities. If you don't explicitly specify any value for one or more fields, when you click the Create button, the default value will be used.

Delete a Queue

To delete an existing queue, you can select the corresponding node in the treeview and then perform one of the following actions:

- Right-click the node and choose **Delete Queue** from the context menu.
- Choose **Delete Queue** command under the Actions menu.
- Click the **Delete** button in the Main Panel.

Test a Queue

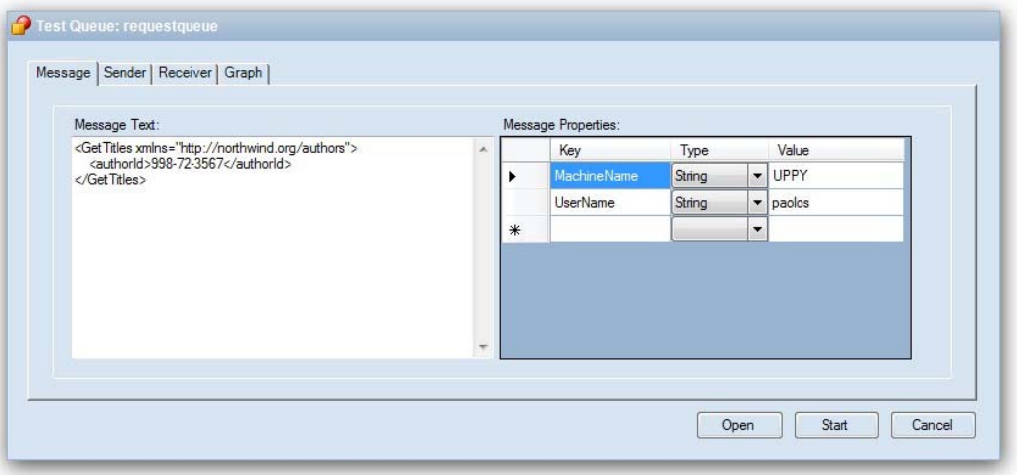
To test a queue in the current namespace, you can select the corresponding node in the treeview and then perform one of the following actions:

- Right-click the node and choose **Test Queue** from the context menu.
- Choose **Test Queue** command under the **Actions** menu.

This operation shows a custom control in the Main panel that is composed of four tabs:

- Message Tab
- Sender Tab
- Receiver Tab
- Graph Tab

Message Tab



The Message tab allows the user to specify the following information:

- **Message Text**: specifies the payload of a BrokeredMessage as a TEXT or XML message. The tool provides the ability to read the message from the file system by pressing the Open button and selecting a file.

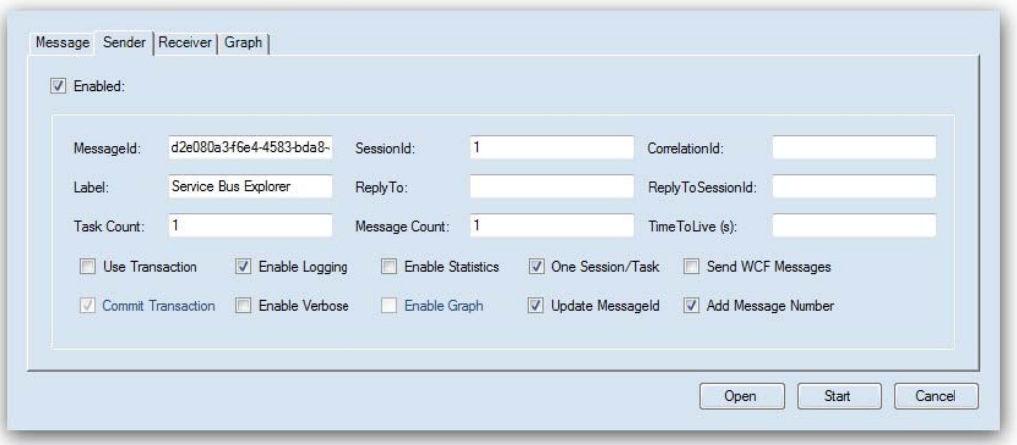
- **Message Properties**: you can use the datagrid to define the key\value pairs that will be added to the Properties dictionary of a BrokeredMessage object. The Properties collection allows to define application specific message properties. This is probably the most important feature of a BrokeredMessage entity as user-defined properties can be used for the following:
- Carry the payload of a message. In this context, the body of the message could be empty.
- Define application specific properties that can be used by a worker process to decide how to process the current message.

Specify filter and action expressions that can be used to define routing and data enrichment rules at a subscription level.

Note

As indicated in Windows Azure AppFabric Service Bus Quotas, the maximum size for each property is 32K. Cumulative size of all properties cannot exceed 64K. This applies to the entire header of the BrokeredMessage, which has both user properties as well as system properties (such as SequenceNumber,Label, MessageId, and so on). The space occupied by properties counts towards the overall size of the message and its maximum size of 256K. If an application exceeds any of the limits mentioned above, a SerializationException exception is generated, so you should expect to handle this error condition..

Sender Tab



The Sender tab allows the user to specify the following information:

- **Enabled**: this checkbox allows enabling or disabling sender tasks when testing a queue.
- **MessageId**: this field specifies a value for the MessageId property of a BrokeredMessage object. When a queue is configured to use the duplicate detection mechanism, this property is used to individuate duplicates or already received messages.
- **SessionId**: this field specifies a value for the SessionId property of a BrokeredMessage object. A SessionId is a logical and lightweight way to correlate messages to each other. For example, the sender can set SessionId = "SessionId" of several messages as an indicator that these messages are related to each other.

- **CorrelationId**: the CorrelationId property can be used to implement a request-reply message exchange pattern where the client application uses the MessageId property of an outgoing request message and the CorrelationId property of an incoming response message to correlate the two messages.
- Label: specifies a value for the Label property of a BrokeredMessage object.
- **ReplyTo**: the ReplyTo property can be used to define the address of the queue to reply to. In an asynchronous request-reply scenario where a client application A sends a request message to a server application B via a Service Bus queue or topic and waits for a response message, by convention the client application A can use the ReplyTo property of the request message to indicate to the server application B the address of the queue or topic where to send the response.
- **ReplyToSessionId**: by definition, the ReplyToSessionId property of a BrokeredMessage object can be used by a client application to indicate to a service the session identifier to reply to. When a client application A sends a flow of request messages to a server application B via a session-enabled queue or topic and waits for the correlated response messages on a separate session-enabled queue or a subscription, the client application A can assign the id of the receive session to the ReplyToSessionId property of outgoing messages to indicate to the application B the value to assign to the SessionId property of response messages.
- **Task Count**: this field can be used to define the number of publisher tasks that will be used to send messages. Every task runs on a separate .NET thread pool thread.
- **Message Count**: this integer field can be used to specify the number of messages to write to the queue.
- **Time to Live**: specifies the value in seconds for the TimeToLive property of a BrokeredMessage object. This is the duration after which the message expires, starting from when the message is sent to the Service Bus. Messages older than their TimeToLive value will expire and no longer be retained in the message store. Subscribers will be unable to receive expired messages.
- **Use Transaction**: when this option is checked, publisher tasks will use a TransactionScope object to send a batch of messages within a transaction context. For more information, see the code snippet in the table below.
- **Commit Transaction**: when this option is checked, publisher tasks will invoke the Complete method to commit the transaction and persist messages to the queue. Please note that when a publisher sends one or multiple messages within a TransactionScope and at the end does not call the Complete method, messages are not written to the queue. The Use Transaction and Commit Transaction options can be used to test the behavior of send-side transactions. For more information, see the code snippet in the table below.

```
using (var scope = new TransactionScope())
{
    serviceBusHelper.SendMessages(...);
    ...
    if (checkBoxSenderCommitTransaction.Checked)
    {
        scope.Complete();
        ...
```

```
    }
    ...
}
```

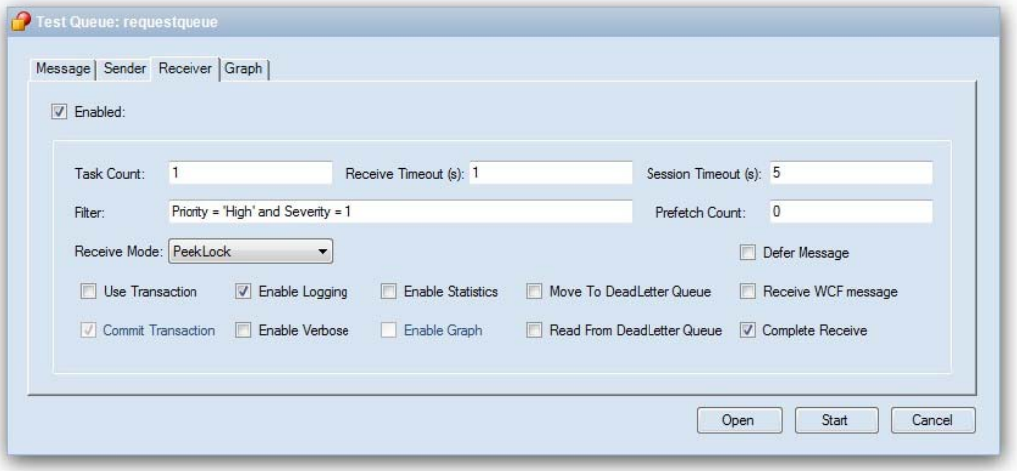
- **Enable Logging**: when this option is checked, sender tasks will track the operations performed in the log view.
- **Enable Verbose**: when this option is checked, sender tasks will log the payload and properties of transmitted messages.
- **Enable Statistics**: when this option is checked, sender tasks will update statistics in the Graph tab.
- **Enable Graph**: when this option is checked, sender tasks will update the latency and throughput charts in the Graph tab.
- **One Session/Task**: when this option is checked, each sender task will use a different value for the SessionId property. This feature allows to test session-enabled queues with multiple sender and consumer tasks.
- **Update MessageId**: when this option is checked, the tool will generate a different MessageId for each outbound message. You can disable this option when testing the duplicate message detection feature of a queue.
- **Add Message Number**: when this option is checked, sender tasks will add the actual message number as a user-defined property to the Properties collection of the outgoing messages.
- **Send WCF Messages**: when this option is checked, outgoing messages are wrapped in a SOAP envelope. This feature can be used to send messages to one or multiple consumers that use WCF and NetMessagingBinding to receive messages from a queue. The following table shows the code used by sender tasks to create a WCF message from a `BrokeredMessage` object used as template. As you can see, the WCF message needs to be encoded using a message encoder that depends on the protocol scheme used by a consumer to receive and decode the message.

```
/// <summary>
/// Create a BrokeredMessage for a WCF receiver.
/// </summary>
/// <param name="messageTemplate">The message template.</param>
/// <param name="taskId">The task Id.</param>
/// <param name="updateMessageId">Indicates whether to use a unique id for each
message.</param>
/// <param name="oneSessionPerTask">Indicates whether to use a different session for
each sender task.</param>
/// <param name="messageText">The message text.</param>
/// <param name="to">The name of the target topic or queue.</param>
/// <returns>The cloned BrokeredMessage object.</returns>
public BrokeredMessage CreateMessageForWcfReceiver(BrokeredMessage messageTemplate,
```

```
                                                        int taskId,
                                                        bool updateMessageId,
                                                        bool oneSessionPerTask,
                                                        string messageText,
                                                        string to)
{

    if (!IsXml(messageText))
    {
        throw new ApplicationException(MessageIsNotXML);
    }

    MessageEncodingBindingElement element;
    if (scheme == DefaultScheme)
    {
        element = new BinaryMessageEncodingBindingElement();
    }
    else
    {
        element = new TextMessageEncodingBindingElement();
    }
    using (var stringReader = new StringReader(messageText))
    {
        using (var xmlReader = XmlReader.Create(stringReader))
        {
            using (var dictionaryReader =
XmlDictionaryReader.CreateDictionaryReader(xmlReader))
            {
                var message = Message.CreateMessage(MessageVersion.Default, "*",
dictionaryReader);
                message.Headers.To = new Uri(namespaceUri, to);
                var encoderFactory = element.CreateMessageEncoderFactory();
                var encoder = encoderFactory.Encoder;
                var outputStream = new MemoryStream();
                encoder.WriteMessage(message, outputStream);
                outputStream.Seek(0, SeekOrigin.Begin);
                var outboundMessage = new BrokeredMessage(outputStream, true)
                {
                    ContentType = encoder.ContentType
                };
                if (!string.IsNullOrEmpty(messageTemplate.Label))
                {
```

```
                    outboundMessage.Label = messageTemplate.Label;
                }
                outboundMessage.MessageId = updateMessageId ?
Guid.NewGuid().ToString() : messageTemplate.MessageId;
                outboundMessage.SessionId = oneSessionPerTask ? taskId.ToString() :
messageTemplate.SessionId;
                if (!string.IsNullOrEmpty(messageTemplate.CorrelationId))
                {
                    outboundMessage.CorrelationId = messageTemplate.CorrelationId;
                }
                if (!string.IsNullOrEmpty(messageTemplate.ReplyTo))
                {
                    outboundMessage.ReplyTo = messageTemplate.ReplyTo;
                }
                if (!string.IsNullOrEmpty(messageTemplate.ReplyToSessionId))
                {
                    outboundMessage.ReplyToSessionId =
messageTemplate.ReplyToSessionId;
                }
                outboundMessage.TimeToLive = messageTemplate.TimeToLive;
                foreach (var property in messageTemplate.Properties)
                {
                    outboundMessage.Properties.Add(property.Key, property.Value);
                }
                return outboundMessage;
            }
        }
    }
}
```

Receiver Tab



The Receiver tab allows the user to specify the following information:

- **Enabled**: this checkbox allows enabling or disabling receiver tasks when testing a queue. When using the Service Bus Explorer tool to send messages to a real application, make sure to uncheck this option.
- **Task Count**: this field can be used to define the number of consumer tasks that will be used to receive messages from the queue.
- **Receive Timeout**: specifies the time span in seconds that consumer tasks wait before a time out occurs. When this happens, consumer tasks stop receiving messages from the queue. This value corresponds to the TimeSpan parameter of the Receive method exposed by the MessageReceiver and QueueClient classes.
- **Session Timeout**: specifies the time span in seconds that consumer tasks wait for a new session before a time out occurs. This value corresponds to the TimeSpan parameter of the AcceptMessageSession method exposed by the QueueClient class.
- **Filter**: the filter field can be used in conjunction with the Defer Message and Move to Deadletter Queue options to determine, respectively, the messages to defer or move to deadletter queue. This field accepts the SQL92 standard syntax accepted by the SqlExpression property of anSqlFilter object. Note: as shown in the following code snippet, a consumer application can use a SqlFilter to check at runtime whether a BrokeredMessage object matches a given condition.

```
...
var sqlFilter = new SqlFilter(!string.IsNullOrEmpty(txtFilterExpression.Text)
                              ? txtFilterExpression.Text
                              : DefaultFilterExpression);
sqlFilter.Validate();
filter = sqlFilter.Preprocess();
...
if (filter.Match(inboundMessage))
```

```
{
    inboundMessage.Defer();
    ...
}
```

- **Prefetch Count**: specifies a value for the PrefetchCount property.of the MessageReceiver objects used by receiver tasks to consume messages from the current queue. This number indicates the number of messages that the message receiver can simultaneously request.
- **Receive Mode**: indicates the receive mode (PeekLock or ReceiveAndDelete) used by consumer tasks.
- **Use Transaction**: when this option is checked, consumer tasks will use a TransactionScope object to receive a batch of messages within a transaction context. For more information, see the code snippet in the table below.
- **Commit Transaction**: when this option is checked, the consumer tasks will invoke the Complete method to commit the transaction and delete messages from the queue. Please note that when a consumer task receives one or multiple messages within a TransactionScope and at the end doesn't call the Complete method, messages are not delete from the queue. The Use Transaction and Commit Transaction options can be used to test the behavior of receive-side transactions. For more information, see the code snippet in the table below.

```
using (var scope = new TransactionScope())
{
    serviceBusHelper.ReceiveMessages(...);
    ...
    if (checkBoxReceiverCommitTransaction.Checked)
    {
        scope.Complete();
        ...
    }
    ...
}
```

- **Enable Logging**: when this option is checked, receiver tasks will track the operations performed in the log view.
- **Enable Verbose**: when this option is checked, receiver tasks will log the payload and properties of transmitted messages.
- **Enable Statistics**: when this option is checked, receiver tasks will update statistics in the Graph tab.
- **Enable Graph**: when this option is checked, receiver tasks will update the latency and throughput charts in the Graph tab.

- **Move to Deadletter queue**: when this option is checked, consumer tasks will move messages matching the SQL expression contained in the Filter field to the deadletter queue.
- **Read from Deadletter queue**: when this option is checked, consumer tasks will read messages from the deadletter queue. Consider disabling sender tasks when using this option.
- **Defer Message**: when this option is selected, consumer tasks will defer messages matching the SQL expression defined in the Filter field. Deferred messages will be processed at the end after the messages that do not match the filter expression. In order to save the SequenceNumber of deferred messages, consumer tasks use the following helper class. You can replace this component with another class that implements the custom IMessageDeferProvider interface. In this case, you need to indicate the fully-qualified-name of the new provider class in the messageDeferProvider setting in the configuration file.

```
public interface IMessageDeferProvider
{
    int Count { get; }
    void Enqueue(long sequenceNumber);
    bool Dequeue(out long sequenceNumber);
}

public class InMemoryMessageDeferProvider : IMessageDeferProvider
{
    #region Private Fields
    private readonly Queue<long> queue = new Queue<long>();
    #endregion

    #region Public Properties
    public int Count
    {
        get
        {
            lock (this)
            {
                return queue.Count;
            }
        }
    }
    #endregion

    #region Public Methods
    public bool Dequeue(out long sequenceNumber)
    {
        sequenceNumber = -1;
```

```
        lock (this)
        {
            if (queue.Count > 0)
            {
                sequenceNumber = queue.Dequeue();
                return true;
            }
        }
        return false;
    }

    public void Enqueue(long sequenceNumber)
    {
        lock (this)
        {
            queue.Enqueue(sequenceNumber);
        }
    }
    #endregion
}
```

- **Read WCF Message**: when this option is checked, incoming messages are extracted from a SOAP envelope. This feature can be used to receive messages from an application that used WCF and NetMessagingBinding to send messages to a queue. The following table shows the code used by consumer tasks to extract a BrokeredMessage object from a WCF message. As you can see, the WCF message needs to be decoded using a message encoder that depends on the protocol scheme used by a WCF publisher to encode and send the message.

```
MessageEncodingBindingElement element;
if (scheme == DefaultScheme)
{
    element = new BinaryMessageEncodingBindingElement();
}
else
{
    element = new TextMessageEncodingBindingElement();
}
var encoderFactory = element.CreateMessageEncoderFactory();
encoder = encoderFactory.Encoder;
...
var stream = inboundMessage.GetBody<Stream>();
if (stream != null)
```

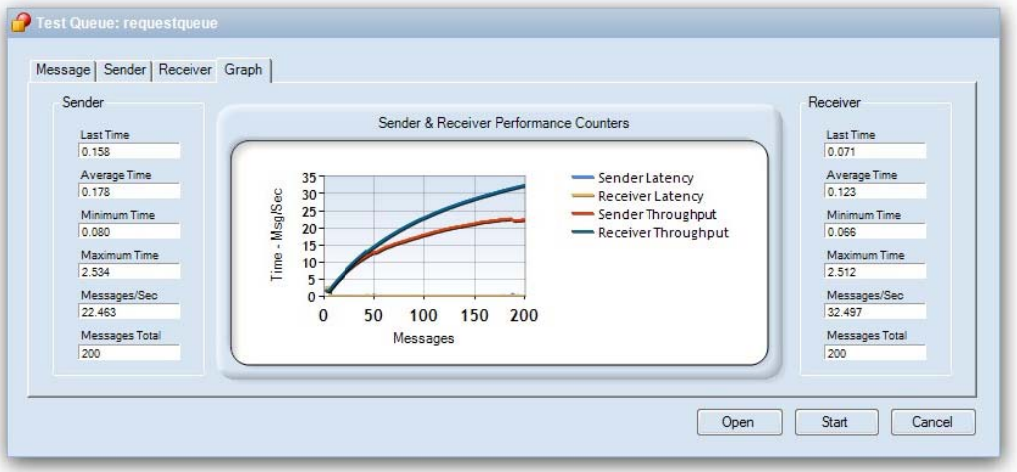
```
{
    var stringBuilder = new StringBuilder();
    var message = encoder.ReadMessage(stream, MaxBufferSize);
    using (var reader = message.GetReaderAtBodyContents())
    {
        // The XmlWriter is used just to indent the XML message
        var settings = new XmlWriterSettings { Indent = true };
        using (var writer = XmlWriter.Create(stringBuilder, settings))
        {
            writer.WriteNode(reader, true);
        }
    }
    messageText = stringBuilder.ToString();
}
```

- **Complete Receive**: when this option is checked and the selected receive mode is equal to PeekLock, consumer tasks will call the Complete method on received messages to complete the receive operation and delete the message from the queue. Conversely, when this option is unchecked, consumer tasks will call the Abandon method on received messages to and relinquish the lock. This option can be used to test and examine the behavior of the Complete and Abandon methods of the BrokeredMessage class.

Graph Tab

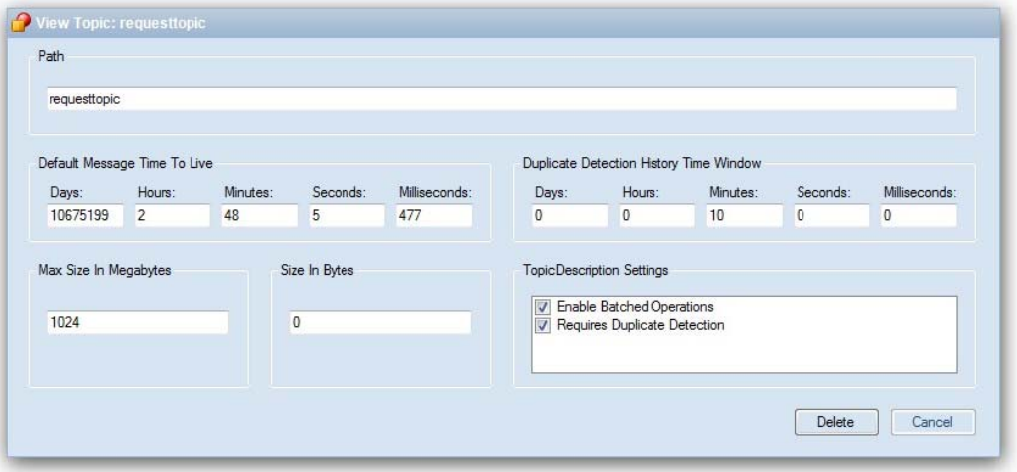
When the Enable Statistics and Enable Graph options are enabled, this tab displays performance statistics for sender and receiver tasks and four distinct chart series:

- Sender Latency
- Receiver Latency
- Sender Throughput
- Receiver Throughput



View a Topic

To view the properties of an existing topic in the Main Panel, you can just select the corresponding node in the treeview.



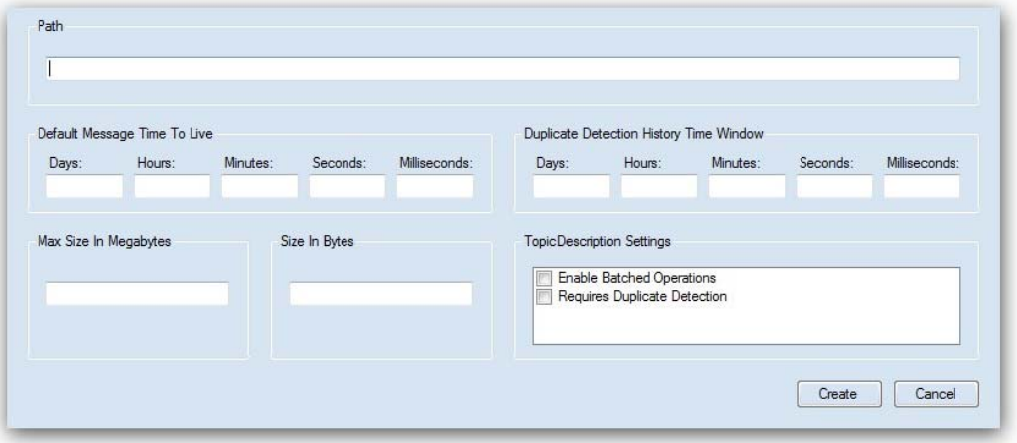
The current version of the Service Bus does not allow to change the properties of an existing topic once created. The same principle applies to the other entities (queues, subscriptions, rules).

Create a Topic

To create a new topic in the current namespace, you can select the Topics node or a path segment node under the Topics subtree and then perform one of the following actions:

1. Right-click the node and choose **Create Topic** from the context menu.
2. Choose **Create Topic** command under the **Actions** menu.

This operation displays a custom control (see the picture below) in the Main Panel that allows the user to specify the following properties for the new topic:



- **Path**: this field defines the path for the topic within the namespace. The value can be an absolute path like erp/prod/topics/ordertopic.
- Default Message Time To Live: this section gives the possibility to enter a TimeSpan value for the DefaultMessageTimeToLive property that specifies the default message time to live.
- **Duplicate Detection History Time Window**: this section gives the possibility to enter a TimeSpan value for the DuplicateDetectionHistoryTimeWindow property that defines the duration of the duplicate detection history.
- Max Size In Megabytes: the MaxSizeInMegabytes property defines the maximum size in megabytes, which is the size of memory allocated for the topic.
- **Enable Batched Operations**: the EnableBatchedOperations property accepts a Boolean value that indicates whether server-side batched operations are enabled.
- Requires Duplicate Detection: this checkbox specifies a Boolean value for the RequiresDuplicateDetection property that enables or disables the duplicate message detection feature.

The SizeInBytes is a read-only property and as such cannot be set when creating a topic. If you don't explicitly specify any value for one or more fields, when you click the Create button, the default value will be used.

Delete a Topic

To delete an existing topic, you can select the corresponding node in the treeview and then perform one of the following actions:

1. Right-click the node and choose Delete Topic from the context menu.
2. Choose Delete Topic command under the Actions menu.
3. Click the Delete button in the Main Panel.

Test a Topic

To test a topic in the current namespace, you can select the corresponding node in the treeview and then perform one of the following actions:

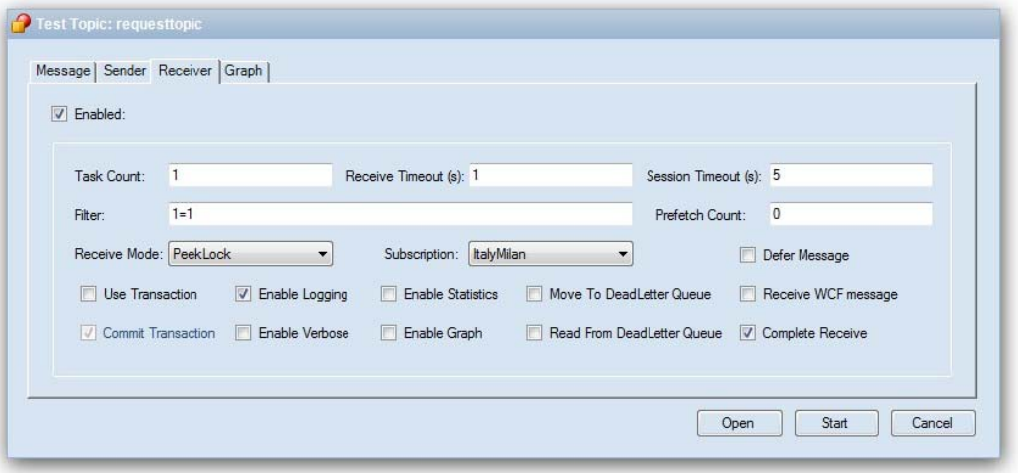
1. Right-click the node and choose Test Topic from the context menu.

2. Choose Test Topic command under the Actions menu.

This operation shows a custom control in the Main panel that is composed of four tabs:

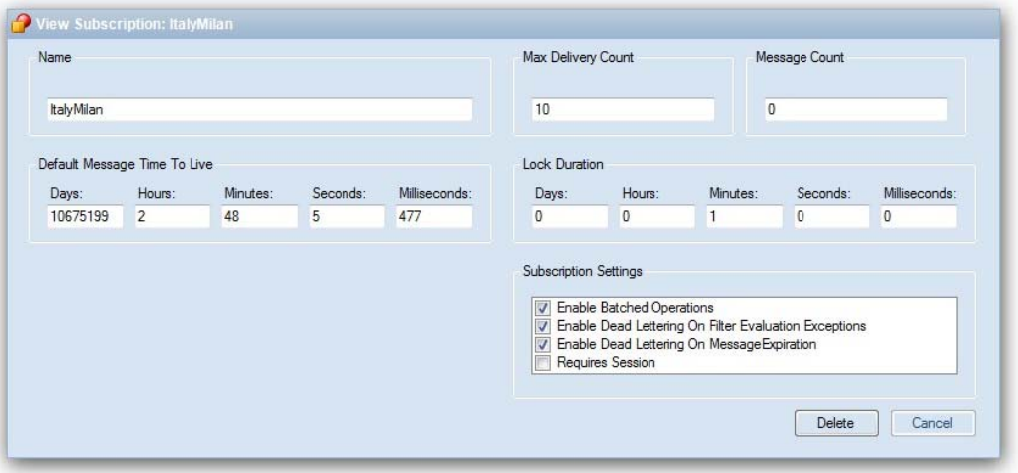
- Message Tab
- Sender Tab
- Receiver Tab
- Graph Tab

The module that allows to test a topic is virtually identical to that we described above and that can be used to test a queue. The only addition is the ability to choose a Subscription in the Receiver tab as shown in the following picture.



View a Subscription

To view the properties of a subscription associated to a given topic, you can expand the Subscriptions node under the topic node and select the corresponding subscription node.



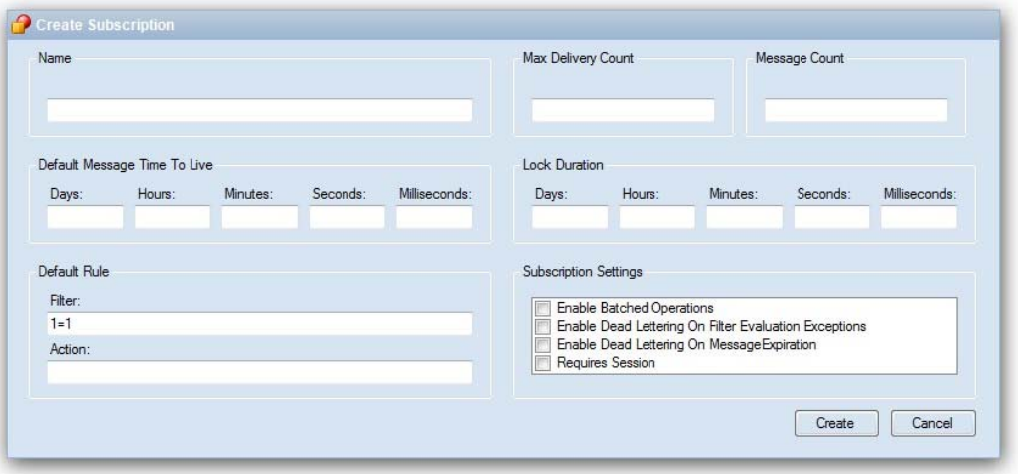
The current version of the Service Bus does not allow to change the properties of an existing subscription once created. The same principle applies to the other entities (queues, topics, rules).

Create a Subscription

To add a new subscription to an existing topic, you can select the topic node in the treeview and then perform one of the following actions:

- Right-click the topic node and choose Create Subscription from the context menu.
- Expand the topic node, right-click the Subscriptions node under the latter and choose Create Subscription from the context menu.
- Choose Create Subscription command under the Actions menu.

This operation displays a custom control (see the picture below) in the Main panel that allows the user to specify the following properties for the new subscription:



- **Name**: this field defines the name for the subscription being created.
- **Max Delivery Count**: the MaxDeliveryCount property defines the maximum delivery count. A message is automatically deadlettered after this number of deliveries.
- **Default Message Time To Live**: this section gives the possibility to enter a TimeSpan value for the DefaultMessageTimeToLive property that specifies the default message time to live.
- **Lock Duration**: this section gives the possibility to enter a TimeSpan value the LockDuration property that defines the duration of the lock used by a consumer when using the PeekLock receive mode.
- **Enable Batched Operations**: the EnableBatchedOperations property accepts a Boolean value that indicates whether server-side batched operations are enabled.
- **Enable Dead Lettering on Message Expiration**: this checkbox specifies a Boolean value for the `EnableDeadLetteringOnMessageExpiration` property that enables or disables the dead-lettering functionality on message expiration.
- **Requires Duplicate Detection**: this checkbox specifies a Boolean value for the RequiresDuplicateDetection property that enables or disables the duplicate message detection feature.

- **Requires Session**: this checkbox specifies a Boolean value for the RequiresSession property that enables or disables sessions support on the subscription being created.
- **Filter**: this field defines the SQL filter expression for the default rule.
- **Action**: this field defines the SQL filter action for the default rule. This field is optional.

Delete a Subscription

To remove an existing subscription, you can select the corresponding node in the treeview and then perform one of the following actions:

1. Right-click the node and choose **Delete Subscription** from the context menu.
2. Choose **Delete Subscription** command under the Actions menu.
3. Click the **Delete** button in the Main panel.

Test a Subscription

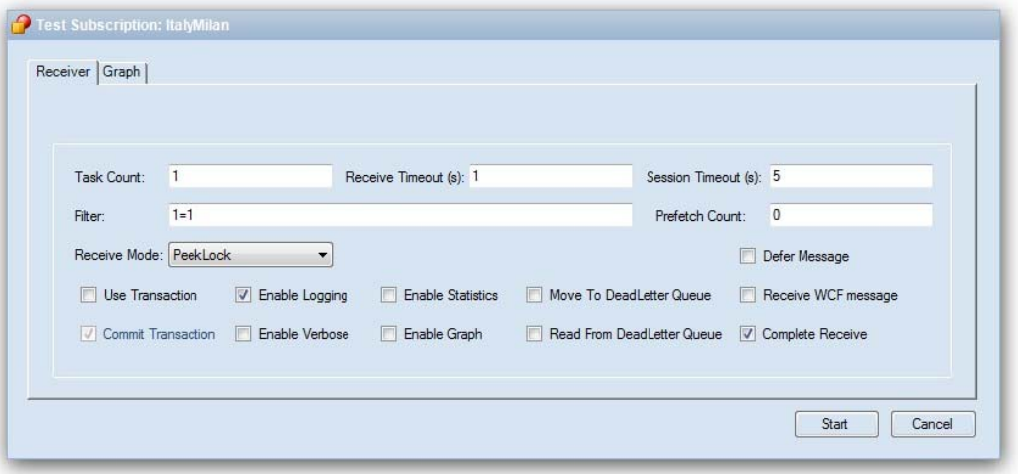
To test a subscription, you can select the corresponding node in the treeview and then perform one of the following actions:

1. Right-click the node and choose **Test Subscription** from the context menu.
2. Choose **Test Subscription** command under the Actions menu.

This operation shows a custom control in the Main panel that is composed of two tabs:

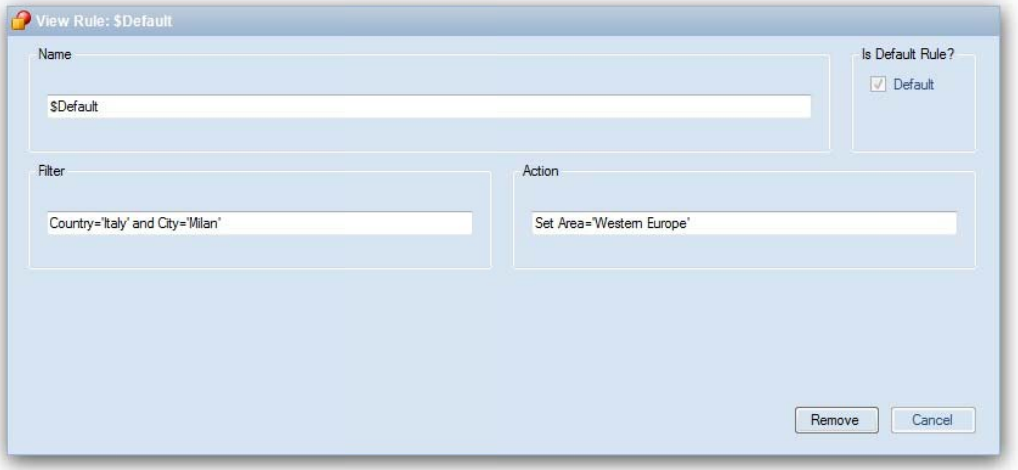
- Receiver Tab
- Graph Tab

Please refer to the Test a Queue section for a description of the functionality and options of the Receiver and Graph tabs.



View a Rule

To view the properties of a rule associated to a given subscription, you can expand the **Rules** node under the subscription node and select the corresponding rule node, as illustrated in the picture below.



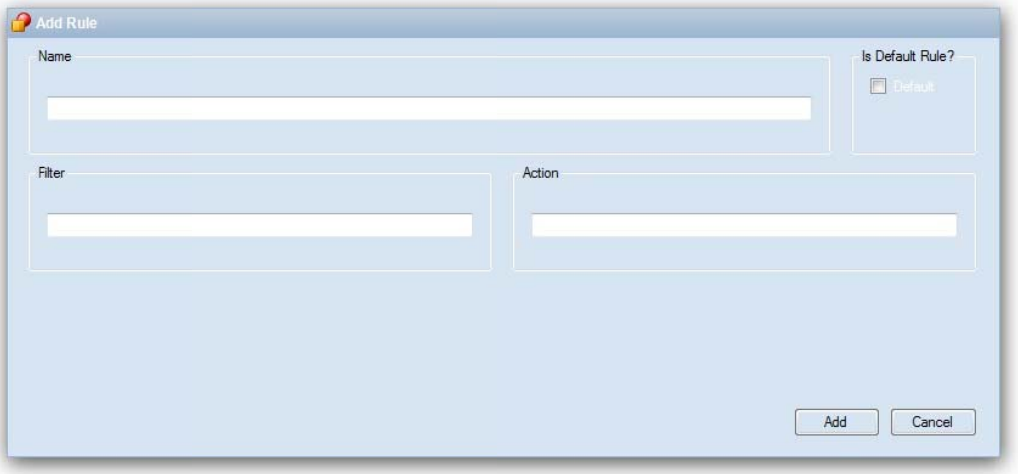
Add a Rule

To add a new rule to an existing subscription, you can select the subscription node in the treeview and then perform one of the following actions:

1. Right-click the subscription node and choose **Add Rule** from the context menu.
2. Expand the subscription node, right-click the **Rules** node under the latter and choose **Add Rule** from the context menu.
3. Choose **Add Rule** command under the Actions menu.

This operation displays a custom control (see the picture below) in the Main panel that allows the user to specify the following properties for the new rule:

- **Name**: this field defines the name for the new rule.
- **Is Default Rule**: this checkbox can be used to indicate that the rule being created is the default rule for the current subscription. If the subscription in question already owns a default rule, this field is disabled.
- **Filter**: this field defines the SQL filter expression for the new rule.
- **Action**: this field defines the SQL filter action for the new rule.



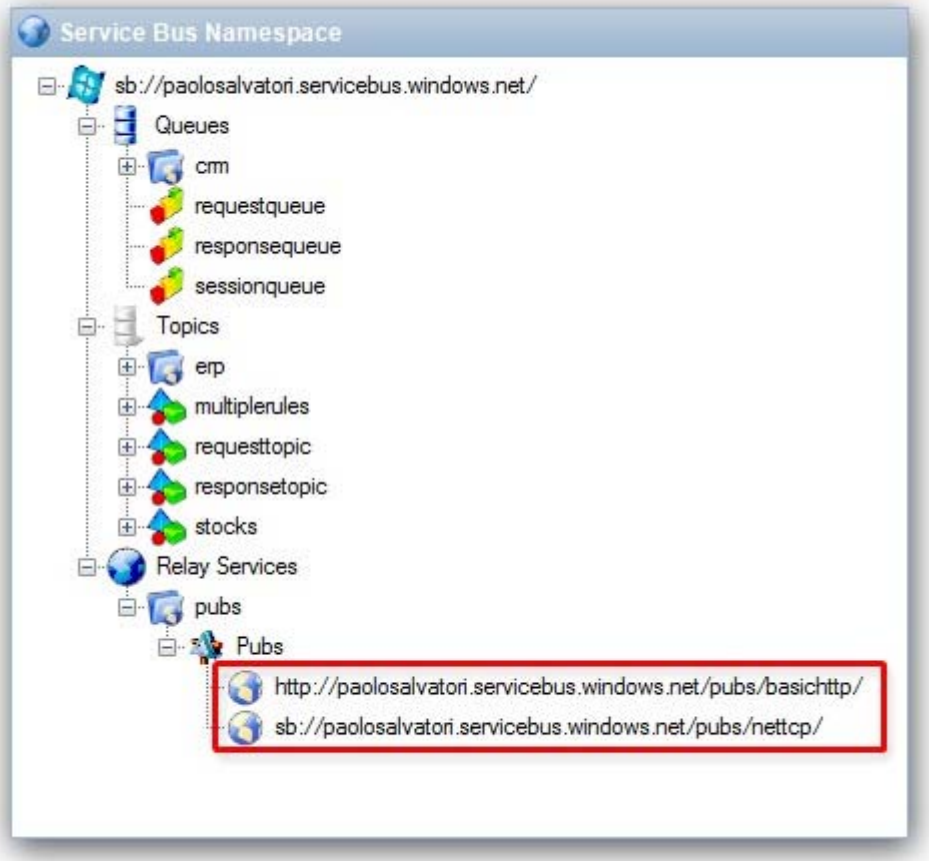
Remove a Rule

To remove an existing rule from a subscription, you can select the corresponding node in the treeview and then perform one of the following actions:

1. Right-click the node and choose **Remove Rule** from the context menu.
2. Choose Remove Rule command under the Edit menu.
3. Click the **Remove** button in the Main Panel.

Test a Relay Service Endpoint

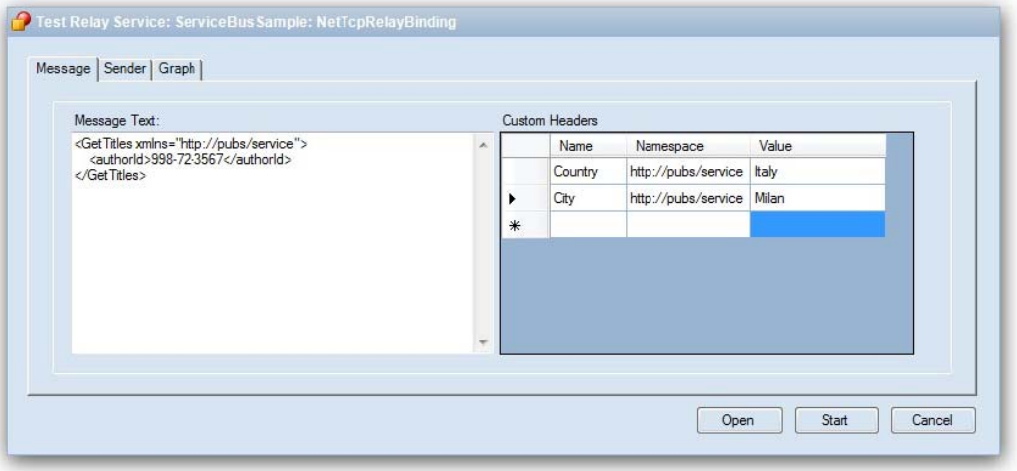
To test one of endpoints exposed by a relay service, you can select the corresponding node in the treeview as highlighted in red in the following picture.



This operation shows a custom control in the Main panel that is composed of three tabs:

- Message Tab
- Sender Tab
- Graph Tab

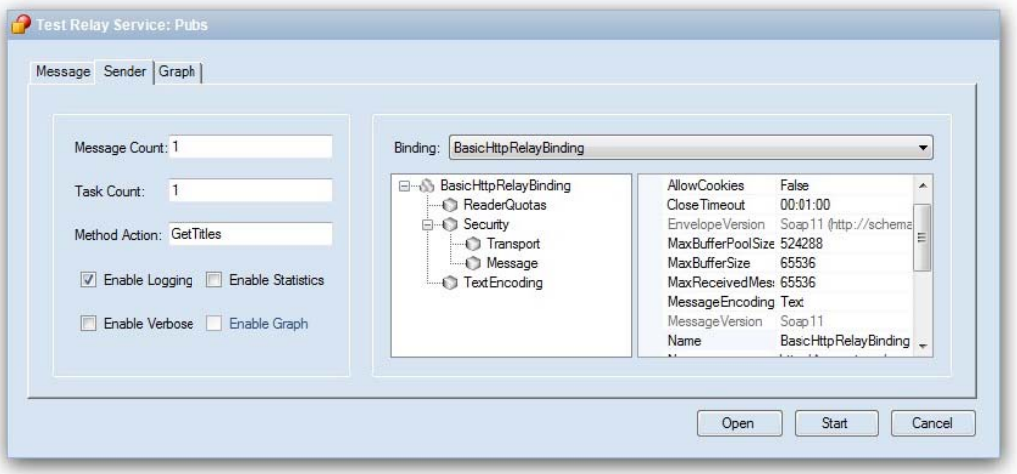
Message Tab



The Message tab allows the user to specify the following information:

- **Message Text**: specifies the payload of a WCF message. The tool provides the ability to read the message from the file system by pressing the Open button and selecting an XML file.
- **Message Properties**: you can use the datagrid to define the custom headers, as shown in the picture above.

Sender Tab



The Sender tab allows the user to specify the following information:

- **Task Count**: this field can be used to define the number of sender tasks that will be used to transmit messages to the relay service. Every task runs on a separate .NET thread pool thread.
- **Message Count**: this integer field can be used to specify the number of messages to send.
- **Method Action**: specified the value for the WS-Addressing Action header.

- **Enable Logging**: when this option is checked, sender tasks will track the operations performed in the log view.
- **Enable Verbose**: when this option is checked, sender tasks will log the payload and headers of request and response messages (if any).
- **Enable Statistics**: when this option is checked, sender tasks will update statistics in the Graph tab.
- **Enable Graph**: when this option is checked, sender tasks will update the latency and throughput charts in the Graph tab.
- **Binding**: you can use this section to select and configure the binding used by the tool to communicate with the underlying relay service. In particular, make sure to properly configure the security section to match settings specified in the configuration file of the relay service.

Graph Tab

When the Enable Statistics and Enable Graph options are enabled, this tab displays performance statistics for sender tasks and two distinct chart series:

- Sender Latency
- Sender Throughput

Conclusions

The Service Bus Explorer is the perfect tool to explore the features provided by the Service Bus and a useful companion for all developers and system administrators that work with it every day. In the future I will continue to develop and extend the tool with new features as they will be made available by the Service Bus team. In the meantime, I invite you to share your ideas on how to improve the tool and I strongly encourage you to download from the MSDN Code Gallery and customize my code to accommodate your needs.

Testing

Using Visual Studio Load Tests in Windows Azure Roles

Load testing is the process of measuring the performance of an application under a simulated usage scenario. For example, you want to load test a website by requesting it to return pages to 1000 users in a five-minute period. By running the load tests in Windows Azure roles, you can reduce your reliance on hardware through the lifecycle of an application.

Authors: Paolo Salvatori, Sidney Higa.

Jaime Alva Bravo also cowrote **Running Load Tests in Mixed Environments**.

Acknowledgements: Stéphane Crozatier, Benjamin Guinebertière in Load testing from Windows Azure | Tests de charge depuis Windows Azure. Also thanks to Janice Choi for technical review.

Visual Studio Ultimate enables you to create *load tests* that are easily managed by a person or a team of testers. The basic method is to create one "controller" and one or more "agents." You have one controller, but as many agents as you require. Each agent generates a part of the load. The controller directs the lifetime of the agent or agents, and records the results of the test. However running a load test typically requires at least two computers: the first runs the controller, and the second runs the agent or agents.

With Windows Azure, you can create worker roles to take the place of multiple computers. Virtualization of computing resources eliminates the need for dedicated hardware for load testing. This series of topics explains the basic mechanics of setting up Windows Azure to run a load test controller and agents in two worker roles.

In This Section

Visual Studio Load Test in Windows Azure Overview

Describes the benefits of using Visual Studio Load Test and outlines the required steps.

Windows Azure Load Test Prerequisites and Setup

Lists the requirements for the solution.

Provisioning Windows Azure For a Load Test

Detailed instructions on how to set up the load test application before publishing.

Publishing the Load Test To Windows Azure

Describes the steps for publishing a Load Test to Azure.

Running Load Tests In Mixed Environments

A mixed environment is one in which the components of a load test (test controller, agents, results repository, and tested system) reside in different environments, such as on-premises and in Windows Azure. This document explains how you can proceed to

configure such a scenario.

Visual Studio Load Test in Windows Azure Overview

The major components of the load test are described here. For a general explanation of the architecture of the application, see Load Testing with Agents running on Windows Azure – part 1. For an overview of load tests in Visual Studio, see Understanding Load Tests.

Note

This document is about a mostly "pure" solution. Except for a copy of Visual Studio, the components of the load test are run as Windows Azure worker roles. There is an alternative to this "mostly pure" scenario: if you run or host any of the components in a mixed environment. For example, you may want to run the test controller on-premises with agents that run on a worker role. Or you want to use SQL Azure to store your load test data. In other words, you can distribute the components of the load test across Windows Azure roles and SQL Azure and on-premises. For more documentation on selecting and setting up these alternative setups, see this topic: Running Load Tests In Mixed Environments.

Benefits

Before continuing with the overview, here are some of the advantages of using Visual Studio load tests in Windows Azure.

Entry Cost

The cost of doing load tests decreases greatly after the initial investment. Your first costs are for Visual Studio Ultimate and for the license to use the Load Test feature. After that, you create the load test harness—a Visual Studio project. You must have a Windows Azure subscription and capacity to deploy and run web and worker roles on an as-needed basis. These costs are balanced against the costs of owing hardware and all of the collateral costs. (Those costs include software, electrical power, a data center, and people to maintain the whole system.)

Maintenance Cost

Using the procedures in here, you can create an easily maintained harness for load testing. After creation, updating the project will be minimal in the foreseeable future.

Elasticity

The load test can be easily modified to accommodate different scenarios and conditions. For example, you can configure the Azure hosted server with a larger number of worker roles. Each role becomes a test agent that can be used to increase the load.

Repeatability

After the initial investment however, you can repurpose the load test against various test subjects with minimal time and costs. Simply reconfigure the test project, and redeploy it on Windows Azure. Run the test for the necessary time and undeploy it. Repeat as needed.

Real-life fidelity

As you host your services in a large data center, the concurrency of the system is greatly increased (more about concurrency later.) Hosting your final application on Windows Azure presents perfect fidelity to the final deployment.

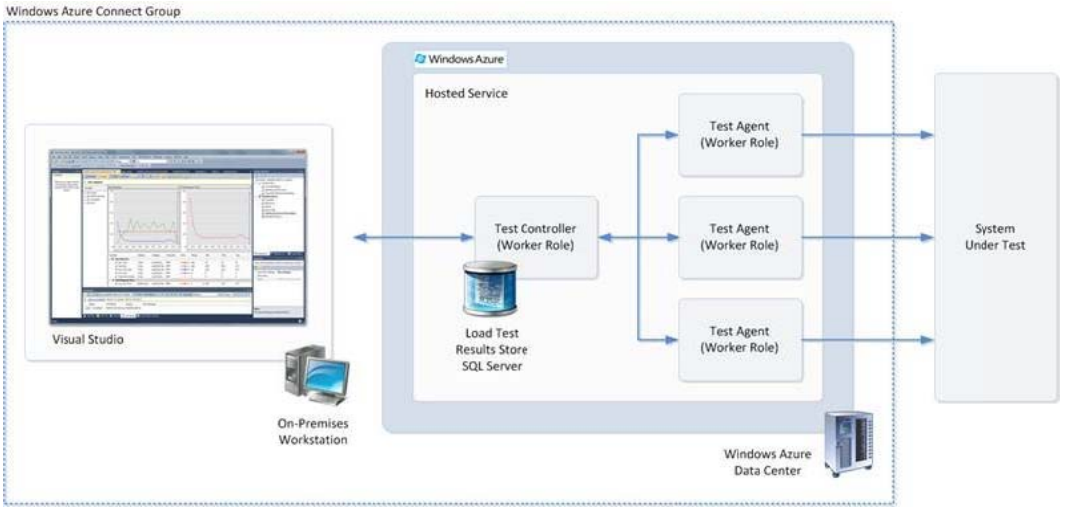
Concurrency

Concurrency is defined as a property of system where several tasks are executed simultaneously, and are possibly interacting. A factor that limits concurrency is the number of available IP addresses. The more IP addresses the system leverages, the greater the concurrent processing. Typically, the number of available addresses depends on the size of your IP provider. If your service level agreement is substantial, it is typically allocated a large number of IP addresses. But such agreements are not common. However, when you use Windows Azure as your platform, you have the benefit of using a Microsoft data center and its resources. That includes a large pool of IP addresses. Hosted services in Windows Azure are assigned virtual IP addresses. In this discussion, the outward facing (Internet) load balancer (not the hosted services) use the IP addresses. And having a large number is an advantage of the Microsoft data center. Also note that not all systems require this level of concurrency. This is an advantage only when testing a high-concurrency system.

This increased capacity for concurrency is another great benefit of running the load tests on Window Azure. This level of concurrency is also hardest to reproduce outside of a large data center.

Overview

This graphic illustrates the components of a load test. A copy of Visual Studio Ultimate is used to configure the load test as well as start and control it. Visual Studio also provides graphic views of the test results. The load test pieces include the test controller, the test agents, and the system under test. The test controller and the agents are running in a Windows Azure data center as worker roles. Except for the system under test, all of the load test components must belong to the same Windows Azure Connect group. That enables the pieces to communicate freely with each other, and with the Visual Studio instance. To create counters sets that track performance counters on the machines that host the system under test, join those machines to the Windows Azure Connect group.



The load test uses a few major technologies. Their functions in the load test are described later in this document. For the complete list of downloads required, see Windows Azure Load Test Prerequisites and Setup

Azure Management Portal

The Management Portal is used to create and manage the primary structures used for the load test. Specifically, you use the following Azure features:

Azure Hosted Service

Create a **hosted service** to run the load test application. The application consists of two worker roles. The first role hosts a single instance of the *test controller*. The controller enables remote execution of the load test. The second role hosts a single agent. You can use the *instance* feature of Azure to create as many *instances* of the agent role as you need. For more information, see How to Scale Applications by Increasing or Decreasing the Number of Role Instances

Storage account

You need a Windows Azure Storage account to hold the compressed code that runs on the worker roles. The storage blobs contain three .zip files. One file contains the set-up software for SQL Express. The second contains the controller code. The third contains the agent code. For more information about creating a storage account, see How to Create a Storage Account for a Windows Azure Subscription

Connect group

To enable communication between the controller and agents, create a Connect group. The Connect feature functions as a virtual private network that allows the controller to conduct the agent activity. It also allows the agents to return test results to the controller. The Connect group also allows you to run the controller from an on-premises copy of Visual Studio. For more information, see Overview of Windows Azure

Connect.

In addition, upload two X.509 certificates to the Azure portal. The certificates are required to authenticate and encrypt the data streams between your on-premises computer and the Azure portal. The first certificate is used to enable **publishing** an application from Visual Studio directly into Azure. That certificate is called a *management certificate* in Azure parlance. The second certificate is used to enable a remote desktop connection between an on-premises computer and an Azure worker role. That certificate is called a *service certificate* in Azure terms.

Visual Studio Ultimate

Visual Studio is the development environment, and it is the only version that provides the load test functionality. For a comparison of Visual Studio versions, see Visual Studio 2010 Comparison. Using Visual Studio Ultimate, you can create test scripts to load test your application.

Visual Studio also has the facility to manage the controller remotely. In a non-Azure scenario, the remote controller would be running on a second computer. In the Azure scenario, the controller runs in a worker role in the cloud.

SQL Server 2012 Express Edition

Collecting and storing the data from the agents requires a database. Fortunately, SQL Server 2012 Express is a free edition, and the load test program uses it by default. All that is required is that the database engine is also deployed onto the worker role that is also running the controller software. Setup is accomplished using code in the RoleEntryPoint.OnStart method.

The SQL data files cannot reside in the virtual hard drive of the worker role because the data is lost when the role is recycled. The recommendation is to mount a shared drive for all roles to write to. Details are found in Using SMB to Share a Windows Azure Drive among multiple Roles.

Another alternative is to use SQL Azure. Download the loadtest2010.dacpac file to provision SQL Azure with the database. For more information, see Running Load Tests In Mixed Environments.

Assemble the pieces you need

To begin, download the software required to provision the Azure portal and to configure the application. You can find details about the downloads in this topic: Windows Azure Load Test Prerequisites and Setup.

Provision Windows Azure and configure the application

Once you have the downloaded all extra software, you can provision the Windows Azure portal and configure the application as outlined later. Details for completing each step in detail are found in the Provisioning Windows Azure For a Load Test.

1. Provision the Windows Azure portal with a new hosted service, a storage account, and Connect virtual network.
2. After creating the storage and hosted-service accounts, use the values from the Windows Azure portal to configure the application.
3. Create three .zip files that contain software that automatically installs on the worker roles.

The application requires both the controller and agent software to run on worker roles. The software for creating the controller and agents are precreated, and available for you to download. In order to get the software onto the worker roles involves one technique: First create two .zip files that contain the test controller and agent software. Use a tool of your choice (there are free versions) to create the .zip files. Once the .zip files are created, upload the files to your Windows Azure storage account. When the application runs, the .zip files are retrieved from the storage, unpacked and loaded onto the worker roles.

There is a third .zip file you must create: one that contains the setup package for SQL Server 2008 SQL Express. As with the controller and agent packages, the SQL Server setup automatically runs and deploys an instance of SQL Express onto the worker role.

4. Upload the .zip files to the Azure storage account. For this task, use the Storage Service Client, a free project that you can download.
5. Create a Windows Azure Connect group in the Windows Azure portal.
6. Install a Connect endpoint onto the computer that is used to connect to the controller role.
7. Configure the application with the Connect activation token. The token allows the worker roles to join the Connect group.

Publish the application

Once both the portal and the application are correctly provisioned and configured, you can publish the application to Windows Azure. Detailed steps are found in Publishing the Load Test To Windows Azure.

1. Retrieve the Windows Azure subscription ID. The ID is used to enable Visual Studio to upload the application to your account.
2. Create and upload a management certificate. The procedure is as follows.
   a. Use the **Publish Windows Azure** dialog box and create a new management certificate. The certificate enables Windows Azure to authorize when you upload the application.
   b. Paste the subscription ID into the dialog box. The ID enables Visual Studio to identify your account.
3. Export the management certificate. To upload the certificate, export it from the computer's local certificate cache.
4. Upload the management certificate. Use the Management Portal to upload the certificate.
5. Publish the application.

Use Remote Desktop

Once the application is published, you use Remote Desktop to connect to it. Visual Studio supplies the tools and features to enable this functionality. The steps for enabling a remote desktop connection are found in Publishing the Load Test To Windows Azure

Next Steps

To begin, download the tools and components of the solution: Windows Azure Load Test Prerequisites and Setup

Windows Azure Load Test Prerequisites and Setup

This is one of a series of topics that cover load testing using Visual Studio Ultimate and Windows Azure. This topic catalogs the prerequisites and samples to download.

Prerequisites

The following are required:

Visual Studio Ultimate

The load testing feature uses Visual Studio to control a remote instance of a controller. The controller is running as a Windows Azure worker role. This capability is only available with Visual Studio Ultimate.

Windows Azure SDK

Please search on the Internet for the latest version of the SDK.

Windows Azure Account

A Windows Azure account is required to complete the examples. The features of Windows Azure that will be used include:

- Windows Azure Connect
- Windows Azure Service Bus
- Windows Azure Storage
- MSDN Subscription

Downloads

The following can be downloaded from the Internet:

1. Load testing from Windows Azure. Save the file to your computer and unzip it. The sample contains the code running the load test.
2. SQL Server 2008 RS Express and Management ToolsThe load test must save the data to a SQL Express database running on the controller.
3. Load Test Key from MSDN. You must sign in to obtain a key, which the load test feature requires to run. The MSDN subscription is included with Visual Studio Ultimate.
4. Visual Studio Test Agents 2010 - ISO. The test agents and the controller code are included in this download.
5. Storage Service Smart Client. Use this tool to upload files to Azure storage.
6. Service Bus Explorer. This tool creates Service Bus relays and message queues or topics.

- In addition, you will need a tool for unpacking and creating .zip files, several of which are available for free on the Internet.

Create the Azure Hosted Service

The first step is to create a hosted service in the Azure portal. But do not deploy it yet.

1. Sign into the Windows Azure portal.
2. In the left pane, click **Hosted Services, Storage Accounts & CDN**.
3. If you manage more than one subscription, select the subscription to use.
4. In the Windows Azure Platform ribbon, click **New Hosted Service**.
5. In the **Create a New Hosted Service** dialog, fill in appropriate values for:
   a. Name of the service
   b. URL for the service. The URL must be unique.
   c. Region or affinity group.
   d. Deployment name. For this exercise, type **Version 1**
6. Uncheck the box for **Start after successful deployment**.
7. Under **Deployment options**, select **Do not deploy**.

For more information about creating hosted services, see How to Create a Hosted Service

Create the Azure storage account

Create an Azure storage account. This will be used to create blobs that contain the zip files for the agents and controller.

Follow the instructions found in this topic: How to Create a Storage Account for a Windows Azure Subscription.

Set up Windows Azure Connect

The Azure Connect feature allows you to create a virtual private network (VPN) of computers. You can add any on-premise computer to the VPN by installing an endpoint onto the computer. You can also add Azure worker roles to the VPN through configuration. Windows Azure Connect is used to establish an IPSec, IPv6 VPN between one or multiple roles and one or multiple local machines. In this scenario, this enables the connection between the controller role and the local Visual Studio machine. Also, if the tested application needs to access local resources (such as a database, file system, LOB application, etc.), be sure to enlist the agent role in the same Windows Azure Connect group as the local machine hosting the resource.

Follow the instructions found in this topic: Tutorial: Setting up Windows Azure Connect.

Next Steps

Download the application (Load testing from Windows Azure) and begin with Provisioning Windows Azure For a Load Test.

Provisioning Windows Azure For a Load Test

When using Visual Studio Ultimate to do a *load test* in Azure, you must provision the Azure with the following components:

1. Hosted Service
2. Storage Account
3. Connect Group

In addition, you must create three .zip files that are uploaded to the Azure portal. The .zip files contain the controller and agent code, as well as the setup files for *SQL Server 2008 R2 Express*.

The sections below walk through the steps of provisioning the Azure Management portal.

For an overview about running visual Studio load test in Azure, see Using Visual Studio Load Tests in Windows Azure Roles

Creating a Hosted Service

For this application, you must create a hosted service, however you should not deploy it.

To create a hosted service

1. Log in to the Windows Azure Management portal.
2. In the left pane, click **Hosted Services, Storage Accounts & CDN**.
3. In the left pane, click the **Hosted Services** node.
4. In the ribbon, click **New Hosted Service**.
5. Select the subscription where the service will be created.
6. Type the name of the service. For this tutorial, type "Azure LoadTest"
7. Type a URL prefix for the service. The URL must be unique; if is not, this message will appear:
8. Choose a region or affinity group for the service.
9. Under **Deployment options** select the **Do not deploy** option.

Creating a Storage Account

A storage account for containing public blobs must be created.

To create the storage account

1. In the left pane, click **Storage Accounts**.
2. Right-click the subscription name that hosts the service, then click **New Storage Account**.
3. In the dialog box, type a unique URL. Note that the URL must be in all lower-case letters, and can contain only letters and numbers.
4. Choose a region or affinity group for the storage. It should be the same region or affinity group as the hosted service.
5. Click **OK**.
6. After the account is created, refresh the browser to see the **Properties** of the account.

7. From the Properties pane, copy the **Name** value, and save it for later use. The value is also referred to as the *storage account name*.
8. Under **Primary access key** click the **View** button.
9. In the dialog box, click the "copy" icon (to the right of the access key value). Save the value for later use. The value is also called the *account key*.

Configuring the Application with the Storage Account Name and Key

With a storage account created, you can configure the application.

To configure the Load Test application

1. Run Visual Studio as an Administrator.
2. Open the AzureLoadTest solution. (See Windows Azure Load Test Prerequisites and Setup.)
3. In Solution Explorer, expand the **AzureLoadTest** project, then expand the **Roles** folder.
4. Right-click the **Agent** role and click **Properties**.
5. In the **Agent [Role]** page, click the **Settings** tab.
6. Select the **Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString** setting.
7. In the **Type** column, set the drop-down value to **Connection String**.
8. At the right of the **Value** column, click the ellipsis (…) button.
9. In the **Storage Account Connection String** dialog box select the **Enter storage account credentials** option.
10. Paste in the account name into the **Account name** box.
11. Paste the primary access key value into the **Account key** box.
12. Click the **OK** button.
13. In Solution Explorer, in the **Roles** folder, right-click the **Controller** role and click **Properties**.
14. In the **Controller [Role]** page, click **Settings**.
15. Repeat steps 6 to 12.

Configuring the Application with the Load Test Key

The Visual Studio load test feature requires that you supply a valid key in order to work. The key can be obtained from your MSDN subscription, which is free with a copy of Visual Studio Ultimate. Once you have the key, configure the application with it. For more information about the load test key key, see:

- Managing Your Virtual User Licenses for Load Testing with a Test Controller
- Microsoft Visual Studio 2010 Load Test Feature Pack
- Visual Studio 2010 Feature Packs Frequently Asked Questions

To obtain the load test key

1. Sign in to your MSDN subscription. Go to: MSDN Subscriptions and click the **Sign In** link.

2. Once signed in, go to the Secure Downloads page.
3. Find the Visual Studio 2010 Load Test Feature Pack entry. In the **Keys** column, click the **View** link.
4. Find the value in the **Product Keys** column.

To configure the application with the load test key

1. In Solution Explorer, open the Properties page of the Controller role.
2. In the Properties page, click the **Settings** tab.
3. Paste the product key into the **TestController_License** setting.

Creating Controller, Agent, and SQL Express .Zip Files

The load test relies on two sets of code that run on the two worker roles. The *controller* and *agent* code are downloaded from the Microsoft Download Center and must be repackaged as .zip files. The .zip files must then be uploaded to the Azure storage account. When the application is deployed, the OnStart method of the RoleEntryPoint class is called. Within the method, installation code is then invoked, and the controller and agents are run as services on the worker roles.

To create the controller agent .zip files

1. Download the *X16-49583VS2010Agents1.iso file* from the Download Center: Visual Studio Agents 2010 – ISO.

   The file contains both the controller and agent software.
2. Find and download .zip utility on the Internet by searching for "free .zip utilities" or similar.
3. Run the utility and open the X16-49583VS2010Agents1.iso file.
4. Extract the testagent and testcontroller folders.
5. Using the utility, create two new .zip files with the contents of the extracted folders. The new .zip files are named as follows:
   - testagent.zip
   - testcontroller.zip

   Important
   The contents of the folders must be at the top level of the .zip file. By default, simply repackaging the folders will put the contents in a folder in the .zip file, which is one level too deep. The contents must be accessible at the first level for the installation code to extract the code.

A third .zip file must be created. The SQL Express engine must also be installed on the controller worker role. In order to do this, you must first retrieve the (free) SQL Server Express installation package, the repackage it as a .zip file.

To create the SQL Server Express installation file

1. Download the the SQL Server 2008 R2 Express database and management setup:SQL Server 2008 RS Express and Management Tools

2. Run the .zip utility and open the SQLEXPRWT_x64_ENU.exe. (Despite the .exe extension, the file is also an .iso archive file.
3. Extract the contents.
4. Use the .zip utility and create a new file named **SQLEXPRWT_x64_ENU.zip**.

   Important
   As with the controller and agent .zip files, the contents of the installation package must be at the top level of the new .zip file.

Uploading the .Zip Files to the Azure Storage Account.

The .zip files must be available to the worker role as it starts up and runs the installation procedures. To do that, you must create a single public blob container, and upload the files there. While you can use code to upload the files, this procedure uses a downloadable tool.

Important
The storage container must be public.

To upload the .zip files

1. Download the tool here: Storage Service Smart Client
2. Open the StorageServicesSmartClient solution in Visual Studio.
3. Configure the application with your storage account name and key.
   a. In Solution Explorer, open the **App.config** file.
   b. Find the **<storageAccounts>** element.
   c. Under the **<storageAccounts>** element, select one of the **<add>** elements. Refer to the following example. In the **key** attribute, replace "<MyStorageName>" with your Azure storage account name. In the **value** attribute, replace "<MyStorageName>" with your storage name. Also in the **value** attribute replace "<MyPrimaryAccessKey>" with your storage account key.

```
<add key="<MyStorageName>"
value="DefaultEndpointsProtocol=https;AccountName=<MyStorageNa
me>;AccountKey=<MyPrimaryAccessKey>"/>
```

4. Run the application.
5. In the lower left corner, under **Storage Accounts**, click the drop-down box and select your storage name.
6. Under **Container Commands** click the **New** button. Leave the default set to **Public**.
7. Give the new container a name and click **OK**.
8. In the left pane, under **Blob Services**, select the new container.
9. In the section named **Blob Commands** click the **New** button.
10. In the **Upload Files** dialog box, click **Select**.
11. Navigate to the directory where the controller, agent, and SQL Express setup .zip files are saved to upload them.

12. Record the name of the new container for later reference. It is used when configuring the application.

Configuring the Application for the .Zip Files

Having created the .zip files and uploaded them to the Azure storage account, you must now configure the Load Test application with the proper values.

To configure the application for the .zip files

1. In Visual Studio, open the**AzureLoadTest** solution.
2. In Solution Explorer, expand the **AzureLoadTest** project, then expand the **Roles** folder.
3. Right-click the **Agent** role and click **Properties**.
4. In the **Agent [Role]** page, click the **Settings** tab.
5. Select the **TestAgent_Zip** setting.
6. In the **Value** box, replace the values for the storage address as shown. Refer to the following example. In the URL, replace "<MyStorageName>" with your Azure storage account name. Then replace "<mycontainername>" with your blob container name.

   ```
   http://<MyStorageName>.blob.core.windows.net/<mycontainername
   >/testagent.zip
   ```

7. In Solution Explorer, right-click the **Controller** role and click **Properties**.
8. In the **Settings** tab, select the **SqlExpress_Zip** setting.
9. In the **Value** box, replace the values for the storage address as shown. Refer to the following example. In the URL, replace "<MyStorageName>" with your Azure storage account name. Then replace "<mycontainername>" with your blob container name.

   ```
   http://<MyStorageName>.blob.core.windows.net/<mycontainername
   >/SQLEXPRWT_x64_ENU.zip
   ```

10. Select the **TestController_Zip** setting. Replace the
11. In the **Value** box, replace the values for the storage address as shown. Refer to the following example. In the URL, replace "<MyStorageName>" with your Azure storage account name. Then replace "<mycontainername>" with your blob container name.

    ```
    http://<MyStorageName>.blob.core.windows.net/<mycontainername
    >/testcontroller.zip
    ```

Creating a Connect Group

The *Connect* feature of Azure allows you to create a Virtual Private Network. Members of the group can include on-premise computers as well as Azure role instances. For the Load Test solution, the Connect group allows communication between the test controller and the agents. For more details about setting up a Connect group, see http://msdn.microsoft.com/en-us/library/gg508836.aspx

If you have not created a Connect group before, install a local endpoint onto the development computer (the computer used to develop and manage the controller instance).

Important

The local endpoint can be installed only once on any computer, and it works only with Connect groups created in a specific Azure subscription. If you have already installed a local endpoint for a particular Azure subscription, you must uninstall it before installing a new local endpoint for a different subscription. For more information, see How to Install Local Endpoints with Windows Azure Connect.

To install a local endpoint

1. In the Azure Management portal, click **Virtual Network** in the left pane.
2. Select the subscription where the application will be hosted.
3. In the ribbon, click **Install Local Endpoint**.
4. In the **Install Windows Azure Connect Endpoint Software** dialog box, click **Copy Link to Clipboard** and click **OK**.
5. Paste the link into the Internet Explorer address box and press Enter.
6. Follow the instructions to install the endpoint.

To create the Connect group

1. In the Azure Management portal, ensure that you have clicked **Virtual Network** in the left pane.
2. Select the subscription in which the application will be hosted.
3. Expand the node where the service is hosted.
4. Select the **Groups and Roles** node.
5. In the ribbon, click **Create Group**.
6. In the dialog box, type a name for the group.
7. Click the **Create** button.
8. In the Windows tray, click the Connect Endpoint tray icon and click **Refresh Policy**.
9. To add the local endpoint to the group:
   a. In the Azure Management portal, select the newly-created group.
   b. In the ribbon, click **Edit Group**.
   c. In the **Edit Endpoint Group** dialog box, in the **Connect from** section, click the **Add** button
   d. In the **Select Local Endpoints** dialog box select the computer and click **OK**.

      If the computer does not appear, see Troubleshooting Windows Azure Connect
   e. Click the **Save** button.

At this point, you cannot add the worker role endpoints to the group. Instead, the roles are added when you configure the worker roles with the activation token, as shown below in "Configuring the Application for the Connect Group" below. Additionally, the roles only appear in the Connect group after the application is deployed, and the roles are created.

Configuring the Application for the Connect Group

To configure the application for use in the Connect group

1. In the Windows Azure Management portal, click **Virtual Network** to open the Connect user interface.
2. Select the subscription that hosts the Load Test application Connect group.
3. In the ribbon, click **Get Activation Token**.
4. Click the **Copy Token to Clipboard** button and click **OK**.
5. In Visual Studio, open the **AzureLoadTest** solution.
6. In Solution Explorer, expand the **AzureLoadTest** project, then expand the **Roles** folder.
7. Right-click the **Agent** role and click **Properties**.
8. In the **Agent [Role]** page, click the **Virtual Network** tab.
9. Select the **Activate Windows Azure Connect** setting.
10. Paste the activation token value into the box.
11. In Solution Explorer, right-click the **Agent** role, and click **Properties**.
12. Repeat steps 8 to 10.

Next Steps

The steps here have configured both the Azure Management portal and the Load Test application with the necessary files and values to get to the next stage of development. To continue, go to Publishing the Load Test To Windows Azure.

Publishing the Load Test To Windows Azure

Publishing an application to Azure has one requirement: creating and supplying a management certificate to the Azure Management portal. Once the portal is correctly configured, you can upload a new version of an application. If there is an existing application already running, you have the option of stopping and overwriting it.

Publishing an Application from Visual Studio

To prepare the application

1. Log in to the Azure Management portal.
2. In the left pane, click **Hosted Services, Storage Accounts & CDN**.
3. In the upper left pane, click **Hosted Services**.
4. Select the subscription where you will publish the application.
5. In the **Properties** pane, find the **Subscription ID** value. Select it and copy it.

   Store the value, or keep the portal open so you can retrieve the value again if needed.
6. Run Visual Studio as an Administrator.
7. Open the Load Test application. (See Windows Azure Load Test Prerequisites and Setup.)

8. In Solution Explorer, right-click the Azure project node, then click **Publish**. You will see the **Publish Windows Azure Application** dialog box.

   If you have not previously published an application, you will be at the **Sign in** stage. If you have already published an application, you can select an existing target profile. Otherwise, to create a new profile, click the **<Previous** button until you return to the *Sign in* stage.
9. Click the drop-down box, and from the list select **<Manage…>**.
10. In the **Windows Azure Project Management Settings** dialog box, click the **New** button.
11. In the **Windows Azure Project management Authentication** dialog box, under **Create or select an existing certificate for authentication** click the drop-down box.
12. At the bottom of the drop-down list, click **<Create…>**.

    Alternatively, select an existing certificate to use.
13. In the **Create Certificate** dialog box type in a friendly name such as "LoadTestManagementCertificate," Then click the **OK** button. Copy and store the friendly name for later reference. (The name is used in *Exporting and Uploading the Management Certificate* below.) Also copy and store the thumbprint. In the Azure Management portal, you can view the thumbprints of all management certificates, which is a quick way to find a specific certificate.
14. In the **Windows Azure Project management Authentication** dialog box, paste the **Subscription ID** value into the third box.
15. In the box under **Name these credentials** type in a friendly name, such as **LoadTestCredentials**
16. Click the **OK** button.
17. In the **Windows Azure Project Management Setting** dialog box click **Close**.
18. Before you can publish the application, you must export the certificate and upload it to the Azure Management portal, as described below. Keep the **Publish Windows Azure Application** dialog box while doing so.

Exporting and Uploading the Management Certificate

Before you can upload a certificate, you must export it into a file format. Export the certificate using these steps:

To export the certificate

1. Open the Visual Studio Command Prompt. See Visual Studio and Windows SDK Command Prompts
2. In the prompt, type **mmc** and press enter to open the Microsoft Management Console.
3. On the **File** menu, click **Add/Remove Snap In**.
4. In the list of available snap-ins, click **Certificates**. Then click the **Add>** button.
5. In the Certificates snap-in dialog box, ensure **My user account** is selected and click **Finish**. Then click the **OK** button.
6. Expand the following **Certificates-Current User** node. Then expand the **Personal** node, and finally expand the **Certificates** node.

7. In the list of certificates, right-click the certificate that was just created, then click **All Tasks**, and click **Export**.
8. In the **Certificate Export Wizard**, click **Next**. Ensure that the **No, do not export the private key** option is selected and click **Next**.
9. In the **Export File Format** dialog use the default settings and click **Next>**.
10. Click the **Browse** button and navigate to a folder where you will save the file. Copy the path for the next step (uploading the certificate).
11. Type in a file name and click **Save**.
12. Click the **Next** button.
13. In the final page click the **Finish** button.

To upload the certificate to the Azure Management Portal

1. In the Azure Management portal, click **Hosted Services, Storage Accounts & CDN**.
2. In the left pane, click the **Management Certificates** folder.
3. Select the subscription where the application will be published.
4. In the ribbon, click **Add Certificate**.
5. In the **Add New Management Certificate** dialog box, click the **Browse** button.
6. Navigate to the folder where the certificate is saved, and select the certificate.
7. Close the dialog box.

Creating a Local User To Manage a Load Test Controller

The Load Test feature of Visual Studio allows you to remotely access the load test controller from Visual Studio. For example, you can change the settings of test agents using this functionality.

However, to do this, the identity running the Visual Studio application must be authenticated to the load test controller, which will be running on an Azure worker role. To enable the remote computer to authenticate the Visual Studio user, first create a new local user on the computer that will be running Visual Studio.

To create a new local user

1. Open a command prompt. At the Start menu, type **cmd** and press Enter.
2. In the command prompt, type **mmc** and press Enter to run the **Microsoft Management Control**.
3. On the **File** menu click **Add/Remove Snap-in**.
4. In the **Add or Remove Snap-ins** dialog box, double-click **Local Users and Groups**.
5. In the **Choose Target Machine** dialog box keep the default, **Local computer**, and click **Finish**.
6. In the **Add or Remove Snap-ins** dialog box, click **OK**. The Console Root and Local Users and Groups nodes appear.
7. Expand the **Local Users and Groups** node.
8. Right-click the **Users** folder and click **New User**.
9. In the **New User** dialog, clear the **User must change password at next logon** box.

10. In the **User Name**, type the new user name.
11. In the Password and Confirm Password boxes type a valid password.
12. Click the **Create** button, and close the **New User** dialog box.
13. In the management console, right-click the new user and click **Properties**. You will now add the user to the Administrators group of the computer.
14. In the user **Properties** dialog box, click the **Member Of** tab.
15. Click the **Add** button.
16. In the **Select Groups** dialog box, type **Administrators** and click the **Check Names** button. The name of the computer with a slash and **Administrators** should appear.
17. Click the **OK** button.
18. In the user properties dialog box, click **OK** to close it.
19. Close the management console.

Managing a Load Test Controller From Visual Studio

After the application is published on Windows Azure(see "Publishing the Application" below for instructions), you will be able to connect to the virtual machine running on the worker role.

To manage load test controllers from Visual Studio

1. Log onto the computer as the new local user.
2. Run Visual Studio as an Administrator.
3. In Visual Studio, on the **Test** menu, click **Manage Test Controllers**. If you have correctly configured the application, you will see a list of virtual machines running on Windows Azure. Note that all worker roles and the computer you are running Visual Studio on must all be part of the same Azure Connect group. If the connection fails, check the Azure Connect configuration. Ensure that the local computer is in the same Connect group as the worker role. Also check the ActivationToken value of the controller and agent service definitions are identitcal:
4. In the left pane of the Windows Azure Management Portal, click **Virtual Network**.
5. Click the subscription containing the load test service.

Configuring the Application for Remote Desktop

Before publishing, you must also enable a remote desktop connection to the application. Doing so allows you to remotely connect to any running instance of the application. Remote desktop authentication requires a certificate; you must also configure the application to use the remote desktop feature.

Configuration consists of two procedures. The first procedure is to enable a remote desktop connection to the application, as shown below in "To configure the application for Remote Desktop connection."

You must also specify which users can access the remote desktop. When you do so, you must also specify a password to authenticate the user at log-in. If you are developing the Azure application on a Windows Server machine, you have the added option of creating a new user account specifically for using the Load test application.

To configure the application for Remote Desktop connection

1. In Visual Studio, open the Load Test application.
2. Right-click the **AzureLoadTest** project and click **Configure Remote Desktop**.
3. Select the **Enable connections for all roles** box.
4. Click the drop-down list and click **<Create>**.
5. Type a friendly name for the certificate, such as "RemoteDesktopServiceCertificate," and click **OK**.

At this point, Visual Studio creates the certificate for you. When you publish the application, the correct certificate is also uploaded.

Viewing the running instance using Remote Desktop

After the application is published on Windows Azure(see "Publishing the Application" below for instructions), you can connect to the deployed, running instance.

To view the Controller instance using Remote Desktop

1. Go to the Azure Management portal.
2. In the left pane, click **Hosted Services, Storage Accounts & CDN**.
3. Under the hosted service node, select the **Controller** node.
4. In the Remote Access section of the ribbon, click **Configure**.

   The **Set Remote Destkop Credentials** dialog box appears. The **User name** box will be filled with a default value.
5. In the **Password** box type the value that will be used when you connect to the controller instance. Retype the same password in the second box.
6. Click the **OK** button.
7. Select the instance. By default, it is named **Controller_IN_0**.
8. In the Remote Access section of the ribbon, click **Connect**.
9. If prompted to "Open" or "Save," choose **Open**.
10. In the **Remote Desktop Connection** dialog, click **Connect**.
11. Type the password for the user.
12. In the **Remote Desktop Connection** dialog box click **Yes**.

    The connection using Remote Desktop will be made and you will be able to examine the worker role.

Publishing the Application

If you have not closed the **Publish Windows Application** dialog box, you can click **Publish**.

If you have closed the dialog box, but have uploaded the management certificate to the Windows Azure Management portal, do the following:

- In Solution Explorer, right-click the AzureLoadTest project and click **Publish**. Then click the **Publish** button.

Running Load Tests In Mixed Environments

This topic covers running Visual Studio Load tests that run in a mixed environment. A "mixed environment" is one where the components of the load test are hosted in different contexts, such as on-premises, or as Windows Azure worker roles. For example, a mixed environment would be one where the test agents run on-premises and the data repository is sited on a SQL Azure server. This is in contrast to the system described in Visual Studio Load Test in Windows Azure Overview. In that topic, all of the components, except for the work station are run as Windows Azure worker roles.

Running the load test in these mixed environments may be necessary due to regulatory reasons, or for reasons specific to your application. At any rate, this topic covers the choices you have, and how to set up such varied scenarios.

Download

The files required for these procedures are found here: VSLoadTestInMixedEnvironments

Concurrency

One factor used in weighing different configurations is the increased *concurrency* from running many processes in a large data center. Concurrency is defined as a property of system where several tasks execute simultaneously, and are possibly interacting. A factor that limits concurrency the number of available IP addresses. The more IP addresses the system leverages, the greater the concurrent processing. Typically, the number of addresses available depends on the size of your IP provider. If your service level agreement is substantial, it is typically allocated a large number of IP addresses. But such agreements are not common. However, when you use Windows Azure as your platform, you have the benefit of using a Microsoft data center and its resources. That includes a large pool of IP addresses. Hosted services in Windows Azure are assigned virtual IP addresses. In this discussion, the IP addresses are used by the outward facing (Internet) load balancer (not the hosted services). And having a large number is an advantage of the Microsoft data center. Also note that not all systems require this level of concurrency.

This increased capacity for concurrency is another great benefit of running the load tests on Window Azure. This level of concurrency is also hardest to reproduce outside of a large data center.

Factors

There are six factors that affect concurrency. The first two factors are the contexts where you run the load test: cloud and on-premises.

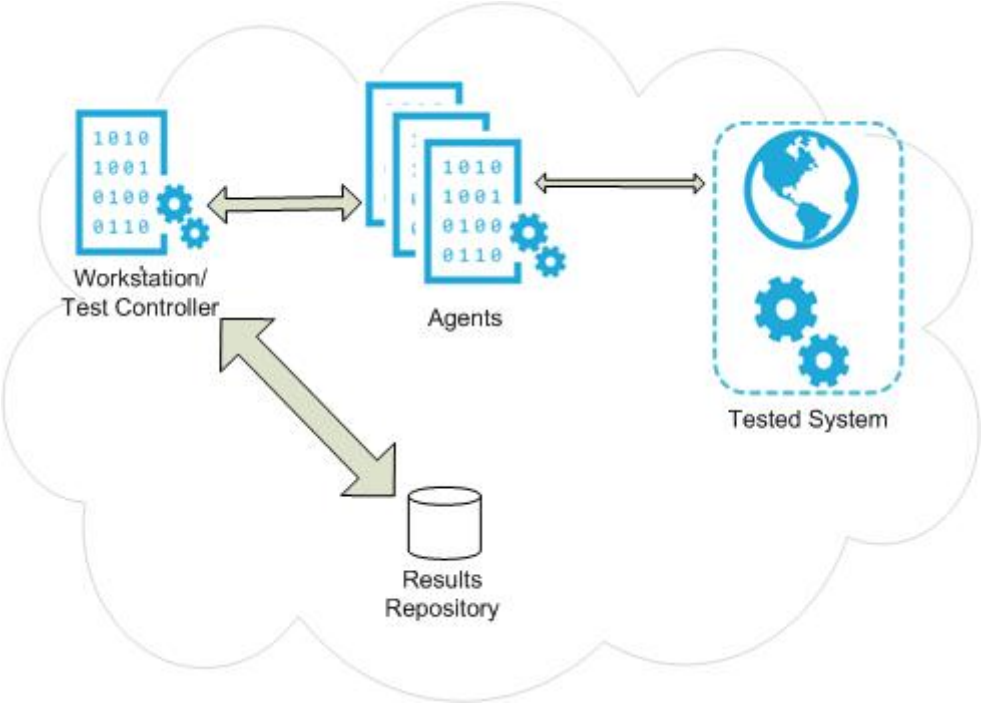
- On-premises
- Cloud

The other four factors are the components of the load test. They are:

- Test Controller
- Test Agents
- Results Repository

- Tested System

The four components are shown here:



In the graphic, the relative widths of the lines represent the amount of data transferred; the thicker the line, the greater the amount of data. The heaviest data transfer occurs between the test controller and the results repository. The lightest load occurs between the tested system and the controller. (However the actual load depends on how much logging the tested system produces.) For reference, see the graphic in Visual Studio Load Test in Windows Azure Overview.

Note

For the sake of clarity, the graphic assumes that the workstation that hosts Visual Studio also hosts the Test Controller. This configuration facilitates the communication between Visual Studio and the test controller. During a load test, the controller streams a large amount of data back to Visual Studio for real-time performance monitoring.

Topologies

Given the four components, and the two contexts, various topological choices become evident. The four components of the load test can exist in one or the other context. For example, the two simplest topologies are stated here.

1. All components on cloud.
2. All components on-premises.

For the sake of clarity, the two simplest choices are shown in this table.

Controller	Agents	Repository	Tested System	Notes
On-premises	On-premises	On-premises	On-premises	Limited concurrency no network traffic outside premises.
Cloud	Cloud	Cloud	Cloud	Large concurrency (More IP addresses) and no network traffic outside the cloud.

Now the topologies are more complicated. To keep things simple, this table shows one major division. In this table, the controller runs on-premises.

The traffic from agents to the tested system is not accounted for, as it is assumed to be part of any test cost. Also note that the network traffic in the following tables has a monetary cost. You are charged for data transferred out of the data center. Internal traffic is not billed. For pricing details, see Pricing Details, and search for "data transfers measured in GB".

This next table shows a major splitting point: when the test controller runs on-premises. In that case, the traffic between the components must cross the boundary, with various degrees of consequences, depending on the component and its level of "chattiness."

Controller Runs On-Premises

Controller	Agents	Repository	Tested System	Notes
On-premises	On-premises	On-premises	Cloud	Limited concurrency, network traffic from tested system to controller.
On-premises	On-premises	Cloud	On-premises	Limited concurrency, network traffic from controller to repository.
On-premises	On-premises	Cloud	Cloud	Limited concurrency, network traffic from tested system to controller and back to the repository.
On-premises	Cloud	On-premises	On-premises	Larger concurrency,

Controller	Agents	Repository	Tested System	Notes
				network traffic from agents to controller.
On-premises	Cloud	On-premises	Cloud	Larger concurrency, network traffic from agents and tested system to controller.
On-premises	Cloud	Cloud	On-premises	Larger concurrency, network traffic from agents to controller and back to the repository.
On-premises	Cloud	Cloud	Cloud	Larger concurrency, network traffic from agent and tested system to controller and back to repository.

This table shows the second major division. In this table, the controller runs in the cloud.

Controller Runs in Cloud

Controller	Agents	Repository	Tested System	Notes
Cloud	On-premises	On-premises	On-premises	Limited concurrency, network traffic from agent and test system to controller and back to repository.
Cloud	On-premises	On-premises	Cloud	Limited concurrency, network traffic from agent to controller and back to repository.
Cloud	On-premises	Cloud	On-premises	Limited

Controller	Agents	Repository	Tested System	Notes
				concurrency, network traffic from agent and test system to controller.
Cloud	Cloud	Cloud	Cloud	Larger concurrency, network traffic from agents to controller.
Cloud	Cloud	On-premises	On-premises	Larger concurrency, network traffic from tested system to controller and back to repository.
Cloud	Cloud	On-premises	Cloud	Larger concurrency, network traffic from controller to repository.
Cloud	Cloud	Cloud	On-premises	Larger concurrency, network traffic from tested system to controller.
Cloud	Multi-cloud	Cloud	Cloud	Largest concurrency, at least from tested system in DC1 to controller in DC2 (possibly more, depending on setup).

Recommendations

The topologies are shown for the sake of completeness. The choice of a topology may not be yours. For example, if you require more than 100 GB of SQL Server data storage, you must store it on-premises; currently, 100 GB is the limit of SQL Azure. However, if you have some leeway, here are the best topologies. These recommendations are the most efficient and give the highest concurrency levels in the two major divisions (controller on-premises or controller on-cloud).

Best When Controller Runs On-Premises

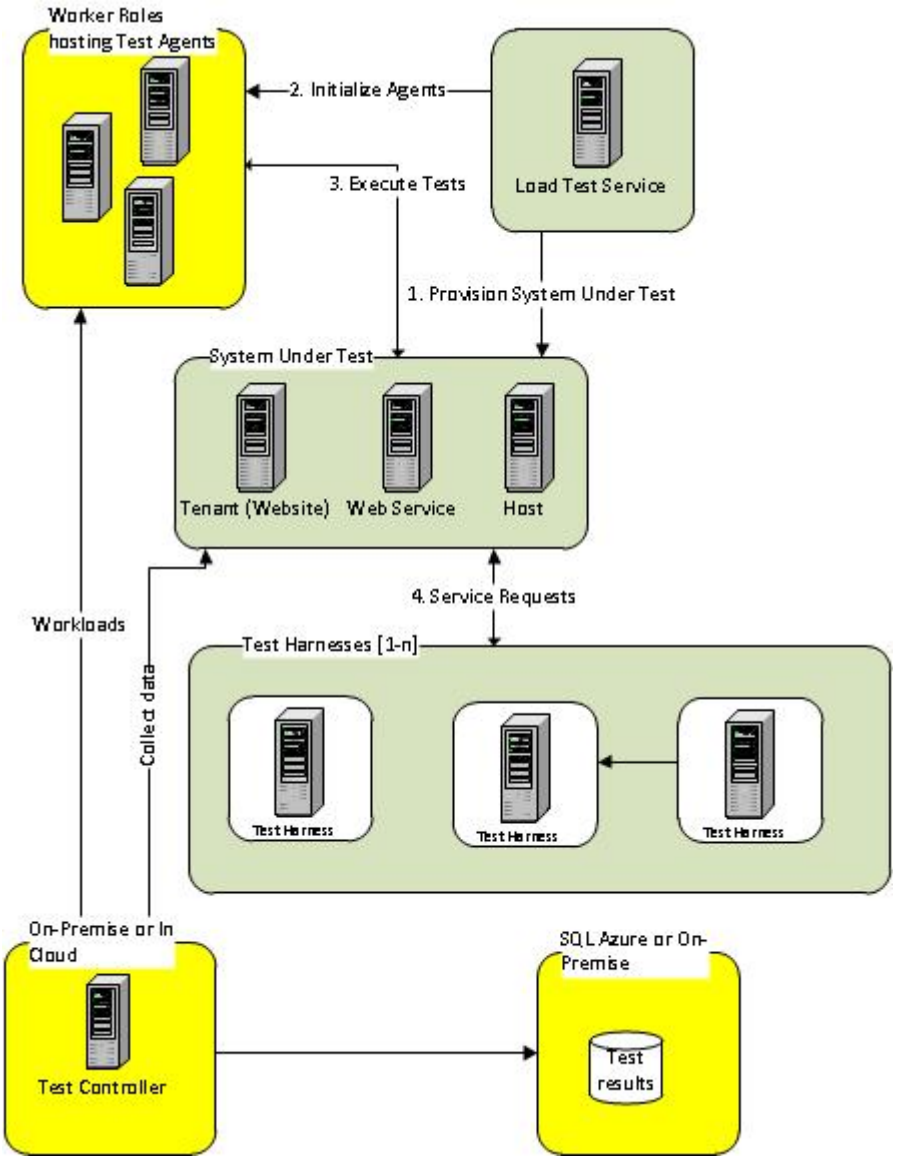
Controller	Agents	Repository	Tested System	Notes
On-premises	Cloud	On-premises	On-premises	Larger concurrency, network traffic from agents to controller.
On-premises	Cloud	On-premises	Cloud	Larger concurrency, network traffic from agents and tested system to controller.

Best When Controller Runs in Cloud

Controller	Agents	Repository	Tested System	Notes
Cloud	On-premises	Cloud	On-premises	Limited concurrency, network traffic from agent and test system to controller.
Cloud	Cloud	Cloud	Cloud	Larger concurrency, network traffic from agents to controller.
Cloud	Multi-cloud	Cloud	Cloud	Largest concurrency, at least from tested system in DC1 to controller in DC2 (possibly more, depends on setup)

Increased Complexity Demands

In the real world, your actual layout of components can be complex. A system under test can have many sub-components, each running as a worker role or web role, or using on-premises services. For example, a component initializes the system, or provides validation services. It is likely then that many of the components must communicate to other parts of the system. This set of components is in addition to the load-test system itself (consisting of controller, agents, and repository). The following graphic shows a system that has many parts. It is only meant to illustrate that a real solution can many pieces, and with many communication requirements. The details of the system are not the main point.



Given this amount of complexity, it is still possible to site various components in Windows Azure, or on-premises, as needed. The next section outlines the required steps.

Setup

The setup here is a set of alternatives to the basic setup found in this document: Provisioning Windows Azure For a Load Test. The procedures there specify how to set up the Management Portal with the correct pieces. For example, it explains how to set up a storage account, and how to set up a Windows Azure Connect group.

The first alternative here enables you to use SQL Azure as a results repository. The second alternative instructs how to set up endpoints on each worker role. Endpoints enable communication between the agents, tested system, the test controller, and (if needed) an auxiliary factor—the workstation that hosts Visual Studio. In the cleanest mode, the machine is

sited on-premises. But if your application requires you to use it during the test execution, it can also be hosted as a Windows Azure worker role.

Prerequisites

The following are required:

- Download the LoadTest2010.bacpac file.
- Optional. Install Visual Studio 2010 Ultimate on a worker role.

  If you intend to run the test controller on a worker role, install Visual Studio 2010 Ultimate on the worker role. Add a worker role to the Windows Azure project in Visual Studio. Ensure that Remote Desktop is enabled. Add the worker role to a Connect group in order to connect it to your on-premises site. Use Remote Desktop to connect to the worker role, and thence install Visual Studio 2010 Ultimate.

Using a SQL BACPAC to Provision SQL Azure

Use this method to store results on a SQL Azure database to store the load test data. Upload the file named **LoadTest2010.bacpac** to your Azure Storage account. You must also have the following prerequisites:

1. A SQL Azure account.
2. A SQL server instance created in a subscription. For more information, see How to Create a SQL Azure Server.
3. A log-in name and password that has permission to modify the server instance.

To provision SQL Azure with a BACPAC

1. Upload the LoadTest2010.bacpac file to your Azure Storage account.

   Use the StorageServicesSmartClient to upload the file to a public container. Once the .dacpac file is uploaded, you can use the Import function of the Management Portal to create the database.
2. In the Management Portal, click **Database**.
3. Expand the subscription that contains the SQL Azure server you use, and select the server.
4. In the **Database** section of the ribbon, click **Import**.
5. Follow the instructions here to import the LoadTest2010.bacpac file: How to: Import a Data-tier Application (SQL Azure).

Once the repository is created in SQL Azure, configure the Test Controller. Specifically, set the connection string for the *results store*. The dialog for setting the value is found only in Visual Studio Ultimate. However before you proceed, you have a choice to make. Decide whether to run the Test Controller:

1. From an on-premises computer.

   If you this option, from an on-premises computer, follow the next set of instructions.
2. From a worker role on Windows Azure, running an instance of Visual Studio.

   If you have decided to run the Test Controller from a worker role on Windows Azure:

   a. Create the Windows Azure Connect Group.

b. Add the on-premises workstation to the group.
c. After deploying the hosted service, use Remote Desktop to connect to the worker role that will host Visual Studio.
d. Install Visual Studio 2010 Ultimate from a setup (.msi) file on your network

To configure the On-Premises instance of the Test Controller

1. Run Visual Studio as an Administrator.
2. In Visual Studio, on the **Test** menu, click **Manage Test Controllers**.
3. Under the text **Load test results store**, click the ellipsis button.
4. In the **Connection Properties** dialog box, paste in the name of the SQL Azure server.
5. Under the text **Log on to the server**, select the **Use SQL Server Authentication** option.
6. Set the **User name** value to the log-in name for the SQL Azure administrator.
7. Set the **Password** field value to the password value for the SQL Azure administrator.
8. In the **Connect to a database** section, select the **LoadTest2010** database.

   If the connection string and the credentials are correct, examine the **Configure Test Controller** dialog box. The dialog is populated with correct values.

Using Endpoints Instead of an Azure Connect Group

There is an alternative to using the Connect group: configuring endpoints on each worker role to communicate between the instances. There is some advantage here in that the connection is more reliable. Use the following steps to try this alternative.

To configure endpoints on worker role instances

1. In Visual Studio, open the Load Test project.
2. Configure each worker role with internal TCP endpoints. For general instructions on adding endpoints to a role, see How to Define Internal Endpoints for a Role. For each endpoint, specify a different private port number. The following table shows the endpoint configurations for each role. Notice that all of these endpoints are private ports using the TCP prototol:

Role Name	Port Name	Port Number
Agent	SQL	1433
Agent	WFS	137-445
Agent	Agent Listener Port	6910
Agent	AgentDownload	80
Controller	SQL	1433
Controller	WFS	137-445
Controller	Controller	6910

Controller	AgentDownload	80
Controller	Controller Listener Port	6901
Controller	Agent	6910
Controller/Workstation	Talkback	49152

3. Configure a specific port to allow the controller to send messages to the agents. Use the following registry key on the worker role that hosts the controller:
   a. HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\VisualStudio\10.0\EnterpriseTools\QualityTools\ListenPortRange\PortRangeStart
   b. HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\VisualStudio\10.0\EnterpriseTools\QualityTools\ListenPortRange\PortRangeEnd
4. In the startup of the worker role, run the setupwfw.cmd file.
5. Remote Desktop to each role and do the following:
   a. Open the following directory: \Windows\System32\drivers\etc\

      Open the *hosts* file to edit it. The file can be opened using Notepad.exe. Place the "hosts" file on each system that has the host name and IP address. Editing the file is a manual process. To find the IP address of a role, open a command window on each role and type **IPCONFIG**. An example of the Hosts file:
   b.
6. Each role can communicate and you can install the binaries on each system. To speed up the process, place the binaries in blob store. Also, run additional commands that as part of the startup task. For more information, see How to Define Startup Tasks for a Role.)
7. On SQL Server, create a local SQL Account for the controller and workstation to access the results database. Then configure the repository to use those accounts. Create the database when you set up the controller and then configure it to use the account from the IDE.

Guidance on Efficiently Testing Azure Solutions

Author: Suren Machiraju

Reviewers: Jaime Alva Bravo and Steve Wilkins

After designing, coding and deploying your Azure AppFabric solution you may find that it does not work. This article provides guidance on how to efficiently test your Azure AppFabric applications through the software development life cycle both at the business logic and the comprehensive end-to-end scenario tests. In this article, we demonstrate how to:

- Develop unit tests for the business logic components while eliminating dependencies on Azure AppFabric components.
- Design integration end-to-end tests and eliminate overhead for setup, initialize, cleanup and teardown of the Azure AppFabric Service resources (for example, Queues) for every test run. Further eliminate creating duplicate infrastructures for ACS namespaces, Service Bus Queues, and other resources.

Additionally, this article provides an overview of the various testing technologies and techniques for testing your Azure AppFabric applications.

The Types of Tests

In this article, we will focus on the following tests:

- Unit Tests are narrowly focused tests designed to exercise one specific bit of functionality. These tests are often referred to as the Code Under Test or CUT. Any dependencies taken by the CUT are somehow removed.
- Integration Tests are broader tests for exercising multiple bits of functionality at the same time. In many cases, these resemble unit tests that cover multiple feature areas and include multiple dependencies.

Overall our tests focus on creating and using Test Doubles. We will use the following types of Test Doubles:

- Mocks are simulated objects that imitate the behavior of real objects in controlled ways. They are replacements of what is dynamically produced by a framework.
- Fakes are simulated objects that implement the same interface as the object that they represent. Fakes return pre-defined responses. Fakes contain a set of method stubs and serve as replacements for what you build by hand.
- Stubs simulate the behavior of software objects.

Our tests, upon executing can verify both state (for example, after you make a call a specific value is returned) and behavior (the method is called in a certain order or a certain number of times).

The Azure Dependencies

One of the major goals of unit testing is to eliminate dependencies. For the Azure framework, these dependencies include the following:

- Service Bus Queues
- Access Control Service

- Cache
- Windows Azure Tables, Blobs, Queues
- SQL Azure
- Cloud Drive
- Other Web Services

When building tests for Azure Applications we will replace these dependencies to better focus our tests on exercising the desired logic.

In this article, we will provide examples of Service Bus Queues with full knowledge that the tools and techniques demonstrated here will also apply to all other dependencies.

The Testing Framework

To implement the testing framework for your Azure AppFabric applications, you will need:

- A unit a testing framework to define and run your tests.
- A mocking framework to help you isolate dependencies and build narrowly scoped unit tests.
- Tools to help with automatic unit test generation for increased code coverage.
- Other frameworks that can help you with testable designs by taking advantage of dependency injection and applying the Inversion of Control Pattern.

Unit Testing with MS Test

Visual Studio includes a command line utility MS Test that you can use to execute unit tests created in Visual Studio. Visual Studio also includes a suite of project and item templates to support testing. You typically create a new test project and then add classes (known as test fixtures) adorned with [TestClass] attributes that contain methods adorned with [TestMethod]. Within MS Test, various windows within Visual Studio enable you to execute unit tests defined in the project, and review the results after you execute them.

Note

Visual Studio Express and Test Professional editions do not contain MS Test.

In MS Test, unit tests follow an AAA pattern - Arrange, Act, Assert.

- Arrange - build up any pre-requisite objects, configurations, and all other necessary preconditions and inputs needed by the CUT.
- Act - perform the actual, narrowly scoped test on the code.
- Assert - verify that the expected results have occurred.

The MS Test framework libraries include PrivateObject and PrivateType helper classes that use reflection to make it easy to invoke non-public instance members or static members, from within the unit test code.

Premium and ultimate editions of Visual Studio include enhanced unit test tooling that integrate with MS Test that enable you to analyze the amount of code exercised by your unit tests, and visualize the result by color coding the source code, in a feature referred to as Code Coverage.

Mocking with Moles

A goal of unit testing is to be able to test in isolation. However, often the code under test cannot be tested in isolation. Sometimes the code is not written for testability. To rewrite the code would be difficult because it relies on other libraries that are not easily isolated, such as those that interact with external environments. A mocking framework helps you isolate both types of dependencies.

A list of mocking frameworks you might consider is available in the links section at the end of this article. For our purposes, we will focus on how to use Moles, a mocking framework from Microsoft Research.

Most mocking frameworks will create mock types dynamically by deriving from the type you indicate for the members you want to change. Fewer frameworks have support for handling sealed classes, non-virtual methods, static members, delegates or events that require alternative techniques (such as using the .NET Profiling API). Moles framework provides this advanced support.

Most of the mocking frameworks do not enable you to rewrite the logic within a constructor. If you are creating an instance of a type that you have to isolate, and the constructor of this type takes dependencies you want to isolate, you will have to creatively mock the calls made by the constructor instead of the constructor itself. Moles can be used to insert logic before constructor calls, but it stops short of letting you completely rewrite the constructor's logic.

Moles provides all these features at no cost to you. In addition, it integrates well with Visual Studio 2010 and MS Test, and tests generated with it are fully debuggable within Visual Studio.

Automated Test Creation with PEX

PEX (Program EXploration) is a white-box test generation tool available to MSDN subscribers from Microsoft Research. You can use PEX to automatically create unit tests with high code coverage by intelligently selecting input parameters for CUT, and recursively for parameterized unit tests. PEX is capable of building the smallest number of tests that produce the greatest code coverage.

PEX installs and runs fully integrated within Visual Studio 2010, and installs Moles as part of the installation process.

Frameworks for Dependency Injection and Inversion of Control (IoC) Containers

You can use Microsoft Unity for extensible dependency injection and IoC. It supports interception, constructor injection, property injection, and method call injection.

Microsoft Unity and similar tools help you build testable designs that let you insert your dependencies at all levels of your application, assuming that your application was designed and built with dependency injection and one of these frameworks in mind.

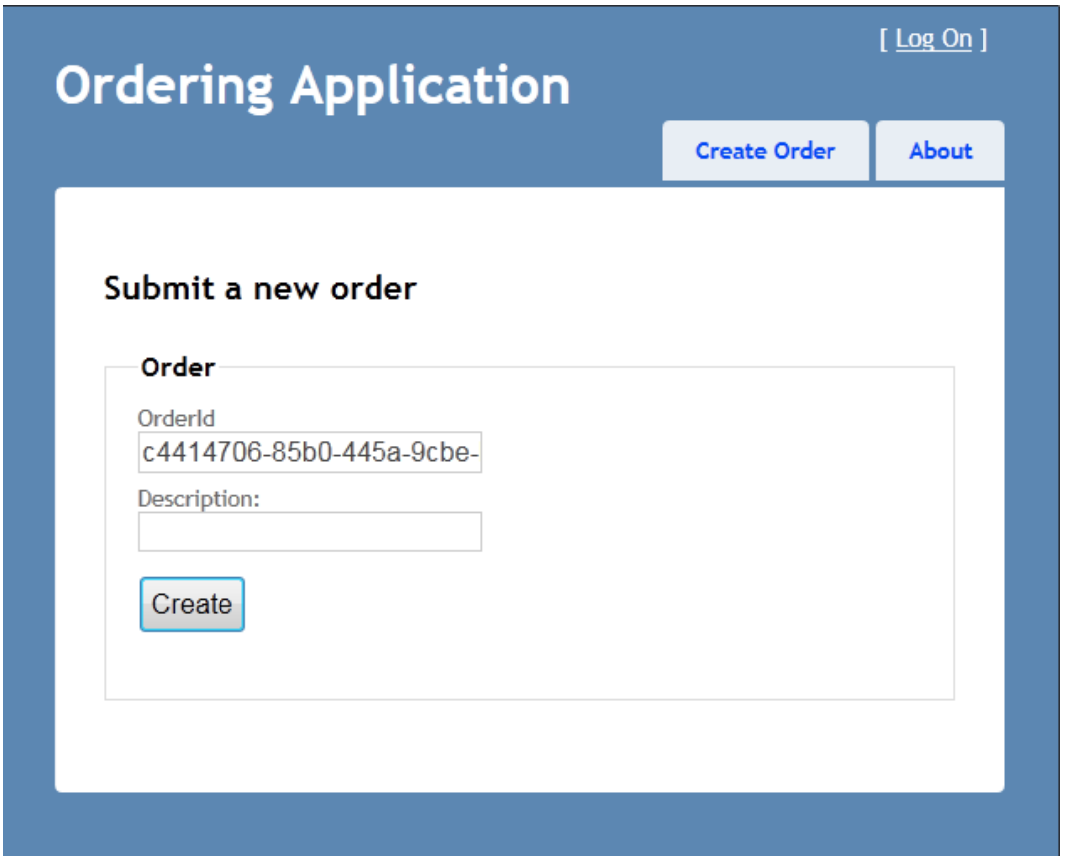
These frameworks are great for writing testable code, and ultimately good code, but can be somewhat heavy in their up-front design requirements. We will not cover DI or IoC containers in this article.

Testing Azure Solutions

In this section, we will describe a solution that includes a web role hosted web site that pushes messages to a Queue and a worker's role that processes messages from the Queue. We are interested in testing aspects of all three.

Unit Testing Controller Actions in a Web Role

Let us say you have a website that creates orders, and these orders are queued for processing by using an AppFabric Queue. Our example web page looks like this:



When the user clicks Create, it posts the new order to the Create action on the associate controller. The action is implemented as follows:

```
[HttpPost]
public ActionResult Create(Order order)
{
    try
    {
        string qName = RoleEnvironment.GetConfigurationSettingValue("QueueName");
        string qNamespace =
RoleEnvironment.GetConfigurationSettingValue("QueueServiceNamespace");
        string qUsername = RoleEnvironment.GetConfigurationSettingValue("QueueUsername");
```

```
        string qPassword = RoleEnvironment.GetConfigurationSettingValue("QueuePassword");

        //Push order to AppFabric Queue
        AppFabricQueue queue = new AppFabricQueue(new QueueSettings(qNamespace, qName,
Username, qPassword));

        queue.Send(order);

        queue.CloseConnections();

        return View("OrderCreated");
    }
    catch (Exception ex)
    {
        Trace.TraceError(ex.Message);
        return View("Error");
    }
}
```

Notice that the code retrieves settings from configuration through RoleEnvironment and sends a message that contains the order to the queue. The AppFabricQueue class we use here is a wrapper class that uses the Service Bus .NET APIs (MessageSender, MessageReceiver, etc.) to interact with Service Bus Queues.

Let us build a unit test for this method using Moles. Notice that the Create action uses the following methods:

- GetConfigurationSettingsValue (RoleEnvironment class)
- Send (AppFabricQueue class)

What we want to do is build detours for these methods using Moles to control their behavior and remove the dependencies on the real environment. Controlling behavior of these methods and removing their dependencies on the real environment will eliminate the need to run the tests in the emulator, on Azure, or to call out to the Service Bus Queue. We will exercise the Create action on the controller to verify that the Order input is the one sent to the queue (checking that it has the Order Id and Description as input to the action) and that the OrderCreated view is displayed as a result.

Accomplishing this with moles is easy. Within the Test project, right click the assembly containing the types you want to mock and select Add Moles Assembly.

In our example, we select Microsoft.Windows.Azure.ServiceRuntime and select Add Moles Assembly. An XML file named Microsoft.WindowsAzure.ServiceRuntime.moles will be added to the test project. Then repeat for the AF.Queues assembly.

When you build the Test project, references will be added to the auto-generated moled versions of these assemblies.

In our example these are Microsoft.Windows.Azure.ServiceRuntime.Moles and AF.Queues.Moles.

Create a new unit test method and decorate it with [HostType("Moles")]. Within the Unit Test, you use the "moled" types by accessing them through a Moles namespace. All moled types start with an M.

For example, the moled type for AppFabricQueue is accessed through AF.Queues.Moles.MAppFabricQueue.

You define your detours (the alternate logic to execute) by assigning a delegate to the method you want to replace.

For example, to detour RoleEnvironment.GetConfigurationSetting we define a delegate for MRoleEnvironment.GetConfigurationSettingValueString.

The complete unit test is as follows. Observe the detours we provide for RoleEnvironment.GetConfigurationSetting, the QueueSettings constructor, and the Send and CloseConnection methods of the AppFabricQueue class.

```
[TestMethod]
[HostType("Moles")]
public void Test_Home_CreateOrder()
{
    //Arrange

Microsoft.WindowsAzure.ServiceRuntime.Moles.MRoleEnvironment.GetConfigurationSettingValue
String = (key) =>
    {
        return "mockedSettingValue";
    };

    bool wasQueueConstructorCalled = false;
    AF.Queues.Moles.MAppFabricQueue.ConstructorQueueSettings = (queue, settings) =>
    {
        wasQueueConstructorCalled = true;
    };

    Order orderSent = null;
    AF.Queues.Moles.MAppFabricQueue.AllInstances.SendObject = (queue, order) =>
    {
       orderSent = order as Order;
    };

    bool wasConnectionClosedCalled = false;
    AF.Queues.Moles.MAppFabricQueue.AllInstances.CloseConnections = (queue) =>
    {
       wasConnectionClosedCalled = true;
    };
```

```
    Order inputOrder = new Order()
    {
       OrderId = "Test123",
       Description = "A mock order"
    };

    HomeController controller = new HomeController();

    //Act
    ViewResult result = controller.Create(inputOrder) as ViewResult;

    //Assert
    Assert.IsTrue(wasConnectionClosedCalled);
    Assert.IsTrue(wasQueueConstructorCalled);
    Assert.AreEqual("OrderCreated", result.ViewName);
    Assert.IsNotNull(orderSent);
    Assert.AreEqual(inputOrder.OrderId, orderSent.OrderId);
    Assert.AreEqual(inputOrder.Description, orderSent.Description);

}
```

Unit Testing Worker Roles that use AppFabric Queues

The web role took care of adding an order to the queue. Let us consider a worker's role that would process the orders by retrieving them from the queue and see how we can test that. The core item to test in the worker role is defined in the implementation of RoleEntryPoint, typically in the OnStart and Run methods.

The Run method in our worker role periodically polls the queue for orders and processes them.

```
public override void Run()
{
        AppFabricQueue queue = null;

        try
        {
                string qName = RoleEnvironment.GetConfigurationSettingValue("QueueName");
                string qNamespace =
RoleEnvironment.GetConfigurationSettingValue("QueueServiceNamespace");
                string qUsername =
RoleEnvironment.GetConfigurationSettingValue("QueueUsername");
                string qPassword =
RoleEnvironment.GetConfigurationSettingValue("QueuePassword");
```

```
                queue = new AppFabricQueue(new QueueSettings(qNamespace, qName,
qUsername, qPassword));
                queue.CreateQueueIfNotExists();

                while (true)
                {
                    Thread.Sleep(2000);

                    //Retrieve order from AppFabric Queue
                    TryProcessOrder(queue);
                }
        }
        catch (Exception ex)
        {
                if(queue !=null)
                    queue.CloseConnections();

                System.Diagnostics.Trace.TraceError(ex.Message);
        }

}
```

We want a test verifying that the routine correctly retrieves a message.

In this case we add a moled assembly for the worker role. For our specific project, this is the Mvc3Web assembly because we host the RoleEntryPoint there. The following is the complete unit test for run method of the worker role.

```
[TestMethod]
[HostType("Moles")]
public void Test_WorkerRole_Run()
{
    //Arrange

Microsoft.WindowsAzure.ServiceRuntime.Moles.MRoleEnvironment.GetConfigurationSettingValue
String = (key) =>
    {
        return "mockedSettingValue";
    };

    bool wasQueueConstructorCalled = false;
    AF.Queues.Moles.MAppFabricQueue.ConstructorQueueSettings = (queue, settings) =>
    {
        wasQueueConstructorCalled = true;
    };
```

```
    bool wasEnsureQueueExistsCalled = false;
    int numCallsToEnsureQueueExists = 0;
    AF.Queues.Moles.MAppFabricQueue.AllInstances.CreateQueueIfNotExists = (queue) =>
    {
        wasEnsureQueueExistsCalled = true;
        numCallsToEnsureQueueExists++;
    };

    bool wasTryProcessOrderCalled = false;
    int numCallsToTryProcessOrder = 0;
    Mvc3Web.Worker.Moles.MWorkerRole.AllInstances.TryProcessOrderAppFabricQueue =
    (actualWorkerRole, queue) =>
    {
         wasTryProcessOrderCalled = true;
         numCallsToTryProcessOrder++;

         if (numCallsToTryProcessOrder > 3) throw new Exception("Aborting Run");
    };

    bool wasConnectionClosedCalled = false;
    AF.Queues.Moles.MAppFabricQueue.AllInstances.CloseConnections = (queue) =>
    {
         wasConnectionClosedCalled = true;
    };

    Mvc3Web.Worker.WorkerRole workerRole = new Worker.WorkerRole();

            //Act
    workerRole.Run();

            //Assert
    Assert.IsTrue(wasConnectionClosedCalled);
    Assert.IsTrue(wasQueueConstructorCalled);
    Assert.IsTrue(wasEnsureQueueExistsCalled);
    Assert.IsTrue(wasTryProcessOrderCalled);
    Assert.AreEqual(1, numCallsToEnsureQueueExists);
    Assert.IsTrue(numCallsToTryProcessOrder > 0);
}
```

This time we focus on detouring the instance methods of the WorkerRole class. You should set the delegate of the AllInstances property of the generated mole type. By using the delegate, any

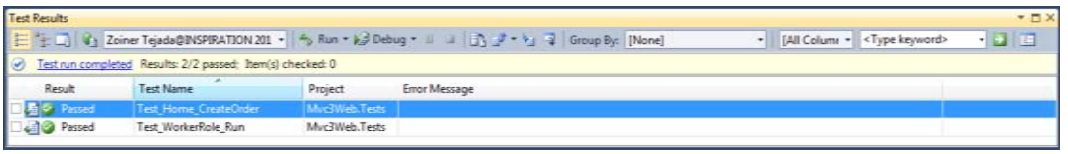
instance you create of the actual type will detour through any of the methods for which you have defined delegates.

In our example, we want to use the original instance's Run method, but provide detours for the instance methods CreateQueueIfNotExists and TryProcessOrder (in addition to the static method detours we have already seen for RoleEnvironement.GetConfigurationSettings). In our code we throw an exception so that the infinite loop maintained within Run is exited at a predetermined time.

You may be asking yourself, why not just use the MessageSender/MessageReceiver and related classes from the Service Bus SDK directly instead of injecting a helper type? In order to completely isolate our code from calling the real-world Service Bus, we have two choices - write fakes that inherit from the abstract classes in the Microsoft.ServiceBus namespace, or let moles create mock types for them all. The problem with either approach is complexity. With both approaches, you will ultimately find yourself reflecting into classes like TokenProvider and QueueClient, burdened with creating derived types from these abstract types that expose all their required overrides, struggling to expose the internal types that real versions of these classes actually rely on, and recreating their constructors or factory methods in clever ways just to excise the dependency on the real Service Bus. On the other hand, if you insert your own helper type, then that is all you need to mock and detour in order to isolate yourself from the real-world Service Bus.

Enabling Code Coverage

To analyze what these unit tests actually tested, we can examine code coverage data. If we run both of the unit tests with MS Test, we can see that they pass and also see related run details from the Test Results dialog that appears.

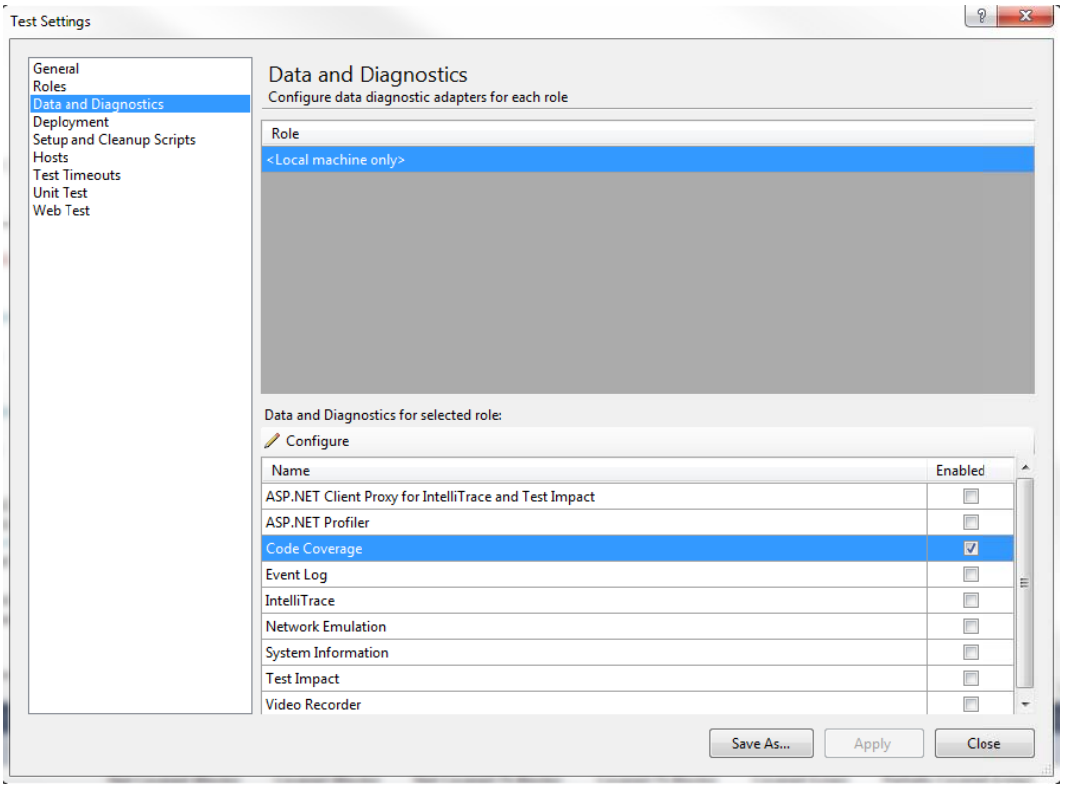


By clicking the Show Code Coverage Results button (the rightmost button on near the top of the Test Results window), we can see the code covered by all unit tests executed in the run. However, if we do this now, we will not see any data. Instead, we will get a message indicating that no coverage data could be found. We first have to enable collection of code coverage data before running our tests.

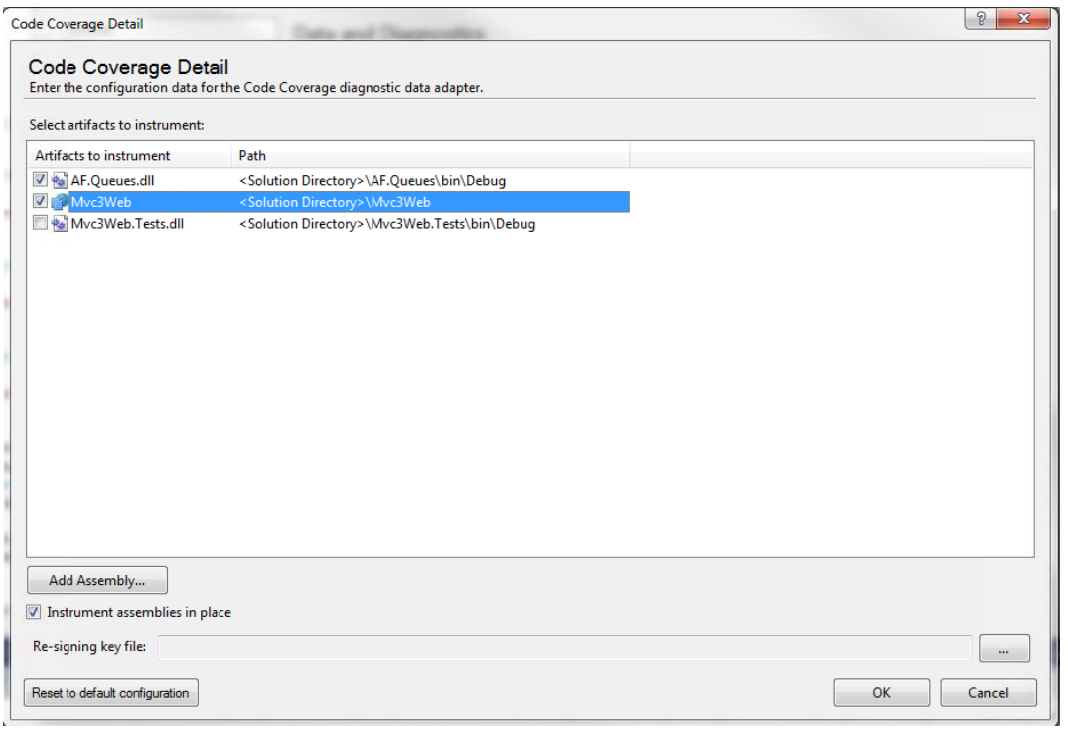
Code coverage data collection is enabled simply by configuring the Local.testsettings under the Solution Items. Opening this file brings up the Test Settings editor.

Note

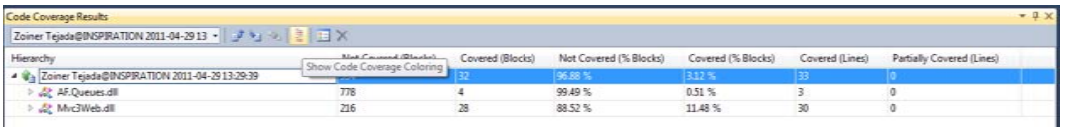
If you have a solution that does not have Local.testsettings file, you can add it using Add New Item to the solution and then select Test >Test Settings.



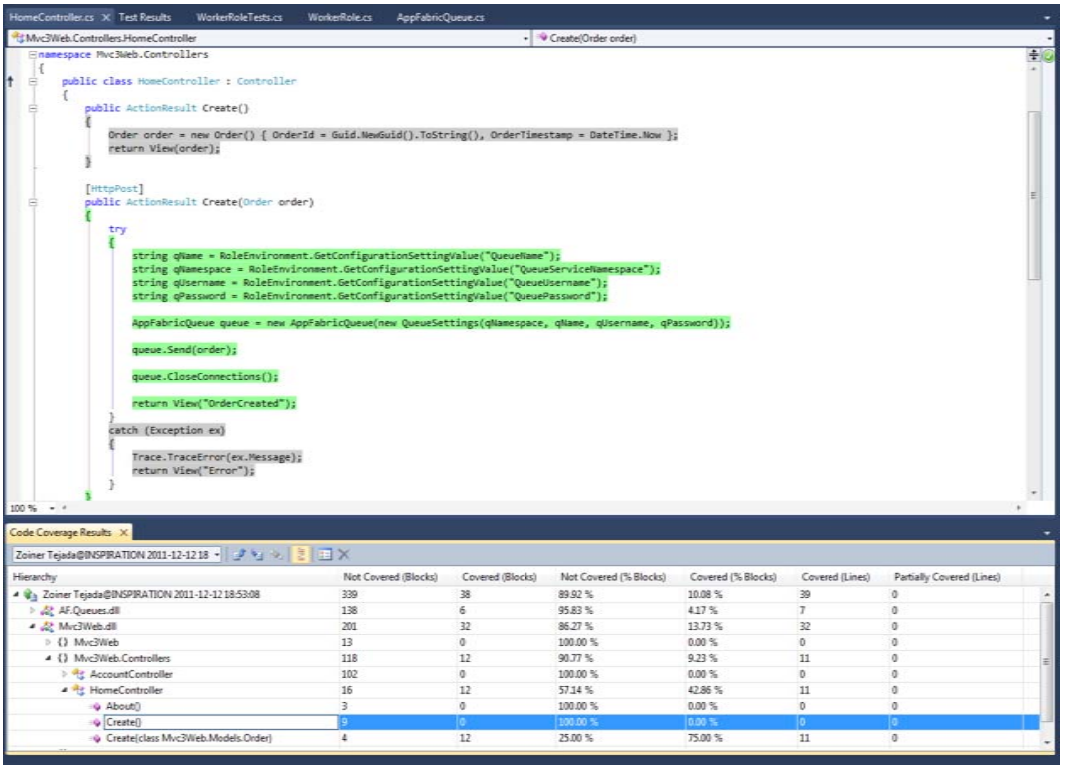
Click the Data and Diagnostics tab, and the check the box to the right of the Code Coverage row. Next, click the Configure button (the one just above the grid containing Code Coverage). In the Code Coverage Detail window select all your assemblies under test and click OK.



Then click Apply and Close to dismiss the Test Settings editor. Re-run your tests and press the code coverage button. Now your Code Coverage Results should look as follows:

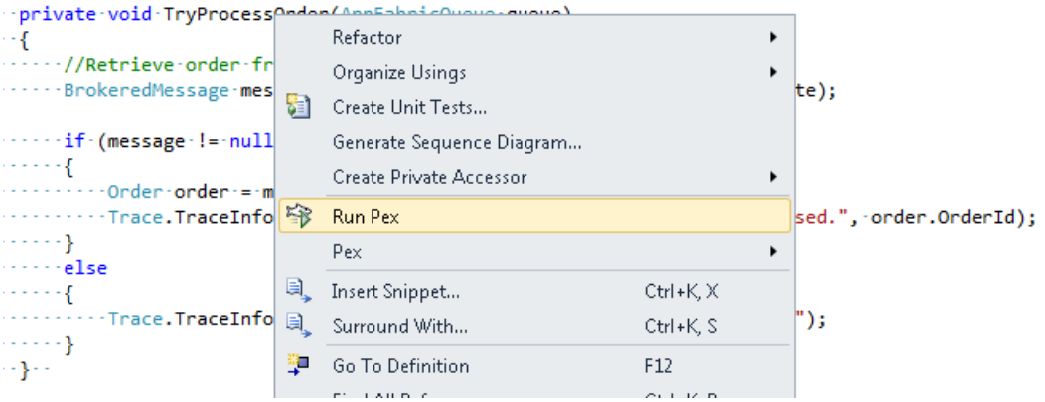


Click the Show Code Coverage Coloring icon, and then drill down to a method in the grid and double-click it to see the source code colored to reflect areas that were tested (green in the following figure), partially tested or untested (gray) the unit tests just run.

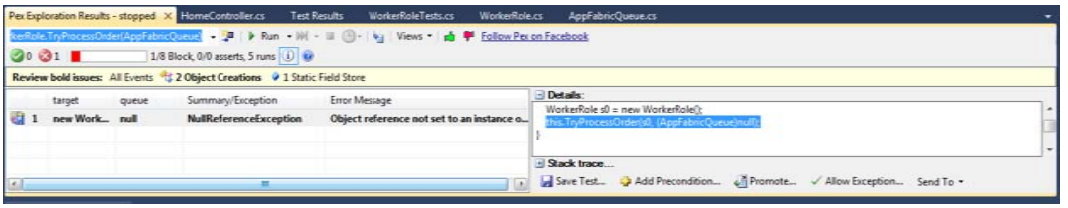


Using PEX to Enhance Code Coverage

Manually creating these unit tests is very valuable, but PEX can help you intelligently take your unit tests to the next level by trying parameter values you may not have considered. After installing PEX, you can have it explore a method simply by right clicking the method in the code editor, and selecting Run PEX.



After a short while, PEX will finish with its processing and the results will stabilize. For example, here is what we got when ran PEX on our worker role's TryProcessOrder method. Notice that PEX was able to create one test that resulted in an exception, and it shows you the inputs it crafted that generated that exception (a null value for the queue parameter). What is even more important, in the details pane you can see the code PEX was executing. The results of all PEX generated tests can be added to your project simply by selecting them and clicking Promote.



Try it out!

The sample used in this article is available at sample_link.

Useful Links

Mocking Frameworks

Moles

[Moq](Moq)

NMock3

Rhino Mocks

EasyMock.NET

NSubstitute

FakeItEasy

TypeMock Isolator

JustMock

Unit Test Related

MSDN – Verifying Code by Using Unit Tests

MSDN - Unit Testing in MVC Applications

Building Testable Applications

Pex and Moles

PrivateObject and PrivateType Introduced

CLR Profiling API Blog

Mocks Aren't Stubs

Dependency Injection and IoC

Microsoft Unity

Structure Map

MSDN Magazine - Aspect Oriented Programming, Interception and Unity 2.0

MSDN Magazine - Interceptors in Unity

MSDN Magazine - MEF v.s. IoC

Database

Data Migration to SQL Azure: Tools and Techniques

This document provides guidance on migrating data definition (schema) and data to SQL Azure. It is primarily for one-time migration from SQL Server to SQL Azure. For on-going data sharing and SQL Azure backup, see SQL Azure Data Sync Overview.

Migration Considerations

Microsoft Windows Azure offers several choices for data storage. You may choose to use one or many in your projects.

SQL Azure Database is SQL Server technology delivered as a service on the Windows Azure Platform. The cloud-based SQL Azure database solutions can provide many benefits, including rapid provisioning, cost-effective scalability, high availability, and reduced management overhead. SQL Azure supports the same tooling and development practices used for on-premise SQL Server applications. Therefore, it should be a familiar experience for most developers.

The long term goal for SQL Server and SQL Azure is for symmetry and parity in both features and capabilities, however there are currently differences in architecture and implementation that need to be addressed when migrating databases to SQL Azure and developing SQL Azure solutions.

Before migrating a database to SQL Azure, it is important to understand when to migrate and the differences between SQL Azure and SQL Server.

When to Migrate

There are three main storage offerings on the Windows Azure platform. Windows Azure Storage contains Table, Blob, and Queue. When designing a Windows Azure solution, you shall evaluate different options, and use each storage mechanism to provide the best performance for the part of the solution.

Storage Offering		Purpose	Maximum Size
SQL Azure Database		Relational database management system	50GB
Windows Azure Storage	Blob	Durable storage for large binary objects such as video or audio	200GB or 1TB
	Table	Durable storage for structured data	100TB
	Queue	Durable storage for inter-process messages	100TB
Local Storage		Per-instance temporary storage	250GB to 2TB

Local storage provides temporary storage for a local running application instance. A local store is only accessible by the local instance. If the instance is restarted on different hardware, such as in the case of hardware failure or hardware maintenance, data in the local store will not follow the instance. If your application requires reliable durability of the data, want to share data between instances, or access the data outside of Windows Azure, consider using a Windows Azure Storage account or SQL Azure Database instead.

SQL Azure provides data-processing capabilities through queries, transactions and stored procedures that are executed on the server side, and only the results are returned to the application. If you have an application that requires data processing over large data sets, then SQL Azure is a good choice. If you have an application that stores and retrieves large datasets but does not require data processing, then Windows Azure Table Storage is a better choice.

Because the SQL Azure size limitation is currently set at 50 GB, and SQL Azure is much more expensive than Windows Azure storage, you may consider moving the blob data into Windows Azure Blog Storage. So that you can reduce the pressure on the database size limit and reduce the operational cost.

For more information, see Data Storage Offerings on the Windows Azure platform.

Compare SQL Azure to SQL Server

Similar to SQL Server, SQL Azure exposes a tabular data stream (TDS) interface for Transact-SQL-based database access. This allows your database applications to use SQL Azure in the same way that they use SQL Server.

Unlike SQL Server administration, SQL Azure abstracts the logical administration from the physical administration; you continue to administer databases, logins, users, and roles, but Microsoft administers and configures the physical hardware such as hard drives, servers, and storage. Since Microsoft handles all of the physical administration, there are some differences between SQL Azure and SQL Server in terms of administration, provisioning, Transact-SQL support, programming model, and features.

The following list provides a high-level overview of some of the major differences:

- **Database Size**

  SQL Azure currently offers two editions:

  - Web Editions, in 1GB and 5GB sizes.
  - Business Editions, in 10, 20, 30, 40 and 50GB sizes.

  It is important to check the size of your database and how it fits within the database allowances used by SQL Azure. If your database is larger than SQL Azure size limitation, then you must examine your database and see if it can be broken down into smaller databases (i.e. sharding), or moving large data to Window Azure blob storage. For more information on database sharding, see Federation: Building Scalable, Elastic, and Multi-tenant Database Solutions with SQL Azure.

- **Authentication**

SQL Azure supports only SQL authentication. You must consider whether changes are needed to the authentication scheme used by your application. For more information on the security limitations, see Security Guidelines and Limitations.

- **SQL Server Database Version**

  SQL Azure is based on SQL Server 2008 (level 100). If you want to migrate your SQL Server 2000 or SQL Server 2005 databases to SQL Azure, you must make sure your databases are compatible with SQL Server 2008. You will find the best path is to migrate from a SQL Server 2008 to SQL Azure. You can go through an on-premises upgrade to SQL Server 2008 before you migrate to SQL Azure. Here are some great resources to help you with migrating from older versions of SQL Server: Upgrading to SQL Server 2008 R2 and Microsoft SQL Server 2008 Upgrade Advisor.

- **Schema**

  SQL Azure does not support heaps. ALL tables must have a clustered index before data can be inserted. For more information on the clustered index requirement, see Inside SQL Azure.

- **Transact-SQL Supportability**

  SQL Azure Database supports a subset of the Transact-SQL language. You must modify the script to only include supported Transact-SQL statements before you deploy the database to SQL Azure. For more information, see Supported Transact-SQL Statements, Partially Supported Transact-SQL Statements, and Unsupported Transact-SQL Statements.

- **The Use Statement**

  In Microsoft SQL Azure database, the USE statement does not switch between databases. To change databases, you must directly connect to the database.

- **Pricing**

  Pricing for your SQL Azure subscription is per database, and it is based on the edition. There are also additional charges for data transfer volume, any time data comes into or out of the data center. You have your choice of running your application code on your own premises and connecting to your SQL Azure database in the data center, or running your application code in Windows Azure, which is hosted in the same data center as your SQL Azure database. Running application code in Windows Azure avoids the additional data transfer charges. In either case, you should be aware of the Internet network latency that cannot be mitigated using either model. For more information, see Pricing Overview.

- **Feature Limitations**

  Some of the SQL Server features are not currently supported by SQL Azure. They include: SQL Agent, Full-text search, Service Broker, backup and restore, Common Language Runtime, and SQL Server Integration Services. For a detailed list, see SQL Server Feature Limitations.

Connection Handling

When using a cloud based database like SQL Azure, it requires connections over the internet or other complex networks. Because of this, you should be prepared to handle unexpected dropping of connections.

SQL Azure provides a large-scale multi-tenant database service on shared resources. In order to provide a good experience to all SQL Azure customers, your connection to the service may be closed due to several conditions.

The following is a list of causes of connection terminations:

- **Network latency**

  Latency causes an increase in time required to transfer data to SQL Azure. The best way to mitigate this effect is to transfer data using multiple concurrent streams. However, the efficiency of parallelization is capped by the bandwidth of your network.

  SQL Azure allows you to create your database in different datacenters. Depending on your location and your network connectivity, you will get different network latencies between your location and each of the data centers. To help reducing network latency, select a data center closest to majority of your users. For information on measuring network latency, see Testing Client Latency to SQL Azure.

  Hosting your application code in Windows Azure is beneficial to the performance of your application because it minimizes the network latency associated with your application's data requests to SQL Azure.

  Minimizing network round trips can also help with reducing network related problems.

- **Database Failover**

  SQL Azure replicates multiple redundant copies of your data to multiple physical servers to maintain data availability and business continuity. In the case of hardware failures or upgrades, SQL Azure provides automatic failover to optimize availability for your application. Currently, some failover actions result in an abrupt termination of a session.

- **Load Balance**

  Load balancer in SQL Azure ensures the optimal usage of the physical servers and services in the data centers. When the CPU utilization, input/output (I/O) latency, or the number of busy workers for a machine exceeds thresholds, SQL Azure might terminate the transactions and disconnect the sessions.

- **Throttling**

  To ensure that all subscribers receive an appropriate share of resources and that no subscriber monopolizes resources at the expense of other subscribers, SQL Azure may close or “throttle” subscriber connections under certain conditions. SQL Azure Engine Throttling service continually monitors certain performance thresholds to evaluate the health of the system and may initiate varying levels of throttling to particular subscribers depending on the extent to which these subscribers are impacting system health.

  The following performance thresholds are monitored by SQL Azure Engine Throttling:

  - Percentage of the space allocated to a SQL Azure physical database which is in use, soft and hard limit percentages are the same.
  - Percentage of space allocated for SQL Azure log files that is in use. Log files are shared between subscribers. Soft and hard limit percentages are different.
  - Milliseconds of delay when writing to a log drive, soft and hard limit percentages are different.

- Milliseconds of delay when reading data files, soft and hard limit percentages are the same.
- Processor usage, soft and hard limit percentages are the same.
- Size of individual databases relative to maximum size allowed for database subscription, soft and hard limit percentages are the same.
- Total number of workers serving active requests to databases, soft and hard limit percentages are different. If this threshold is exceeded, the criteria of choosing which databases to block are different than in the case of other thresholds. Databases utilizing the highest number of workers are more likely to be throttling candidates than databases experiencing the highest traffic rates.

For more information, see SQL Azure Connection Management, and SQL Azure Performance and Elasticity Guide.

The best way to handle connection loss it to re-establish the connection and then re-execute the failed commands or query. For more information, see Transient Fault Handling Framework.

Optimize Databases for Data Import

There are a few things you can do to the databases to improve the migration performance:

- Delay creation of non-clustered indexes or disable non-clustered indexes. Addition indexes created before loading the data can significantly increase the final database size and the time taken to load the same amount of data.
- Disable triggers and constrain checking. The triggers may fire when a row is inserted into a table causing the row to be reinserted into another table. The trigger can cause delays and you might not want those types of inserts to be reinserted.
- Bulk import performance is improved if the data being imported is sorted according to the clustered index on the table. For more information, see Controlling the Sort Order When Bulk Importing Data.

Transfer Large Data to SQL Azure

SQL Server Integration Services (SSIS) and bcp utility perform well with large data migration.

When loading large data to SQL Azure, it is advisable to split your data into multiple concurrent streams to achieve the best performance.

By default, all the rows in the data file are imported as one batch. To distribute the rows among multiple batches, specify a batch size whenever it is available. If the transaction for any batch fails, only insertions from the current batch are rolled back. Batches already imported by committed transactions are unaffected by a later failure. It is best to test a variety of batch size settings for your particular scenario and environment to find the optimal batch size.

Choose Migration Tools

There are various tools available for migration database to SQL Azure. In general, database migration involves schema migration and data migration. There are tools supporting one of each or both. You could even use the Bulk Copy API to author your own customized data upload application.

Migrate from SQL Server

Tools	Schema	SQL Azure Compatibility Check	Data	Data Transfer Efficiency	Note
DAC Package	Yes	Yes	No	N/A	• Entity containing all database objects, but no data • Full SQL Azure support
DAC BACPAC Import Export	Yes	Yes	Yes	Good	• Export/import of DAC plus data with DAC framework • Service for cloud-only support available • SQL DAC Examples available on CodePlex
SSMS Generate Scripts Wizard	Yes	Some	Yes	Poor	• Has explicit option for SQL Azure scripts generation • Good for smaller database
bcp	No	N/A	Yes	Good	• Efficient transfer of data to existing table • Each bcp command transfer one database
SQL Azure Migration Wizard	Yes	Yes	Yes	Good	• Great capabilities, e.g. evaluate trace files

					• Open source on CodePlex • Not supported by Microsoft
SQL Server Integration Services	No	N/A	Yes	Good	• Most flexibility
SQL Server Import and Export Wizard	No	N/A	Yes	Good	• Simple UI on top of SSIS; also available in SSMS

Migrate from Other RDMSs

SQL Azure Migration Assistant can be used to migration database from Access, MySQL, Oracle, Sybase to SQL Azure.

Microsoft Codename “Data Transfer” can transfer data in CSV or Excel file to SQL Azure.

Migrate between SQL Azure Databases

To migrate data from one SQL Azure database to another SQL Azure database, you can use SQL Azure *database copy* and *SQL Azure Data Sync*.

SQL Azure supports a database copy feature. This creates a new database in SQL Azure which is a transactionally consistent copy of an existing database. To copy database, you must be connected to the master database of the SQL Azure server where the new database will be created, and use the CREATE DATABASE command:

```
CREATE DATABASE destination_database_name AS COPY OF

[source_server_name.]source_database_name
```

The new database can be on the same server, or on a different server. The user executing this statement must be in the *dbmanager* role on the destination server (to create a new database) and must be *dbowner* in the source database. For more information see Copying Databases in SQL Azure .

SQL Azure Data Sync enables creating and scheduling regular synchronizations between SQL Azure and either SQL Server or other SQL Azure databases. For more information, see SQL Azure Data Sync Overview.

Using Migration Tools

Data-tier Application DAC Package

Data-tier Applications (DAC) were introduced in SQL Server 2008 R2, with developer tool support in Visual Studio 2010. They are useful for packaging the schema, code and configuration of a database for deploying to another server. When a DAC is ready to deploy, it is built into a DAC package (.bacpac), which is a compressed file that contains the DAC definitions in the XML format. From SQL Server Management Studio, you can export database schema to a DAC package, and then deploy the package to SQL Azure.

Note

The DACPAC format is different from the BACPAC format. The BACPAC format extends the DACPAC format to include a metadata file and JavaScript Object Notation (JSON) encoded table data, in addition to the standard .dacpac file contents. The BACPAC format is discussed in the DAC Import Export section.

You have the option to modify DAC package using Visual Studio 2010 before deployment. Within the DAC project, you can specify pre-deployment and post-deployment scripts. These are Transact-SQL scripts that can perform any action, including inserting data in the post-deployment scripts. However it is not recommended to insert a large amount of data using a DAC package.

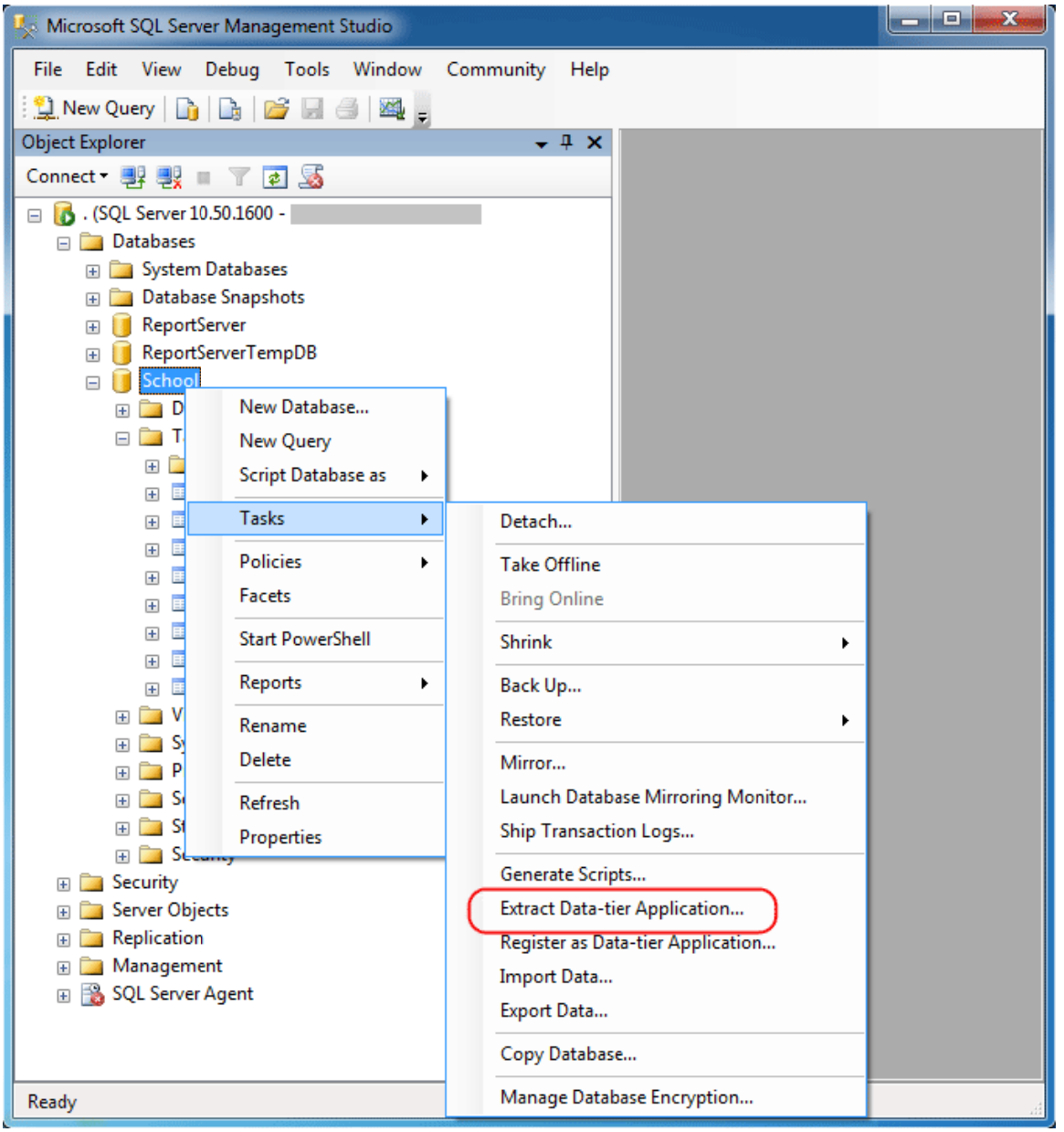
Installation and Usage

DAC is shipped with SQL Server 20008 R2. There are two main steps involved migrating SQL Server database schema to SQL Azure:

1. **Extract a DAC package from a SQL Server database:**

   The *Extract Data-tier Application Wizard* can be used to build a DAC package based on an existing database. The DAC package contains the selected objects from the database, and associated instance-level objects such as the logins that map to database users.

   Here is a screenshot of opening the wizard:



The wizard involves the following main steps:

a. Set the DAC properties, including DAC application name, version, description, and the package file location.

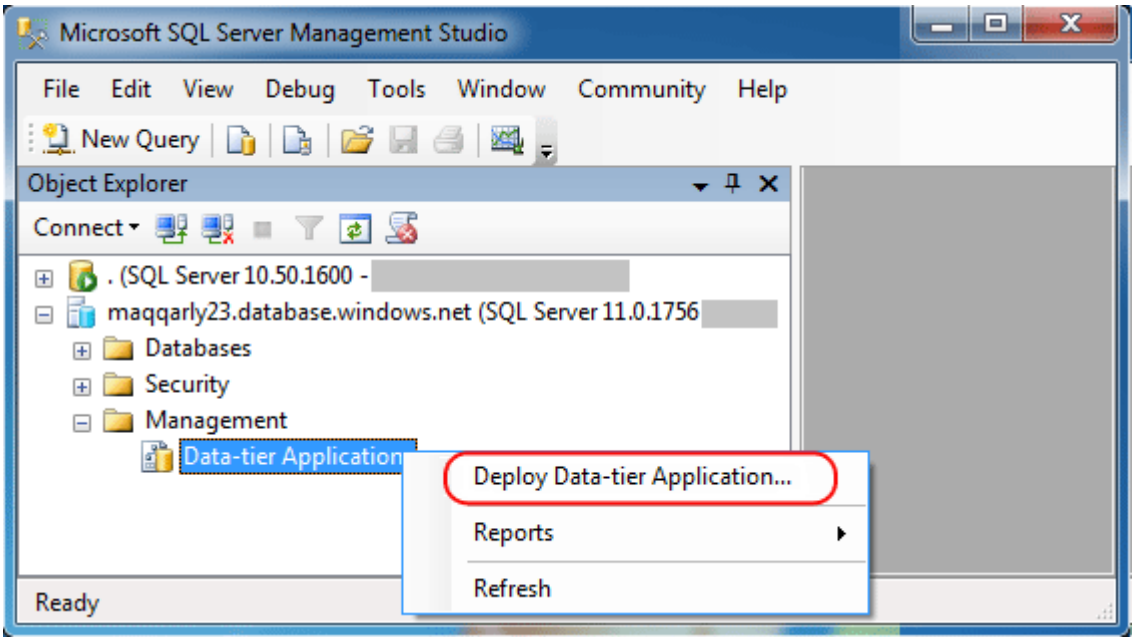
b. Validate that all the database objects are supported by a DAC.

c. Build the package.

A DAC can only be extracted from a database in SQL Azure, or SQL Server 2005 Service Pack 4 (SP4) or later. You cannot extract a DAC if the database has objects that are not supported in a DAC, or contained users. For more information about the types of objects supported in a DAC, see DAC Support For SQL Server Objects and Versions.

2. **Deploy the DAC package to SQL Azure:**

   The *Deploy Data-tier Application Wizard* can be used to deploy a DAC package. You must first connect to the SQL Azure server from SQL Server Management Studio. The wizard creates the database if the database doesn't exist. The wizard deploys the DAC package to the instance of the Database Engine associated with the node you selected in the Object Explorer hierarchy. For example, in the following screenshot, it deploys the package to the SQL Server called *maqqarly23.database.windows.net*:



   Important

   It is a good practice to review the contents of a DAC package before deploying it in production, especially when the deployed package was not developed in your organization. For more information, see Validate a DAC Package.

   The wizard involves the following main steps:

   a. Select the DAC package.
   b. Validate the content of the package.
   c. Configure the database deployment properties, where you specify the SQL Azure database.
   d. Deploy the package.

Other than using the wizard, you can also use PowerShell with the dacstore.install() method to migration schema to SQL Azure.

Resources

- Understand Data-tier Applications.
- Extract a DAC From a Database
- Deploy a Data-tier Application

Data-tier Application BACPAC Package

A data-tier application (DAC) is a self-contained unit for developing, deploying, and managing data-tier objects. DAC enables data-tier developers and database administrators to package Microsoft SQL Server objects, including database objects and instance objects, into a single entity called a DAC package (.dacpac file). The BACPAC format extends the DACPAC format to include a metadata file and JavaScript Object Notation (JSON)–encoded table data in addition to the standard .dacpac file contents. You can package your SQL Server database into a .bacpac file, and use it to migrate the database to SQL Azure.

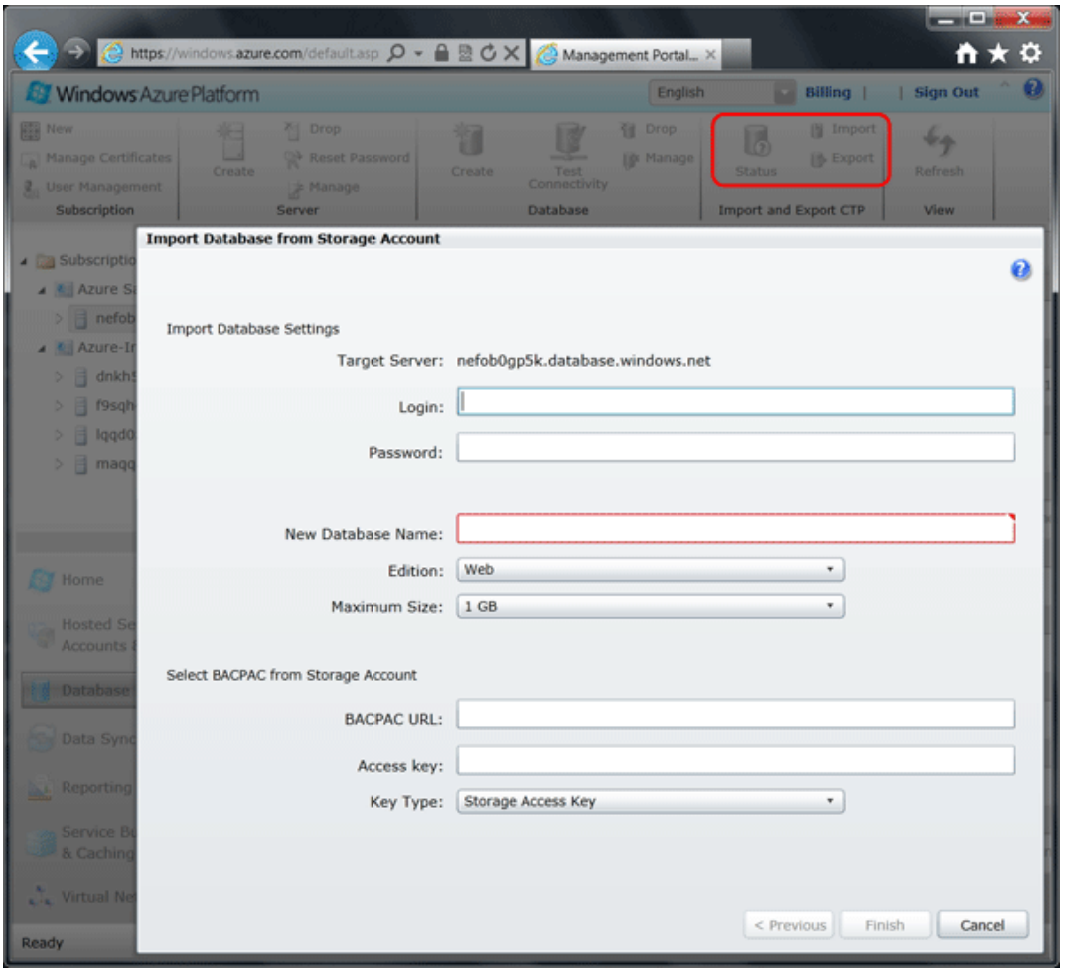
Note

DACPAC & BACPAC are similar but actually target very different scenarios. DACPAC is focused on capturing and deploying schema. Its primary use case is for deploying to development, testing, and then production environment.

BACPAC is focused on capturing schema and data. It is the logical equivalent of a database backup and cannot be used to upgrade existing databases. BACPAC's primary use cases are moving a database from one server to another (or to SQL Azure) and archiving an existing database in an open format.

The *Import and Export Service for SQL Azure* is currently available as a public CTP. The service can directly import or export BACPAC files between a SQL Azure database and Windows Azure Blob storage. The Import and Export Service for SQL Azure provides some public REST endpoints for the submission of requests.

The Windows Azure Platform Management Portal has an interface for calling the Import and Export Service for SQL Azure.



SQL Server Management Studio currently does not support exporting a database into a BACPAC file. You can leverage the DAC API to import and export data.

The *SQL DAC Examples* shows how to use the Data-tier Application Framework API to migrate databases from SQL Server to SQL Azure. The package provides two command line utilities and their source code:

- **DAC Import and Export Client-side tools** can be used to export and import bacpac files.
- **DAC Import and Export Service client** can be used to call the Import and Export Service for SQL Azure to import and export bacpac files between Windows Azure Blob storage and a SQL Azure database.

  One way to copy a bacpac file to Windows Azure Blob storage is to use Microsoft Codename "Data Transfer". For more information, see the Microsoft Codename Data Transfer section.

Note

The ability to import and export data to SQL Azure using the Data-tier Application (DAC) framework is currently only available as CodePlex examples. The tools are only supported by the community.

Installation and Usage

This section demonstrates how to use the client tools of the SQL DAC Examples for migrating database from SQL Server to SQL Azure.

The SQL DAC Examples can be downloaded from CodePlex. To run the sample, you must also install Data-Tier Application Framework on your computer.

Before using the tools to migrate database, you must create the destination SQL Azure database first. There are two steps involved using the tool for migration:

1. **Export a SQL Server Database**

   Assume a database exists that is running on SQL Server 2008 R2, which a user has integrated security access to. The database can be exported to a ".bacpac" file by calling the sample EXE with the following arguments:

   ```
   DacCli.exe -s serverName -d databaseName -f
   C:\filePath\exportFileName.bacpac -x -e
   ```

2. **Import the Package to SQL Azure**

   Once exported, the export file can be imported to a SQL Azure database with the following arguments:

   ```
   DacCli.exe -s serverName.database.windows.net -d databaseName -f
   C:\filePath\exportFileName.bacpac -i -u userName -p password
   ```

Resources

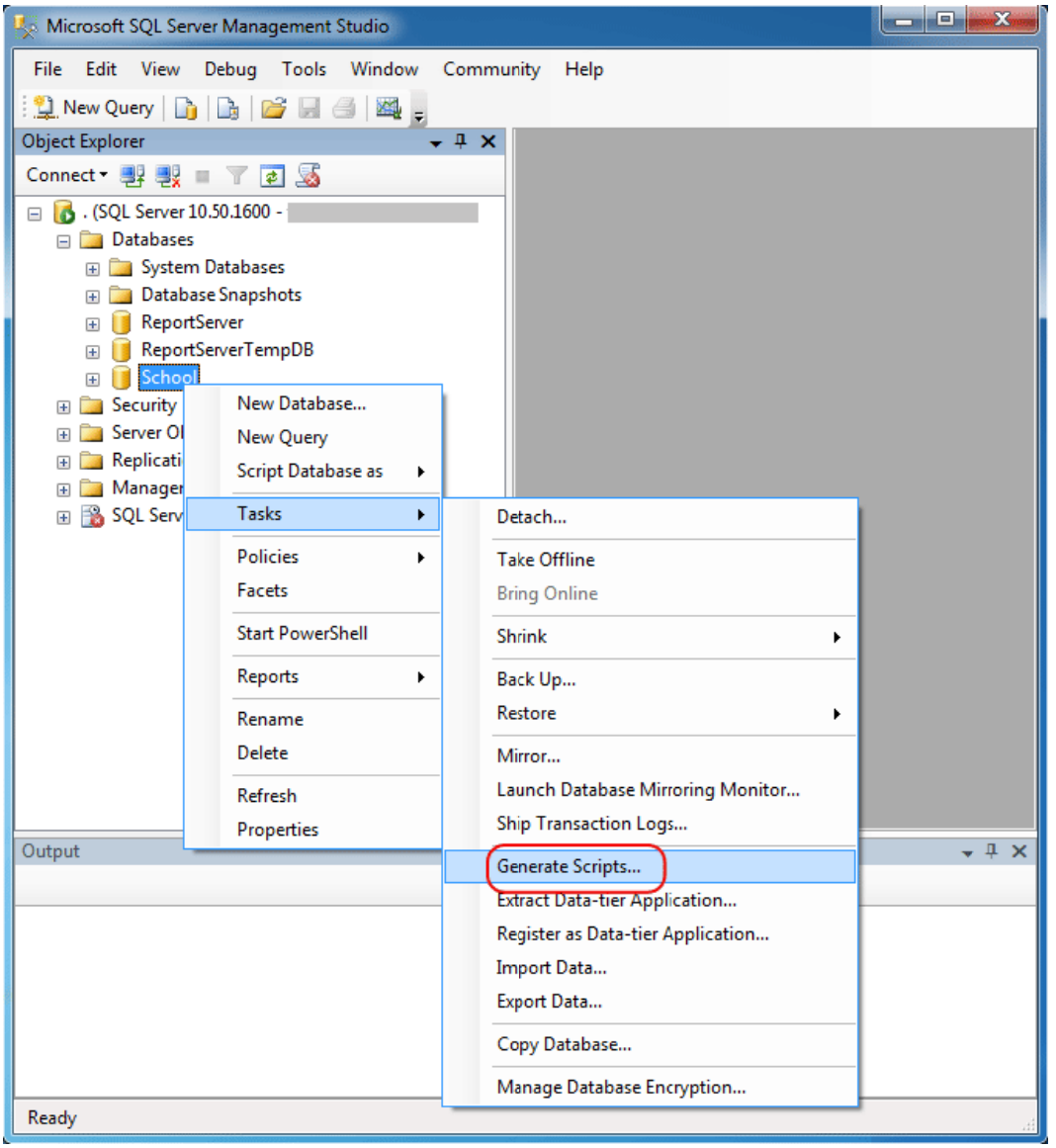
- How to Use Data-Tier Application Import and Export with SQL Azure
- DAC Framework Client Side Tools Reference

Generate Scripts Wizard

The Generate Scripts Wizard can be used to create Transact-SQL scripts for SQL Server database and/or related objects within the selected database. You can then use the scripts to transfer schema and/or data to SQL Azure.

Installation and Usage

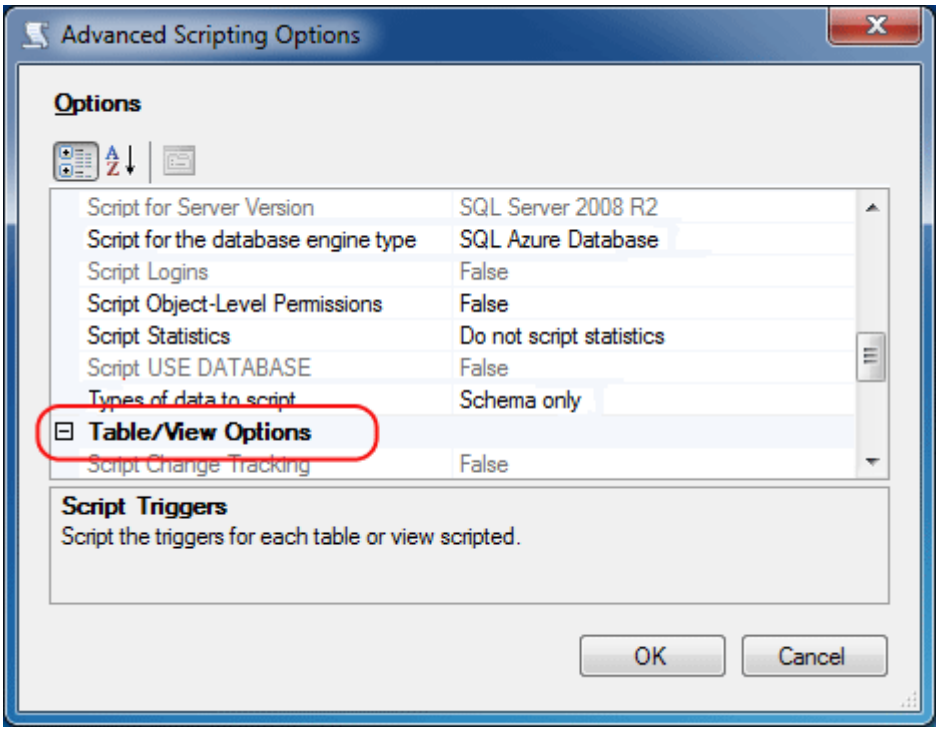
The Generate Scripts Wizard is installed with SQL Server 2008 R2. The wizard can be opened from SQL Server Management Studio 2008 R2. The following screenshot shows how to open the wizard:



The wizard involves the following main steps:

1. Choose objects to export.
2. Set scripting options. You have the options to save the script to file, clipboard, new query window; or publish it to a Web service.
3. Set advanced scripting options.

   By default, the script is generated for stand-alone SQL Server instance. To change the configuration, you must click the **Advanced** button from the **Set Scripting Options** dialog, and then set the **Script for the database engine type** property to **SQL Azure**.



You can also set the **Types of data to script** to one of the following based on your requirements: **Schema only**, **Data only**, **Schema and data**.

After the script is created, you have the option to modify the script before running the script against a SQL Azure database to transfer the database.

Resources

- How to: Migrate a Database by Using the Generate Scripts Wizard (SQL Azure Database)

bcp

The bcp utility is a command line utility that is designed for high performing bulk upload to SQL Server or SQL Azure. It is not a migration tool. It does not extract or create schema. You must first transfer the schema to SQL Azure database using one of the schema migration tools.

Note

You can use bcp to backup and restore your data on SQL Azure.

Note

The SQL Azure Migration Wizard uses bcp.

Installation and Usage

The bcp utility is shipped with SQL Server. The version shipped with SQL Server 2008 R2 is fully supported by SQL Azure.

There are two steps involved using bcp:

1. **Export the data into a data file.**

   To export data out of SQL Server database, you can run the following statement at command prompt:

   ```
   bcp tableName out C:\filePath\exportFileName.dat –S serverName –T –n -q
   ```

   The out parameter indicates copying data out of SQL Server. The -n parameter performs the bulk-copy operation using the native database data types of the data. The -q parameter executes the SET QUOTED_IDENTIFIERS ON statement in the connection between the bcp utility and your SQL Server instance.

2. **Import the data file into SQL Azure**

   To import data into your SQL Azure database, you must first create the schema in the destination database, and then run the bcp utility on from a command prompt:

   ```
   Bcp tableName in c:\filePath\exportFileName.dat –n –U userName@serverName –S tcp:serverName.database.windows.net –P password –b batchSize
   ```

   The –b parameter specifies the number of rows per batch of imported data. Each batch is imported and logged as a separate transaction that imports the whole batch before being committed. Identify the best batch size and using the batch size is a good practice for reducing connection lose to SQL Azure during data migration.

The following are some best practices when you use bcp to transfer a large amount of data.

1. Use the –N option to transfer data in the native mode so that no data type conversion is needed.
2. Use the –b option to specify the batch size. Be default, all the rows in the data file are imported as one batch. In case of a transaction failure, only insertions from the current batch are rolled back.
3. Use the –h “TABLOCK, ORDER(…)” option. The –h “TABLOCK” specifies that a bulk update table-level lock is required for the duration of the bulk load operation; otherwise, a row-level lock is acquired. It can reduce lock contention on the table. The –h “ORDER(…)” option specify the sort order of the data in the data file. Bulk import performance is improved if the data being imported is sorted according to the clustered index on the table.

You can use the –F and –L options to specify the first and last rows of a flat file for the upload. This was useful to avoid having to physically split the data file to achieve multiple stream upload.

Resources

- bcp utility

SQL Azure Migration Wizard

The SQL Azure Migration Wizard is an open source UI tool that helps migrating SQL Server 2005/2008 databases to SQL Azure. Other than migrating data, it can also be used to identify any compatibility issues, fix them where possible and notify you of all issues it knows about.

The SQL Azure Migration Wizard has built-in logic for handling connection lose. It breaks down the transactions into smaller bunches and runs until SQL Azure terminates the connection. When

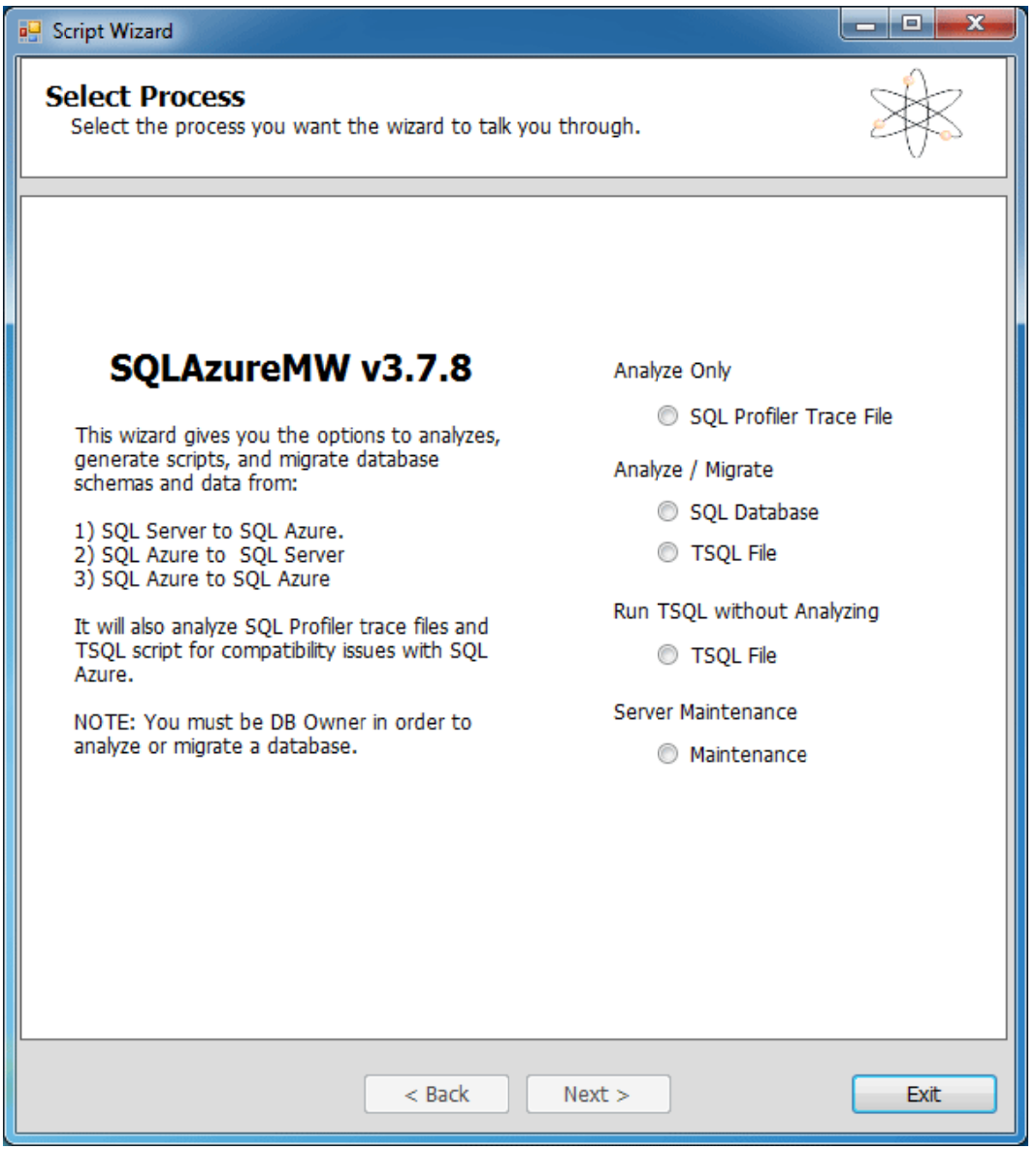
the wizard encounters the connection error, it reestablishes a new connection with SQL Azure and picks up processing after the last successful command. In the same manner, when using bcp to upload the data to SQL Azure, the wizard chunks the data into smaller sections and uses retry logic to figure out the last successful record uploaded before the connection was closed. Then it has bcp restart the data upload with the next set of records.

Note

The SQL Azure Migration Wizard is an open source tool built and supported by the community.

Installation and Usage

The SQL Azure Migration Wizard can be downloaded from http://sqlazuremw.codeplex.com. Unzip the package to your local computer, and run SQLAzureMW.exe. Here is a screenshot of the application:



The wizard involves the following steps:

1. Select the process you want the wizard to talk you through.
2. Select the source you want to script.
3. Select database objects to script.
4. Generate the script. You have the option to modify the script afterwards.
5. Enter information for connecting to target server. You have the option to create the destination SQL Azure database.
6. Execute script against destination server.

Resources

- Using the SQL Azure Migration Wizard (video)

SQL Server Integration Services

SQL Server Integration Services (SSIS) can be used to perform a broad range of data migration tasks. It is a powerful tool when operating on multiple heterogeneous data sources and destinations. This tool provides support for complex workflow and data transformation between the source and destination. Even though SSIS is not currently supported by SQL Azure, you can run it on an on-premise SQL Server 2008 R2 to transfer data to SQL Azure Database.

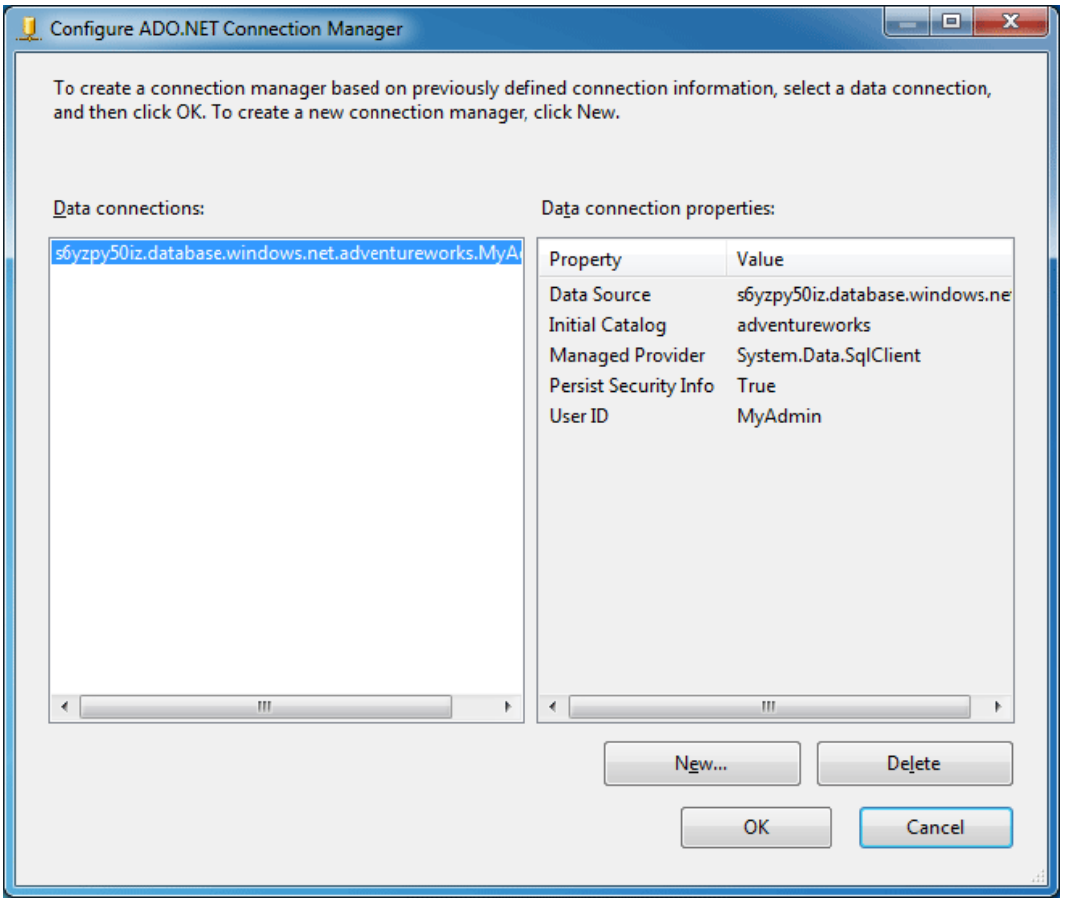
The SSIS Import/Export Wizard lets the user create packages that move data from a single data source to a destination with no transformations. The Wizard can quickly move data from a variety of source types to a variety of destination types, including text files and other SQL Server instances.

Installation and Usage

You must use the SQL Server 2008 R2 version of SSIS to connect to SQL Azure.

The ADO.NET adapters have the required support for SQL Azure. It provides an option to bulk load data specifically for SQL Azure. Use ADO.NET Destination adapter to transfer data to SQL Azure. Connecting to SQL Azure Database by using OLEDB is not supported.

The following is a screenshot for configuring the ADO.NET Connection to SQL Azure:



Because your package might fail due to throttling or network issue, you might consider designing the packages in a way so that it can be resumed at the point of failure, as different to restarting the package.

When you configure the ADO.NET destination, make sure to use the “Use Bulk Insert when possible” option. That allows you to use bulk load capabilities to improve the transfer performance.

One way to improve the performance is to split source data into multiple files on the file system. In SSIS Designer, you can reference the files using the Flat File Component. Each input file is then connected to an ADO .Net Component that has the Use Bulk Insert when possible flag checked.

Resources

- Designing and Implementing Packages (Integration Services)

SQL Server Import and Export Wizard

The SQL Server Import and Export Wizard offers the simplest method to create a SQL Server Integration Services package for a simple import or export. This wizard configures the connections, source, and destination, and adds any data transformations that are required to let

you run the import or export immediately. After a package is created, you have the option to modify the package in SSIS Designer.

The wizard supports the following data sources:

- .NET Framework Data Provider for ODBC
- .NET Framework Data Provider for Oracle
- .NET Framework Data Provider for SQL Server
- Flat File Source
- Microsoft OLE DB Provider for Analysis Services 10.0
- Microsoft OLE DB Provider for Search
- Microsoft OLE DB Provider for SQL Server
- SQL Native Client
- SQL Server Native Client 10.0

The SQL Server Import and Export Wizard can only transfer data. Before using the wizard, you must transfer the schema using one of the schema migration tools, Generate Scripts Wizard or DAC package.

Note

On a 64-bit computer, Integration Services installs the 64-bit version of the SQL Server Import and Export Wizard (DTSWizard.exe). However, some data sources, such as Access or Excel, only have a 32-bit provider available. To work with these data sources, you might have to install and run the 32-bit version of the wizard. To install the 32-bit version of the wizard, select either Client Tools or Business Intelligence Development Studio during setup.

Installation and Usage

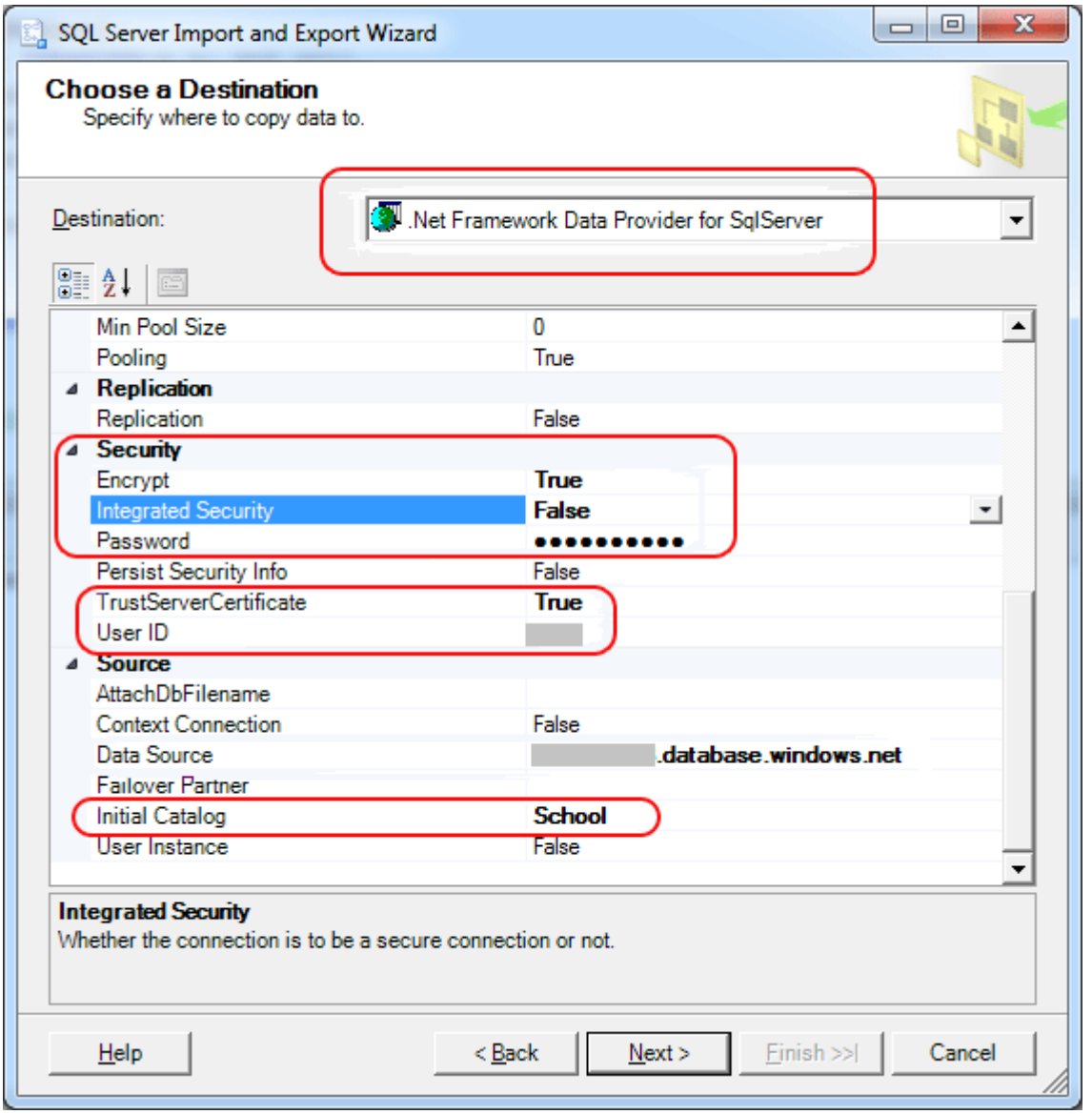
In SQL Server 2008 R2 or later, the SQL Server Import and Export Wizard supports for SQL Azure. There are several ways to start the wizard:

1. On the **Start** menu, point to **All Programs**, point to **Microsoft SQL Server 2008**, and then click **Import and Export Data**.
2. In Business Intelligence Development Studio, right-click the **SSIS Packages** folder from Solution Explorer, and then click **SSIS Import and Export Wizard**.
3. In Business Intelligence Development Studio, on the **Project** menu, click **SSIS Import and Export Wizard**.
4. In SQL Server Management Studio, connect to the Database Engine server type, expand Databases, right-click a database, point to **Tasks**, and then click **Import Data** or **Export data**.
5. In a command prompt window, run DTSWizard.exe, located in C:\Program Files\Microsoft SQL Server\100\DTS\Binn.

The migration involves the following main steps:

1. Choose a data source from which to copy data.
2. Choose a destination where to copy data to.

   To export data to SQL Azure, you must choose the *.NET Framework Data Provider for SQLServer* as the destination:



3. Specify table copy or query.
4. Select source objects.
5. Save and run the package.

After the SQL Server Import and Export Wizard has created the package, you can optionally save the package to run it again later, or to refine and enhance the package in SQL Server Business Intelligence (BI) Development Studio.

Note

If you save the package, you must add the package to an existing Integration Services project before you can change the package or run the package in BI Development Studio.

Resources

- How to: Run the SQL Server Import and Export Wizard
- Using the SQL Server Import and Export Wizard to Move Data

Microsoft Codename "Database Transfer"

Microsoft Codename "Data Transfer" is a cloud service that lets you transfer data from your computer to a SQL Azure database or Windows Azure Blob storage. You can upload any data format to Windows Azure Blob storage, and data that is stored in comma-separated value (CSV) or Microsoft Excel format (.xlsx) to a SQL Azure database. When you upload data to a SQL Azure database, it is transformed into database tables.

Usage

The Data Transfer service is accessible at https://web.datatransfer.azure.com/. From the home page, you have the options to import data, manage your datasets and your stores.

The data import to SQL Azure process involves the following steps:

- Enter your SQL Azure credentials.
- Choose a file to transfer.
- Analyze the data file and then transfer the data.

Resources

- Microsoft Codename "Data Transfer" Tutorial

SQL Server Migration Assistant

SQL Server Migration Assistant (SSMA) is a family of products to reduce the cost and risk of migration from Oracle, Sybase, MySQL and Microsoft Access databases to SQL Azure or SQL Server. SSMA automates all aspects of migration including migration assessment analysis, schema and SQL statement conversion, data migration as well as migration testing.

Installation and Usage

SSMA is a Web download. To download the latest version, see the SQL Server Migration Tools product page. As of this writing, the following are the most recent versions:

- Microsoft SQL Server Migration Assistant for Access v5.1
- Microsoft SQL Server Migration Assistant for MySQL v5.1
- Microsoft SQL Server Migration Assistant for Oracle v5.1
- Microsoft SQL Server Migration Assistant for Sybase v5.1
- Microsoft SQL Server Migration Assistant 2008 for Sybase PowerBuilder Applications v1.0

SSMS is installed by using a Windows Installer-based wizard. SSMA is free, but you must download a registration key. After you install and run the application, the application prompts you to register and download the registration key.

The migration process of the SSMA for Access involves the following steps:

1. Create a new migration wizard. Make sure selecting SQL Azure in the **Migrate To** box.
2. Add Access databases.

3. Select the Access objects to migrate.
4. Connect to SQL Azure.
5. Link tables. If you want to use your existing Access applications with SQL Azure, you can link your original Access tables to the migrated SQL Azure tables. Linking modifies your Access database so that your queries, forms, reports, and data access pages use the data in the SQL Azure database instead of the data in your Access database.
6. Convert selected objects.
7. Load converted objects into SQL Azure database.
8. Migrate data for selected Access objects.

Resources

- SQL Server Migration Assistant
- Migrating Microsoft Access Applications to SQL Azure (video)
- SQL Server: Manage the Migration

See Also

Migrating Database to SQL Azure

Using the ReportViewer ASP.NET Control in Windows Azure

This article was written using: Windows Azure SDK 1.6, Visual Studio 2010 ReportViewer control for ASP.NET (Microsoft.ReportViewer.WebForms.dll version 10.0.40219.329)

The MSDN articles Getting Started Guide for Application Developers (SQL Azure Reporting) and How to: Use ReportViewer in a Web Site Hosted in Windows Azure contain information on using the **ReportViewer** control for ASP.NET in a Windows Azure application with SQL Azure Reporting. This article provides key additional information for successful use of **ReportViewer** in a Windows Azure website that uses more than one web role instance.

This article will outline two solutions that will allow you to deploy **ReportViewer** solutions in a multi-instance Windows Azure website.

IReportServerCredentials causes ASP.NET Session State to be used

As noted in the documentation for IReportServerCredentials, when the **ReportViewer** uses an implementation of this interface, it will use session state to store the object and the **ReportViewer** state.

When using a single web role instance, the ASP.NET **ReportViewer** control will work correctly without explicitly setting sessionState in the web.config. It will work just by implementing **IReportServerCredentials** and setting **ReportViewer.ServerReport.ReportServerCredentials** to an instance of your **IReportServerCredentials** implementation. This will work because it will automatically use the default in-process session state provider. With only a single web role instance, requests will always hit the same instance, and the session state will always exist on the instance. Of course the instance will be rebooted for Windows patches, and it could be rebooted to move to another physical host machine in the Azure data center. Reboots of the instance will cause the in-process session state to be lost.

Multiple web role instances hosting the ReportViewer

When you use multiple instances of the web role (for high availability and scale-out) for your solution, and you are using the default in-process session state provider, you will intermittently receive the following error when loading the web page that contains a **ReportViewer** control:

```
ASP.NET session has expired or could not be found
```

This error occurs because your initial request hit a single web role instance and had its session state stored in-process on that web role instance. A subsequent request from the same browser might hit one of the other web role instances, where the session state does not exist.

The most direct approach is to use the Azure Caching session state provider to provide session state across all instances as described in Session State Provider for Winddows Azure Caching. Unfortunately, the use of Azure Caching session state provider surfaces a bug in the **NetDataContractSerializer** when using the **ReportViewer** control that comes with Visual Studio 2008 or Visual Studio 2010. The ASP.NET Session State Provider for Windows Azure Caching uses the **NetDataContractSerializer** to serialize objects into session state.

The bug causes an exception to be raised by the web page that contains a **ReportViewer** control (along with a stack trace):

```
Server Error in '/' Application.

Type 'Microsoft.Reporting.WebForms.SyncList`1[[System.String, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089]]' is an invalid collection type since
it does not have a default constructor.

Description: An unhandled exception occurred during the execution of the current web
request. Please review the stack trace for more information about the error and where it
originated in the code.

Exception Details: System.Runtime.Serialization.InvalidDataContractException: Type
'Microsoft.Reporting.WebForms.SyncList`1[[System.String, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089]]' is an invalid collection type since
it does not have a default constructor.
```

Note

The **ReportViewer** control that will ship with the next version of Visual Studio after Visual Studio 2010 should have this issue fixed.

Solutions

There are two possible solutions for successful use of **ReportViewer**:

1. Make the **ReportViewer** "session-less" to prevent it from using ASP.NET session state. **This is the preferred method.**
2. Install hotfix 2624480 (http://support.microsoft.com/kb/2624480) as a startup task in the role instance. This hotfix fixes an issue with the **NetDataContractSerializer**. This approach will cause all role instance reboots to be slower because the startup task to install the hotfix must complete before the instance is available.

"Session-less" ReportViewer

You can find a discussion of making **ReportViewer** “session-less” in the following articles:

- Custom Credentials in the Report Viewer
- Specifying Connections and Credentials for the ReportViewer Web Server Control

Instead of implementing **IReportServerCredentials** and assigning an instance to **ReportViewer.ServerReport.ReportServerCredentials**, you can implement IReportServerConnectionor IReportServerConnection2 and point to this class in your web.config file.

The following code is an implementation of **IReportServerConnection** (with error handling removed for brevity). This implementation simply retrieves the Report Server user name, password, and server name from the web.config AppSettings:

Note

Note that this code is based on the Windows Azure ReportViewer Control Sample linked to from the SQL Azure Reporting Samples TechNet Wiki page.

```
namespace ReportViewerRemoteMode
{
    public sealed class ReportServerConnection : IReportServerConnection
    {
        public bool GetFormsCredentials(out Cookie authCookie, out string user,
                                        out string password, out string authority)
        {
            authCookie = null;
            user = ConfigurationManager.AppSettings["RSUSERNAME"];
            password = ConfigurationManager.AppSettings["RSPASSWORD"];
            authority = ConfigurationManager.AppSettings["RSSERVER_NAME"];
            return true;
        }

        public WindowsIdentity ImpersonationUser
        {
            get { return null; }
        }

        public ICredentials NetworkCredentials
        {
            get { return null; }
        }

        public Uri ReportServerUrl
        {
            get
            {
                return new Uri(String.Format("https://{0}/reportserver",
                               ConfigurationManager.AppSettings["RSSERVER_NAME"]));
            }
        }

        public int Timeout
        {
            get { return 60000; }
        }
    }
```

```
}
```

Assuming that your assembly name is **ReportViewerRemoteMode**, the following entry in the <appSettings> section in the web.config will cause the **ReportViewer** control to use this **IReportServerConnection** implementation:

```
<appSettings>
<!-- Custom Report Server Connection implementation-->
<add key="ReportViewerServerConnection"
         value="ReportViewerRemoteMode.ReportServerConnection, ReportViewerRemoteMode"/>
</appSettings>
```

Do not set the **ReportViewer.ServerReport.ReportServerCredentials** or the **ReportViewer.ServerReport.ReportServerUrl** in your web page code-behind class.

This solution will not work for all scenarios because the web.config can only point to one **IReportServerConnection** implementation.

Hotfix 2624480 installed as startup task

Another solution is to install hotfix 2624480, http://support.microsoft.com/kb/2624480, as a startup task in the role instance. This hotfix fixes an issue with the **NetDataContractSerializer** that will allow the **ReportViewer** control to successfully serialize to session state without throwing an exception.

In order to implement this solution, you will need the "x64" version of the hotfix executable downloaded from http://support.microsoft.com/kb/2624480. Add the file "NDP40-KB2624480-x64.exe" to your Visual Studio project, and set the file properties to copy to the output directory. Then in the ServiceDefinition.csdef file, you need to add a `<Startup>`element inside the `<WebRole>` element as follows. This causes the hotfix to silently install without user prompting or intervention.

Note

Note the "/q" parameter for the .EXE

```
<Startup>
<Task commandLine="NDP40-KB2624480-x64.exe /q"
        executionContext="elevated" />
</Startup>
```

Now when the role instance starts, it will install the hotfix before the instance starts accepting web requests.

One drawback of this approach is that it will cause all role instance reboots and deployments to be slower because the startup task to install the hotfix must complete before the instance is available. This can cause deploy times and startup times to be significantly slower. A test of this approach with deploying two "small" web role instances to Windows Azure from within Visual Studio showed that deployments took twice as long when using this startup task to install the hotfix.

If possible, make the **ReportViewer** "session-less". If this is not possible, then install the hotfix as a startup task as a fallback method.

Additional Resources

Custom Credentials in the Report Viewer

Specifying Connections and Credentials for the ReportViewer Web Server Control

Getting Started with Defining Startup Tasks for Roles in Windows Azure

Introduction to Windows Azure Startup Tasks

Author: Suren Machiraju

Reviewers: Jaime Alva Bravo and Steve Wilkins

After designing, coding and deploying your Azure AppFabric solution you may find that it does not work. This article provides guidance on how to efficiently test your Azure AppFabric applications through the software development life cycle both at the business logic and the comprehensive end-to-end scenario tests. In this article, we demonstrate how to:

- Develop unit tests for the business logic components while eliminating dependencies on Azure AppFabric components.
- Design integration end-to-end tests and eliminate overhead for setup, initialize, cleanup and teardown of the Azure AppFabric Service resources (for example, Queues) for every test run. Further eliminate creating duplicate infrastructures for ACS namespaces, Service Bus Queues, and other resources.

Additionally, this article provides an overview of the various testing technologies and techniques for testing your Azure AppFabric applications.

The Types of Tests

In this article, we will focus on the following tests:

- Unit Tests are narrowly focused tests designed to exercise one specific bit of functionality. These tests are often referred to as the Code Under Test or CUT. Any dependencies taken by the CUT are somehow removed.
- Integration Tests are broader tests for exercising multiple bits of functionality at the same time. In many cases, these resemble unit tests that cover multiple feature areas and include multiple dependencies.

Overall our tests focus on creating and using Test Doubles. We will use the following types of Test Doubles:

- Mocks are simulated objects that imitate the behavior of real objects in controlled ways. They are replacements of what is dynamically produced by a framework.
- Fakes are simulated objects that implement the same interface as the object that they represent. Fakes return pre-defined responses. Fakes contain a set of method stubs and serve as replacements for what you build by hand.
- Stubs simulate the behavior of software objects.

Our tests, upon executing can verify both state (for example, after you make a call a specific value is returned) and behavior (the method is called in a certain order or a certain number of times).

The Azure Dependencies

One of the major goals of unit testing is to eliminate dependencies. For the Azure framework, these dependencies include the following:

- Service Bus Queues
- Access Control Service
- Cache
- Windows Azure Tables, Blobs, Queues
- SQL Azure
- Cloud Drive
- Other Web Services

When building tests for Azure Applications we will replace these dependencies to better focus our tests on exercising the desired logic.

In this article, we will provide examples of Service Bus Queues with full knowledge that the tools and techniques demonstrated here will also apply to all other dependencies.

The Testing Framework

To implement the testing framework for your Azure AppFabric applications, you will need:

- A unit a testing framework to define and run your tests.
- A mocking framework to help you isolate dependencies and build narrowly scoped unit tests.
- Tools to help with automatic unit test generation for increased code coverage.
- Other frameworks that can help you with testable designs by taking advantage of dependency injection and applying the Inversion of Control Pattern.

Unit Testing with MS Test

Visual Studio includes a command line utility MS Test that you can use to execute unit tests created in Visual Studio. Visual Studio also includes a suite of project and item templates to support testing. You typically create a new test project and then add classes (known as test fixtures) adorned with [TestClass] attributes that contain methods adorned with [TestMethod]. Within MS Test, various windows within Visual Studio enable you to execute unit tests defined in the project, and review the results after you execute them.

Note

Visual Studio Express and Test Professional editions do not contain MS Test.

In MS Test, unit tests follow an AAA pattern - Arrange, Act, Assert.

- Arrange - build up any pre-requisite objects, configurations, and all other necessary preconditions and inputs needed by the CUT.
- Act - perform the actual, narrowly scoped test on the code.
- Assert - verify that the expected results have occurred.

The MS Test framework libraries include PrivateObject and PrivateType helper classes that use reflection to make it easy to invoke non-public instance members or static members, from within the unit test code.

Premium and ultimate editions of Visual Studio include enhanced unit test tooling that integrate with MS Test that enable you to analyze the amount of code exercised by your unit tests, and visualize the result by color coding the source code, in a feature referred to as Code Coverage.

Mocking with Moles

A goal of unit testing is to be able to test in isolation. However, often the code under test cannot be tested in isolation. Sometimes the code is not written for testability. To rewrite the code would be difficult because it relies on other libraries that are not easily isolated, such as those that interact with external environments. A mocking framework helps you isolate both types of dependencies.

A list of mocking frameworks you might consider is available in the links section at the end of this article. For our purposes, we will focus on how to use Moles, a mocking framework from Microsoft Research.

Most mocking frameworks will create mock types dynamically by deriving from the type you indicate for the members you want to change. Fewer frameworks have support for handling sealed classes, non-virtual methods, static members, delegates or events that require alternative techniques (such as using the .NET Profiling API). Moles framework provides this advanced support.

Most of the mocking frameworks do not enable you to rewrite the logic within a constructor. If you are creating an instance of a type that you have to isolate, and the constructor of this type takes dependencies you want to isolate, you will have to creatively mock the calls made by the constructor instead of the constructor itself. Moles can be used to insert logic before constructor calls, but it stops short of letting you completely rewrite the constructor's logic.

Moles provides all these features at no cost to you. In addition, it integrates well with Visual Studio 2010 and MS Test, and tests generated with it are fully debuggable within Visual Studio.

Automated Test Creation with PEX

PEX (Program EXploration) is a white-box test generation tool available to MSDN subscribers from Microsoft Research. You can use PEX to automatically create unit tests with high code coverage by intelligently selecting input parameters for CUT, and recursively for parameterized unit tests. PEX is capable of building the smallest number of tests that produce the greatest code coverage.

PEX installs and runs fully integrated within Visual Studio 2010, and installs Moles as part of the installation process.

Frameworks for Dependency Injection and Inversion of Control (IoC) Containers

You can use Microsoft Unity for extensible dependency injection and IoC. It supports interception, constructor injection, property injection, and method call injection.

Microsoft Unity and similar tools help you build testable designs that let you insert your dependencies at all levels of your application, assuming that your application was designed and built with dependency injection and one of these frameworks in mind.

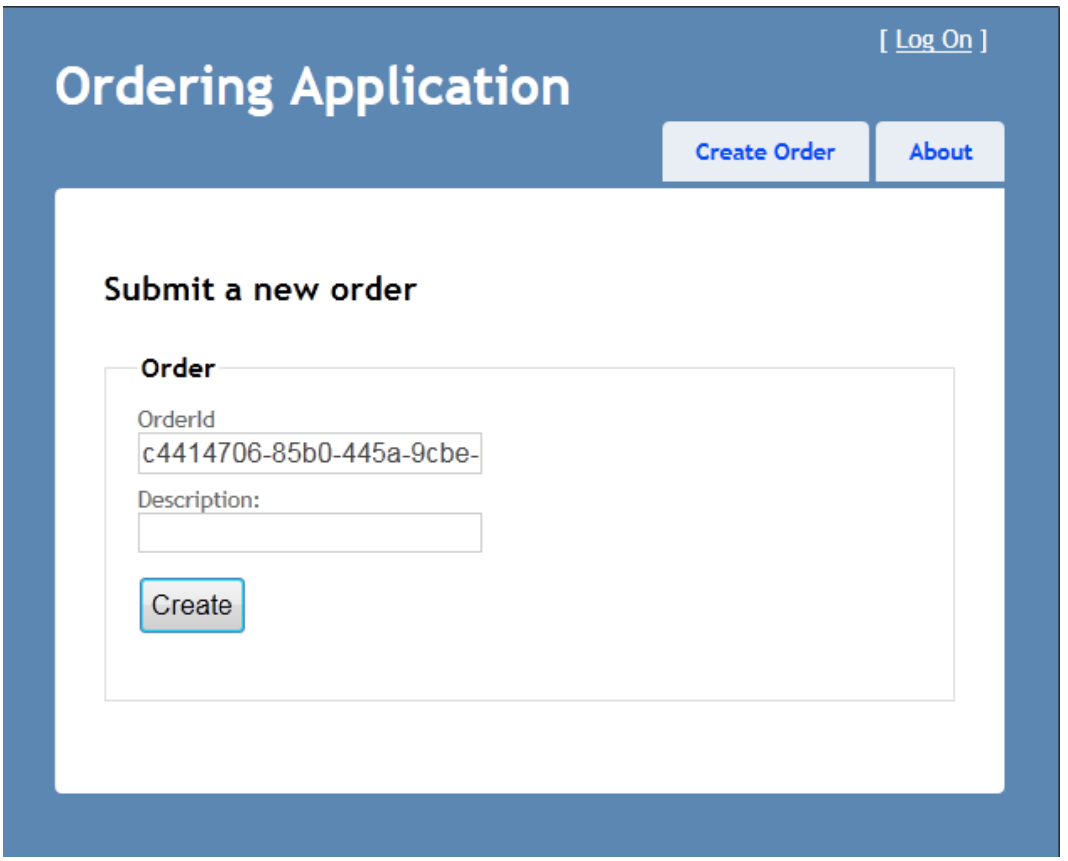
These frameworks are great for writing testable code, and ultimately good code, but can be somewhat heavy in their up-front design requirements. We will not cover DI or IoC containers in this article.

Testing Azure Solutions

In this section, we will describe a solution that includes a web role hosted web site that pushes messages to a Queue and a worker's role that processes messages from the Queue. We are interested in testing aspects of all three.

Unit Testing Controller Actions in a Web Role

Let us say you have a website that creates orders, and these orders are queued for processing by using an AppFabric Queue. Our example web page looks like this:



When the user clicks Create, it posts the new order to the Create action on the associate controller. The action is implemented as follows:

```
[HttpPost]
public ActionResult Create(Order order)
{
    try
    {
```

```
        string qName = RoleEnvironment.GetConfigurationSettingValue("QueueName");
        string qNamespace =
RoleEnvironment.GetConfigurationSettingValue("QueueServiceNamespace");
        string qUsername = RoleEnvironment.GetConfigurationSettingValue("QueueUsername");
        string qPassword = RoleEnvironment.GetConfigurationSettingValue("QueuePassword");

        //Push order to AppFabric Queue
        AppFabricQueue queue = new AppFabricQueue(new QueueSettings(qNamespace, qName,
Username, qPassword));

        queue.Send(order);

        queue.CloseConnections();

        return View("OrderCreated");
    }
    catch (Exception ex)
    {
        Trace.TraceError(ex.Message);
        return View("Error");
    }
}
```

Notice that the code retrieves settings from configuration through RoleEnvironment and sends a message that contains the order to the queue. The AppFabricQueue class we use here is a wrapper class that uses the Service Bus .NET APIs (MessageSender, MessageReceiver, etc.) to interact with Service Bus Queues.

Let us build a unit test for this method using Moles. Notice that the Create action uses the following methods:

- GetConfigurationSettingsValue (RoleEnvironment class)
- Send (AppFabricQueue class)

What we want to do is build detours for these methods using Moles to control their behavior and remove the dependencies on the real environment. Controlling behavior of these methods and removing their dependencies on the real environment will eliminate the need to run the tests in the emulator, on Azure, or to call out to the Service Bus Queue. We will exercise the Create action on the controller to verify that the Order input is the one sent to the queue (checking that it has the Order Id and Description as input to the action) and that the OrderCreated view is displayed as a result.

Accomplishing this with moles is easy. Within the Test project, right click the assembly containing the types you want to mock and select Add Moles Assembly.

In our example, we select Microsoft.Windows.Azure.ServiceRuntime and select Add Moles Assembly. An XML file named Microsoft.WindowsAzure.ServiceRuntime.moles will be added to the test project. Then repeat for the AF.Queues assembly.

When you build the Test project, references will be added to the auto-generated moled versions of these assemblies.

In our example these are Microsoft.Windows.Azure.ServiceRuntime.Moles and AF.Queues.Moles.

Create a new unit test method and decorate it with [HostType("Moles")]. Within the Unit Test, you use the "moled" types by accessing them through a Moles namespace. All moled types start with an M.

For example, the moled type for AppFabricQueue is accessed through AF.Queues.Moles.MAppFabricQueue.

You define your detours (the alternate logic to execute) by assigning a delegate to the method you want to replace.

For example, to detour RoleEnvironment.GetConfigurationSetting we define a delegate for MRoleEnvironment.GetConfigurationSettingValueString.

The complete unit test is as follows. Observe the detours we provide for RoleEnvironment.GetConfigurationSetting, the QueueSettings constructor, and the Send and CloseConnection methods of the AppFabricQueue class.

```
[TestMethod]
[HostType("Moles")]
public void Test_Home_CreateOrder()
{
    //Arrange

Microsoft.WindowsAzure.ServiceRuntime.Moles.MRoleEnvironment.GetConfigurationSettingValue
String = (key) =>
    {
        return "mockedSettingValue";
    };

    bool wasQueueConstructorCalled = false;
    AF.Queues.Moles.MAppFabricQueue.ConstructorQueueSettings = (queue, settings) =>
    {
        wasQueueConstructorCalled = true;
    };

    Order orderSent = null;
    AF.Queues.Moles.MAppFabricQueue.AllInstances.SendObject = (queue, order) =>
    {
       orderSent = order as Order;
    };

    bool wasConnectionClosedCalled = false;
    AF.Queues.Moles.MAppFabricQueue.AllInstances.CloseConnections = (queue) =>
```

```
    {
       wasConnectionClosedCalled = true;
    };

    Order inputOrder = new Order()
    {
       OrderId = "Test123",
       Description = "A mock order"
    };

    HomeController controller = new HomeController();

    //Act
    ViewResult result = controller.Create(inputOrder) as ViewResult;

    //Assert
    Assert.IsTrue(wasConnectionClosedCalled);
    Assert.IsTrue(wasQueueConstructorCalled);
    Assert.AreEqual("OrderCreated", result.ViewName);
    Assert.IsNotNull(orderSent);
    Assert.AreEqual(inputOrder.OrderId, orderSent.OrderId);
    Assert.AreEqual(inputOrder.Description, orderSent.Description);

}
```

Unit Testing Worker Roles that use AppFabric Queues

The web role took care of adding an order to the queue. Let us consider a worker's role that would process the orders by retrieving them from the queue and see how we can test that. The core item to test in the worker role is defined in the implementation of RoleEntryPoint, typically in the OnStart and Run methods.

The Run method in our worker role periodically polls the queue for orders and processes them.

```
public override void Run()
{
        AppFabricQueue queue = null;

        try
        {
                string qName = RoleEnvironment.GetConfigurationSettingValue("QueueName");
                string qNamespace =
RoleEnvironment.GetConfigurationSettingValue("QueueServiceNamespace");
                string qUsername =
RoleEnvironment.GetConfigurationSettingValue("QueueUsername");
```

```
                string qPassword =
RoleEnvironment.GetConfigurationSettingValue("QueuePassword");

                queue = new AppFabricQueue(new QueueSettings(qNamespace, qName,
qUsername, qPassword));
                queue.CreateQueueIfNotExists();

                while (true)
                {
                    Thread.Sleep(2000);

                    //Retrieve order from AppFabric Queue
                    TryProcessOrder(queue);
                }
        }
        catch (Exception ex)
        {
                if(queue !=null)
                    queue.CloseConnections();

                System.Diagnostics.Trace.TraceError(ex.Message);
        }

}
```

We want a test verifying that the routine correctly retrieves a message.

In this case we add a moled assembly for the worker role. For our specific project, this is the Mvc3Web assembly because we host the RoleEntryPoint there. The following is the complete unit test for run method of the worker role.

```
[TestMethod]
[HostType("Moles")]
public void Test_WorkerRole_Run()
{
    //Arrange

Microsoft.WindowsAzure.ServiceRuntime.Moles.MRoleEnvironment.GetConfigurationSettingValue
String = (key) =>
    {
        return "mockedSettingValue";
    };

    bool wasQueueConstructorCalled = false;
    AF.Queues.Moles.MAppFabricQueue.ConstructorQueueSettings = (queue, settings) =>
```

```
    {
        wasQueueConstructorCalled = true;
    };

    bool wasEnsureQueueExistsCalled = false;
    int numCallsToEnsureQueueExists = 0;
    AF.Queues.Moles.MAppFabricQueue.AllInstances.CreateQueueIfNotExists = (queue) =>
    {
        wasEnsureQueueExistsCalled = true;
        numCallsToEnsureQueueExists++;
    };

    bool wasTryProcessOrderCalled = false;
    int numCallsToTryProcessOrder = 0;
    Mvc3Web.Worker.Moles.MWorkerRole.AllInstances.TryProcessOrderAppFabricQueue =
    (actualWorkerRole, queue) =>
    {
         wasTryProcessOrderCalled = true;
         numCallsToTryProcessOrder++;

         if (numCallsToTryProcessOrder > 3) throw new Exception("Aborting Run");
    };

    bool wasConnectionClosedCalled = false;
    AF.Queues.Moles.MAppFabricQueue.AllInstances.CloseConnections = (queue) =>
    {
         wasConnectionClosedCalled = true;
    };

    Mvc3Web.Worker.WorkerRole workerRole = new Worker.WorkerRole();

            //Act
    workerRole.Run();

            //Assert
    Assert.IsTrue(wasConnectionClosedCalled);
    Assert.IsTrue(wasQueueConstructorCalled);
    Assert.IsTrue(wasEnsureQueueExistsCalled);
    Assert.IsTrue(wasTryProcessOrderCalled);
    Assert.AreEqual(1, numCallsToEnsureQueueExists);
    Assert.IsTrue(numCallsToTryProcessOrder > 0);
}
```

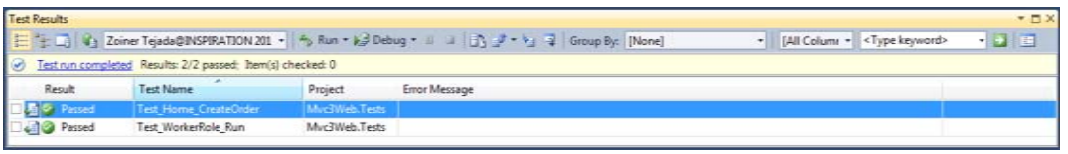
This time we focus on detouring the instance methods of the WorkerRole class. You should set the delegate of the AllInstances property of the generated mole type. By using the delegate, any instance you create of the actual type will detour through any of the methods for which you have defined delegates.

In our example, we want to use the original instance's Run method, but provide detours for the instance methods CreateQueueIfNotExists and TryProcessOrder (in addition to the static method detours we have already seen for RoleEnvironement.GetConfigurationSettings). In our code we throw an exception so that the infinite loop maintained within Run is exited at a predetermined time.

You may be asking yourself, why not just use the MessageSender/MessageReceiver and related classes from the Service Bus SDK directly instead of injecting a helper type? In order to completely isolate our code from calling the real-world Service Bus, we have two choices - write fakes that inherit from the abstract classes in the Microsoft.ServiceBus namespace, or let moles create mock types for them all. The problem with either approach is complexity. With both approaches, you will ultimately find yourself reflecting into classes like TokenProvider and QueueClient, burdened with creating derived types from these abstract types that expose all their required overrides, struggling to expose the internal types that real versions of these classes actually rely on, and recreating their constructors or factory methods in clever ways just to excise the dependency on the real Service Bus. On the other hand, if you insert your own helper type, then that is all you need to mock and detour in order to isolate yourself from the real-world Service Bus.

Enabling Code Coverage

To analyze what these unit tests actually tested, we can examine code coverage data. If we run both of the unit tests with MS Test, we can see that they pass and also see related run details from the Test Results dialog that appears.

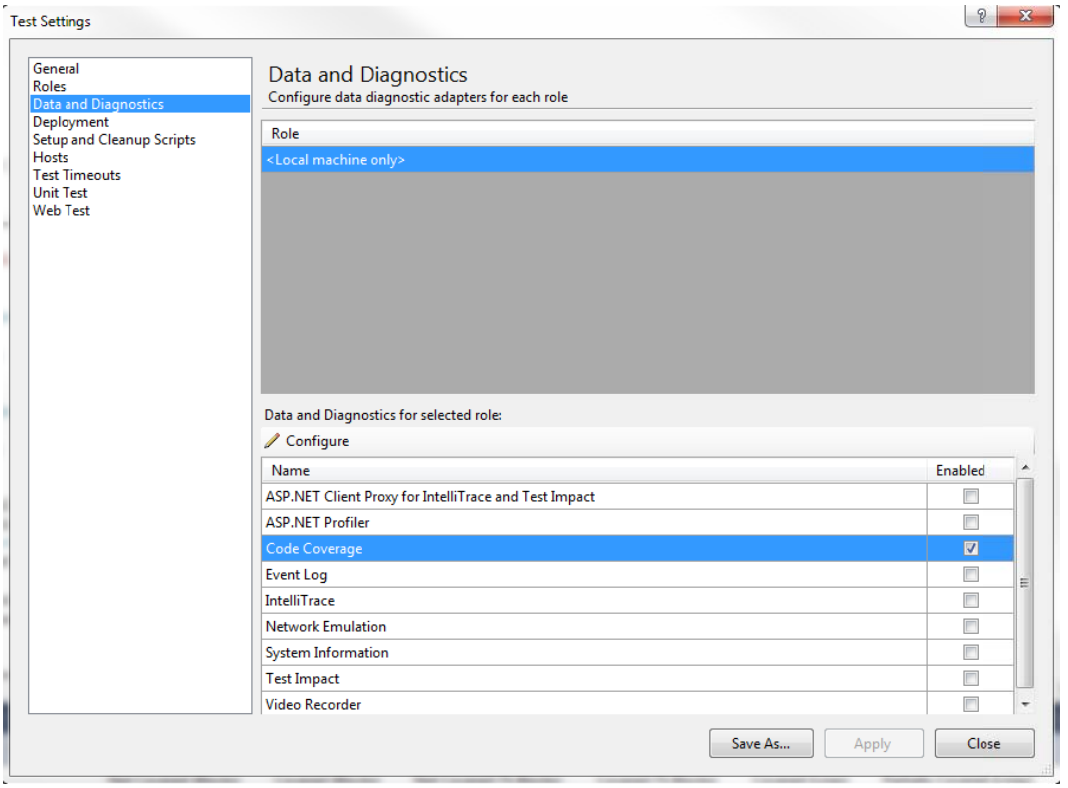


By clicking the Show Code Coverage Results button (the rightmost button on near the top of the Test Results window), we can see the code covered by all unit tests executed in the run. However, if we do this now, we will not see any data. Instead, we will get a message indicating that no coverage data could be found. We first have to enable collection of code coverage data before running our tests.

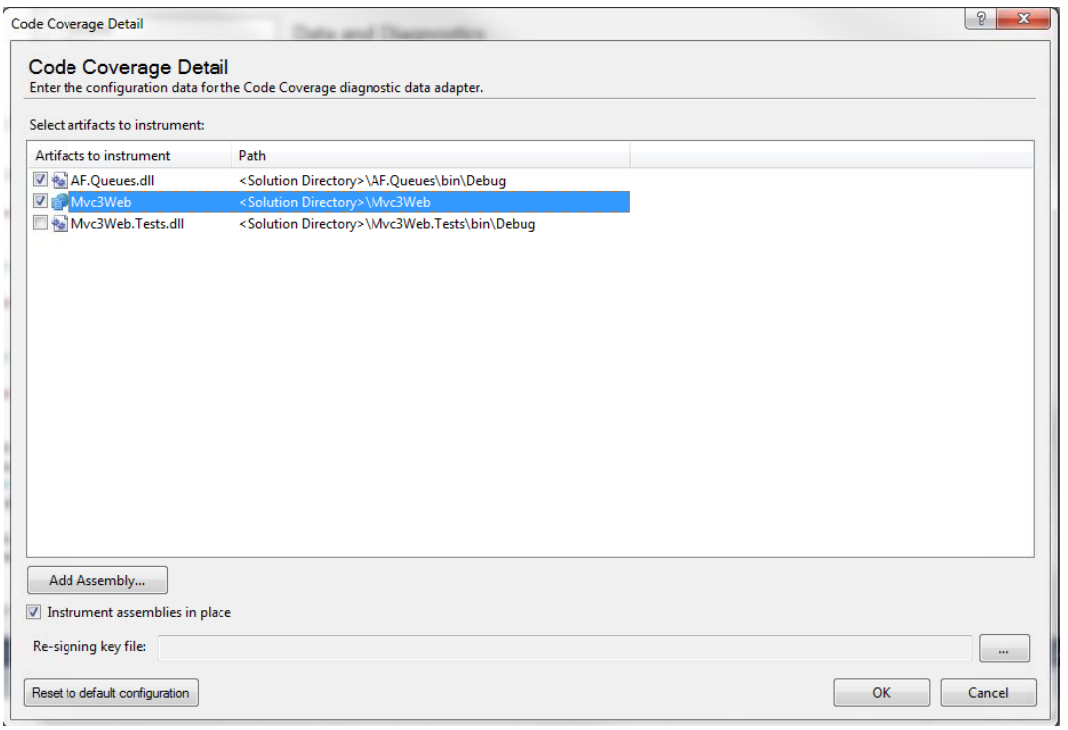
Code coverage data collection is enabled simply by configuring the Local.testsettings under the Solution Items. Opening this file brings up the Test Settings editor.

Note

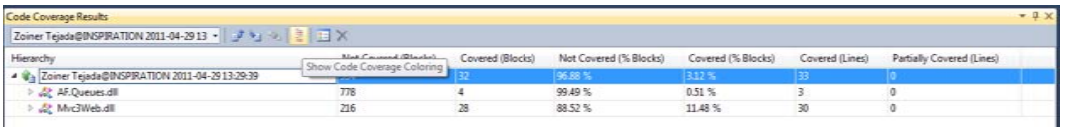
If you have a solution that does not have Local.testsettings file, you can add it using Add New Item to the solution and then select Test >Test Settings.



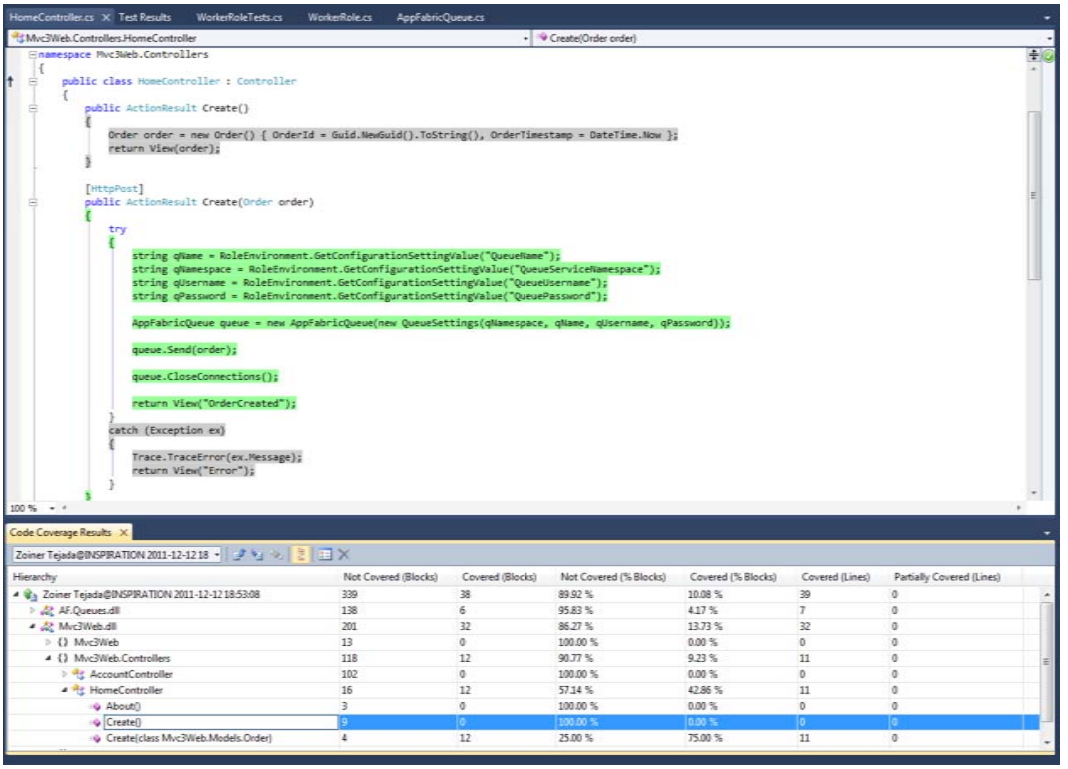
Click the Data and Diagnostics tab, and the check the box to the right of the Code Coverage row. Next, click the Configure button (the one just above the grid containing Code Coverage). In the Code Coverage Detail window select all your assemblies under test and click OK.



Then click Apply and Close to dismiss the Test Settings editor. Re-run your tests and press the code coverage button. Now your Code Coverage Results should look as follows:

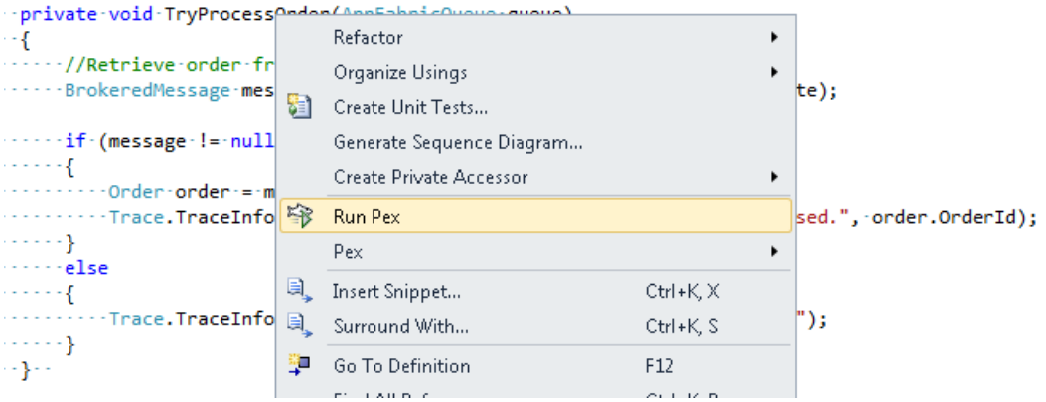


Click the Show Code Coverage Coloring icon, and then drill down to a method in the grid and double-click it to see the source code colored to reflect areas that were tested (green in the following figure), partially tested or untested (gray) the unit tests just run.

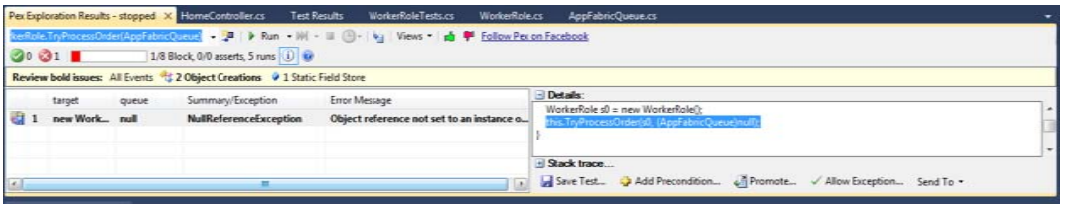


Using PEX to Enhance Code Coverage

Manually creating these unit tests is very valuable, but PEX can help you intelligently take your unit tests to the next level by trying parameter values you may not have considered. After installing PEX, you can have it explore a method simply by right clicking the method in the code editor, and selecting Run PEX.



After a short while, PEX will finish with its processing and the results will stabilize. For example, here is what we got when ran PEX on our worker role's TryProcessOrder method. Notice that PEX was able to create one test that resulted in an exception, and it shows you the inputs it crafted that generated that exception (a null value for the queue parameter). What is even more important, in the details pane you can see the code PEX was executing. The results of all PEX generated tests can be added to your project simply by selecting them and clicking Promote.



Try it out!

The sample used in this article is available at sample_link.

Useful Links

Mocking Frameworks

Moles

[Moq](Moq)

NMock3

Rhino Mocks

EasyMock.NET

NSubstitute

FakeItEasy

TypeMock Isolator

JustMock

Unit Test Related

MSDN – Verifying Code by Using Unit Tests

MSDN - Unit Testing in MVC Applications

Building Testable Applications

Pex and Moles

PrivateObject and PrivateType Introduced

CLR Profiling API Blog

Mocks Aren't Stubs

Dependency Injection and IoC

Microsoft Unity

Structure Map

MSDN Magazine - Aspect Oriented Programming, Interception and Unity 2.0

MSDN Magazine - Interceptors in Unity

MSDN Magazine - MEF v.s. IoC